UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ASPECTOS DE DESEMPENHO DA COMPUTAÇÃO PARALELA EM CLUSTERS E GRIDS PARA PROCESSAMENTO DE IMAGENS

por

GIORGIA DE OLIVEIRA MATTOS

Tese submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como parte dos requisitos para a obtenção do grau de Doutora em Engenharia Elétrica.

ORIENTADOR: RAFAEL DUEIRE LINS, Ph.D

Recife, Dezembro de 2008.

M444a **Mattos, Giorgia de Oliveira.**
Aspectos de desempenho da computação paralela em *Clusters* e *Grids* para processamento de imagens / Giorgia de Oliveira Mattos. - Recife: O Autor, 2008.
vi, 158 folhas, il : tabs. Grafs.

Tese (Doutorado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Elétrica, 2008.

Inclui Bibliografia e Anexos.

1. Engenharia Elétrica. 2. Processamento Paralelo. 3. Clusters. 4.Processamento de Imagens. 5. Balanceamento de Carga. I Título

UFPE **621.3**

BCTG/ 2009-027

Universidade Federal de Pernambuco

Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE TESE DE DOUTORADO

GIORGIA DE OLIVEIRA MATTOS

TÍTULO

"ASPECTOS DE DESEMPENHO DA COMPUTAÇÃO PARALELA EM CLUSTERS E GRIDS PARA PROCESSAMENTO DE IMAGENS"

A comissão examinadora composta pelos professores: RAFAEL DUEIRE LINS, DES/UFPE, JOAQUIM FERREIRA MARTINS FILHO, DES/UFPE, SIANG WUN SONG, DME/USP, FRANCISCO HERON DE CARVALHO JÚNIOR, DC/UFC e MARIA LENCASTRE PINHEIRO DE MENEZES CRUZ, DSC/UPE sob a presidência do primeiro, consideram a candidata **GIORGIA DE OLIVEIRA MATTOS APROVADA**.

Recife, 12 de dezembro de 2008.

EDUARDO FONTANA
Coordenador do PPGEE

RAFAEL DUEIRE LINS
Orientador e Membro Titular Interno

FRANCISCO HERON DE CARVALHO JÚNIOR
Membro Titular Externo

JOAQUIM FERREIRA MARTINS FILHO
Membro Titular Interno

MARIA LENCASTRE PINHEIRO DE MENEZES CRUZ
Membro Titular Externo

SIANG WUN SONG
Membro Titular Externo

Dedico aos meus pais, Isaias Aleixo de Mattos e Irene de Oliveira Mattos, que sempre me incentivaram e me apoiaram em todas as minhas decisões.

Dedico ao meu marido Sabiniano Araújo Rodrigues pela constante motivação e incentivo em enfrentar e superar as dificuldades que existiram.

AGRADECIMENTOS

A Deus, fonte de vida e sabedoria, que me concedeu a graça de ter chegado até aqui.

A minha família que sempre me apoiou incondicionalmente.

A minha amiga Marily, pelo apoio, pela companhia e pelas boas conversas.

Ao professor Rafael Dueire Lins pela sua orientação e ensinamentos durante os anos em que estivemos trabalhando juntos.

Aos professores Siang Song, Francisco Heron, Joaquim Martins e Maria Lencastre pelas suas sugestões e comentários, que foram de grande relevância para esta tese e pelos ensinamentos, os quais levarei para a minha vida profissional.

Aos colegas do doutorado, especialmente ao Andrei de Araújo Formiga pelas discussões e trabalhos que realizamos juntos; e a Giovanna Angelis pela amizade e apoio constantes.

Ao Ricardo Adriano, responsável pelo Laboratório da Telemática que sempre esteve pronto a me ajudar em tudo o que precisei.

Aos funcionários e professores do Departamento de Eletrônica e Sistemas da UFPE.

Ao Isaac, colega que conheci em Paris e que me deu dicas preciosas sobre a compilação de programas em Visual Studio.

A UTFPR Campus Pato Branco que me concedeu a oportunidade de realizar a pós-graduação em nível de doutorado e principalmente aos colegas e professores da COINF com quem aprendi e amadureci profissionalmente e pessoalmente ao longo do tempo em que fui professora daquela coordenação.

Resumo da Tese apresentada à UFPE como parte dos requisitos necessários para a obtenção do grau de Doutora em Engenharia Elétrica.

ASPECTOS DE DESEMPENHO DA COMPUTAÇÃO PARALELA EM CLUSTERS E GRIDS PARA PROCESSAMENTO DE IMAGENS

Giorgia de Oliveira Mattos
Dezembro/2008

Orientador: Rafael Dueire Lins, Dr.
Área de Concentração: Comunicações.
Palavras-chave: Processamento paralelo, clusters, grades, processamento de imagens, balanceamento de carga.
Número de Páginas: 157.

RESUMO: O processo de digitalização de documentos de maneira automática geralmente resulta em imagens com características indesejáveis como orientação errada, inclinação incorreta das linhas do texto e até mesmo a presença de ruídos causados pelo estado de conservação, integridade física e presença ou ausência de poeira no documento e partes do scanner. O simples processamento de cada imagem é efetuado em poucos segundos, a sua transcrição ou busca de palavras-chave para indexação necessita vários segundos de processamento. O lote produzido por um único scanner de linha de produção pode levar dias para ser filtrado, dependendo da capacidade de processamento do computador utilizado. Faz-se necessário distribuir tal tarefa para que o processamento das imagens acompanhe a taxa de digitalização. Esta tese analisa a viabilidade da computação paralela em cluster e grades para o processamento de grandes quantidades de imagens de documentos digitalizados. Diferentes cenários de configuração de computadores e de distribuição de tarefas foram analisados, sob o ponto de vista do processamento das tarefas e do tráfego de rede gerado durante o processamento.

Abstract of Thesis presented to UFPE as a partial fulfillment of the requirements for the degree of Doctor in Electrical Engineering.

ASPECTS OF PERFORMANCE OF PARALLEL COMPUTING IN CLUSTERS AND GRIDS FOR IMAGE PROCESSING

Giorgia de Oliveira Mattos
December /2008

Supervisor: Rafael Dueire Lins, Ph.D.
Area of Concentration: Communications.
Keywords: Parallel processing, clusters, grids, image processing, load balancing.
Number of Pages:157.

ABSTRACT: The digitalization of documents with automatically fed scanners generally yield images with undesirable characteristics such as noise document framing, wrong orientation, document skew and even the presence of noise caused by the state of conservation, physical integrity and the presence or absence of dust in the document and parts of the scanner. The simple processing of each image is made in seconds, while its OCR transcription or the search for keywords for indexing requires several seconds of processing. The batch produced by one single production-line scanner may take days to be filtered, depending on the computer ability of processing used. Thus, it is necessary to paralelize the filtering task to meet the digitalization rate. This thesis examines the viability of parallel computing in cluster and grid large batches of images of scanned documents. Different scenarios of configuration of computers and task distribution are analyzed from the point of view of task processing of and the network traffic generated during processing.

SUMÁRIO

1. INTRODUÇÃO

A evolução tecnológica dos sistemas computacionais tem sido grande e rápida. As tecnologias móveis, a contínua e crescente capacidade computacional de processamento, armazenamento e transmissão de dados, e a adoção de sistemas e padrões abertos trouxeram mudanças significativas à Tecnologia da Informação.

Uma área em crescente expansão é a computação paralela [1] que até a alguns poucos anos estava restrita ao processamento de aplicações especiais em universidades, centros de pesquisa e algumas grandes empresas. Hoje, com os computadores pessoais dotados de mais de um processador, a computação paralela está disponível para os usuários domésticos.

Essa tese apresenta a computação paralela como solução viável para o processamento de grandes quantidades de imagens de documentos digitalizados, através da utilização de ambientes de processamento paralelo. Apresenta também a análise dos resultados obtidos com o processamento das imagens de documentos possibilitando a escolha da melhor solução para o problema proposto.

Para um maior entendimento sobre a evolução dos computadores até a computação paralela, a seguir são apresentados alguns marcos importantes dessa evolução.

Embora as tecnologias empregadas na construção dos computadores tenham evoluído, os conceitos básicos remontam à década de 1940 quando apareceram os primeiros computadores. O ENIAC (Computador e Integrador Numérico Eletrônico – *Eletronic Numerical Integrator and Computer*), projetado e construído sob a supervisão de John Mauchly e John Presper Eckert na Universidade da Pensilvânia, foi o primeiro computador eletrônico digital de propósito geral em todo o mundo [2]. Ele era enorme, pesando 30 toneladas ocupava o espaço de aproximadamente 140 metros quadrados e continha mais de 180 mil válvulas. A operação dessa máquina consumia 140 quilowatts de energia elétrica, era muito mais rápida do que qualquer computador eletromecânico, sendo capaz de executar cinco mil adições por segundo. O ENIAC permaneceu em operação até 1955 quando foi desativado.

Em 1947, Mauchly e Eckert fundaram a Eckert-Mauchly Computer Corporation para fabricar computadores comercialmente. Sua primeira máquina de sucesso foi o UNIVAC I (Computador Automático Universal – *Universal Automatic Computer*) que tinha o propósito de servir tanto aplicações científicas quanto aplicações comerciais, tais como cálculos algébricos sobre matrizes, resolução de problemas estatísticos, cálculo de prêmio de seguro para uma companhia seguradora e solução de problemas logísticos [3]. No final dos anos 50 foi lançado o UNIVAC II, ele possuía maior capacidade de memória e maior desempenho que o UNIVAC I, e ilustra

tendências que permaneceram na indústria de computadores. A primeira é que os avanços da tecnologia permitiram que as companhias continuassem a desenvolver computadores cada vez mais poderosos e maiores; e a segunda é que cada companhia procurava construir suas novas máquinas de modo que fossem compatíveis com as máquinas anteriores [3].

A primeira grande mudança nos computadores eletrônicos veio com a substituição da válvula pelo transistor [4]. O transistor é menor, mais barato e dissipa menos calor e, assim como a válvula, também pode ser utilizado para a construção de computadores. Ao contrário da válvula, que requer o uso de fios, placas de metal, cápsula de vidro e vácuo, o transistor é um dispositivo de estado sólido, feito de silício [3].

O uso dos transistores criou a segunda geração de computadores. É comum classificar os computadores em gerações [2] [3] [4], de acordo com a tecnologia básica de hardware empregada. Cada geração é caracterizada por computadores com maior velocidade, maior capacidade de memória e menor tamanho que os computadores da geração anterior. A Tabela 1 mostra as gerações dos computadores e algumas de suas características.

Tabela 1.1 – *Geração de computadores.*

Geração	***Datas aproximadas***	***Tecnologia***	Velocidade típica (operações por segundo)
1	1946-1957	Válvula	40.000
2	1958-1964	Transistor	200.000
3	1965-1971	Integração em baixas e médias escalas	1.000.000
4	1972-1977	Integração em grande escala	10.000.000
5	1978-	Integração em escala muito grande	100.000.000

O transistor foi inventado pela *Bell Laboratories* em 1947 e iniciou uma revolução na indústria eletrônica nos anos 50. Entretanto, apenas no final da década de 50, computadores totalmente transistorizados tornaram-se comercialmente disponíveis. As empresas pioneiras nesse novo tipo de computador foram a NCR, RCA e a IBM com a série 7000. A IBM que era um dos maiores fabricantes de dispositivos de processamento de cartões perfurados lançou em 1953 seu primeiro computador eletrônico programável, o 701, sendo inicialmente voltado para aplicações científicas. Em 1955 foi lançado o modelo 702 que possuía várias características de hardware que o tornavam adequado para aplicações comerciais. Esses foram os primeiros de uma longa série de computadores 700/7000 que estabeleceram a IBM como um dos maiores fabricante de computadores do mercado.

Durante a década de 50 e o início dos anos 60, os equipamentos eletrônicos eram compostos basicamente de componentes discretos – transistores, resistores, capacitores e assim por diante.

Esses componentes eram fabricados separadamente, encapsulados em seus próprios recipientes e soldados ou ligados com fios nas placas de circuito, que eram então instaladas nos computadores, osciloscópios e outros equipamentos eletrônicos. O processo completo de fabricação, desde o transistor até a placa de circuito, era caro e complicado, e isso começava a criar problemas na indústria de computadores, pois os computadores do início da segunda geração continham cerca de 10 mil transistores. Esse número cresceu até centenas de milhares, tornando cada vez mais difícil a fabricação de máquinas novas e mais poderosas. Assim, em 1958 foi desenvolvida uma nova técnica que revolucionou os equipamentos eletrônicos e iniciou a era da microeletrônica: a invenção do circuito integrado. Esse circuito caracteriza a terceira geração de computadores.

A invenção dos circuitos integrados proporcionou mais um grande avanço na evolução da computação. Os transistores foram substituídos por essa nova tecnologia e sua utilização tornou os equipamentos mais compactos, rápidos e confiáveis, com menor consumo de energia e custo que os das gerações anteriores. Inicialmente, era possível fabricar e empacotar em uma única pastilha apenas um pequeno número de componentes e por isso esses primeiros circuitos integrados são chamados circuitos com integração em baixa escala (*Small-Scale Integration* – SSI). Com o passar do tempo, foi possível empacotar mais e mais componentes em uma mesma pastilha.

A partir da terceira geração de computadores, existe um menor consenso sobre a definição das demais gerações de computadores. A Tabela 1 sugere a existência de uma quarta e uma quinta gerações, com base na evolução da tecnologia de circuitos integrados. Com a introdução da integração em grande escala (*Large-Scale Integration* – LSI), mais de mil componentes podem ser colocados em uma única pastilha de circuito integrado. A integração em escala muito grande (*Very-Large-Scale Integration* – VLSI) atingiu mais de 10 mil componentes por pastilha, e as pastilhas VLSI atuais contêm milhões de componentes.

1.1 Aspectos Históricos e Atuais da Computação Paralela

Nos anos 70 surgiram os primeiros supercomputadores [3]. Até aquele momento, as arquiteturas eram simples e o aumento da eficiência computacional estava limitado pelo desenvolvimento tecnológico, principalmente pelo fato dos processadores terem que terminar uma tarefa antes de iniciar outra [2]. Dessa forma, percebeu-se que a divisão de tarefas traria avanços significativos quanto ao desempenho das máquinas, surgindo a partir desta época dois caminhos distintos: arquiteturas paralelas e sistemas distribuídos.

Em 1975 nasce o ILLIAC IV, dotado de 64 processadores, considerado o computador mais rápido até o surgimento do supercomputador Cray-1, fabricado por Seymor Cray em 1976. O Cray-1 foi o primeiro supercomputador a ficar mundialmente famoso por iniciar o que seria chamado de processamento vetorial ou *pipeline*, ou seja, uma mesma operação é aplicada em uma grande

quantidade de dados, ao mesmo tempo, desde que não exista dependência entre os dados. Seu processador executava uma instrução dividindo-a em partes, ou seja, enquanto a segunda parte de uma instrução estava sendo processada, a primeira parte de outra instrução era inicializada.

O desempenho desse tipo de computador era extremamente alto e seu preço chegava a alguns milhões de dólares. O Cray-1 era capaz de processar até 133 megaflops, 133 milhões de operações aritméticas de ponto flutuante por segundo. Em 1985, seu sucessor, o Cray-2, tinha capacidade de desempenho de 1,9 gigaflops, 1,9 bilhões de operações aritméticas de ponto flutuante por segundo, tendo a maior memória do mundo, 2 gigabytes. Essas arquiteturas dominaram por cerca de duas décadas o mercado da supercomputação. Com o avanço da microeletrônica e dos circuitos integrados miniaturizando os componentes de um computador, uma nova vertente de supercomputadores surgiu: os compostos de milhares de processadores simples, as chamadas máquinas massivamente paralelas. Tais computadores têm um custo menor do que as arquiteturas vetoriais e envolvem processadores já bem conhecidos pelos programadores, o que facilitava a programação. Atualmente, os supercomputadores possuem altíssimo poder de processamento se comparados aos existentes há duas décadas; são fabricados por empresas como IBM, DELL, CRAY, etc.

A Tabela 2 mostra a relação dos dez supercomputadores mais rápidos do mundo até junho de 2008, disponibilizada através do site http://www.top500.org. Hoje em dia há um revezamento entre as duas vertentes de supercomputadores. Observando a lista dos supercomputadores mais rápidos do mundo observa-se um domínio da IBM, com duas arquiteturas diferentes: Blue Gene e Roadrunner.

O Blue Gene [5] [6] é uma parceria entre a IBM e instituições de pesquisa para desenvolver um computador com alto custo/benefício quando aplicado a simulações de biologia, farmácia e outras aplicações que necessitem de processamento massivo de dados. A estratégia de desenvolvimento foi criar um computador composto por milhares de processadores simples, podendo chegar a centenas de milhares deles em alguns casos. O Blue Gene ocupou o topo da lista do Top500 até junho de 2008, com um desempenho de cerca de 500 teraflops, 500 trilhões de operações aritméticas de ponto flutuante por segundo. Atualmente, quatro entre os dez computadores mais rápidos são da família Blue Gene. O Roadrunner [7], atualmente o computador mais rápido do mundo, combina processadores *Dual Core* da AMD (*Advanced Micro Devices*), similares aos encontrados em notebooks, com o chip *Cell Broadband Engine*, da IBM, anunciados inicialmente para uso no videogame Sony Playstation 3. O supercomputador foi encomendado pelo Departamento Nacional de Administração e Segurança de Energia Nuclear dos Estados Unidos, para ser usado no Laboratório Nacional de Los Alamos, Novo México, com a finalidade de realizar simulações que permitam atestar a confiabilidade do arsenal de armas nucleares norte-americanas

sem ter de recorrer a testes nucleares subterrâneos. Seu poder de processamento pode chegar a 1,2 petaflops, 1,2 quadrilhões de operações aritméticas de ponto flutuante por segundo.

Tabela 2.2 – *Relação dos 10 maiores supercomputadores do mundo.*

Fonte: www.top500.org.

Rank	**Fabricante**	**Computador**	**Localização**	**Proces-sador**	**Desempenho TFlops**
1	IBM	Roadrunner – Blade Center QS22/LS21 Cluster, PowerXCell 8i 3.2Ghz/ Opteron DC 1.8 GHz, Voltaire Infiniband	DOE/NNSA/ LANL (Estados Unidos)	122,400	1026,00
2	IBM	BlueGene/L - eServer Blue Gene Solution	DOE/NNSA/ LANL (Estados Unidos)	212,992	478,20
3	IBM	Argonne National Laboratory	Blue Gene/P Solution (Estados Unidos)	163,840	450,30
4	Sun Micro systems	Texas Advanced Computing Center/ University of Texas	Ranger – SunBlade x6420, Opteron Quad 2Ghz, Infiniband (Estados Unidos)	62,976	326,00
5	Cray Inc.	Jaguar - Cray XT4 QuadCore 2.1 GHz	DOE/Oak Ridge National Lab. (Estados Unidos)	30,976	205,00
6	IBM	JUGENE - Blue Gene/P Solution	Forschungszentrum Juelich (FZJ) (Alemanha)	65,536	180,00
7	SGI	Encanto - SGI Altix ICE 8200, Xeon quad core 3.0 GHz	New México Computing Applications Center (NMCAC) (Estados Unidos)	14,336	133,20
8	Hewlett-Packard	EKA - Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband	Computational Research Lab. TATA SONS (Índia)	14,384	132,80
9	IBM	Blue Gene/P Solution	IDRIS (França)	40,960	112,50
10	SGI	SGI Altix ICE 8200EX, Xeon quad core 3.0 GHz	Total Exploration Production (França)	10,240	106,10

De acordo com projetos em andamento da IBM, a tendência é usar a tecnologia de chips de alta densidade e de múltiplos núcleos (*cores*) para construir chips onde cada unidade é constituída de dezenas ou centenas de processadores, praticamente um supercomputador. Com isso, será possível observar máquinas com desempenhos superiores aos atuais, contribuindo com o

desenvolvimento de novas tecnologias, as quais provavelmente serão usadas em novos computadores, mantendo assim o ciclo de desenvolvimento.

Outra forma de realizar o processamento paralelo a um custo inferior ao obtido com os supercomputadores é a utilização das redes de computadores para tal. O conceito inicial de utilização de uma rede de computadores para o processamento paralelo foi desenvolvido na década de 1960 pela IBM com o intuito de interligar grandes mainframes e obter uma solução comercialmente viável de paralelismo. Mas foi em 1980 que os clusters [8] ganharam força quando três tendências convergiram: os microprocessadores de alto desempenho, as redes de alta velocidade e as ferramentas padronizadas para computação distribuída de alto desempenho. A crescente necessidade de poder de processamento para aplicações científicas e comerciais, unida ao alto custo e a baixa acessibilidade dos tradicionais supercomputadores também contribuíram para a consolidação dos clusters. O uso das redes de computadores abriu novos horizontes computacionais, tornando o processamento de alto desempenho (*High Performance Computing* – HPC) acessível a usuários das mais diversas realidades econômicas. Surgiram assim os *clusters* computacionais e por isso, em geral, eles são associados a uma rede local, um conjunto de computadores interligados para o processamento de determinada tarefa.

Os clusters estão sendo usados em uma grande quantidade de aplicações, especialmente aquelas que exigem alto poder de processamento, como por exemplo, a previsão meteorológica, simulações de eventos no solo, renderização de efeitos especiais, simulações financeiras, distribuição de carga em sistemas de potência, etc. Sua utilização inclui ainda qualquer tipo de aplicação crítica, ou seja, aplicações que não podem parar de funcionar ou não podem perder dados, como os sistemas de bancos, por exemplo.

As grades computacionais [9] ou *grids* surgiram na década de 90 através de pesquisas da comunidade de alto desempenho com o objetivo de utilizar recursos computacionais ociosos de computadores independentes e amplamente dispersos para executar aplicações paralelas. As grades, portanto estiveram sempre ligadas a redes geograficamente espalhadas em WANs (redes de longa distância) ou internet. A possibilidade de alocar uma quantidade enorme de recursos para uma aplicação paralela (centenas de máquinas conectadas através da Internet) e fazer isso com custo mais baixo que alternativas tradicionais, baseada em supercomputadores paralelos, são algumas das principais vantagens da utilização das grades computacionais, além do grande poder de armazenamento de dados, disponibilidade de dados e tempo de acesso aos dados. No entanto os softwares de grade são considerados de complexidade elevada.

As grades computacionais diferem da computação distribuída convencional ou cluster computacional por ter seu foco no compartilhamento de recursos em larga escala, sobre o ambiente de redes amplamente conectadas, tipicamente a Internet [10]. Como aconteceu no cluster

computacional, o acesso transparente aos recursos através da rede ainda é o principal interesse da grade computacional, mas com complexidade maior para as redes de larga escala.

A computação em grade ganhou destaque nos últimos tempos com algumas empresas investindo no desenvolvimento dessa tecnologia. Um exemplo é a IBM que tem investido em pesquisa e desenvolvimento de ferramentas em grade para ambiente corporativo, sendo uma das principais colaboradoras no desenvolvimento do Globus Toolkit [11]. No meio científico é possível encontrar várias grades em funcionamento espalhadas por alguns países, muitas sendo projetos multi-institucionais. Como exemplo tem-se: o Datagrid [12] [13] do CERN (*Conseil Européen pour la Recherche Nucléaire* – Conselho Europeu para a Pesquisa Nuclear), um projeto financiado pela Comunidade Européia com o objetivo de atuar em áreas de pesquisa como astronomia, física e biologia; o BIRN [14] (*Biomedical Informatics Research Network* – Rede de Pesquisa em Informática Biomédica), projeto multi-institucional que conta com quinze universidades norte-americanas, voltado para a pesquisa neurológica; e o projeto Mammogrid [15], uma iniciativa da comunidade européia para formar uma base de mamografias que abrange toda a Europa com o intuito de fornecer material de estudo e campo para o desenvolvimento de tecnologias em grades computacionais. No Brasil, um exemplo é o Sprace [16], projeto de uma grade do Instituto de Física da Universidade de São Paulo que participa do processamento dos dados provenientes do projeto D0, projeto que reúne pesquisadores do mundo todo para analisar os dados gerados pelo acelerador de alta energia *Tevatron Collider*, localizado em Illinois, Estados Unidos.

Por permitirem o emprego de vários computadores de uso pessoal na resolução de problemas que exigem mais poder computacional do que um único deles pode oferecer, os clusters e as grades são, atualmente, duas maneiras possíveis de tratar tais problemas que são objeto de estudo da computação de alto desempenho – possíveis quando comparado ao uso de supercomputadores, por exemplo.

Um exemplo de tarefa de alto desempenho de grande relevância para as organizações é o tratamento de imagens de documentos digitalizados. Com a necessidade de interligar os fluxos de documentos de uma organização aos seus sistemas de Tecnologia da Informação é preciso digitalizar, manipular e armazenar os documentos em papel existentes. O processo de digitalização freqüentemente resulta em imagens digitais com características indesejáveis; por exemplo, o documento quando digitalizado dificilmente estará orientado de maneira satisfatória, incluindo uma inclinação das linhas do texto com relação à horizontal, conforme ilustrado pela Figura 1.1. A correção desses artefatos da digitalização pode ser feita automaticamente através de programas especializados [17] [18] [19] [20] [21], resultando em imagens mais adequadas para o armazenamento em um acervo de documentos. O simples processamento de cada imagem é efetuado em poucos segundos, a sua transcrição ou busca de palavras-chave para indexação necessita vários segundos de processamento. O processamento de um lote produzido por um único

scanner pode demorar dias, dependendo da capacidade de processamento do computador utilizado isoladamente. Então é necessário distribuir a tarefa para que o processamento das imagens acompanhe a taxa de digitalização.

O problema apresentado é um exemplo real que necessita de uma solução de computação de alto desempenho. A natureza desse problema, tal qual muitos outros, é de paralelismo de dados, e o processamento de uma imagem independe do processamento das outras. Problemas dessa natureza são naturalmente passivos de processamento paralelo, podendo-se utilizar cluster ou grade para resolvê-lo.

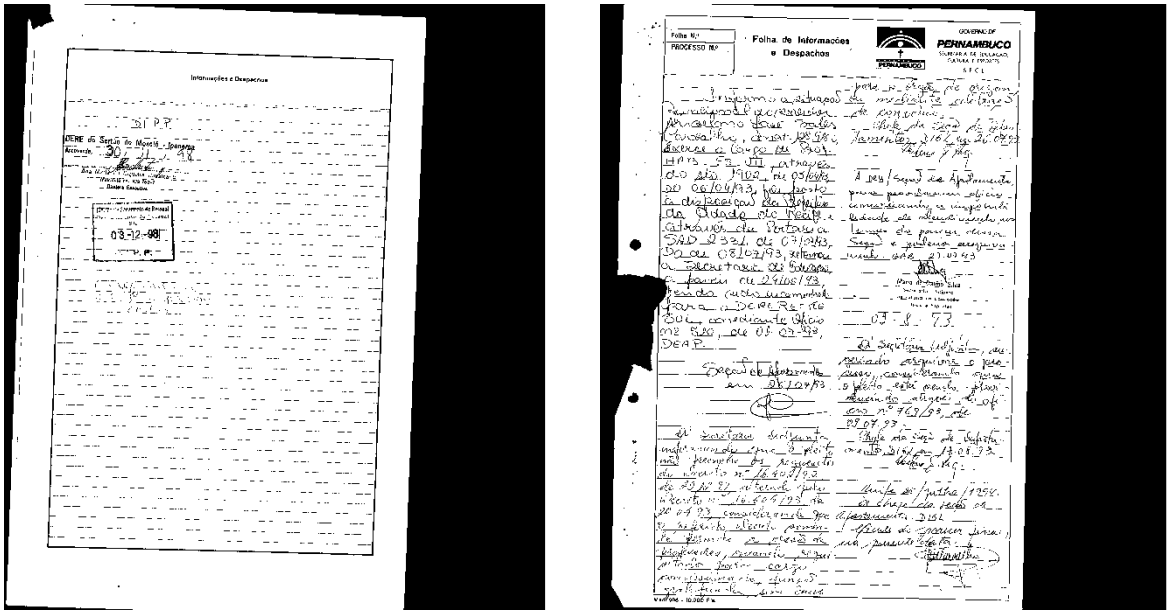


Figura 1.1 – *Exemplos de imagens digitalizadas com scanner.*

1.2 Estado da Arte em Processamento Paralelo de Imagens

Nos últimos anos, o processamento de imagens e a multimídia tornaram-se uma tecnologia chave para a computação moderna. Além das tradicionais aplicações de processamento de imagem surgiram novos domínios, como a vídeo-conferência e o vídeo sob demanda, enquanto outros tiveram um lugar de destaque como o processamento de imagens médicas. Tais aplicações geram grandes conjuntos de dados para armazenagem e recuperação das imagens, gerando a necessidade de realizar análises complexas nas imagens em vários domínios. Conseqüentemente aumenta as exigências computacionais para manipulá-las. O esforço computacional necessário e a necessidade de tempo de resposta rápido, combinado com a natureza inerentemente paralela dos dados das imagens, levaram ao uso da computação paralela em aplicações de processamento de imagens.

Por muitos anos, as arquiteturas paralelas foram estudadas para resolver problemas do processamento de imagens. Durante os anos 50, algumas propostas para arquiteturas maciçamente paralelas apareceram [22]. Mas foi ao longo dos anos 70 que se percebeu que muitos processamentos locais e idênticos eram necessários, de forma que se tornou natural pensar em replicar o executor de instruções, um para cada pixel ou para cada instrução.

No início dos anos 80 o desenvolvimento de programas para arquiteturas multiprocessadas ainda era uma tarefa difícil mesmo quando utilizados microprocessadores comerciais como o Z80. Além disso, a passagem para algoritmos mais sofisticados de análise de imagens levaram à necessidade de enriquecer os mecanismos de comunicação entre os processadores elementares. Assim, durante os anos 80, os pesquisadores consideraram o uso de várias topologias permitindo o enriquecimento das comunicações entre os nós por uma máquina paralela: pirâmide, hipercubo, etc. A maioria dos resultados em processamento de imagens relatam o uso da topologia em pirâmide; cuja interconexão reflete os movimentos dos dados presentes em muitos algoritmos de processamento de imagens, em particular o processamento de multiresolução. Diversos projetos foram assim realizados [23], [24] e [25].

Do ponto de vista das ferramentas de programação, várias abordagens surgiram com o intuito de integrar os conhecimentos de domínio específicos. Assim, linguagens de programação específicas para o processamento paralelo de imagens apareceram. Como exemplos, é possível citar as linguagens Apply [26] e Adapt [27], a linguagem denominada IAL (*Image Algebra SIMD Programming Language* [28] - Linguagem de programação SIMD de álgebra da imagem), I-BOL [29] e Tulip [30], extensões da IAL, além de várias bibliotecas para o desenvolvimento de aplicações paralelas como ANET.

Por outro lado, arquiteturas de uso geral e conceitos de programação foram desenvolvidos e são amplamente utilizados atualmente em todo o mundo, tornando-se uma ferramenta quotidiana de um grande número de usuários. Dois exemplos dessas arquiteturas são os *clusters* e as *grades*. Clusters estão presentes na maioria dos laboratórios e em um número crescente de indústrias. Mais recentemente, a tecnologia emergente de grade [31] promete conectar por completo os recursos computacionais a fim de fornecer poder computacional virtualmente ilimitado. Um dos avanços da investigação consiste no desenvolvimento de camadas especializadas de alto nível que levam em conta as exigências específicas para o processamento de vídeo e imagem. Algumas recentes experiências bem sucedidas no processamento de imagens biomédicas podem ser encontradas em [32] e [33].

A grande maioria dos trabalhos abrange o tópico da computação paralela de propósito geral para aplicações de processamento de imagem. O trabalho apresentado em [34] é dedicado aos sistemas paralelos de recuperação da imagem baseado no conteúdo (CBIR). Esta aplicação destina-se a recuperar imagens baseadas na similaridade do seu conteúdo e é uma questão importante para

lidar com as imagens imensas atuais e futuras arquivadas; a demanda pela implementação paralela, principalmente implementações de memória distribuída, através da utilização de um cluster de PCs, é extremamente necessária. Assim, esse trabalho relata a utilização de *wavelet*s e métodos baseados em Gabor para extrair características da imagem. Técnicas para o particionamento da formação da imagem, execução paralela de consultas e estratégias para balanceamento de carga são explicados, considerando a base de dados paralela de imagens CAIRO como exemplo.

Em [35], Oh e Aizawa apresentam um sistema de sensoriamento experimental de imagens em larga escala. O sistema utiliza sensores inteligentes de imagens de amostragem espacialmente-variante (SVS) para reduzir o volume de dados durante a aquisição da imagem através de esquemas de redução da resolução espacial/temporal, considerando a importância das regiões no cenário. Todo o tráfego do sistema é dinamicamente controlado pela sub-amostragem em uma região inativa (IR), ou seja, a região sem uma mudança significativa, em nível de aquisição e transmissão de imagem IR com a taxa temporal mais baixa do que a região ativa (AR), a região alterada detectada pela diferença do *frame*. O sistema, implementado em ambiente real, preserva os recursos da rede.

Colombo, Del Bimbo e Valli apresentam em [36] o monitoramento não-intrusivo do movimento do corpo humano e o seu uso para animação avatar. Os autores propõem e descrevem um sistema para monitoramento da postura corporal baseado na visão, o qual conta com estações assimétricas interconectadas que estão incumbidas de tarefas específicas no sistema de monitoramento. As tarefas selecionadas incluem (a) processamento de imagem de baixo nível e monitoramento do corpo em ambas as imagens, esquerda e direita; (b) análise estéreo e manipulação da oclusão; (c) reconstrução da postura corporal do dado estéreo através da cinemática inversa e (d) atualização do caractere virtual baseado em computação gráfica. O sistema mostra ser capaz de animar um boneco virtual em 3D através dos movimentos do corpo além de ser eficiente em termos de tempo de processamento, precisão da reconstrução 3D, monitoramento e matching na presença de oclusões e renderização gráfica.

Isgrò e Tegolo [37] enfrentam um problema típico de restauração de vídeo digital, ou seja, o restabelecimento de arranhões da linha vertical. O problema computacionalmente intensivo sobre um vídeo de algumas horas é resolvido sobre um sistema de memória distribuída, especificamente uma rede de estações de suporte de sistemas operacionais heterogêneos. Os autores projetam um algoritmo distribuído para resolver o problema pela adoção de um algoritmo genético *ad hoc*.

Preocupado com o ambiente de programação paralela, Jonker, Olk e Nicolescu [38] relatam uma estrutura de programação que considera simultaneamente o paralelismo baseado em pixel e objeto. Eles correspondem a mecanismos fundamentais em processamento paralelo (paralelismo de dados e tarefas) e ambos são importantes para uma utilização eficiente do hardware paralelo.

1.3 Esta Tese

Essa seção descreve a motivação e relevância do trabalho de pesquisa foco desta tese, bem como a metodologia adotada e estrutura dessa tese.

1.3.1 Motivação e Relevância

A fim de automatizar o processo de digitalização de documentos é habitual utilizar scanners para tal tarefa. Tais imagens de documentos são obtidas por scanners de linha de produção com alimentação automática, sendo produzidos, dependendo do tamanho dos documentos, cerca de mais de 1.000 imagens de documentos por hora. O processo de digitalizar as imagens utilizando scanners de linha de produção com alimentação automática geralmente introduz uma série de características indesejáveis às imagens dos documentos, sendo necessário o processamento das imagens digitalizadas a fim de se obter documentos digitais úteis.

Processar milhares de documentos não é possível para um único computador isolado, de maneira que é necessário distribuir a tarefa entre vários computadores para que o processamento das imagens acompanhe a taxa de digitalização. Como o processamento de uma imagem não depende do processamento de outra imagem, isso caracteriza o paralelismo dos dados e a possibilidade do problema do processamento ser resolvido através do processamento paralelo. Em geral, a escolha da arquitetura de cluster ou de grade depende das características intrínsecas do problema e dos computadores disponíveis; visto que clusters são normalmente organizados em redes locais, com recursos dedicados e homogêneos, enquanto as grades geralmente são utilizadas em WANs e Internet, não dedicadas e heterogêneas. Tal maneira de dividir as arquiteturas trouxe associada a preconceituosa idéia de que o custo computacional de uma grade seria maior do que um cluster, devido à latência na comunicação, roteamento inerentemente mais complexo, etc. Entretanto, não há na literatura qualquer referência que realmente justifique tal preconceito ou mesmo que quantifique os *overheads* do uso da arquitetura grade em relação a cluster.

Nesse contexto, foram estudadas algumas características do processamento de imagens de documentos em clusters e grades, como o tempo de processamento das tarefas, a quantidade de computadores disponíveis para processar as tarefas e ainda o tipo de computador utilizado. Com isso, foi possível observar o comportamento de sistemas de cluster e grade no mesmo ambiente físico de rede a fim de comparar o desempenho das duas soluções em uma rede local, bem como estabelecer critérios de ajuste para as mesmas, possibilitando aumentar a vazão desses sistemas. Essa comparação pretende estabelecer, por exemplo, se vale à pena utilizar uma solução do tipo grade para resolver problemas tipicamente resolvidos com organizações do tipo cluster.

1.3.2 Metodologia

Para realizar a análise comparativa entre as plataformas de cluster e grade para o processamento de imagens foram necessárias algumas ações como a escolha do software de grade a ser utilizado, o desenvolvimento de ferramentas para o escalonamento das tarefas no caso do cluster, preparação da base de imagens a ser utilizada nos experimentos e a análise dos dados obtidos.

Existem vários softwares de grade disponíveis no mercado, dentre eles o Condor [39], o Globus [11] e o Ourgrid [40] [41]. Nos experimentos realizados, optou-se por utilizar o OurGrid por ser uma ferramenta de código aberto que atendia aos requisitos necessários ao estudo a ser realizado além da fácil configuração e utilização.

Da mesma forma, há disponível uma grande variedade de bibliotecas de software para cluster e softwares que podem ser usados para ajudar a gerenciar tarefas em um cluster, como por exemplo, OpenMosix [42], Condor [43] e Microsoft Cluster Server [44] [45]. Porém, é mais comum que aplicações sejam explicitamente desenvolvidas para o cluster a ser utilizado, incorporando a divisão de tarefas entre os nós e a comunicação entre eles. A programação de tarefas para o cluster geralmente utiliza bibliotecas especializadas, tais como MPI [46] e OpenMP [47]. Dessa forma, inicialmente foi desenvolvido um software com o objetivo de distribuir as tarefas para os nós do cluster. Posteriormente outro software para o cluster foi desenvolvido utilizando o padrão MPI. Assim, foi possível comparar o cluster usando software escrito com MPI e o programa escrito especialmente para o processamento dos documentos no cluster.

Experimentos foram realizados utilizando diversos cenários de configurações tanto com relação aos computadores utilizados quanto aos dados processados. Por fim, a análise dos resultados obtidos foi realizada.

1.3.3 Estrutura da Tese

Esta tese é formada por 7 capítulos além deste de Introdução. O capítulo 1 faz uma introdução sobre a evolução dos computadores até a computação de alto desempenho apresentando as arquiteturas de cluster e grades computacionais. Apresenta ainda a motivação, relevância e metodologia adotada no desenvolvimento dessa tese. O capítulo 2 apresenta o sistema paralelo através da sua definição e classificação. Aspectos relacionados ao desempenho desses sistemas bem como os fatores que podem degradar o desempenho dos sistemas paralelos também são apresentados.

No capítulo 3 são tratados em detalhes os ambientes de cluster e grade computacional com as suas definições, classificações e exemplos, além de outros fatores necessários ao entendimento desses ambientes. No capítulo 4 é apresentado em detalhes o problema que esse trabalho se propõe a resolver além de todos os elementos necessários ao seu desenvolvimento.

O capítulo 5 mostra os resultados de experimentos realizados com o processamento de imagens de documentos nos ambientes de clusters e grade. A análise sobre os resultados obtidos também é mostrada nesse capítulo. O capítulo 6 traz a análise sobre o impacto do tráfego de rede gerado durante o processamento das imagens de documentos. Finalmente, o capítulo 7 apresenta as conclusões e perspectivas de trabalhos futuros visando a continuidade do estudo.

2. SISTEMAS PARALELOS DE COMPUTAÇÃO

Desde o surgimento do primeiro computador digital eletrônico, o ENIAC, em 1946, a computação passou por um processo evolutivo intenso em nível de hardware e software, a fim de proporcionar maior desempenho e ampliar o leque de aplicações que podem ser computacionalmente resolvidas de maneira eficiente.

O processamento paralelo implica na divisão de uma determinada aplicação de maneira que ela possa ser executada por vários elementos de processamento, que por sua vez deverão cooperar entre si (comunicação e sincronização), buscando eficiência através da quebra do paradigma da execução seqüencial do fluxo de instruções. Este capítulo introduz os conceitos de processamento paralelo, a classificação desses sistemas bem como os elementos utilizados para medir o desempenho desses sistemas.

2.1 O Sistema Paralelo

No final dos anos 40, John von Neumann e um grupo de pesquisadores da Universidade da Pensilvânia propuseram o ENIAC [2] que deu início a era dos computadores modernos. Passados 50 anos, a maioria dos computadores ainda seguia, mais ou menos, o projeto original onde um computador consiste essencialmente de uma única unidade de processamento, ou processador, que executa uma seqüência única de instruções em uma seqüência única de dados [48], como mostrado na Figura 2.1.

A seqüência de instruções é o programa que diz ao processador como resolver determinado problema, enquanto a seqüência de dados é uma instância do problema. A cada passo durante o processamento, a unidade de controle envia uma instrução ao processador que opera com um par de números, por exemplo, obtido da memória. Pode-se tomar como exemplo alguma operação aritmética ou lógica e colocar o resultado na memória. O processador tem uma pequena memória local, formada de um número constante de registradores de tamanho fixo para realizar o processamento. Ele também está conectado a uma unidade de entrada e de saída de dados para que se comunique com o mundo exterior. Este modelo de processamento é conhecido como **processamento seqüencial** ou **serial**.

Nos anos 80 acreditava-se que o desempenho dos computadores seria melhorado pela criação de processadores mais rápidos e mais eficientes. Essa idéia foi mudada pelo processamento paralelo, cuja essência está em ligar dois ou mais computadores para juntos solucionarem algum

problema computacional. Desde o início dos anos 90 com a rápida melhoria na disponibilidade de componentes de alto desempenho para estações de trabalho e redes e a conseqüente diminuição do seu custo, as redes de computadores tornaram-se o caminho para a computação paralela [8].

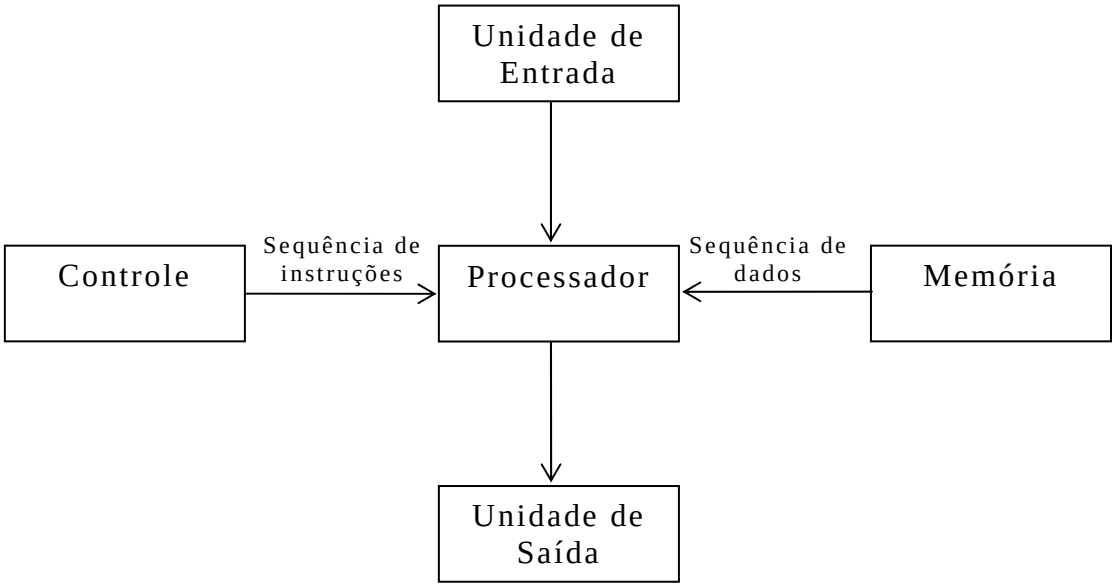


Figura 2.1 – *Um computador seqüencial.*

No **processamento paralelo** existem vários processadores, dois ou mais. Dado um problema a ser resolvido, esse é quebrado em um número de subproblemas que são resolvidos simultaneamente, cada um em um processador diferente. Ao fazê-lo, os processadores podem comunicar-se entre si para troca de resultados parciais. Finalmente, os resultados são combinados para gerar uma resposta ao problema original.

A Figura 2.2 ilustra um computador paralelo com memória compartilhada [48]. Nessa figura existem 5 processadores conectados em uma única memória compartilhada onde os processadores usam essa memória para comunicar-se entre si. Um par de processadores que deseje trocar dados pode fazê-lo através da memória compartilhada: um processador escreve seus dados em determinada localização da memória, o qual é então lido por outro processador. Para a solução de um determinado problema, o programa é o mesmo para todos os processadores, e cada processador tem memória local suficiente para armazenar sua própria cópia do programa. Esses processadores trabalham de maneira síncrona, todos executando a mesma instrução do programa simultaneamente, cada um com seus dados diferentes. Todos os processadores têm uma unidade de entrada e de saída de dados para se comunicarem com o mundo exterior. Este computador paralelo é um exemplo de uma grande classe de computadores cujos processadores compartilham uma memória em comum.

Computadores paralelos são usados principalmente para acelerar o processamento. Um algoritmo paralelo pode ser mais rápido que a melhor solução seqüencial possível. Existe um número crescente de aplicações – por exemplo, em ciências, engenharia, negócios, medicina e outros – que exigem processamento rápido que não podem ser realizadas por qualquer computador atual ou que possa vir a ser desenvolvido. Essas aplicações envolvem o processamento de grandes quantidades de dados, e/ou realizam um grande número de iterações, levando a um tempo de execução impraticável para arquiteturas de computadores seqüenciais.

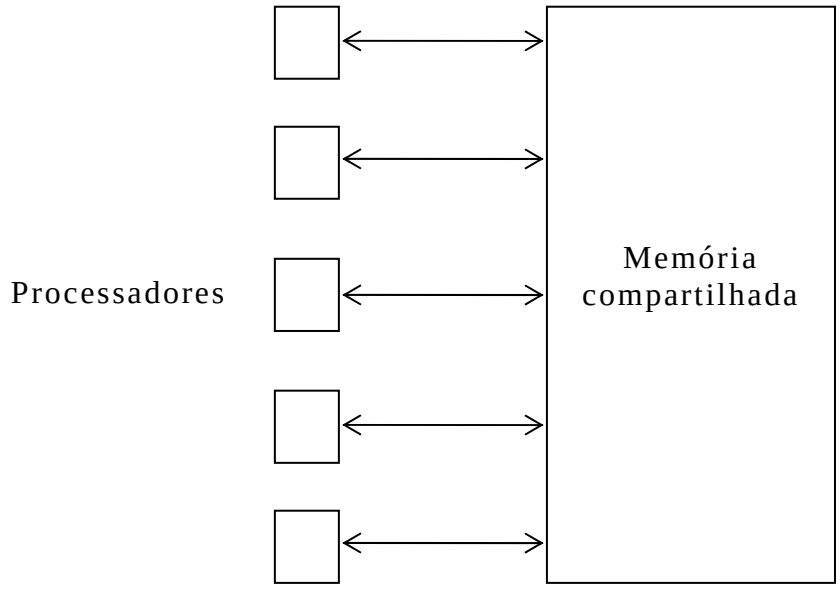


Figura 2.2 – *Um computador paralelo com memória compartilhada.*

A principal motivação para criar e usar computadores paralelos é que o paralelismo é uma das melhores maneiras de vencer o gargalo de velocidade de um único processador [8]. Somado a isso, o custo da taxa de desempenho de um pequeno cluster baseado em computadores paralelos em relação a outras tecnologias como supercomputadores é muito menor e conseqüentemente melhor. Em resumo, desenvolver e produzir sistemas de velocidade razoável usando arquiteturas paralelas é muito mais barato que o desempenho equivalente de um sistema seqüencial.

Computadores paralelos são interessantes porque oferecem o potencial de concentrar recursos computacionais, sejam eles processadores, memória, ou largura de banda de entrada e saída, na solução de problemas computacionais de relevância [49]. Para se ter um desempenho máximo, uma máquina paralela precisa extrair desempenho máximo de seus componentes individuais.

A computação paralela é utilizada em vários tipos de aplicações que necessitam dessas máquinas para serem executadas, dentre elas estão a previsão do tempo, o seqüenciamento de

DNA, simulação de explorações petrolíferas, simulações astrofísicas, entre outras. Essas aplicações críticas exigem alto poder de processamento e alto desempenho para que cumpram seus objetivos.

2.2 Classificação dos Sistemas Paralelos

Existem diversas maneiras de classificar os sistemas paralelos. Uma maneira amplamente utilizada é a classificação proposta por Michael Flynn [50] [51], baseada no fluxo de dados e de instruções. A Figura 2.3 representa essa classificação.

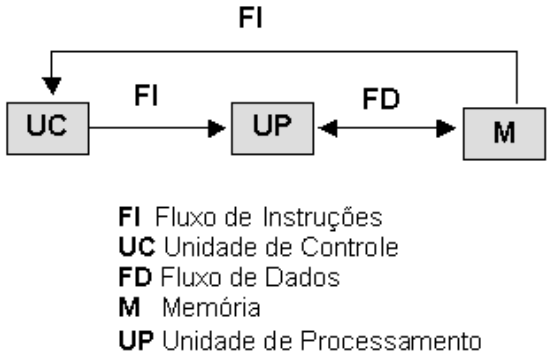
Segundo Flynn, o processo computacional deve ser visto como uma relação entre fluxos de instruções e fluxos de dados. Um fluxo de instruções equivale a uma seqüência de instruções executadas, em um processador, sobre um fluxo de dados aos quais essas instruções estão relacionadas. Baseando-se na possível unicidade e multiplicidade dos fluxos de dados e das instruções, as arquiteturas de computadores são divididas em 4 classes.

A arquitetura SISD (S*ingle Instruction Single Data* – Fluxo único de instruções Fluxo único de dados) corresponde ao modelo tradicional de computação, um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados. É também chamada de computador serial.

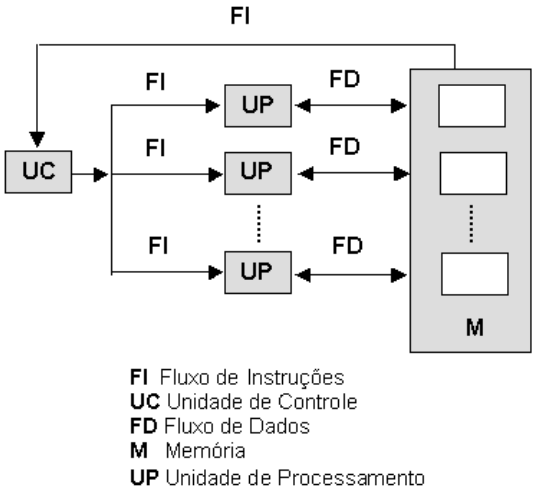
Na arquitetura SIMD (*Single Instruction Multiple Data* – Fluxo único de instruções Fluxo múltiplo de dados) múltiplos processadores escravos sob o controle de uma única unidade de controle mestre, executam simultaneamente a mesma instrução em diversos conjuntos de dados. A arquitetura mostrada apresenta *n* unidades de processamento (UP), sendo que cada uma delas trabalha sobre um dado distinto, que vem de cada um dos *n* módulos de memória (M). O ponto importante é que todas as unidades de processamento trabalham sincronizadas e todas executam a mesma instrução, ou seja, cada instrução é passada, ao mesmo tempo, para as *n* UPs. Assim, os processadores executam a mesma instrução, porém sobre um dado diferente. Um exemplo dessa arquitetura são os processadores vetoriais. Os computadores vetoriais surgiram da necessidade de máquinas com alta capacidade de processamento para lidar com cálculos científicos complexos, como por exemplo, cálculos relacionados à previsão do tempo. Um computador vetorial é caracterizado por poder realizar operações aritméticas sobre vetores ou matrizes de números de pontos flutuantes através de uma simples instrução.

A arquitetura MISD (*Multiple Instruction Single Data* – Fluxo múltiplo de instruções Fluxo único de dados) envolve múltiplos processadores que executam diferentes instruções sobre um único conjunto de dados. Nessa arquitetura, apesar de existir um único fluxo de dados, existem vários dados sendo operados ao mesmo tempo. Na prática, isso quer dizer que várias instruções usam a mesma memória ao mesmo tempo.

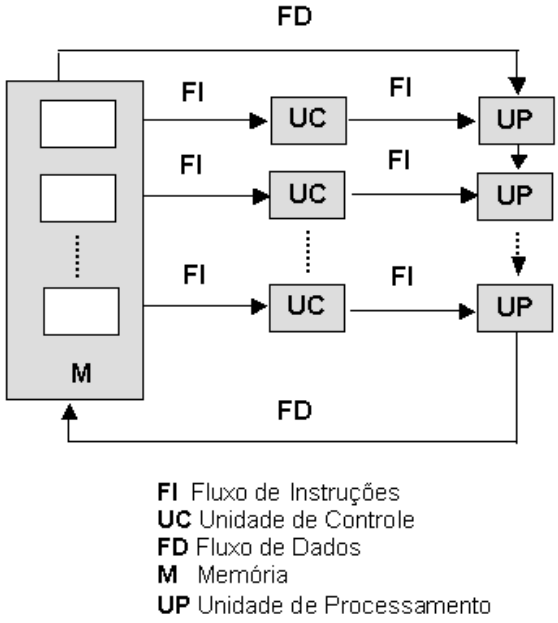
E a arquitetura MIMD (*Multiple Instruction Multiple Data* – Fluxo múltiplo de instruções Fluxo múltiplo de dados) envolve múltiplos processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente. Logo, têm-se vários dados sendo operados por várias instruções, simultaneamente. Essa é a arquitetura mais usada pelos modernos supercomputadores. Nesse caso, é importante que as unidades de processamento possam comunicar-se entre si para fazer a sincronização e trocar informações. Além disso, é possível ter uma memória, chamada de global, onde todos os processadores possam disponibilizar, para os demais, os resultados intermediários.



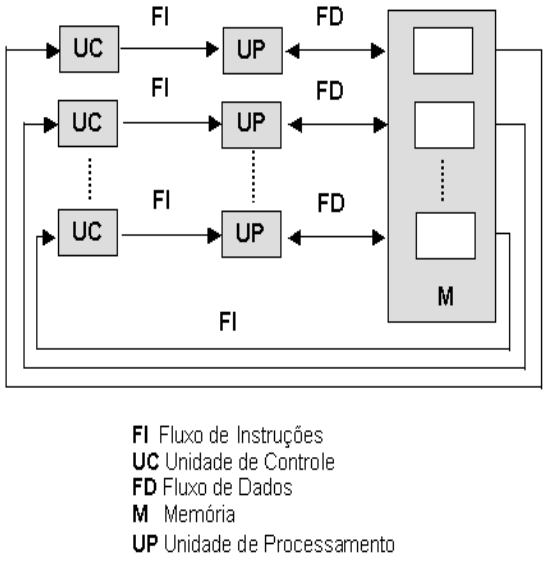
Arquitetura SISD



Arquitetura SIMD



Arquitetura MISD



Arquitetura MIMD

Figura 2.3 – *As quatro arquiteturas da classificação Flynn.*

A classe MIMD pode ainda ser dividida de acordo com o tipo de acesso à memória, se a memória é compartilhada ou se não é compartilhada. Nas máquinas com memória compartilhada, todos os processadores podem acessar toda a memória. Também chamadas de multiprocessadores, existe apenas um espaço de endereçamento que é usado de forma implícita para a comunicação entre os processadores. O acesso à memória é feito através das operações de *load/store* e o acesso aos dados remotos por hardware. Sua vantagem é que o modelo de programação é similar ao caso seqüencial e as principais desvantagens são o alto custo do hardware para acesso aos dados remotos e difícil expansão para um número alto de processadores.

Quando a memória não é compartilhada, existem múltiplos espaços de endereçamento privados, um para cada processador. Essas máquinas são também conhecidas como de memória compartilhada ou multicomputadores, a interligação entre processadores e memória é feita através de uma rede de interconexão. O acesso aos dados remotos é realizado através de troca de mensagens com as operações de *send/receive*. Entre as suas principais vantagens estão a expansão natural de arquiteturas seqüenciais, por exemplo, os clusters de estações de trabalho, e a fácil expansão para um alto número de processadores. Como desvantagens podem ser citadas a programação mais complexa e a granularidade comunicação/processamento é crítica para atingir um bom desempenho.

2.3 O Desempenho dos Sistemas Paralelos

Em qualquer tipo de sistema o desempenho é um fator que deve ser buscado durante todo o seu desenvolvimento. Em sistemas computacionais, a busca pelo desempenho ótimo é crucial para programas que são muito utilizados ou que devem ser executados durante muito tempo a cada ativação. Programas que são executados em sistemas paralelos pertencem a essa categoria, pois, em geral, são programas com carga computacional elevada.

Dada a importância de obter-se o melhor desempenho possível para um determinado sistema, são necessárias formas de defini-lo e medi-lo, fornecendo dados que balizem o trabalho. Tais dados podem vir de uma ferramenta para análise de desempenho ou de um especialista humano na área.

Para que seja possível fazer qualquer análise de desempenho de um programa é preciso que se tenham medidas sobre como será o seu comportamento no sistema para o qual ele está sendo projetado. Mesmo quando se quer apenas uma estimativa do desempenho, algum tipo de medida deve ser utilizada. Conseqüentemente, as ferramentas usadas para análise/estimativa de desempenho devem incluir algum mecanismo de instrumentação que forneça tais medidas.

As medidas de desempenho são geradas através de modelos de desempenho que por sua vez podem ser classificados de várias maneiras, sendo a mais simples delas baseada no tipo de abordagem utilizada [49] [1].

Medidas de desempenho geradas através de modelos analíticos apresentam como vantagem fundamental a simplicidade da realização das medidas, uma vez que na realidade existem apenas estimativas do tempo de execução nos vários trechos de um programa. Essa simplicidade, no entanto, faz com que os resultados tenham sua precisão dependente da qualidade das estimativas. Se estas forem precisas, então o modelo analítico resultante também o será. Mas, se as estimativas não forem precisas ou as relações entre elas não forem adequadas, então o resultado será provavelmente ruim. Apesar de seus problemas de precisão em sistemas muito complexos, esse tipo de medição é muito útil quando ainda não existe código para ser executado experimentalmente. Este é o caso de comportamentos dinâmicos, como a alocação de processadores durante a execução, algoritmos adaptativos, etc.

Outro problema potencial é que a geração destes modelos exige um conhecimento mais profundo da aplicação e da arquitetura. Em resumo, a construção de um modelo analítico deve levar em conta dois pontos: o nível de detalhe a ser modelado, e conseqüentemente sua precisão, e a tratabilidade do modelo.

Na modelagem estatística as funções dos modelos analíticos são substituídas por distribuições que fornecem o comportamento esperado dos diversos parâmetros. Desta forma o comportamento exato não precisa ser conhecido, porém perde-se a precisão. A principal vantagem está na flexibilidade do modelo, mas por outro lado estes modelos fornecem um comportamento assintótico e algumas ferramentas são muito custosas. As Redes de Petri, por exemplo, embora sejam úteis para a verificação da estabilidade e algumas outras propriedades, são inviáveis para a análise do tempo de execução de um sistema complexo.

A simulação, outra abordagem, permite um controle maior dos diversos parâmetros de execução sendo mais fácil a obtenção de informações para a construção e validação de modelos de desempenho. Com essa abordagem é possível a construção de um simulador específico, a emulação de uma arquitetura em outra, a utilização de traços de execução, ou ainda a utilização de informações fornecidas pelo sistema de compilação. Neste último caso combinadas com modelos analíticos. Os maiores problemas com esta abordagem são o alto custo e a necessidade de um conhecimento detalhado do equipamento a ser simulado, com isto a precisão não é sempre satisfatória.

Modelos empíricos se baseiam em medições de trechos de programas que representam as operações mais comuns como, por exemplo, *broadcasting*, sincronização de barreiras, chamada remota de procedimento, criação de canais de comunicação virtual, etc. Desta forma, obtém-se diretamente os parâmetros necessários, evitando a construção de modelos que descrevam os níveis mais básicos da aplicação. O maior problema deste método é a dependência no modo de construção da aplicação, isto é, o tipo da máquina, as bibliotecas e o compilador utilizado. Uma variação desta

abordagem são os modelos dinâmicos, que usam informações retiradas de uma execução do programa para completar o modelo.

Há ainda a combinação das diversas abordagens de modo a construir um modelo híbrido, que tenta minimizar as desvantagens de cada método.

Um algoritmo seqüencial geralmente é avaliado em relação ao seu tempo de execução, expresso como uma função do tamanho da sua entrada de dados. O tempo de execução de um algoritmo paralelo depende, além do tamanho da sua entrada de dados, também da arquitetura do computador paralelo e o número de processadores. Um sistema paralelo é a combinação de um algoritmo e a arquitetura paralela na qual ele foi implementado.

Medir o desempenho de um sistema paralelo é uma tarefa complexa; é preciso considerar além do tempo de execução e a escalabilidade, os mecanismos pelos quais os dados são gerados, armazenados e transmitidos pela rede, movidos de e para o disco e passados pelos diferentes estágios do processamento. Assim, as métricas pelas quais o desempenho é medido podem ser tão diversas quanto o tempo de execução, a eficiência paralela, exigências de memória, taxa de transferência do sistema, taxa de transferência da rede, latência, exigências de hardware, portabilidade, escalabilidade, custos de desenvolvimento e tantos outros.

A importância dessas diversas métricas varia de acordo com a natureza do problema em questão. Uma especificação pode proporcionar números rigorosos para algumas métricas, exigir que outras sejam otimizadas e ainda ignorar outras. Por exemplo, seja o processamento de imagens formado por diversas filtragens concorrentes, cada uma realizando uma transformação diferente em partes da imagem. Uma pode estar interessada não com o tempo total para processar certa quantidade de imagens, mas principalmente com o número de imagens que podem ser processadas por segundo (vazão) ou o tempo que leva uma única imagem para ser processada (latência).

As seções seguintes tratam de dois assuntos importantes no desempenho de sistemas paralelos, as métricas de avaliação de desempenho desses sistemas e a sua escalabilidade.

2.3.1 Métricas de Desempenho de Sistemas Paralelos

Algumas métricas são comumente utilizadas para medir o desempenho de sistemas paralelos. São elas, o tempo de execução, fator de aceleração ou *speedup*, eficiência e o custo.

O **tempo de execução** de um algoritmo paralelo é definido como o tempo necessário pelo algoritmo para resolver um problema computacional. É o tempo compreendido entre o momento em que o processamento paralelo inicia a sua execução até o momento em que o último processador termina a execução [1]. O tempo de execução é medido contando o número de passos consecutivos executado pelo algoritmo, no pior caso, do começo ao fim do processamento [48]. T_s indica o tempo de execução serial e T_p o tempo de execução paralelo.

O número de passos e, portanto, o tempo de execução, de um algoritmo paralelo é uma função do tamanho da entrada de dados e do número de processadores utilizado. Mais ainda, para

um problema de tamanho *n*, o pior caso com relação ao tempo de execução de um algoritmo paralelo é denotado por *T(n)*. Assim, quando o algoritmo tem um tempo de execução de *T(n)*, significa que *T(n)* é o número de unidades de tempo necessárias para a execução do algoritmo.

Segundo Ian Foster [49] o tempo de execução é formado por três componentes, o tempo de processamento, o tempo de comunicação e o tempo ocioso. O tempo de processamento de um algoritmo é o tempo gasto executando o processamento sem contar a comunicação e a ociosidade. O tempo de processamento depende de algumas medidas do tamanho do problema, se o tamanho é representado por um único parâmetro *N* ou por um conjunto de parâmetros N_1, N_2,..., N_m. Se o algoritmo paralelo replica o processamento, então o tempo de processamento depende também do número de tarefas por processador. Em sistemas heterogêneos, o tempo de processamento pode variar de acordo com o processador utilizado para realizar o processamento. O tempo de processamento depende ainda de características do processador e seu sistema de memória. Assim, não é possível assumir que o tempo total de processamento permanecerá constante à medida que o número de processadores muda.

O tempo de comunicação de um algoritmo é o tempo que suas tarefas gastam enviando e recebendo mensagens. Dois tipos de comunicação podem ser distinguidos: a comunicação entre-processadores e a comunicação intra-processadores. Na comunicação entre-processadores as tarefas que se comunicam estão localizadas em processadores diferentes (uma tarefa por processador). Na comunicação intra-processadores as tarefas que se comunicam estão no mesmo processador.

Tanto o tempo de processamento quanto o tempo de comunicação são explicitamente especificados em um algoritmo paralelo; por isso é geralmente simples determinar as suas contribuições para o tempo de execução. O tempo ocioso pode ser mais difícil de determinar. Contudo, muitas vezes ele depende da ordem na qual as operações são realizadas. Um processador pode estar ocioso devido à falta de processamento ou falta de dados. No primeiro caso, o tempo ocioso pode ser evitado utilizando-se técnicas de balanceamento de carga e no segundo caso, o processador está ocioso enquanto o processamento e a comunicação necessários para gerar dados remotos são realizados. Este tempo ocioso algumas vezes podem ser evitados através da estruturação de um programa de maneira que os processadores realizem outros processamentos ou comunicação, enquanto esperam por dados remotos.

A segunda medida de desempenho de um sistema paralelo é o seu **fator de aceleração** ou ***speedup*** [49] [1]. Em um sistema paralelo quando esse é avaliado, há o interesse em saber o quanto se ganhou em desempenho pela paralelização de uma aplicação em relação à sua implementação seqüencial. Assim, o fator de aceleração é uma medida que capta a vantagem relativa de resolver um problema em paralelo e é definida como a razão entre o tempo necessário para solucionar um problema em um único processador e o tempo necessário para resolver o mesmo problema em um computador paralelo com *p* processadores idênticos.

Um determinado problema tem mais que uma solução seqüencial disponível, mas todas elas podem não ser igualmente adequadas para a paralelização. Quando um computador serial é utilizado, é natural usar o algoritmo seqüencial para resolver o problema no tempo mínimo. Dado um algoritmo paralelo, não é possível julgar seu desempenho em relação ao algoritmo seqüencial mais rápido para resolver o mesmo problema em um único processador. Às vezes, o algoritmo seqüencial mais rápido para resolver o problema não é conhecido, ou seu tempo de execução tem uma constante muito grande que torna impraticável a sua implementação. Em tais casos, toma-se o algoritmo mais rápido conhecido que seria uma escolha prática para um computador serial para ser o melhor algoritmo seqüencial. A comparação do desempenho de um algoritmo paralelo para resolver o problema é realizada com o melhor algoritmo seqüencial para resolver o mesmo problema. A definição formal do fator de aceleração ou *speedup*, S, é a razão entre o tempo de execução serial do melhor algoritmo seqüencial para solucionar o problema e o tempo gasto pelo algoritmo paralelo para resolver o mesmo problema em *p* processadores. Os *p* processadores utilizados pelo algoritmo paralelo devem ser idênticos àquele utilizado pelo algoritmo seqüencial. Essa definição, formalizada por Gene Amdahl, ficou conhecida como Lei de Amdahl [49] ou limites de *speedup*.

A argumentação de Amdahl procurava demonstrar que o ganho de velocidade não seria infinito mesmo se existissem infinitos processadores em paralelo. Na realidade, o ganho seria proporcional à razão entre as componentes, paralela e serial do programa, assim $S = \left(T_s + T_p\right) \Big/ \left(T_s + T_p \Big/ p\right)$, onde, *p* é o número de processadores em paralelo, T_s representa a porção serial do programa e T_p sua porção paralela. Por isso, $T_s + T_p = 1$ e pela Lei de Amdahl quando o número de processadores tende ao infinito, o ganho será no máximo de $1 \Big/ T_s$. Essa conclusão é correta se for assumido que o tamanho do problema não varia com o número de processadores. Esse fato faz com que aparentemente todo programa tenha um grau de paralelismo baixo, o que não é verdade.

Teoricamente, o fator de aceleração nunca pode exceder o número de processadores, *p*. Se o melhor algoritmo seqüencial leva T_s unidades de tempo para resolver um dado problema em um único processador, então o fator de aceleração de *p* pode ser obtido em *p* processadores se nenhum dos processadores gasta menos que T_s/p unidades de tempo na resolução do problema. Nesse caso, um único processador poderia imitar os *p* processadores e resolver o problema em menos que T_s unidades de tempo. Isso é uma contradição porque o fator de aceleração, por definição, é calculado em relação ao melhor algoritmo seqüencial. Se T_s é o tempo de execução serial daquele algoritmo,

então o problema não pode ser resolvido em menos que T_s unidades de tempo em um único processador.

Na prática, um fator de aceleração maior que *p* algumas vezes é observado (conhecido como *fator de aceleração super-linear*). Isso geralmente acontece devido a um algoritmo seqüencial não ótimo ou às características do hardware que coloca o algoritmo seqüencial em desvantagem.

Somente um sistema paralelo ideal contendo *p* processadores pode obter um fator de aceleração igual a *p*. Na prática, o comportamento ideal não é atingido porque enquanto executa um algoritmo paralelo, o processador não pode dedicar cem por cento do seu tempo para o processamento do algoritmo. Assim, a **eficiência** [1] [52] é uma medida da fração do tempo em que o processador é utilmente empregado; ela é definida como a razão entre o fator de aceleração e o número de processadores. Em um sistema paralelo ideal, o fator de aceleração é igual a *p* e a eficiência é igual a um. Na prática, o fator de aceleração é menos que *p* e a eficiência está entre zero e um, dependendo do grau de eficácia com que o processador é utilizado. Matematicamente é dada por $E = \frac{S}{p}$.

O **custo** para solucionar um problema em um sistema paralelo é o produto do tempo de execução paralelo e o número de processadores utilizado [1]. O custo reflete a soma de tempo que cada processador gasta resolvendo o problema. A eficiência também pode ser expressa como a razão entre o tempo de execução do algoritmo seqüencial mais rápido conhecido para resolver o problema e o custo de resolver o mesmo problema em *p* processadores.

O custo de resolver o problema em um único processador é o tempo de execução do algoritmo seqüencial mais rápido conhecido. Um sistema paralelo é dito ser de custo ótimo se o custo de resolver um problema em um computador paralelo é assintoticamente equivalente ao tempo de execução do algoritmo seqüencial mais rápido conhecido em um único processador.

2.3.2 Escalabilidade de Sistemas Paralelos

O número de processadores é o limite superior do fator de aceleração que pode ser alcançado por um sistema paralelo. O fator de aceleração é um para um único processador, mas se mais processadores são utilizados, o fator de aceleração é habitualmente menor que o número de processadores. Para um determinado problema, o fator de aceleração não aumenta linearmente conforme o número de processadores aumenta. O fator de aceleração tende a tornar-se saturado e a sua curva fica nivelada. Isso é uma conseqüência da Lei de Amdahl, a eficiência cai com o aumento do número de processadores. Outra conseqüência é que uma grande instância do mesmo problema rende um fator de aceleração e eficiência maiores para o mesmo número de processadores, apesar de ambos continuarem a cair com o aumento de *p*.

A capacidade de manter a eficiência em um valor fixo através do aumento simultâneo do número de processadores e o tamanho do problema é o que acontece em muitos sistemas paralelos e por isso eles são chamados de sistemas paralelos escaláveis. A escalabilidade de um sistema paralelo é uma medida da sua capacidade em aumentar o fator de aceleração em proporção ao número de processadores [1]. Isso reflete a habilidade do sistema paralelo em utilizar de maneira eficaz o aumento dos recursos de processamento.

2.4 Fatores que Degradam o Desempenho dos Sistemas Paralelos

A função *overhead* [1] [49] [52] de um sistema paralelo é definida como a diferença entre o custo e o tempo de execução serial do algoritmo mais rápido conhecido para resolver o mesmo problema. A função *overhead* encapsula todas as causas da ineficiência do sistema paralelo, seja devido ao algoritmo, a arquitetura, ou a interação algoritmo-arquitetura. As maiores fontes de *overhead* em um sistema paralelo são a comunicação entre processadores, a carga desbalanceada e o processamento extra.

Qualquer sistema paralelo não trivial exige comunicação entre processadores. O tempo para transferir dados entre os processadores é geralmente a fonte mais significante de *overhead* do processamento paralelo. Se cada um dos p processadores gasta t_{comm} unidades de tempo realizando a comunicação, então a comunicação entre processadores contribui $t_{comm} * p$ para a função *overhead*.

Em muitas aplicações paralelas, tais como pesquisa e otimização, por exemplo, é difícil saber o tamanho das tarefas atribuídas aos vários processadores. Por isso, o problema não pode ser dividido estaticamente entre os processadores a fim de manter a carga de trabalho uniforme. Se diferentes processadores têm cargas de trabalho diferentes, alguns processadores podem estar ociosos durante parte do tempo em que outros estão trabalhando na solução do problema.

Com freqüência alguns ou todos os processadores devem sincronizar-se em certos pontos durante a execução do problema paralelo. Se todos os processadores não estão prontos ao mesmo tempo, então alguns deles estarão ociosos. Qualquer que seja a causa da ociosidade, o tempo total ocioso de todos os processadores contribui para a função *overhead*.

Um caso especial de *overhead* devido à ociosidade do processador é a presença de um componente seqüencial em um algoritmo paralelo. Parte de um algoritmo pode ser não paralelizável, permitindo que apenas um único processador trabalhe. O tamanho do problema para tal algoritmo é expresso como a soma do trabalho devido ao componente seqüencial, W_s, mais o trabalho devido ao componente paralelizável, W_p.

Enquanto um processador está trabalhando em W_s, os demais processadores, $(p-1)$, estão ociosos. Como resultado, um componente serial de W_s contribui $(p-1)*W_s$ para a função *overhead* de um sistema paralelo com *p* processadores.

O algoritmo seqüencial mais rápido conhecido para um problema pode ser difícil e em alguns casos impossível de ser paralelizável, forçando o uso de um algoritmo baseado em um algoritmo seqüencial inferior, mas facilmente paralelizável, ou seja, com um grau maior de concorrência. Seja *W* o tempo de execução do algoritmo seqüencial mais rápido conhecido para o problema e *W'* o tempo de execução de um algoritmo inferior, porém paralelizável, para o mesmo problema. Então a diferença $W' - W$ deve ser considerada como parte da função overhead porque ela expressa a quantidade de trabalho extra realizado para resolver o problema em paralelo.

Um algoritmo paralelo baseado no melhor algoritmo serial pode ainda agregar mais processamento que a sua versão seqüencial. Um exemplo dessa situação é o caso de um algoritmo que em sua versão seqüencial reutiliza os resultados de alguns processamentos. No entanto, na versão paralela esses resultados não podem ser reutilizados porque eles são gerados por processadores diferentes. Assim, alguns processamentos são realizados múltiplas vezes em diferentes processadores, contribuindo para a função *overhead*.

3. O PROCESSAMENTO PARALELO EM CLUSTERS E GRADES

Um dos principais fatores que explicam a necessidade de processamento paralelo é a busca por maior desempenho. As diversas áreas nas quais a computação se aplica, sejam científicas, industriais ou militares, requerem cada vez mais poder computacional, em virtude dos algoritmos complexos que são utilizados e do tamanho do conjunto de dados a ser processado.

A evolução do hardware e a diminuição dos seus custos, os avanços nas tecnologias de comunicação nas redes locais e nas redes de longa distância, permitiram o surgimento dos conceitos de clusters e grades para o processamento paralelo. Este capítulo apresenta conceitos dos ambientes de cluster e grades, bem como os elementos necessários à sua implementação.

3.1 O Processamento Paralelo em Clusters

O cluster é um tipo de sistema de processamento paralelo ou distribuído que consiste de um conjunto de computadores interconectados trabalhando em conjunto para executar aplicações e integrando recursos computacionais [8]. Seu objetivo é fazer com que todo o processamento da aplicação seja distribuído aos computadores, de forma que pareça um único computador. Com isso, é possível realizar processamentos que até então somente computadores de alto desempenho seriam capazes de fazer. As características fundamentais para a construção dessa plataforma são o aumento da confiabilidade, distribuição de carga e desempenho.

Cada computador do cluster, também chamado de nó ou nodo, pode ser um sistema multiprocessado ou não com memória, facilidades de I/O e um sistema operacional. Segundo [8] a arquitetura típica de um cluster é mostrada na Figura 3.1.

O hardware de rede atua como um processador de comunicação e é responsável por transmitir e receber pacotes de dados entre os nós do cluster através da rede/switch. O software de comunicação oferece um meio de comunicação rápido e confiável entre os nós do cluster e o ambiente externo. Os nós do cluster podem trabalhar coletivamente, como um recurso computacional integrado, ou podem agir como computadores individuais. O *middleware* do cluster é responsável por oferecer uma ilusão de um sistema unificado e a disponibilidade de uma coleção de computadores independentes, mas interligados. O ambiente de programação oferece ferramentas portáteis, eficientes e de fácil utilização para o desenvolvimento de aplicações, incluindo as bibliotecas de passagem de mensagens, depuradores e análise de desempenho das aplicações. Os clusters podem ser usados para a execução de aplicações seqüenciais ou paralelas.

O uso de clusters para executar aplicações paralelas é uma alternativa cada vez mais popular do que o uso de plataformas de computação paralela que são extremamente caras. Um fator importante que tem proporcionado essa popularização dos clusters é a padronização de muitas ferramentas utilizadas por aplicações paralelas. Um exemplo dessa padronização é a biblioteca de passagem de mensagens chamada MPI [46] [53]. Nesse contexto, a padronização permite que aplicações sejam desenvolvidas, testadas e mesmo executadas em diversas plataformas computacionais sendo necessárias pequenas adaptações para executá-la em plataformas paralelas dedicadas onde o tempo de CPU é cobrado e contabilizado.

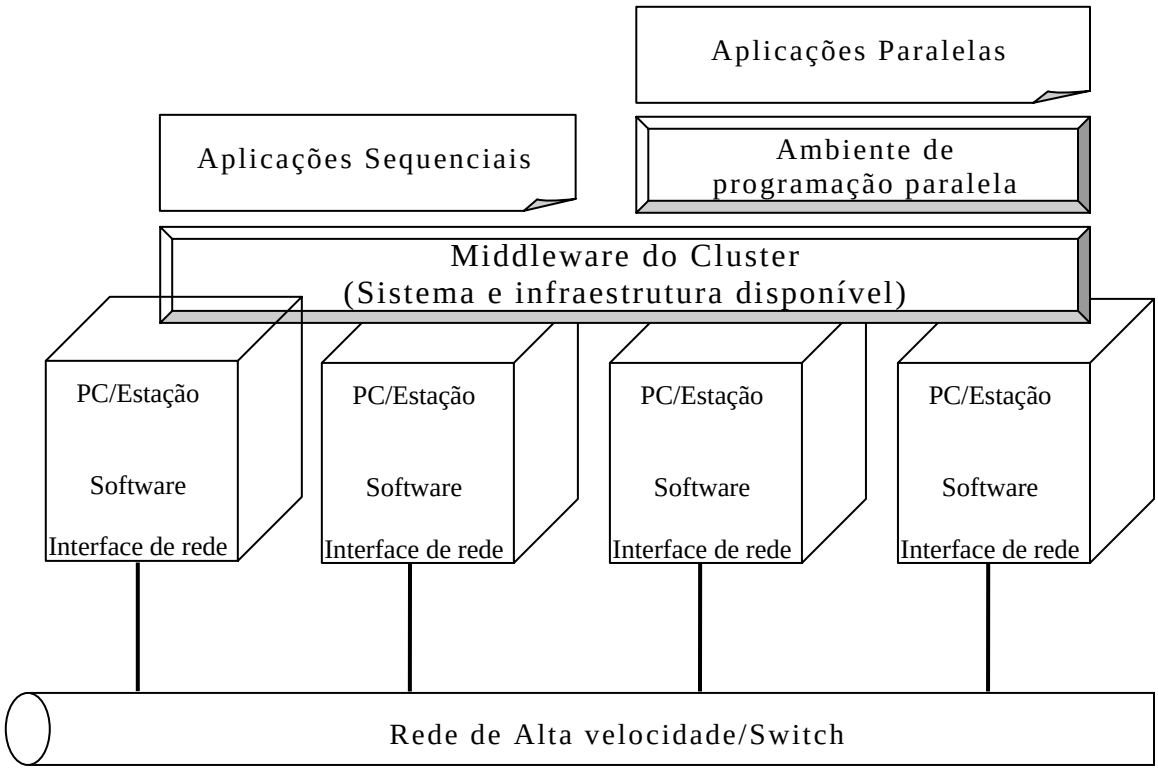


Figura 3.1 – *Arquitetura de um cluster de computadores.*

Os clusters oferecem funcionalidades como alto desempenho, expansão e escalabilidade, alta taxa de transferência e alta disponibilidade a um custo relativamente baixo.

Independente do sistema operacional usado é preciso utilizar um software que permita a instalação e configuração do cluster. Esse software é responsável, entre outras coisas, pela distribuição do processamento, sendo este um ponto crucial na montagem do cluster. É preciso que o software trabalhe de forma que erros e defeitos sejam detectados, oferecendo meios de providenciar reparos, mas sem interromper as atividades do cluster. Esse tipo de necessidade pode ser controlado através de equipamento específico, não dependendo apenas do software.

Os clusters podem ser classificados [8] como segue.

3.1.1 Cluster de Alta Disponibilidade (High Availability – HA)

Estes modelos de clusters são construídos para prover disponibilidade de serviços e recursos de forma ininterrupta através do uso da redundância implícita ao sistema. A idéia geral é que se um nó do cluster vier a falhar as aplicações ou serviços possam estar disponíveis em outro nó. Estes tipos de cluster são utilizados para bases de dados de missões críticas, correio, servidores de arquivos e aplicações, e costumam ter meios eficientes de proteção e detecção de falhas.

3.1.2 Cluster de Balanceamento de Carga (Load Balancing – LB)

Este modelo de cluster distribui de maneira equilibrada o tráfego entrante ou requisições de recursos e o processamento aos nós do cluster. Neste tipo de cluster, é necessário que haja monitoração constante da comunicação e mecanismos de redundância; caso contrário, qualquer falha pode interromper o funcionamento do cluster. Se um nó falhar, as requisições são redistribuídas entre os nós disponíveis no momento. É bastante usado na Internet, em servidores de e-mail e comércio eletrônico.

Um exemplo deste tipo de cluster é o MOSIX (*Multicomputer Operating System for UnIX*) [54]. Trata-se de um conjunto de ferramentas de cluster para o sistema operacional Linux (ou sistemas baseados em Unix). Uma de suas principais características é não necessitar de aplicações e recursos de software voltados ao cluster, como acontece com o Beowulf [55]. O MOSIX é eficiente na distribuição dinâmica de processamento entre os computadores do cluster, sendo amplamente utilizado por universidades em pesquisas e projetos. Por ser baseada em Linux, sua implementação é transparente, ou seja, seu código é aberto, além de fácil instalação. De maneira generalizada, O MOSIX é uma extensão para Linux de um sistema de cluster que trabalha como se fosse um único supercomputador, por meio de conceitos de distribuição de processos e balanceamento de carga.

Outro exemplo é o OpenMosix [42], uma extensão do projeto MOSIX.

3.1.3 Cluster Combo ou Combinação HA e LB

Combina as características dos clusters de alta disponibilidade e de balanceamento de carga, aumentando assim a disponibilidade e escalabilidade de serviços e recursos. Esse tipo de configuração de cluster é bastante utilizado em servidores de *web*, *e-mail*, *news* ou *ftp*.

3.1.4 Cluster de Processamento Distribuído ou Processamento Paralelo

Este modelo de cluster aumenta a disponibilidade e o desempenho para as aplicações, particularmente as grandes tarefas computacionais. Uma grande tarefa computacional pode ser dividida em pequenas tarefas que são distribuídas às estações como se fossem um supercomputador massivamente paralelo. É comum associar este tipo de cluster ao projeto Beowulf [55].

O Beowulf foi um cluster voltado à computação paralela, implantado em 1994 pela NASA. Era formado por 16 computadores pessoais 486 DX-100 conectados através de uma rede Ethernet, com a finalidade de processar as informações espaciais que a entidade recolhia. Seu objetivo era

suprir a crescente e elevada capacidade de processamento em diversas áreas cientificas com o intuito de construírem sistemas computacionais poderosos e economicamente viáveis. Para manter a independência do sistema e baixar os custos, os desenvolvedores optaram por utilizar o Linux. A evolução constante no desempenho dos processadores colaborou com a aproximação entre computadores pessoais e *workstations*; a diminuição dos custos das tecnologias de rede e dos próprios processadores e o sistema operacional aberto e gratuito, como o GNU/Linux em muito influenciam as pesquisas para melhoria deste método de processamento de alto desempenho em clusters.

O sistema é dividido em um nó denominado *front-end* (mestre), cuja função é controlar o cluster, monitorando e distribuindo as tarefas, atua como servidor de arquivos e executa o elo entre os usuários e o cluster. Grandes sistemas em cluster podem distribuir diversos servidores de arquivos, nó de gerenciamento da rede para não sobrecarregar o sistema. Os demais nós são conhecidos como clientes ou *back-end* (escravos) e são exclusivamente dedicados para processamento das tarefas enviadas pelo nó controlador. A Figura 3.2 ilustra este tipo de cluster.

Uma característica chave de um cluster Beowulf é o software utilizado, que possui desempenho elevado e é gratuito na maioria das suas ferramentas; como exemplo pode-se citar os sistemas operacionais GNU/Linux e FreeBSD sobre os quais estão instaladas as diversas ferramentas que viabilizam o processamento paralelo, como é o caso das API's (*Application Programming Interface*), MPI (*Message Passing Interface*) e PVM (*Parallel Virtual Machine*). Este fato permitiu que alterações no sistema operacional Linux fossem realizadas a fim de dotá-lo de novas características que facilitaram a implementação de aplicações paralelas.

O que distingue o Beowulf de outros tipos de cluster são as seguintes características, aplicadas de acordo com a finalidade do cluster:

- Podem ser usados computadores comuns, inclusive modelos considerados obsoletos, e a conexão dos nós pode ser feita por redes do tipo Ethernet (mais comum), não sendo necessário utilizar equipamentos próprios para clusters;
- Existe um servidor responsável por controlar todo o cluster, principalmente quanto à distribuição de tarefas e processamento (pode haver mais de um servidor, dedicado a tarefas específicas, como monitoração de falhas). Este servidor é chamado de Front-end;
- O sistema operacional é baseado em Linux, sendo necessário que ele contenha todos os programas para cluster.

De maneira geral, o cluster Beowulf permite a construção de sistemas de processamento que podem alcançar altos valores de *gigaflops*, com o uso de computadores comuns e de um sistema operacional com código-fonte livre, ou seja, além de gratuito, pode ser melhorado para a sua

finalidade. Tais características fizeram do Cluster Beowulf um tema muito explorado em universidades e conseqüentemente aplicado para vários fins.

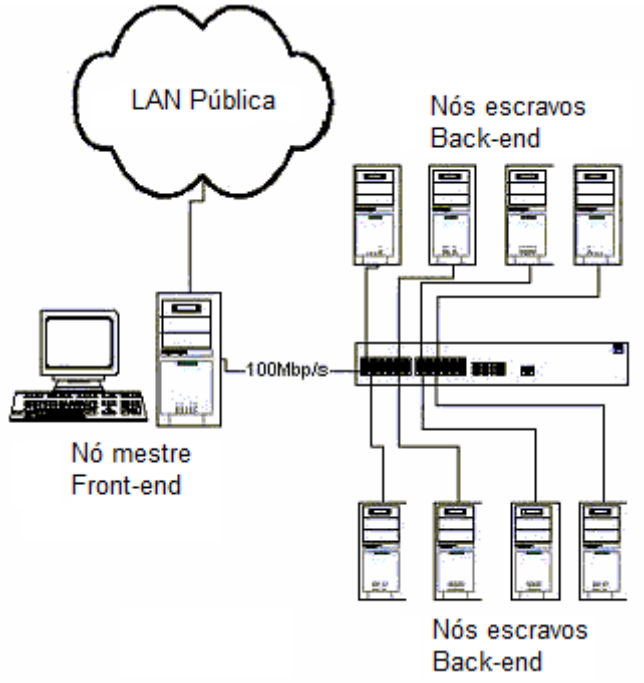


Figura 3.2 – *O cluster Beowulf.*

Entre os requisitos para o sistema operacional de um cluster Beowulf estão a necessidade de ter as bibliotecas para PVM ou para MPI. Ambos os tipos são usados para a troca de mensagens entre os nós do cluster, sendo que MPI é mais avançada que a PVM, pois pode trabalhar com mensagens para todos os computadores (broadcast) ou para apenas um determinado grupo (*multicast*), por exemplo, quando somente esse grupo vai realizar determinada tarefa. Em compensação, PVM suporta ambientes heterogêneos, algo que foi considerado fonte de sobrecarga no projeto da MPI. MPI pode ser dita mais avançada por ter sido desenvolvida com base na PVM e outras interfaces proprietárias. Porém tem suposições de projeto diferentes da PVM.

Empresas especializadas, centros de pesquisas e universidades costumam estudar este assunto a fundo. Como conseqüência, existe clusters com até milhares de nós. No Brasil, um exemplo é o cluster desenvolvido no ano de 2003 por um aluno da Universidade Estadual Paulista (UNESP), no estado de São Paulo [56]. Baseado no tipo Beowulf, esse cluster ficou bastante conhecido, por ajudar na pesquisa de medicamentos para o tratamento da tuberculose. O valor gasto neste projeto foi 60 mil reais; se tivesse sido utilizado um supercomputador de capacidade equivalente, os gastos seriam até 17 vezes maior.

O Beowulf é um projeto bem sucedido. A opção feita por seus criadores de usar hardware popular e software aberto tornou-o fácil de replicar e modificar. A prova disso é a grande

quantidade de sistemas construídos à moda Beowulf em diversas universidades, empresas e até residências. Mais do que um experimento foi obtido um sistema de uso prático que continua sendo aperfeiçoado constantemente.

3.2 O Padrão MPI

Passagem de Mensagem é um paradigma de programação [46] amplamente usado em computadores paralelos, especialmente nos Computadores Paralelos Escalares (*Scalable Parallel Computers* – SPC) com memória distribuída e nas Redes de Estações de Trabalho *(Networks of Workstations* – NOW).

MPI [46] [53] [57] [58], a Interface de Passagem de Mensagem, é um sistema padronizado portável de passagem de mensagem projetado por um grupo de pesquisadores da academia e da indústria para funcionar em computadores paralelos. O padrão define a sintaxe e a semântica de um conjunto de rotinas úteis para uma grande quantidade de usuários que desenvolvem programas portáteis de passagem de mensagem nas linguagens de programação Fortran, C ou C++. A primeira versão do padrão, chamado MPI-1 [46] [59] foi liberada em 1994 e desde então a especificação MPI tornou-se o padrão de passagem de mensagem para computadores paralelos. A segunda versão do padrão, a MPI-2 [53] [59], foi disponibilizada em 1997.

Existem várias implementações de MPI em uma ampla variedade de plataformas. Cada vendedor de computadores paralelos de alto desempenho oferece uma implementação MPI como parte do sistema de software padrão, além de existir um número de implementações gratuitas disponíveis para redes de estações heterogêneas, tanto para Unix quanto para Windows.

Os atrativos do paradigma de passagem de mensagem pelo menos parcialmente impedem o avanço de sua ampla portabilidade. Programas escritos dessa forma podem ser executados em multicomputadores de memória distribuída, multiprocessadores de memória compartilhada, redes de estações e combinações de todos eles. O paradigma não se tornará obsoleto por combinar arquiteturas considerando memória compartilhada e memória distribuída, ou pelo aumento na velocidade da rede. MPI é implementada em uma grande variedade de máquinas, incluindo aquelas máquinas que consistem de coleções de outras máquinas, paralelas ou não, conectadas por uma rede de comunicação de dados.

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e o recebimento de mensagens entre os processos. A Figura 3.3 ilustra a estrutura básica de um programa MPI, escrito em linguagem de programação C. A implementação mais utilizada nas aplicações MPI utiliza o modelo SPMD (*Single Program Multiple Data – Único Programa Múltiplos Dados*), ou seja, todos os processadores executam o mesmo programa, trabalhando com diferentes dados. O processo mestre é responsável por

distribuir as tarefas, os escravos executam as tarefas e enviam ao mestre os resultados parciais. O que define qual o código que será executado é justamente o *rank* do processo (o processo mestre recebe o *rank* zero), conforme pode ser observado na figura.

```
#include "mpi.h"

main (int argc, char *argv[])
{
    ...

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &meurank);

    if (meurank == 0)
    {
       // Processo Mestre
       ...
    }
    else
    {
       // Processo Escravo
       ...
    }
    MPI_Finalize ();
}
```

Figura 3.3 – *Estrutura básica de um programa MPI.*

Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. O mecanismo de comunicação do padrão MPI é a transmissão de dados entre um par de processos, dos quais um tem o papel de enviar a mensagem e o outro de recebê-la. Esse tipo de comunicação é chamado de comunicação ponto-a-ponto [59]. O MPI disponibiliza um conjunto de funções de envio e recebimento que permite a comunicação de dados tipados com uma *tag* associada. As *tags* são usadas para distinguir os diferentes tipos de mensagens que um processo pode enviar ou receber. Essa tipagem é necessária para o suporte à heterogeneidade – o tipo da informação é devido à forma que conversões corretas das representações dos dados possam ser realizadas de acordo como o dado foi enviado de uma arquitetura para outra. O balanceamento de carga pode ser feito de maneira estática, antes do início da execução da aplicação, ou dinâmica, onde as decisões do balanceamento são tomadas durante a execução da aplicação. O balanceamento de carga estático é o mais usual.

3.3 O Processamento Paralelo em Grades

A grade computacional [9] [60] é uma infraestrutura de gerenciamento de dados que fornece recursos eletrônicos para uma sociedade global em negócios, administração, pesquisa, ciência e entretenimento sem levar em conta a localização geográfica. As grades integram redes, comunicação, processamento e informação com o objetivo de prover uma plataforma virtual para processamento e gerenciamento de dados da mesma forma que a Internet integra recursos para formar uma plataforma virtual para obter informações. As grades de larga escala são intrinsecamente distribuídas, heterogêneos e dinâmicos; eles têm transformado a ciência, as empresas, a saúde e a sociedade.

A essência da grade computacional pode ser resumida como Recursos sob-Demanda (*RoD – Resources on Demand*), ou seja, o fornecimento transparente de recursos da grade necessários às aplicações ou serviços que necessitam de pouco ou nenhum atraso [61]. A má qualidade de serviços na rede pode prejudicar significativamente o fornecimento eficiente de RoD.

Existem várias plataformas de computação em grade, tais como o Globus [11], Condor [43], OurGrid [40] [41], dentre outras. A seguir é apresentada uma breve descrição de cada uma dessas plataformas.

3.3.1 Globus

O Globus Toolkit [11] é um conjunto de serviços que facilita a computação em grade permitindo a submissão e o controle de aplicações, descoberta de recursos, movimentação de dados e segurança no ambiente da grade. Tendo sido a solução de maior impacto na comunidade da computação de alto desempenho, o Globus e os protocolos definidos em sua arquitetura tornaram-se um padrão de fato como infraestrutura para computação em grade. A Figura 3.4 ilustra a arquitetura do Globus.

A utilização dos serviços do Globus pressupõe a instalação e configuração de uma considerável infra-estrutura de suporte. Cada recurso é gerenciado por uma instância do *Globus Resource Allocation Manager* (GRAM) [63], o responsável por instanciar, monitorar e reportar o estado das tarefas alocadas para o recurso. A partir da autenticação única do usuário na grade, o GRAM verifica se o usuário pode utilizar o recurso solicitado. Caso o usuário tenha o acesso permitido, é criado um gerenciador de trabalhos, que é responsável por iniciar e monitorar a tarefa submetida. As informações sobre o estado da tarefa e do recurso são constantemente reportadas ao serviço de informação e diretório do Globus, o *Metacomputing Directory Service* (MDS).

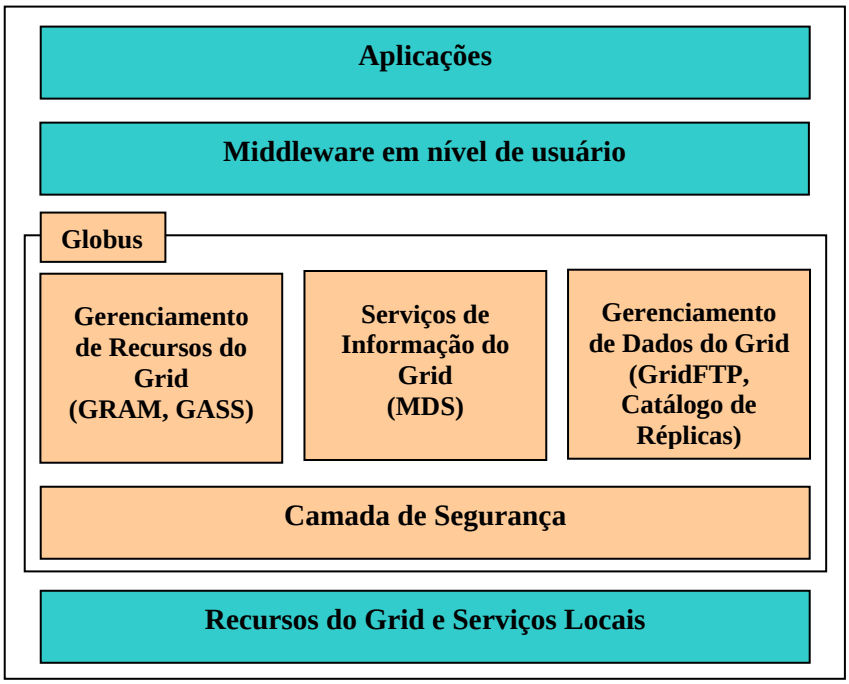


Figura 3.4 – *A arquitetura Globus [62].*

Um aspecto importante para a grande aceitação do Globus é que os serviços oferecidos são razoavelmente independentes, possibilitando que se utilize apenas parte dos serviços Globus em uma dada solução. Essa possibilidade do uso parcial de Globus ajuda na adaptação de aplicações paralelas existentes para a grade. Pode-se começar usando serviços mais básicos e ir, aos poucos, incorporando funcionalidades mais avançadas. O fato de o Globus ser estruturado como um conjunto de serviços independentes deixa claro que ele não é uma solução pronta e completa (*plug-and-play*) para a construção de grades. O Globus fornece serviços úteis para computação em grade, mas os desenvolvedores, administradores e usuários precisam despender certo esforço para finalizar sua grade. Por exemplo, administradores precisam decidir quais usuários terão acesso a quais recursos que compõem a grade e em quais condições esses acessos se darão; freqüentemente é necessário desenvolver escalonadores de aplicações que tenham conhecimento sobre as aplicações que serão executadas e algumas vezes também sobre a estrutura da grade a ser usada. Por ser complexa, a computação em grade não possibilita soluções *plug-and-play*, logo, o fato do Globus não ser uma solução pronta e completa não é nenhum demérito.

Apesar de resolver satisfatoriamente diversas questões do aspecto operacional, o Globus não ataca o problema da grade do ponto de vista gerencial. Utilizando o mecanismo de acesso hoje disponível, são necessárias negociações com os donos de recursos para obter o acesso a esses e há necessidade do mapeamento dos clientes para usuários locais, fatos que eliminam a escalabilidade desse mecanismo.

3.3.2 Condor

O Condor [43], surgido em 1984, é um *middleware* que aproveita os ciclos ociosos de um conjunto de estações de trabalho em uma rede institucional. Esse conjunto é chamado de Condor pool. Com a utilização desses pools em diversas instituições, surgiram diversos mecanismos para o compartilhamento de recursos disponíveis em diferentes instituições e domínios, antes mesmo do surgimento do conceito de computação em grade [64]. Tendo sido um esforço pioneiro e se adaptado à evolução das perspectivas da computação de alto desempenho para a computação em grade, o projeto Condor tem, em sua história, uma série de lições sobre os problemas gerenciais e operacionais da criação e utilização de grades computacionais.

Condor é um sistema que possui um escopo bem definido e menos genérico que outras soluções para computação de alto desempenho em grades computacionais. Condor objetiva fornecer grande quantidade de poder computacional a médio e longo prazo (dias a semanas) utilizando recursos ociosos na rede [43]. O Condor objetiva a alta vazão (*high throughput*) e não o alto desempenho (*high performance*) [39] [43] [65] [66]. Portanto, o Condor visa fornecer desempenho sustentável a médio e longo prazo, mesmo que o desempenho instantâneo do sistema possa variar consideravelmente.

O funcionamento do Condor pool é ilustrado na Figura 3.5. As aplicações são submetidas através de agentes, responsáveis por armazenar as tarefas até encontrar os recursos adequados para executá-las. Os agentes e recursos reportam seus estados a um *matchmaker*, que é responsável por apresentar recursos adequados às necessidades reportadas para os agentes. Os agentes então precisam descobrir se os recursos ainda são adequados para a execução da tarefa (por exemplo, se ainda estão ociosos) e então alocá-los às tarefas. Para executar a tarefa, os dois lados iniciam um novo processo. Do lado do agente, um *shadow* fornece os detalhes necessários para a correta execução da tarefa, e do lado do recurso um *sandbox* garante a segurança necessária para que a execução da tarefa não seja maléfica ao recurso. Num nível de abstração maior, é possível ver na Figura 3.6 como o agente, os recursos e o *matchmaker* interagem.

Cada um dos componentes do sistema, agentes, recursos e *matchmaker*, é independente e responsável por implementar políticas de compartilhamento definidas por seus donos. Os agentes implementam a definição de quais recursos são confiáveis e adequados para executar as aplicações desejadas. Os recursos definem quais usuários são confiáveis e devem ter acesso ao recurso. O *matchmaker* implementa as políticas da comunidade como o controle de admissão. Ele pode admitir ou rejeitar participantes baseado em seus nomes ou endereços e pode ainda definir limites globais como, por exemplo, a fração do pool que pode ser alocada para um único agente. Cada participante é autônomo, mas a comunidade como uma entidade é definida pela escola comum de um *matchmaker*.

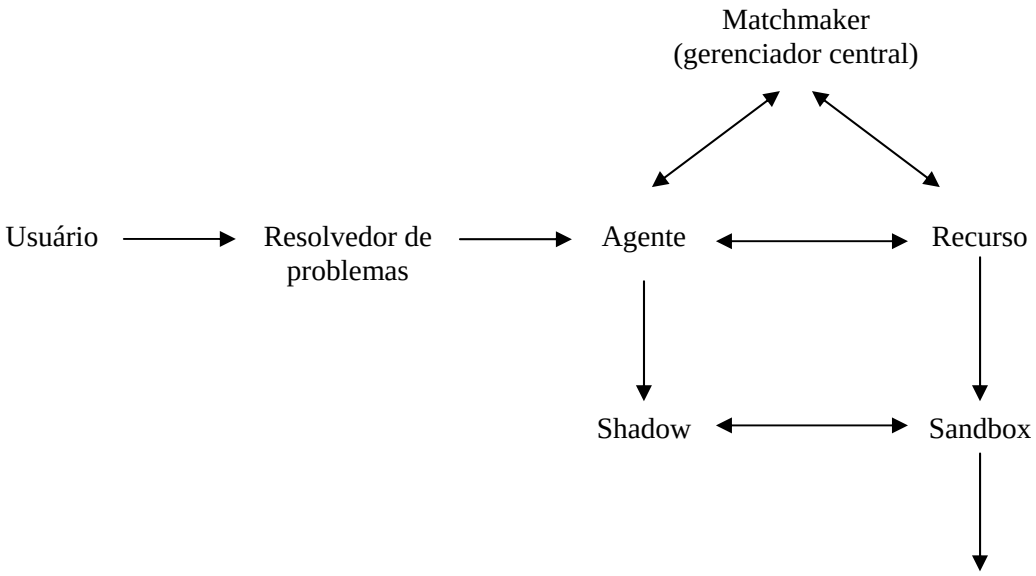


Figura 3.5 – *Funcionamento do Condor-pool [64].*

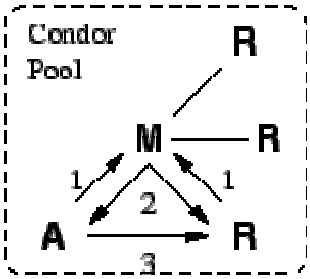


Figura 3.6 – *Interações entre os componentes do Condor-pool [64].*

Com o surgimento do conceito de grade e, em especial do projeto Globus, o mecanismo de compartilhamento do Condor chegou ao seu estágio atual. Um agente Condor foi adaptado para se comunicar com o Globus, e foi chamado de Condor-G.

O Condor-G adota uma visão heterogênea de grade. Além do *Condor Pools*, Condor-G também utiliza recursos via Globus. O *Condor-G Scheduler* controla a execução da aplicação, submetendo tarefas tanto a *Condor Pools* quanto a recursos acessíveis via Globus.

Ao se investigar os mecanismos de acesso aos recursos da grade, uma atenção especial é necessária para o projeto Condor. Sua evolução e os diversos mecanismos de acesso utilizados provêem um conhecimento fundamental sobre a utilização prática de grades computacionais. Dentre os fatos interessantes, pode-se destacar a inadequação do modelo de negociação estática do

gateway flocking (mecanismo para compartilhar recursos) a situações reais; a necessidade da interoperabilidade para a viabilização de utilização de recursos em larga escala, fornecida com o Condor-G; e o mecanismo de automatização do aspecto gerencial da preparação de um condor pool no *gliding in* (técnica de comunicação entre o Globus e o Condor).

3.3.3 OurGrid

OurGrid [40] [41] é uma grade aberta e cooperativa onde sites doam seus recursos computacionais ociosos em troca de recursos ociosos de outros sites quando necessário. Esses sites têm o interesse em trocar “favores” computacionais entre si, então o OurGrid usa uma tecnologia ponto-a-ponto que faz com que cada site interessado em colaborar com o sistema doe seus recursos ociosos. Portanto, existe uma rede ponto-a-ponto de troca de “favores” permitindo que os recursos ociosos de um site sejam disponibilizados para outro quando solicitado.

O OurGrid tem como objetivo fornecer uma solução efetiva para a execução de aplicações *Bag-of-Tasks* (BoT) [67] [68] em grades computacionais. Estas aplicações paralelas são compostas por um conjunto de tarefas independentes que não necessitam de comunicação durante sua execução

A solução OurGrid, ilustrada na Figura 3.7, é composta por três componentes principais: MyGrid ou OurGrid Broker, Peer e UserAgent ou OurGrid Worker.

O componente MyGrid é composto por dois módulos: o escalonador e o executor de réplicas. O escalonador é responsável por receber novos jobs de um usuário e interpretá-los. O escalonador cria réplicas das tarefas, se for necessário, e então contacta o Peer solicitando máquinas para executá-las. O executor de réplicas tem a responsabilidade de gerenciar a execução das réplicas. Quando a execução termina, o executor de réplicas envia os resultados ao escalonador. A arquitetura do MyGrid é apresentada na Figura 3.8.

Atualmente o MyGrid possui duas estratégias de escalonamento: *Workqueue with Replication* (WQR) [69] e *Storage Affinity* [70]. O WQR foi definido para aplicações *cpu-intensive*, enquanto o *Storage Affinity* para melhorar o desempenho de aplicações que processam grandes quantidades de dados.

Um job MyGrid é composto por tarefas independentes que são formadas por três partes: *init*, *remote* e *final*. Estas partes são executadas seqüencialmente sendo que *init* e *final* efetuam transferências de arquivos da tarefa e são executadas na máquina MyGrid. A fase *remote* é executada em uma máquina da grade e realiza o processamento da tarefa propriamente dita.

O Peer OurGrid é executado em uma máquina chamada máquina *peer*. Seu papel principal é organizar e fornecer máquinas para a execução das tarefas da aplicação bem como determinar como e quais máquinas podem ser usadas. Durante a execução do job, o MyGrid solicita ao *peer* associado a ele máquinas para executar as tarefas pertencentes àquele job.

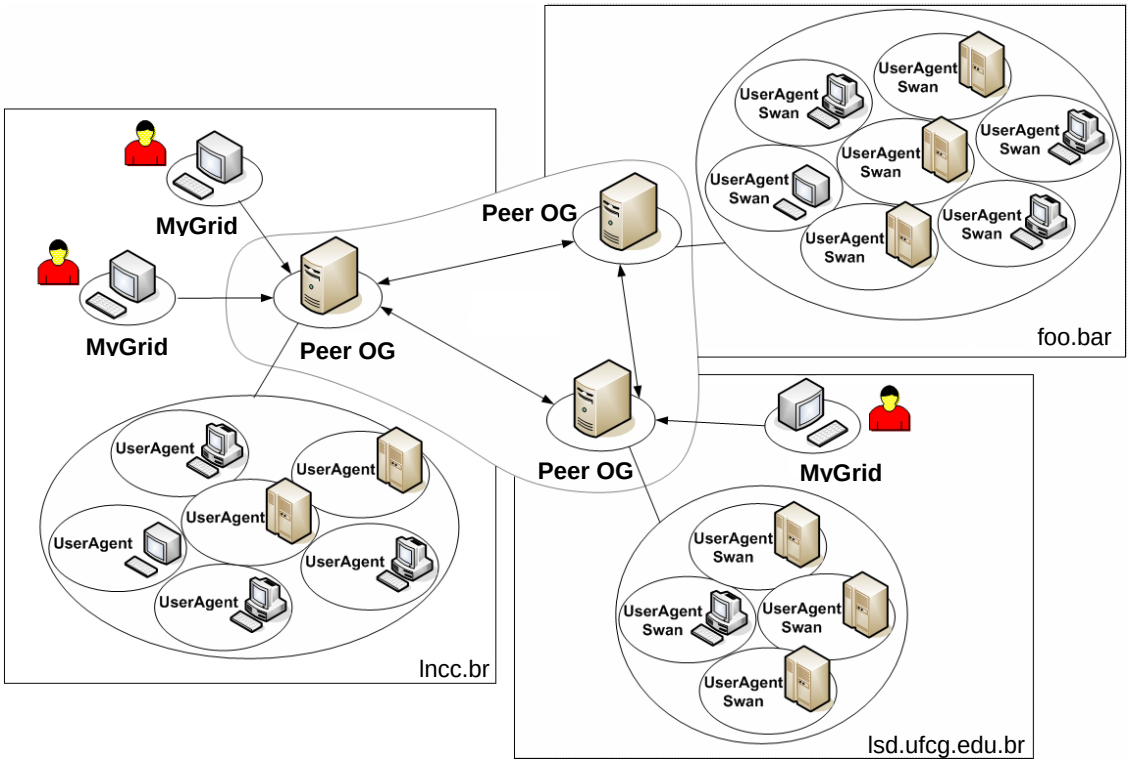


Figura 3.7 – *Os componentes da solução OurGrid. Fonte: http://www.ourgrid.org.*

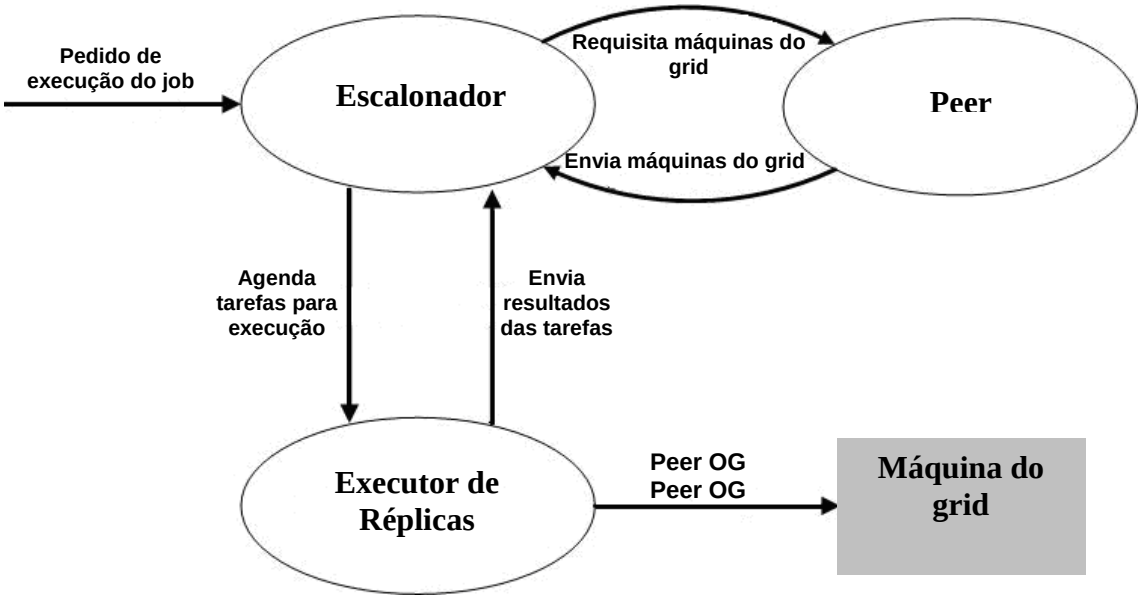


Figura 3.8 – *A arquitetura do MyGrid. Fonte: http://www.ourgrid.org.*

O componente UserAgent é executado em cada máquina que faz parte da grade. Ele fornece o acesso necessário para a máquina que executa o MyGrid. Juntamente com o *peer*, permite a utilização de recursos das máquinas em redes privadas. Executa a fase *remote* da tarefa definida no MyGrid. O módulo UserAgent Swan contém o serviço de segurança, protejendo a máquina provedora de recursos contra a execução de códigos maliciosos.

4. FERRAMENTAS E TECNOLOGIAS UTILIZADAS NA EXECUÇÃO DE APLICAÇÕES DO TIPO *BAG OF TASKS* EM CLUSTERS E GRADES

Aplicações do tipo BoT (*Bag-of-Tasks* – Bolsa de Tarefas) [67] [68] são aquelas aplicações paralelas cujas tarefas são independentes umas das outras. Devido à autonomia das suas tarefas, tais aplicações têm sido executadas com sucesso em grades computacionais distribuídas, conforme demonstrado por SETI@home [71]. Na verdade, pode-se argumentar que aplicações BoT são as mais adequadas para grades computacionais, pois não há a interferência da velocidade da rede no tempo de execução de uma tarefa e nem a necessidade de conhecimento da estrutura de recursos compartilhados. Apesar da sua simplicidade, as aplicações BoT têm sido usadas em uma ampla variedade de cenários como mineração de dados [72], pesquisas extensivas (como quebra de chaves), *parameter sweeps* [73], simulações Monte Carlo [74], cálculos de fractais, biologia computacional [74] e processamento de imagens [75] [76]. Um exemplo de tarefa de alto desempenho de grande relevância para as organizações é o tratamento de imagens de documentos digitalizados.

Embora o processamento da imagem de cada documento possa ser efetuado em poucos segundos, o lote produzido por um único scanner não é passivo de processamento por uma máquina isolada. Faz-se necessário distribuir a tarefa para que o processo de digitalização cumpra seus objetivos. A busca de palavras chave que possibilitem a indexação automática ou a transcrição, ambas efetuadas por OCR, demandam processamento que pode atingir a ordem de minutos de CPU.

Este capítulo descreve as ferramentas e tecnologias utilizadas para a realização dos experimentos dessa tese.

4.1 A Solução BigBatch

Com a necessidade crescente de integrar os fluxos de documentos de uma organização aos seus sistemas de Tecnologia da Informação, é preciso digitalizar, manipular e armazenar os documentos em papel existentes. Para automatizar o processo de digitalização é comum a utilização de *scanners* de linha de produção, como aquele apresentado em [77], para a digitalização de lotes de milhares de documentos. A Figura 4.1 e a Figura 4.2 mostram documentos digitalizados com esse tipo de *scanner*.

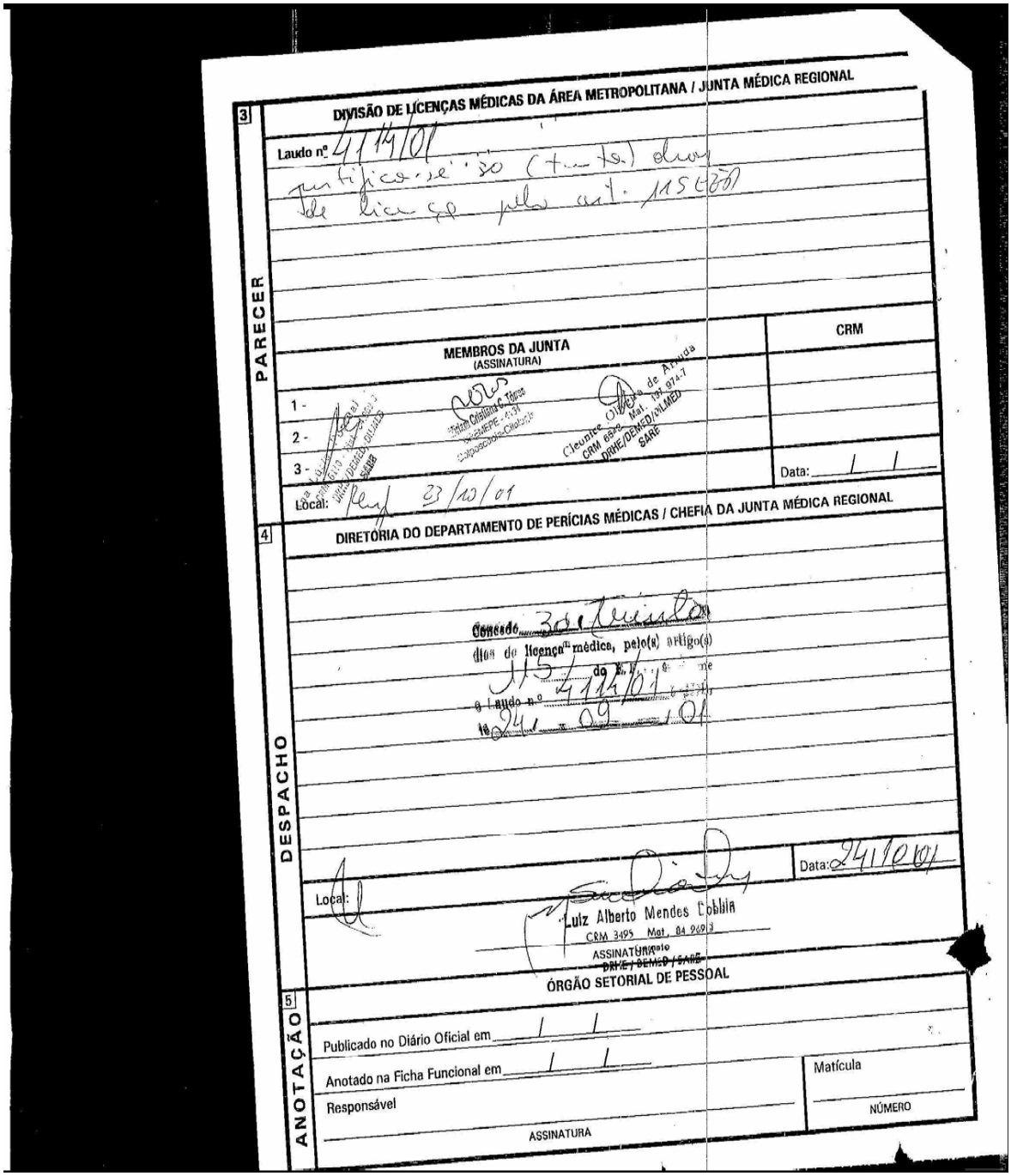


Figura 4.1 – *Imagem de um documento digitalizado em scanners de linha de produção com alimentação automática [21].*

Simplesmente digitalizar um documento, geralmente, não costuma produzir um documento digital útil. O uso de um *scanner*, de alimentação manual ou de uma linha de produção com alimentação automática, introduz características na imagem digitalizada bruta que são indesejáveis aos documentos digitais. Alguns problemas comumente encontrados são: a presença de bordas

pretas, a orientação errada, as deformações no documento, bem como a presença de ruído *salt-and-pepper*.

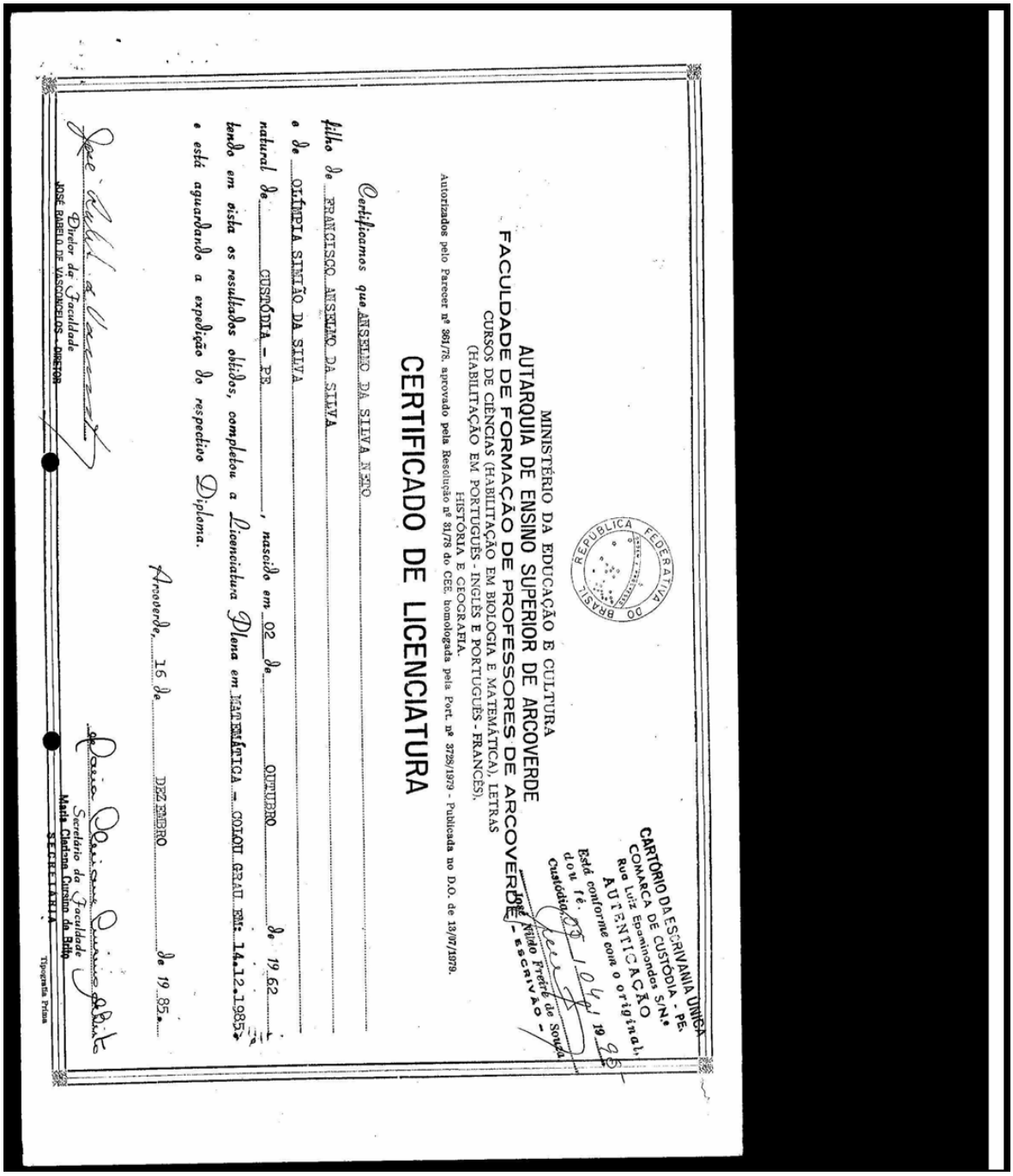
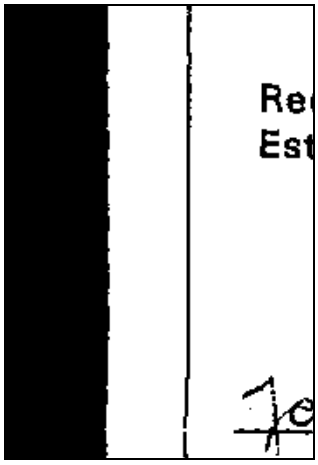


Figura 4.2 – *Imagem de documento digitalizado em scanners de linha de produção com alimentação automática [21].*

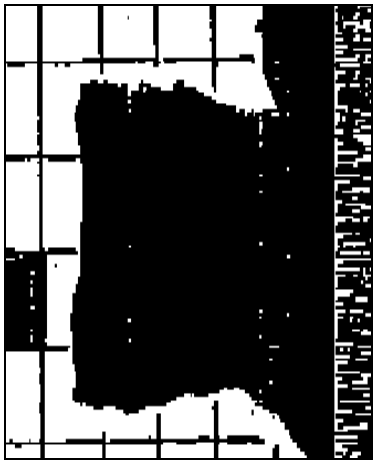
Dependendo de uma série de fatores, tais como o tamanho do documento, o seu estado de conservação e integridade física e a presença de poeira no documento e partes do *scanner*, muito

freqüentemente a imagem gerada é enquadrada por uma sólida borda negra. A Figura 4.1 e a Figura 4.2 ilustram esse tipo de problema e a Figura 4.3 apresenta alguns tipos típicos de ruído de borda.

Esta característica indesejável, também conhecida como ruído de borda ou *marginal noize*, não só diminui a qualidade da imagem resultante para visualização CRT (*Cathode Ray Tube* – Tubo de Raios Catódicos), mas também consome espaço de armazenamento e grandes quantidades de *toner* para impressão. Remover tais bordas manualmente não é prático, demanda tempo e exige usuários especializados. Vários fabricantes de scanners de linha de produção têm desenvolvido ferramentas de software para remover essas bordas. No entanto, muitos desses programas [17] [19] [20] [78] tendem a remover partes essenciais dos documentos ou deixam demasiada borda residual.



Quadro preto sólido ruidoso.



Ruído de borda de forma irregular.

Padrão com listra.

Informação junto com o ruído de borda.

Figura 4.3 – *Tipos de ruído de borda [21].*

Os documentos nem sempre são colocados corretamente na superfície plana do scanner, seja manualmente pelos operadores ou pelo dispositivo automático de alimentação, gerando imagens orientadas e rotacionadas incorretamente, conforme ilustra a Figura 4.1. Para o ser humano, as imagens mal orientadas e rotacionadas são de difícil visualização e leitura. Para os computadores alguns problemas podem acontecer como a necessidade de espaço extra de armazenamento por conter erros de reconhecimento e a transcrição da imagem por ferramentas automáticas de OCR (*Optical Character Recogniser* – Reconhecedor Ótico de Caractere). Assim, correções de orientação e deformação estão presentes em qualquer ambiente de processamento de documentos. No entanto, a precisão da detecção do ângulo de orientação e rotação, a qualidade da correção da deformação bem como o tempo necessário para o processamento dessas operações varia significativamente de uma ferramenta para outra. Três problemas freqüentemente aparecem na rotação das imagens monocromática: buracos brancos aparecem dentro de áreas planas pretas, bordas lisas tornam-se desigual e cheia de ondulações, e áreas vizinhas tornam-se desconectadas. Muitas vezes, o resultado da rotação em imagens monocromáticas mostra efeitos da degradação tal como aqueles apresentados na Figura 4.4.

Figura 4.4 – *Palavra rotacionada por 45º e -45º por algoritmo [21].*

A imagem digitalizada pode também incluir algum ruído em zonas que eram originalmente homogêneas, ou perto de contornos, especialmente o tipo de ruído conhecido como *salt-and-pepper*. A remoção deste tipo de ruído pode freqüentemente melhorar a qualidade do documento e facilitar o processamento de mais documentos, como o reconhecimento de caracteres utilizando ferramentas OCR. No mínimo, uma ferramenta de processamento de documentos digitalizados deve incluir filtros para esses três problemas. Uma ferramenta que atende esses requisitos é o BigBatch [21]. A Figura 4.5 mostra uma imagem digitalizada e a imagem resultante após a correção dos problemas pela ferramenta BigBatch.

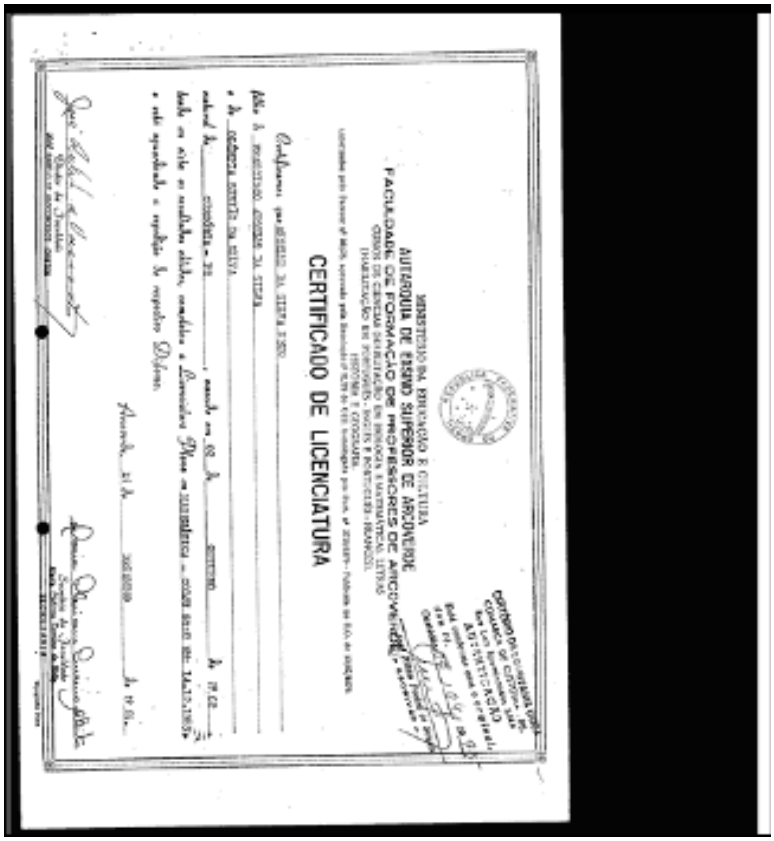


Figura 4.5 – *a) Imagem original digitalizada.*

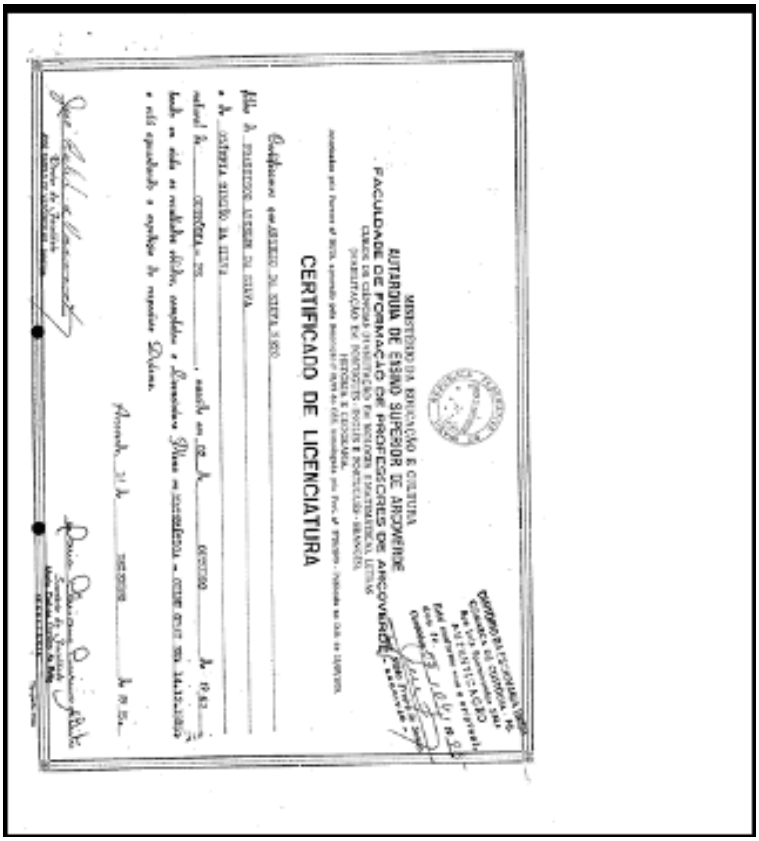


Figura 4.5 – *b) Ruído de borda removido.*

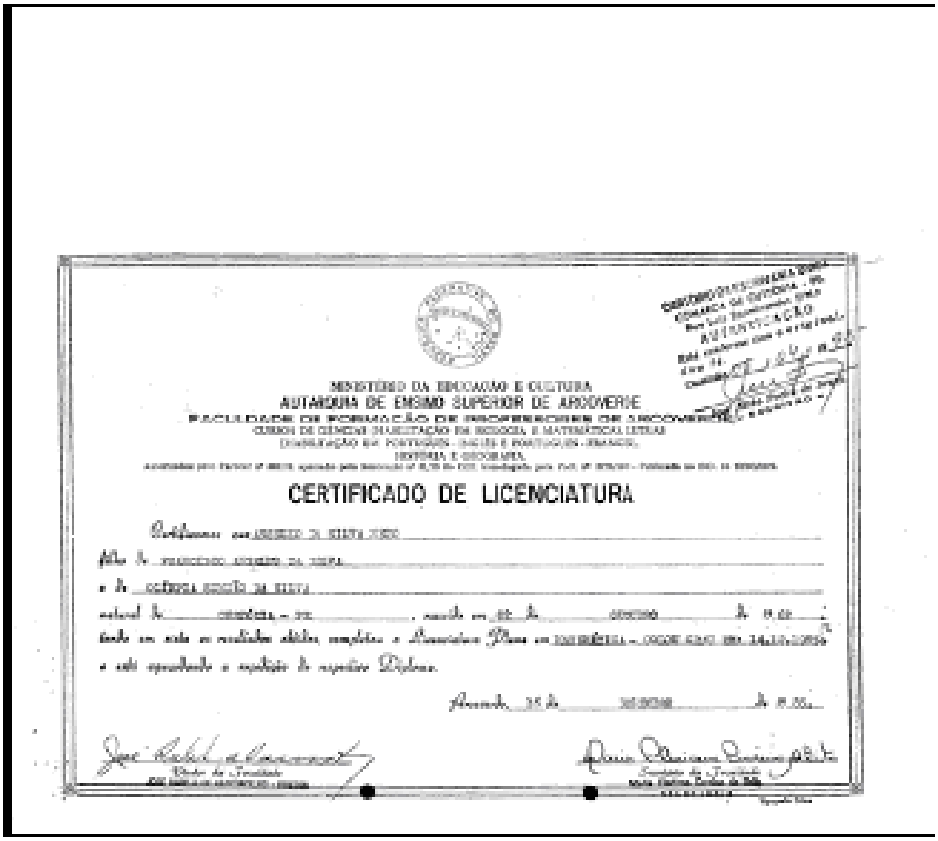


Figura 4.5 – *c) Correção da orientação e deformação.*

Figura 4.5 – *d) Imagem cortada e filtrada.*

Figura 4.5 – *Imagem original digitalizada e após ser corrigida pela ferramenta BigBatch [21].*

A questão está em aplicar a ferramenta de ajuste da imagem dos documentos digitalizados sobre todos os documentos do acervo de uma organização. O que torna essa tarefa exigente do ponto de vista computacional é o grande número de imagens que precisam ser tratadas em casos comuns, embora o tratamento de cada imagem, isoladamente, seja uma tarefa relativamente rápida em computadores atuais. Tais imagens de documentos burocráticos são obtidas por scanners de linha de produção com alimentação automática, sendo produzidos, dependendo do tamanho dos documentos, cerca de mais de 1.000 imagens de documentos por hora.

O problema descrito é um exemplo real que necessita de uma solução de computação de alto desempenho e o processamento de uma imagem não depende do processamento das outras imagens. Problemas dessa natureza são naturalmente passivos de processamento paralelo podendo-se utilizar um cluster ou grade para resolvê-lo. Em geral, a escolha de uma ou outra arquitetura depende das características intrínsecas do problema e dos computadores disponíveis, visto que, clusters são normalmente organizados em redes locais, com recursos dedicados e homogêneos, enquanto grades geralmente são utilizadas em WANs e internet, não dedicadas e heterogêneas. Os resultados da comparação entre utilizar uma ou outra arquitetura foram apresentados nas referências [79] e [80].

As seções seguintes descrevem as plataformas de cluster e de grade utilizadas bem como a estrutura física de computadores e os dados a serem processados.

4.1 O Cluster-Scala

Existe uma ampla variedade de softwares e bibliotecas para clusters que podem ser utilizadas no gerenciamento de tarefas em um cluster, por exemplo, o openMosix [42], Condor [43] e Microsoft Cluster Pack [81] [82] e MPI [58] [83]. Mas geralmente as aplicações são escritas explicitamente para o cluster em questão, incluindo a divisão de tarefas entre os nós do cluster e a maneira como eles se comunicam.

Seguindo essa linha, uma aplicação para distribuir as tarefas entre os nós do cluster que executaram o processamento das imagens com o BigBatch foi desenvolvido utilizando a linguagem de programação Scala [84]. A linguagem de programação Scala é uma linguagem funcional e orientada a objetos que foi projetada para ser executada sobre a Máquina Virtual Java e utiliza as bibliotecas e APIs (*Application Programming Interface* – Interface de Programação de Aplicativos) do Java. Essa linguagem foi escolhida por trabalhar com a plataforma Java, aumentando sua portabilidade e disponibilidade de bibliotecas; além de possuir bom suporte para programação distribuída utilizando a biblioteca Actors [85] para troca de mensagem entre os nós do cluster.

Outro motivo para ter o BigBatch sendo executado sobre a plataforma Java é a facilidade de integração com os componentes da grade, como será explicado mais adiante.

A aplicação é composta pelo módulo mestre, *BigBatch Master Module*, e pelo módulo cliente, *BigBatch Client Module*. A comunicação entre os nós é feita por passagem de mensagem, sempre do mestre para os clientes ou dos clientes para o mestre mas nunca entre os clientes. O agente mestre além de executar a aplicação BigBatch é responsável por coordenar a distribuição das tarefas colocando em uma fila todas as tarefas a serem processadas. Além disso, registra os nós clientes colocando-os em uma lista de clientes registrados e livres para processarem as tarefas. Enquanto a lista de tarefas não está vazia, o mestre simplesmente pega a primeira tarefa na fila e manda para o primeiro cliente na lista de clientes disponíveis, o qual passa a ficar ocupado com a tarefa e, portanto, sai da lista. Quando não há mais clientes disponíveis, o servidor espera algum cliente terminar. Quando o mestre manda uma tarefa para o cliente, envia também o pacote de imagens que deve ser processado. Os clientes, ao serem iniciados, mandam uma mensagem para o mestre se registrando e avisando que estão prontos para receber uma tarefa e executá-la. Ao receber uma tarefa o mestre é marcado como ocupado e assim permanece até concluir a sua tarefa, então ele envia uma mensagem para o mestre dizendo que terminou, e em seguida envia o resultado da tarefa. Nesse momento o cliente é colocado novamente na lista de clientes disponíveis. Esse procedimento continua até que todas as tarefas tenham sido processadas.

O balanceamento de carga é bastante simples devido à homogeneidade da arquitetura do cluster e a natureza do paralelismo dos dados do problema.

O código fonte da aplicação bem como exemplos de arquivos de log gerados estão disponíveis nos anexos a esta tese.

4.2 O Cluster MPI

Como o padrão MPI [58] [83] é amplamente utilizado em aplicações paralelas foi desenvolvida uma segunda aplicação a fim de levar vantagem de um cluster MPI. A distribuição das tarefas foi escrita em linguagem de programação C utilizando chamadas MPI para a comunicação entre os nós do cluster.

A aplicação é composta por um único módulo e reside em um computador do cluster chamado de mestre o qual contém ainda toda a base de tarefas a serem processadas além da aplicação BigBatch. A comunicação entre o mestre e os clientes, como acontece no cluster-Scala, é feita por passagem de mensagem utilizando chamadas MPI e acontece do mestre para os clientes e dos clientes para o mestre, mas nunca entre os clientes.

Cada processo é numerado de zero a *n-1*, dependendo do número de computadores disponíveis no cluster. O processo zero é o mestre, sendo responsável pela distribuição das tarefas

além de coordenar o envio e o recebimento de informações dos demais processos. Inicialmente o processo mestre lê uma lista de tarefas e serem processadas que estão contidas em um arquivo e divide tal lista de acordo com o número de processos disponíveis. Cada processo cliente, por sua vez, recebe a lista de tarefas a serem processadas, realiza o processamento e envia o resultado para o processo mestre. Novamente, devido à homogeneidade da arquitetura do cluster não faz sentido escolher entre um ou outro nó do cluster para executar determinada tarefa.

O código fonte da aplicação bem como exemplos de arquivos de log gerados estão disponíveis nos anexos a esta tese.

4.3 A Grade OurGrid

Existem várias plataformas de grade disponíveis atualmente sendo as mais conhecidas o Globus e o Condor. No Brasil, uma plataforma que vem ganhando espaço é o OurGrid [41], o qual é de fácil instalação e configuração e seu código é aberto. A versão utilizada é o OurGrid 3.3 composto pelos módulos *Peer*, *MyGrid* e *UserAgent*.

Os componentes Peer e MyGrid foram instalados e configurados em um mesmo computador chamado de computador mestre, enquanto os demais computadores, denominados clientes, executaram uma cópia do componente UserAgent. O computador mestre contém ainda toda a base de imagens a ser processada além da solução BigBatch para o processamento das imagens.

Os componentes Peer e MyGrid estão disponíveis na versão Linux enquanto o UserAgent pode ser executado tanto no ambiente Linux quanto no ambiente Windows. Todos eles são executados sobre a plataforma Java sendo necessária a versão 1.5 ou superior da Máquina Virtual Java.

Uma coleção de tarefas relacionadas ao mesmo problema é chamada de job; assim um job no OurGrid é composto por tarefas independentes, sendo cada tarefa composta pelas fases *init, remote* e *final*, executadas seqüencialmente. As fases *init* e *final*, executadas pelo componente MyGrid, geralmente são utilizadas para transferência de arquivos utilizados pela tarefa enquanto a fase *remote*, executada pelo componente UserAgent, processa a tarefa propriamente dita. A execução das tarefas é gerenciada pelo componente MyGrid que escalona as tarefas entre os nós disponibilizados pelo componente Peer de acordo com o método de escalonamento usado. Esse procedimento continua até que todas as tarefas tenham sido executadas. A Figura 4.6 ilustra o formato do arquivo de configuração para emissão das tarefas.

A aplicação BigBatch cria o job baseado em um arquivo de lote de documentos das imagens que devem ser processadas e comunica esse job ao componente MyGrid. Como ambos os componentes são executados sobre a plataforma Java, esta comunicação é realizada através do mecanismo RMI (*Remote Method Invocation* – Método de Invocação Remota) do Java.

```
job:
label: PACOTES_100
task:
init:  put /home/giorgia/pacotes100/filtros filtros
       put /home/giorgia/pacotes100/1.org 1.org
remote: ./filtros 1
final: get 1.pro /home/giorgia/result/1.pro
task:
init:  put /home/giorgia/pacotes100/filtros filtros
       put /home/giorgia/pacotes100/2.org 2.org
remote: ./filtros 2
final: get 2.pro /home/giorgia/result/2.pro
task:
init:  put /home/giorgia/pacotes100/filtros filtros
       put /home/giorgia/pacotes100/3.org 3.org
remote: ./filtros 3
final: get 3.pro /home/giorgia/result/3.pro
.
.
.
task:
init:  put /home/giorgia/pacotes100/filtros filtros
       put /home/giorgia/pacotes100/212.org 212.org
remote : ./filtros 212
final: get 212.pro /home/giorgia/result/212.pro
```

Figura 4.6 – *Formato do arquivo de configuração que descreve um job.*

O MyGrid disponibiliza dois algoritmos de escalonamento de tarefas que podem ser utilizados: o WQR [69] (*WorkQueue with Replication* – WQR com replicação de tarefas) e o *Storage Affinity* [70]. A estratégia de escalonamento WQR foi concebida para resolver o problema de se obter informações precisas sobre o futuro desempenho de tarefas que serão executadas nos recursos da grade. Nessa estratégia, as tarefas são enviadas randomicamente para processadores ociosos e quando um processador termina uma tarefa, ele recebe uma nova tarefa para ser executada. Porém, quando um processador torna-se disponível e não existem tarefas esperando para serem iniciadas, o algoritmo inicia a replicação das tarefas que ainda estão em execução em outros processadores. A primeira réplica a ser completada é tomada como resultado daquela tarefa e todas as outras réplicas desta tarefa são abortadas. A idéia por trás desta estratégia é melhorar o desempenho de tarefas através do aumento de chances de executar uma tarefa em um processador mais rápido do que aquele originalmente atribuído à tarefa.

A estratégia *Storage Affinity* foi concebida para explorar padrões de reutilização de dados a fim de melhorar o desempenho das aplicações. Os padrões de reutilização de dados acontecem em duas situações: entre jobs (um job usa os dados já usados por um outro job anterior a ele) e entre

tarefas (as tarefas compartilham os mesmos dados de entrada). *Storage Affinity* determina o quão fechado para um site é uma dada tarefa, ou seja, quantos bytes do conjunto de dados de entrada de uma tarefa já estão armazenados em um site específico. Assim, esta heurística é dividida em duas partes. A primeira parte consiste em atribuir um processador a cada tarefa e calcular o valor de *Storage Affinity* mais alto para cada tarefa. A tarefa, como o maior valor, é escolhida para ser escalonada, e assim sucessivamente até todas as tarefas serem escalonadas. Na segunda parte, quando não há mais tarefas esperando para serem executadas e existe pelo menos um processador disponível, uma réplica pode ser criada para qualquer tarefa em execução.

WQR atinge bom desempenho para aplicações do tipo *cpu-intensive* sem utilizar qualquer tipo de informação dinâmica sobre processadores, links de rede ou tarefas. O inconveniente está em alguns ciclos de CPU que são desperdiçados com as réplicas que não são completadas. Quanto ao *Storage Affinity* seu inconveniente está no desperdício dos recursos da grade devido a sua estratégia de replicação. Portanto, O algoritmo de escalonamento utilizado foi o *Storage Affinity*, visto que a aplicação é certamente do tipo *data-intensive*.

Os arquivos de configuração necessários à utilização do OurGrid bem como exemplos de arquivos de log gerados pelos módulos estão disponíveis nos anexos a esta tese.

4.4 Windows® HPC Server 2008

O Microsoft® Windows® HPC Server 2008 [44], chamado de HPCS (*High Performance Computing Server*), é o sucessor do Windows® Compute Cluster Server 2003 [45]. Oficialmente lançado em setembro de 2008, o HPCS é um sistema operacional voltado para o mercado da computação de alto desempenho fornecendo ferramentas que tem por objetivo proporcionar melhores desempenhos e escalabilidade para um ambiente altamente produtivo.

O HPCS disponibiliza um ambiente de cluster integrado que inclui o sistema operacional, o escalonador de tarefas, suporte a MPI versão 2 e componentes de gerenciamento e monitoramento do cluster. Construído com a tecnologia Windows Server ® 2008 64-bit, trabalha com vários nós de processamento, podendo escalar de dois a dois mil ou mais nós de servidores e cada nó pode conter de 1 a 4 processadores. O HPC Server 2008 possui funções de administração e diagnóstico que facilitam o trabalho dos administradores, ajudam a monitoração, localização e resolução de problemas mantendo assim o sistema estável.

Atualmente, o Windows HPC Server 2008 é utilizado pelo computador Dawning 5000A – Dawning 5000A, QC Opteron 1.9 GHz, que ocupa o 10º lugar no ranking dos 500 supercomputadores mais rápidos do mundo (http://www.top500.org/list/2008/11/100), no Centro de Supercomputação de Shanghai, China.

4.4.1 Arquitetura da Solução

Windows HPC Server 2008 é composto de um cluster de servidores que inclui um único nó principal chamado de *head node* e um ou mais clientes ou *compute nodes*. A Figura 4.7 apresenta uma rede típica com o Windows HPC Server 2008.

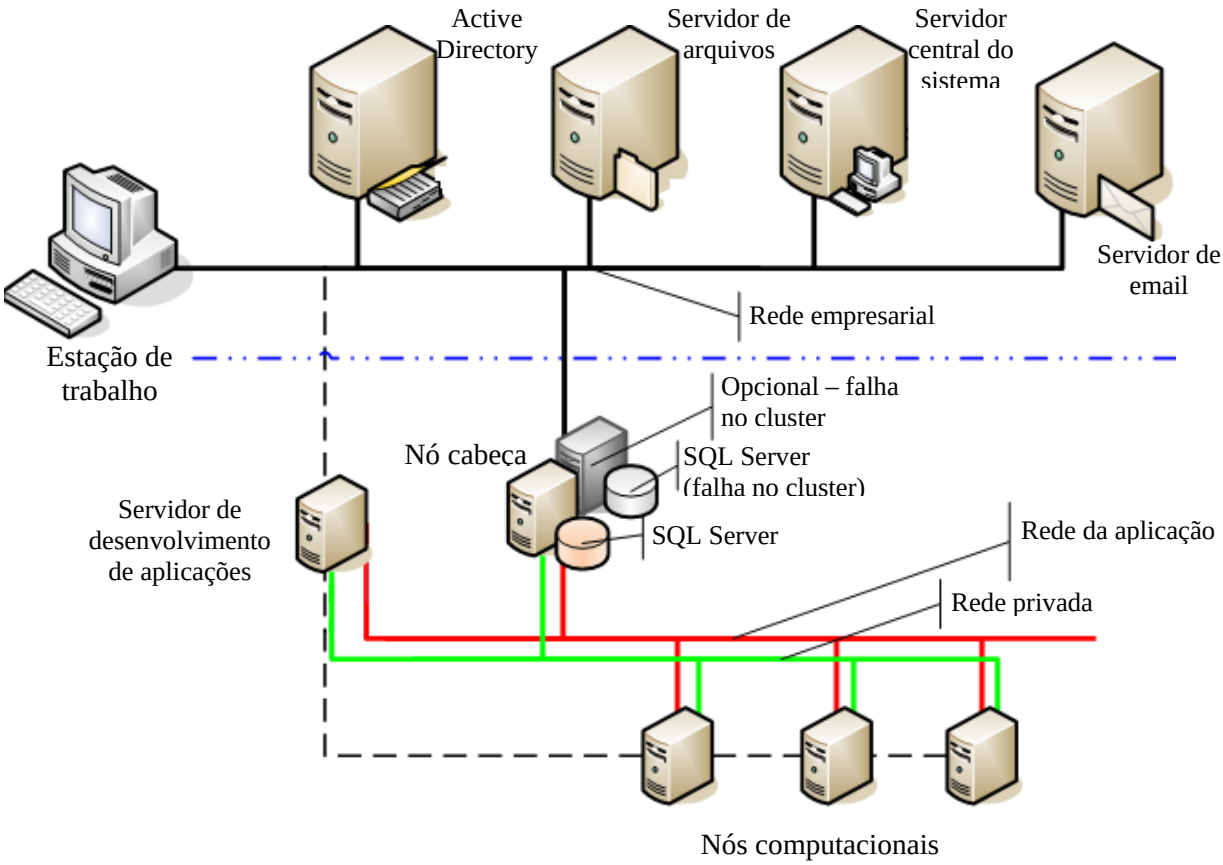


Figura 4.7 – *Rede típica do Windows HPC Server 2008.*

O nó cabeça que controla e serve de intermédio para todos os acessos aos recursos do cluster e é o único ponto de gerenciamento, implantação e escalonamento de tarefas para o cluster. Windows HPC Server 2008 utiliza a infraestrutura do *Active Directory*® [84] para segurança, gerenciamento de contas e todas as demais operações de gerenciamento.

4.4.2 Instalação e Configuração do Windows HPC Server 2008

A instalação do Windows HPC Server 2008 consiste em instalar o sistema operacional no nó cabeça, juntá-lo a um domínio do *Active Directory* e instalar o pacote HPC 2008. Na seqüência, uma lista com os passos necessários para completar a configuração quando da primeira inicialização do console de administração é iniciada. Esses passos incluem a definição da topologia de rede a ser utilizada e outras configurações, além da adição dos nós computacionais ao cluster. A

interface de gerenciamento do Windows HPC Server 2008 inclui testes de diagnóstico que permitem a detecção de problemas de conectividade, inicialização dos nós e *status* das tarefas executadas no cluster.

O Windows HPC Server 2008 suporta cinco diferentes topologias de rede, conforme ilustrado na Figura 4.8, que o usuário escolhe de acordo com as suas necessidades. As topologias têm de uma a três placas de rede em cada nó.

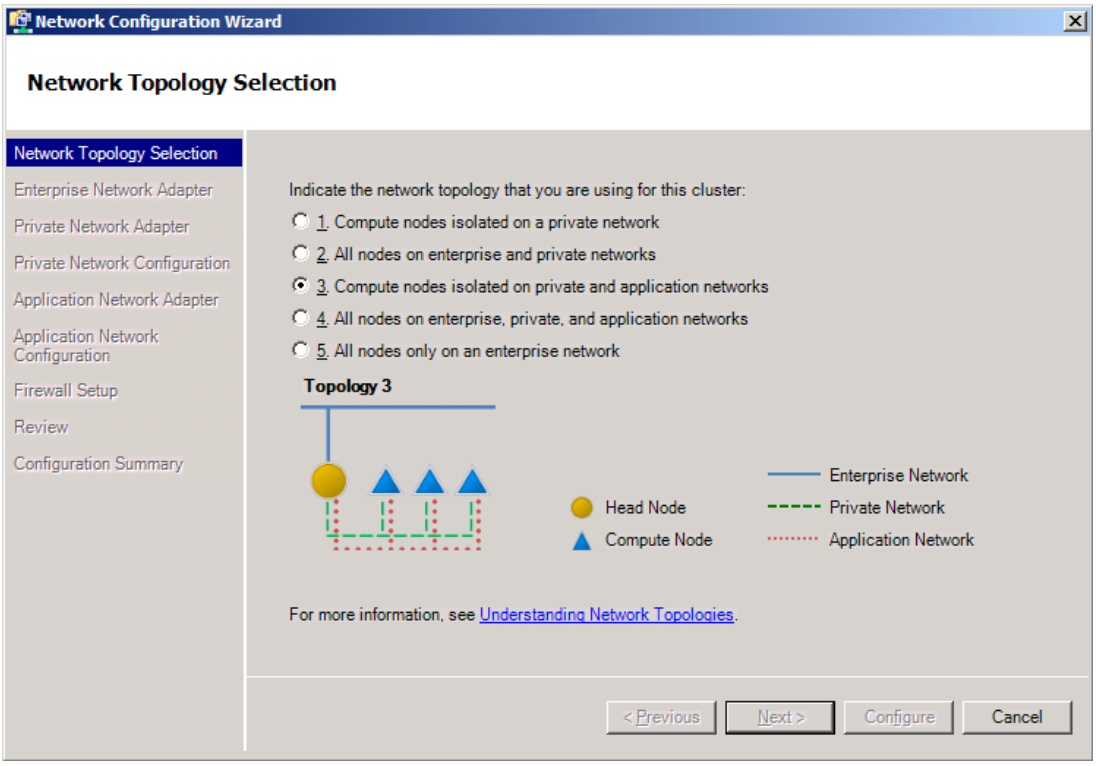


Figura 4.8 – *Tela do Assistente de Configuração da Rede.*

Na primeira topologia, nós computacionais isolados em uma rede privada, o nó cabeça tem duas placas de rede e pode utilizar o serviço de NAT (*Network Address Translation* – Tradução de Endereços de Rede) entre os nós computacionais, os quais têm uma única placa de rede cada conectada à rede privada e à rede empresarial.

Na segunda topologia, todos os nós em ambas as redes pública e privada, uma placa de rede está conectada à rede empresarial e uma está conectada à rede privada, dedicada à rede do cluster.

Na terceira topologia, nós computacionais isolados na rede privada e na rede de aplicação, o nó cabeça tem três placas de rede, uma conectada à rede empresarial, uma conectada à rede privada e a outra conectada à rede de aplicação. O nó cabeça pode fornecer NAT entre os nós computacionais e a rede empresarial, e cada nó computacional tem conexão com a rede privada e os protocolos de alta velocidade como o MPI para a rede de aplicação.

A quarta topologia, todos os nós na rede empresarial, privada e aplicação, uma placa de rede está conectada à rede empresarial, uma conectada à rede privada, dedicada ao gerenciamento da rede do cluster e a outra conectada à alta velocidade, dedicada à rede de aplicação.

Finalmente, a quinta topologia, todos os nós somente na rede empresarial, é um cenário limitado de rede onde cada nó tem uma placa de rede e cada nó computacional deve ser instalado e ativado manualmente.

A Figura 4.9 apresenta a tela do Console de Administração que tem cinco grandes painéis de navegação: configuração, gerenciamento do nó, gerenciamento do job, diagnósticos e gráficos e relatórios.

O painel Configuração inclui as configurações do cluster como o assistente para configuração da rede e o assistente para configuração dos nós computacionais. O Gerenciamento do Nó é utilizado para iniciar qualquer ação específica sobre o nó como, implantação, monitoramento, adição ou remoção. No painel Gerenciamento do Job é possível controlar totalmente o escalonamento da tarefa como adição, remoção ou cancelamento dela. O painel Diagnóstico permite a seleção de um nó ou grupo de nós e executar testes de diagnóstico como conectividade da rede, execução da tarefa, configuração e desempenho. Finalmente, o painel Gráficos e Relatórios mostra relatórios das tarefas e nós do cluster além de permitir relatórios agendados.

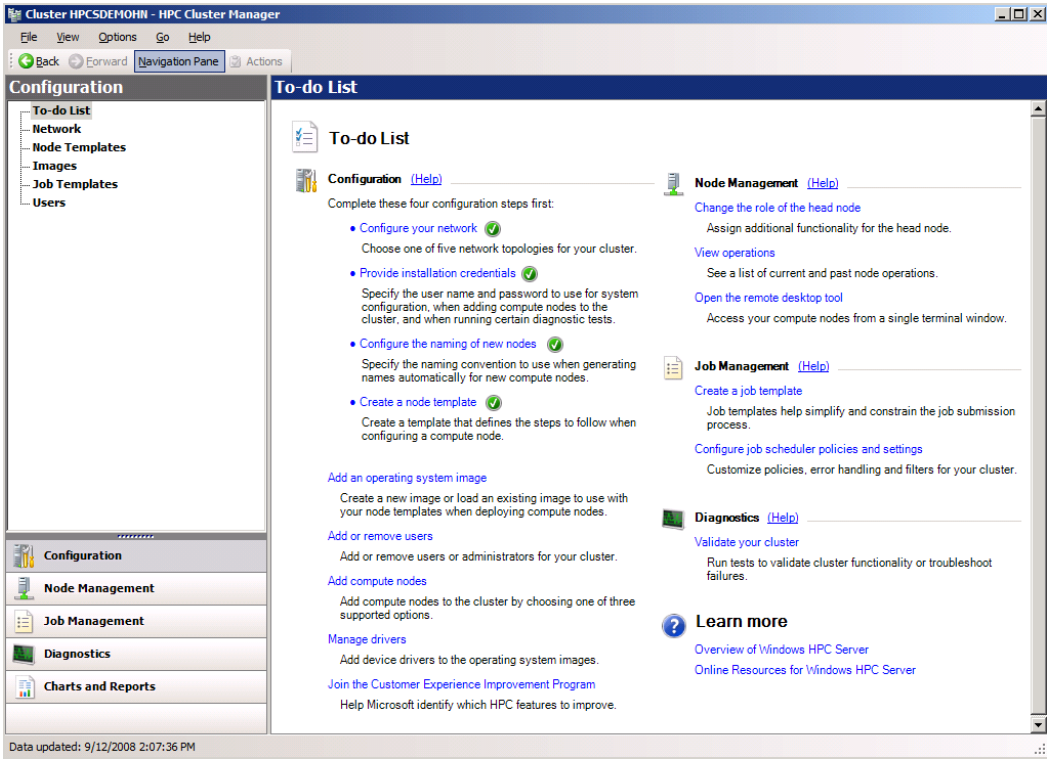


Figura 4.9 – *Tela do Console de Administração.*

4.4.3 Escalonador de Tarefas

O princípio básico de funcionamento das tarefas (*job*) no Windows HPC Server 2008 conta com três conceitos importantes: a submissão, o escalonamento e a execução do *job*. Estes três conceitos constituem a estrutura do ciclo de vida do *job* no HPC. A Figura 4.10 ilustra o relacionamento entre cada aspecto da operação do *job*. Cada vez que um usuário prepara um *job* para ser executado no cluster, este é executado através das três fases.

A submissão do *job* consiste em criá-lo, adicionar tarefas a ele e submetê-lo ao sistema. Os *jobs* podem ser compostos por uma única tarefa ou muitas delas e é possível determinar o número de processadores necessários para àquele *job* além de especificar se os processadores são exclusivamente para o *job* ou podem ser compartilhados com outros. O escalonador de *jobs* [82] mantém uma fila de *jobs* e tarefas associadas a ele. É responsável por fazer a alocação de recursos para os *jobs*, iniciar as tarefas nos nós computacionais do cluster e monitorar o status dos *jobs*, tarefas e nós computacionais. Essas configurações também podem ser definidas pelo próprio usuário. O escalonador utiliza as credenciais dos usuários definidas no *Active Directory* para executar cada *job*. Essas credenciais são criptografadas e armazenadas juntamente com o *job* somente até o término do *job*. Esse comportamento permite que a execução do *job* acesse recursos da rede tais como arquivos e servidores. A Figura 4.11 ilustra a tela do escalonador de *jobs*.

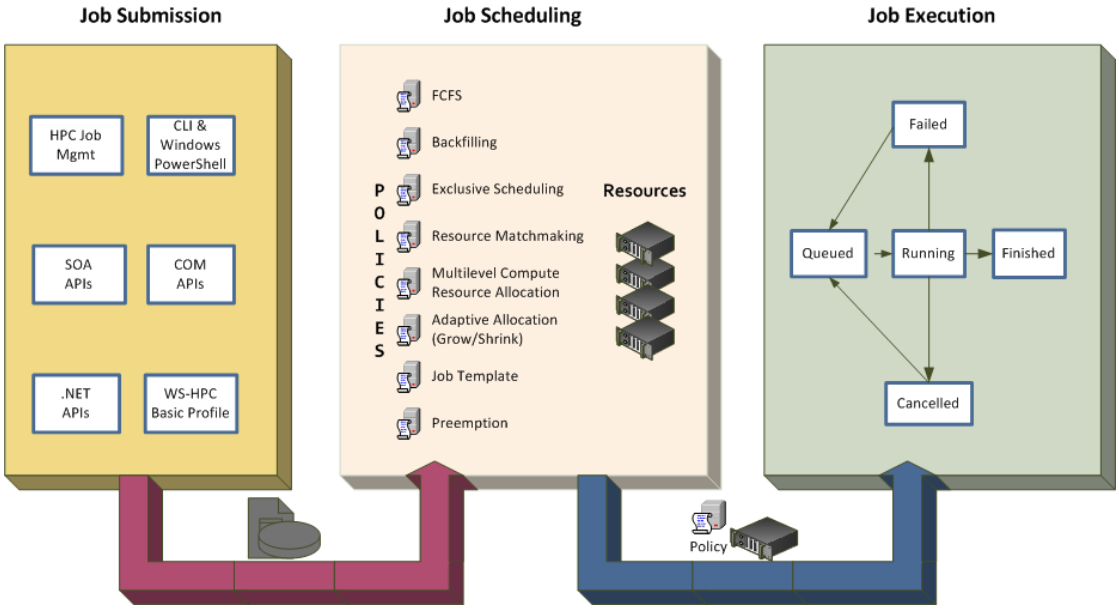


Figura 4.10 – *Ciclo de vido do job no Windows HPC 2008.*

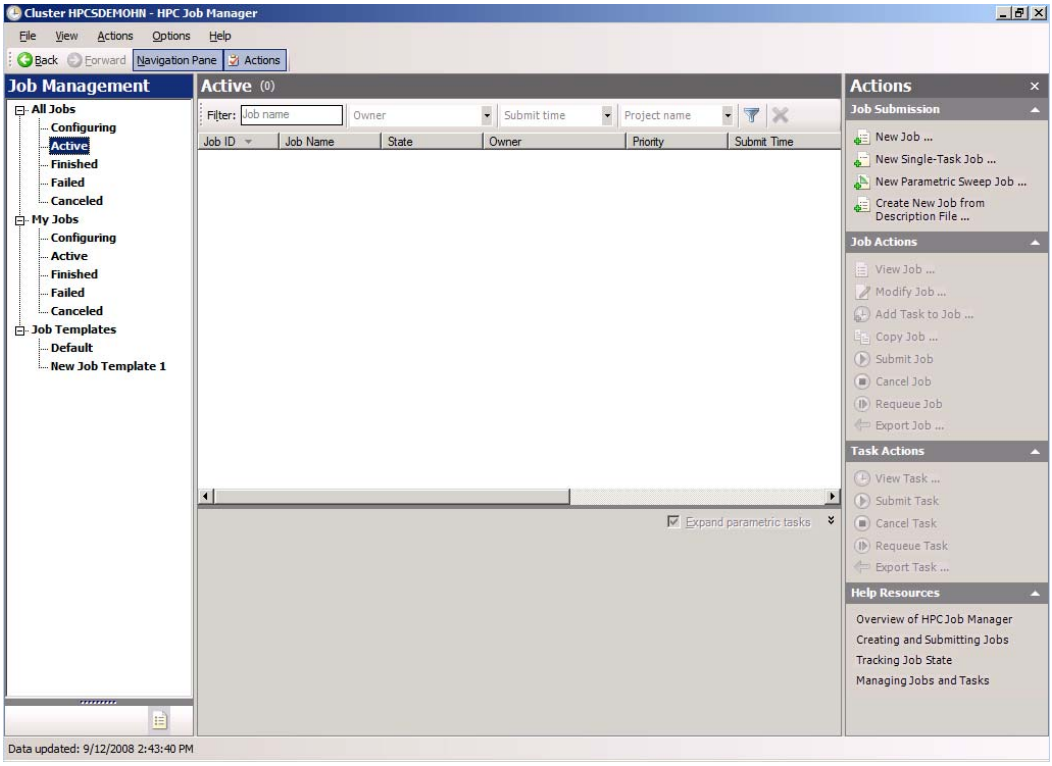


Figura 4.11 – *Tela do Escalonador de jobs.*

Os *jobs* são executados no contexto em que são submetidos pelo usuário. Eles também podem ser automaticamente colocados na fila após uma falha. As tarefas são gerenciadas pelos seus estados de transição, conforme mostrado na Figura 4.12. Os *jobs*, assim como as tarefas tem seu ciclo de vida representado por seu estado. Assim, um *job* ou tarefa pode ser criado, mas não submetido, então seu estado é chamado CONFIGURING. O *job* ou tarefa pode ser submetido e ficar aguardando por ativação, então seu estado é QUEUED. Se ele está em execução, seu estado é chamado RUNNING. Quando sua execução termina com sucesso ele está no estado FINISH. Se algum problema aconteceu e o *job* não pode ser terminado com sucesso então seu estado passa a ser FAILED. E se o usuário interrompe a sua execução ele passa para o estado chamado CONCELLED.

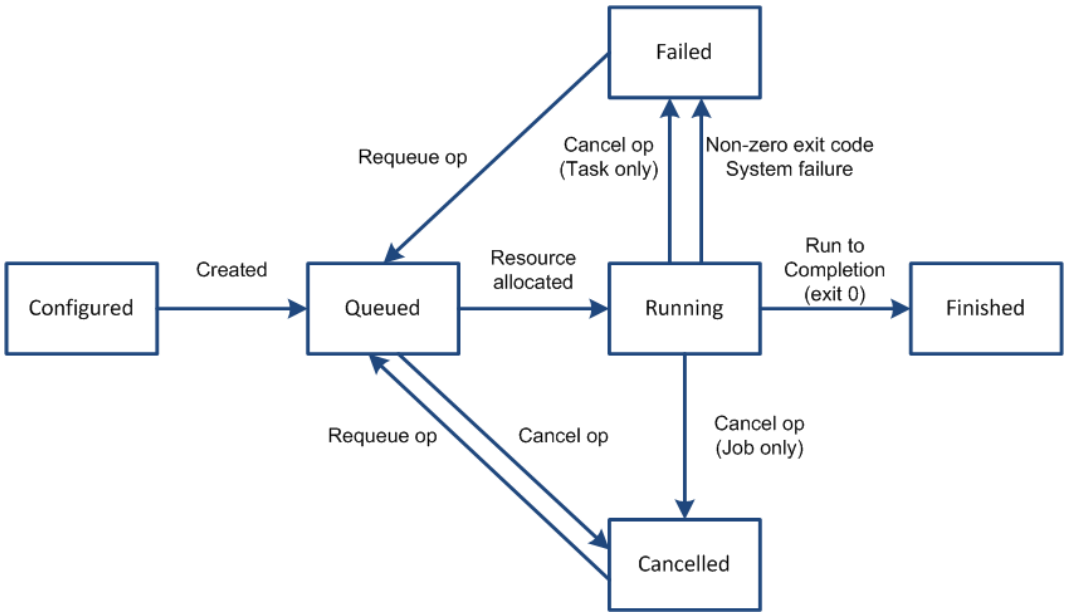


Figura 4.12 – *Estados de transição das tarefas.*

4.4.4 MSMPI

MPI e MPI2 são especificações amplamente aceitas para o gerenciamento de mensagens em clusters de alto desempenho. Entre as implementações do MPI mais amplamente aceitas está a implementação MPICH2 [65], de código aberto, desenvolvida no Argonne National Laboratory. O Windows HPC Server 2008 inclui a implementação MS-MPI [87] [83], baseada e compatível com a implementação MPICH2. MS-MPI é idêntica a quase todas as implementações disponibilizadas na MPICH2 ao mesmo tempo em que reforça a segurança e o gerenciamento dos processos. Usado com MSMPI, o Windows HPC Server 2008 suporta aplicações 32-bit e 64-bit.

MS-MPI usa uma comunicação eficiente com memória compartilhada entre os processadores em um nós computacional e drivers de rede para fornecer redes MPI de alto desempenho para adaptadores Gigabit Ethernet e InfiniBand, e suporta adaptadores que tenham um provedor Network Direct ou Winsock Direct.

MPI utiliza objetos chamados de comunicadores para controlar quais coleções de processos podem se comunicar. Um comunicador é um tipo de “linha de partida” dos processos que podem todos receber mensagens de outros processos, mas que ignoram quaisquer mensagens dentro do comunicador exceto as mensagens direcionadas para eles. Cada processo tem um identificador no comunicador. As rotinas de comunicação da MSMPI também incluem operações coletivas que permitem o programador coletar e avaliar todos os resultados de uma operação particular entre todos os nós em uma chamada no comunicador.

O comando utilizado para iniciar uma aplicação MSMPI no cluster Windows HPC Server 2008 é o mpiexec.exe que inclui muitos parâmetros para controlar a execução da aplicação. É possível especificar, por exemplo, o número de processadores necessários e os nós explicitamente.

4.5 Processadores com Tecnologia *Hyperthreading* e *Dual Core*

A tecnologia *hyperthreading* [88] é uma técnica criada para oferecer maior eficiência na utilização dos recursos de execução do processador. Segundo a Intel, seu desenvolvedor, essa tecnologia oferece um aumento de desempenho de até 30% dependendo da configuração do sistema. A tecnologia *hyperthreading* simula em um único processador físico dois processadores lógicos. Cada processador lógico recebe seu próprio controlador de interrupção programável e conjunto de registradores. Os outros recursos do processador físico, tais como, cache de memória, unidade de execução, unidade lógica e aritmética, unidade de ponto flutuante e barramentos, são compartilhados entre os processadores lógicos.

Uma ilustração da diferença entre um computador com tecnologia *hyperthreading* e um computador sem ela é mostrado na Figura 4.13. AS refere-se aos registradores e controladores de interrupção. Na área denominada Recursos de Execução estão todos os recursos que o processador necessita para executar as instruções. O processador da direita que suporta a tecnologia trabalha duplicando seus registradores e controladores e compartilhando os recursos de execução entre os processadores lógicos, parecendo assim um sistema com dois processadores.

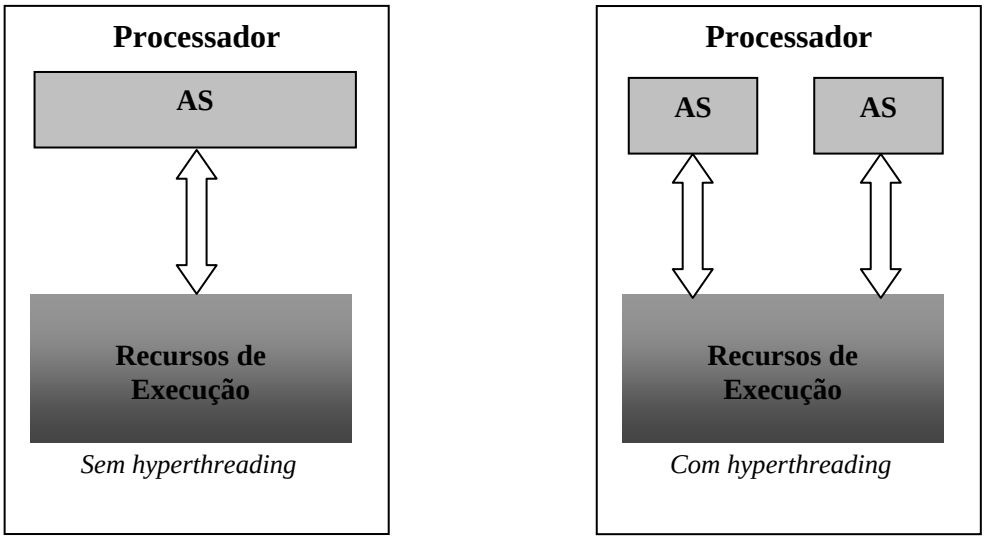


Figura 4.13 – *Comparação de um processador com e sem a tecnologia hyperthreading.*

As pesquisas atuais em arquitetura de computadores indicam o crescimento no uso de multiprocessadores *single-chip*, tecnologia *dual core* [88], os quais tratam tarefas simultâneas ainda melhor. Tal tecnologia refere-se a circuitos integrados que contêm um processador com dois núcleos no mesmo circuito integrado. A Figura 4.14 mostra esquematicamente a arquitetura Core 2 Duo. Tipicamente, isso significa que dois processadores idênticos são fabricados de forma que residam lado a lado no mesmo circuito. É possível escalonar dois processos separadamente dentro do processador.

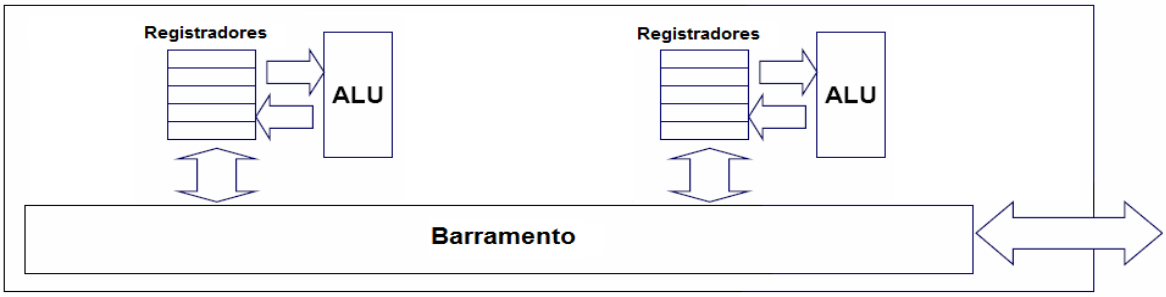


Figura 4.14 – *Desenho esquemático da arquitetura dual core utilizada nos processadores Core 2 Duo.*

4.6 Infraestrutura Utilizada

A infraestrutura utilizada para a execução das tarefas de processamento de imagens nas arquiteturas do cluster e da grade é formada por três configurações diferentes de computadores. A topologia de rede adotada é aquela apresentada na Figura 4.15.

A primeira configuração consiste de oito computadores Intel Pentium IV com tecnologia *Hyperthreading*, clock de 3.2 GHz, 512 MB de memória RAM DDR com cache L1 de 16 KB e L2 de 2048 KB, HD Ultra-ATA de 80 GB, conectados em rede local Ethernet de 100Mbit/s através de um switch Ethernet 3COM, modelo OfficeConnect, com 08 portas mais uma porta de Uplink. O sistema operacional utilizado é o Ubuntu Linux versão 6.06 [89] para a configuração de grade e para o cluster-Scala.

A segunda configuração adotada é composta por oito computadores Intel Pentium IV com tecnologia *Hyperthreading*, clock de 3.2 GHz, 1 GB de memória RAM DDR2 com cache L1 de 16 KB e L2 de 1024 KB, HD SATA-II de 250 GB, conectados em rede local Ethernet de 100Mbit/s através de um switch Ethernet 3COM, modelo OfficeConnect, com 08 portas mais uma porta de Uplink. O sistema operacional utilizado é o Ubuntu Linux versão 6.06 [89] para a configuração de grade e para o cluster-Scala.

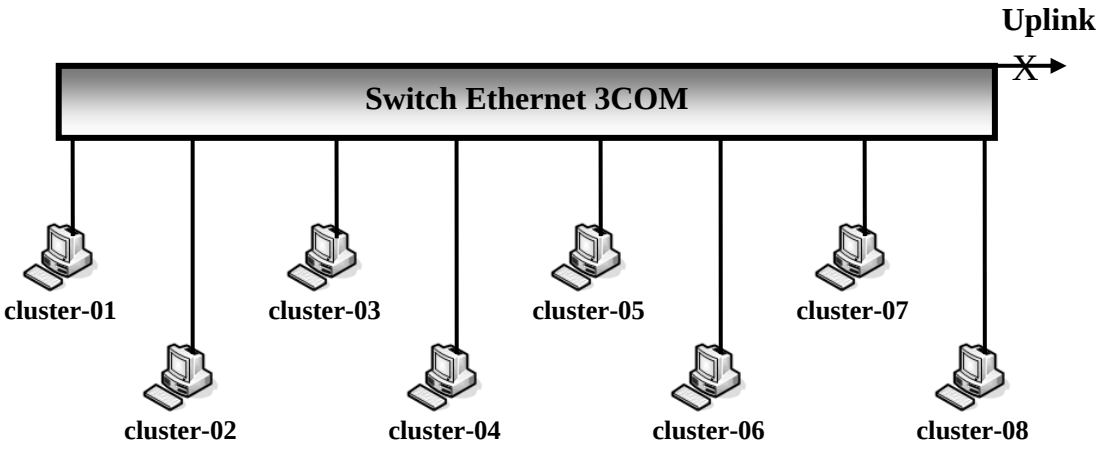


Figura 4.15 – *Topologia de rede utilizada nos experimentos.*

A terceira configuração utilizada é composta por oito computadores Intel Core 2 Duo com clock de 2.66 GHz, 2 GB de memória RAM DDR2 com cache L1 de 64 KB e L2 de 4096 KB, HD SATA II de 250 GB, conectados em rede local Ethernet de 100Mbit/s através de um switch Ethernet 3COM, modelo OfficeConnect, com 08 portas mais uma porta de Uplink. O sistema operacional utilizado é o Ubuntu Linux versão 6.06 [89] para a configuração de grade e para o cluster-Scala. Para o cluster MPI é utilizado o ambiente Windows HPC Server 2008 [44].

4.7 O Experimento

Os resultados utilizados na comparação entre as arquiteturas de cluster e grade foram gerados pela execução da ferramenta *BigBatch* [21], um ambiente para processamento de documentos digitalizados na forma de imagens monocromáticas. As imagens resultantes deste processamento são representações melhores de documentos legados e utilizam menos espaço para armazenamento. A base de imagens a serem processadas consiste de uma coleção de 21.200 imagens reais, carga de processamento que corresponde à produção de 1 dia de trabalho (8 horas) de um scanner de linha de produção com alimentação automática.

A filtragem de cada imagem é independente uma da outra, confirmando que esta aplicação é naturalmente *bag-of-tasks* e sugere o particionamento do problema. Para minimizar *overheads* de rede e de escalonamento, foi decidido atribuir a cada tarefa o processamento de um pacote de imagens, composto de *N* imagens cada, em vez de uma tarefa por imagem. Dessa forma, com pacotes de 100 imagens e um total de 21.200 imagens, 212 pacotes foram gerados, totalizando 212 tarefas independentes a serem executadas nas arquiteturas de cluster e grade.

O número *N* de imagens por pacote foi variado para avaliar o impacto do tamanho das tarefas no desempenho das estruturas. Conjuntos completos de tarefas foram gerados para

processar todas as 21.200 imagens em pacotes de tamanho *N* igual a 25, 50, 100, 500 e 1,000 imagens por pacote.

As tarefas foram executadas sob diferentes condições tanto na grade quanto no cluster sendo que o primeiro cenário consistiu na execução das tarefas com *N* igual a 100 imagens por pacote, com o mestre dedicado, sem participar do processamento das tarefas, e com o mestre não dedicado, ou seja, ele também ajudou no processamento das tarefas. No segundo cenário, a configuração da grade incluiu ainda outra condição: como a grade trabalha com computadores não dedicados, uma carga computacional a mais foi adicionada continuamente através da execução de um DVD com vídeo nos computadores clientes, enquanto esses também executavam o processamento das tarefas. Tal procedimento foi feito para avaliar o quanto uma carga computacional extra, não relacionada às tarefas da grade, afetaria o desempenho dos computadores clientes no processamento das imagens, simulando o comportamento dos computadores que fazem parte da grade em uma organização onde enquanto os usuários utilizam os computadores em suas tarefas corriqueiras o mesmo também trabalha no processamento de tarefas da grade. Isso não aconteceu no cluster devido ao fato dos computadores estarem dedicados exclusivamente ao processamento das tarefas do cluster.

No terceiro cenário foi variado o número de imagens por pacote e as tarefas foram executadas com o computador mestre não dedicado, sendo alocado também como computador cliente ajudando no processamento das tarefas. Isto foi feito para medir o quanto balanceamentos de cargas diferentes afetam o desempenho do processamento total.

Em cada condição observada, o número de computadores utilizados para compor o cluster e a grade foi variado para determinar como a aplicação aumenta em relação ao número de computadores disponíveis. O número de computadores nos experimentos variou de um a oito.

As tarefas de processamento de imagens foram especificadas de acordo com o formato exigido pelo OurGrid, mostrado na Figura 4.6. Os pacotes com as imagens foram inicialmente armazenados no computador mestre, sendo necessário transmiti-los para o computador cliente designado para processar a tarefa. Após o processamento, o computador cliente enviou o pacote resultante com as imagens filtradas para o computador mestre. Na configuração da grade, isto foi realizado pelas fases *init* e *final* das tarefas; enquanto tanto o cluster-Scala quanto o cluster MPI se encarregaram de tal tarefa de acordo com o software de cluster desenvolvido.

5. ANÁLISE COMPARATIVA ENTRE CLUSTERS E GRADES EM APLICAÇÕES *BAG OF TASKS* EM REDES LOCAIS

Com o objetivo de observar os efeitos das variações das condições estabelecidas na Seção 4.7 do Capítulo 4, este capítulo apresenta os resultados obtidos nos experimentos com o processamento das imagens de documentos utilizando a ferramenta BigBatch nos ambientes OurGrid, cluster-Scala e cluster-MPI. A análise sobre os resultados obtidos e algumas considerações também são mostradas.

5.1 Grade versus Cluster-Scala

A Tabela 5.1 mostra os tempos totais de execução das tarefas separadamente para a grade e o cluster-Scala, considerando os pacotes com 100 imagens utilizando computadores *hyperthreading* com 512 MB de memória RAM. Os resultados indicam que dedicar um computador para atuar apenas como mestre, ou não, impõe muito pouca diferença nos tempos totais de execução quando muitos computadores clientes estão presentes; as variações observadas são significativas apenas para um número pequeno de computadores clientes. Pode-se dizer então que o mestre processando as tarefas ajuda efetivamente na vazão do sistema e isso indica que o trabalho de distribuição das tarefas é leve.

A Tabela 5.1 mostra ainda que o fato de adicionar uma carga computacional externa à configuração da grade, simulada pela execução de um vídeo, causa um aumento em torno de 25% no tempo total de execução das tarefas. No cenário de uma grade utilizada por uma organização, é esperado que o número de computadores locais que possam ser alocados para a execução das tarefas seja maior que sete ou oito, indicando que uma carga externa pode ter impacto limitado no desempenho da grade. No entanto, é importante observar que a execução contínua de um vídeo não simula uma carga externa realista nos computadores que são usados como clientes da grade e estações de trabalho da organização; em uma situação real, o usuário coloca uma carga externa que varia de tempos em tempos, possibilitando a execução de suas aplicações que demandam mais. Quando comparado ao cluster, o desempenho da grade é melhor em alguns casos mesmo quando os computadores da grade executam o vídeo e os computadores do cluster estão todos dedicados exclusivamente ao processamento das tarefas.

Tabela 5.1 *– Resultados obtidos na execução das tarefas para N=100 imagens, considerando computadores hyperthreading com 512MB de memória RAM.*
Os tempos estão expressos em minutos.

Número de computadores processando as tarefas	**Grade**			**Cluster-Scala**	
	Tempo com o mestre não dedicado	**Tempo com o mestre dedicado**	**Tempo com o mestre não dedicado e executando o DVD com vídeo**	**Tempo com o mestre não dedicado**	**Tempo com o mestre dedicado**
1	627	804	865	681	822
2	320	378	390	379	402
3	233	238	283	249	251
4	180	187	200	221	226
5	146	154	173	175	177
6	128	135	142	147	150
7	105	119	125	113	114
8	84	---	113	92	---

Com o propósito de mostrar graficamente a comparação entre os tempos totais de execução das tarefas obtidos nos dois ambientes, a Figura 5.1 mostra os resultados para as duas configurações quando as tarefas foram executadas com o computador mestre não dedicado, ou seja, ele também executa as tarefas de processamento de imagens.

A Figura 5.2 apresenta graficamente os resultados obtidos nas configurações de cluster e grade considerando o computador mestre dedicado. Os resultados são bastante próximos e isso pode ser explicado por ineficiências na aplicação do cluster que não foram detectadas durante os experimentos. Como em ambas as configurações a carga é balanceada dinamicamente, o software de cluster, desenvolvido especialmente para o escalonamento das tarefas, está em vantagem somente porque seu algoritmo de escalonamento pode ser muito mais simples e conseguir resultados similares. Por esta razão, é esperado que os tempos totais para ambos, cluster e grade, sejam semelhantes, mesmo o software de cluster sendo melhorado.

Estes resultados indicam ainda que não há perdas quando da execução deste tipo de aplicação BoT, com características de balanceamento de carga, em plataformas de grade em vez de cluster. Vale salientar que ambas as configurações executam exatamente os mesmos aplicativos de processamento de imagens, sobre a mesma massa de dados.

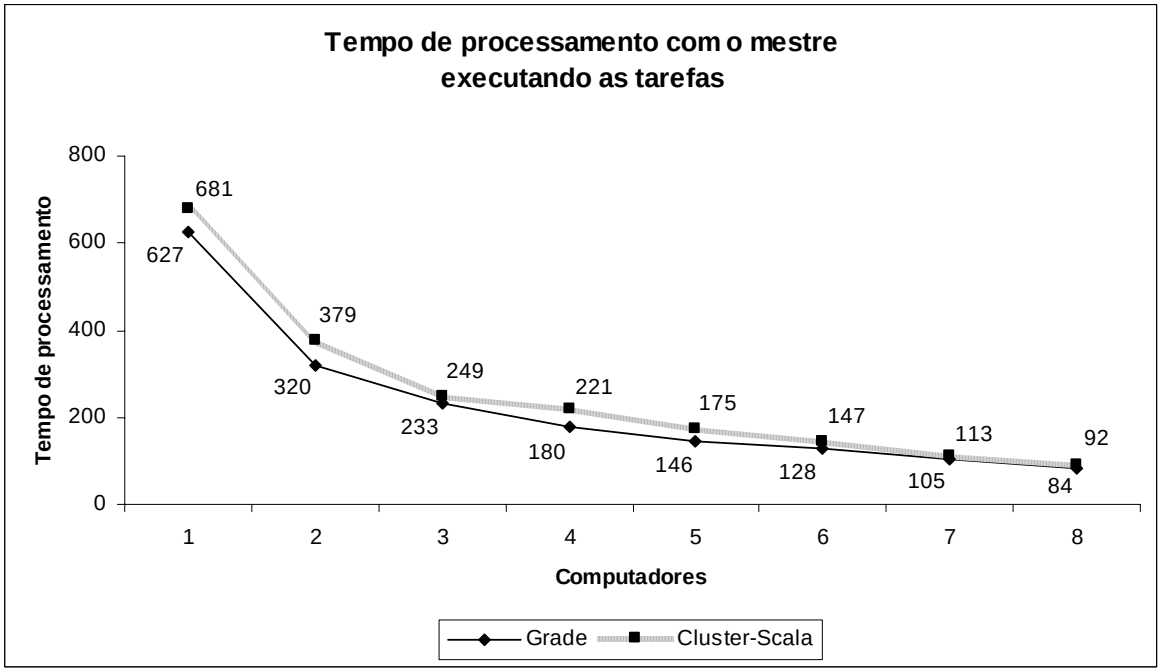


Figura 5.1 – *Tempo de processamento no cluster e na grade com a participação do computador mestre na execução das tarefas (N=100).*

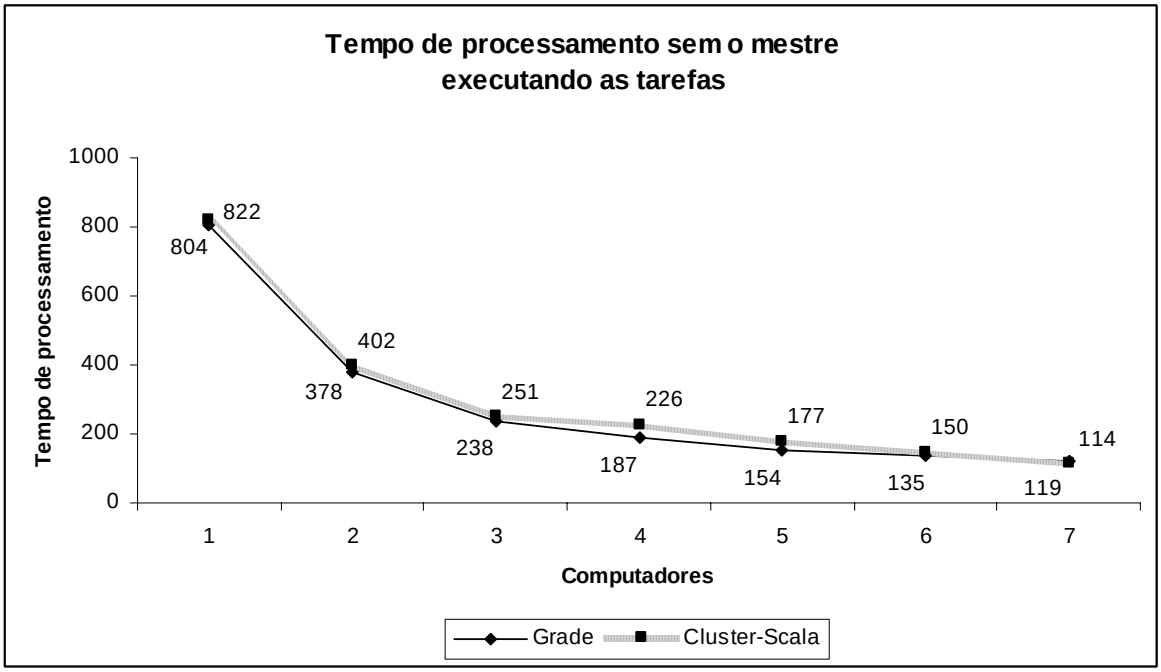


Figura 5.2 – *Tempo de processamento no cluster e na grade sem a participação do computador mestre na execução das tarefas (N=100).*

É possível observar ainda que os resultados obtidos, embora apresentem comportamento linear aproximado, não podem ser considerados lineares. A Figura 5.3 apresenta a curva de *speedup* e a Figura 5.4 a curva de eficiência para a grade e para o cluster ilustrando esse fato. O *speedup* para 8 processadores é aproximadamente 7.4 e não 8.

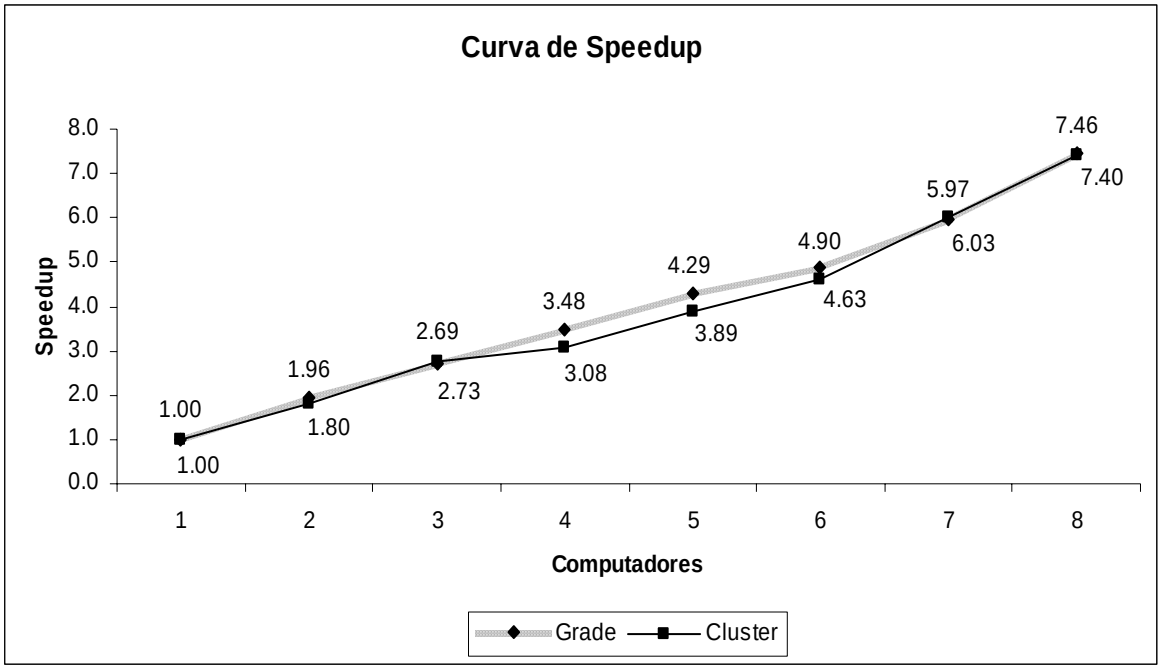


Figura 5.3 – *Curva de Speedup considerando a tecnologia hyperthreading com 512MB de memória RAM.*

A Tabela 5.2 apresenta os resultados obtidos pela grade considerando a variação no número de imagens por pacote quando processadas todas as imagens. Resultados similares aos da grade foram obtidos no cluster e são apresentados na Tabela 5.3. Nos dois casos, os resultados mostram claramente que existe pouca variação no tempo de processamento das tarefas indicando que a variação no tamanho do pacote, nesse caso, não influencia no tempo total de processamento. Assim, pode-se dizer que o balanceamento de carga é adequado ao problema não afetando o desempenho do sistema.

A Figura 5.5 e a Figura 5.6 mostram graficamente os resultados apresentados na Tabela 5.2 e Tabela 5.3. Esses resultados apontam ainda que o tráfego gerado na rede durante o processamento das tarefas é baixo, independentemente do tamanho do pacote utilizado. Assim, processar tarefas de alta granularidade ou fina granularidade não causa impacto no sistema. Os computadores utilizados nos experimentos são homogêneos tanto no cluster quanto na grade.

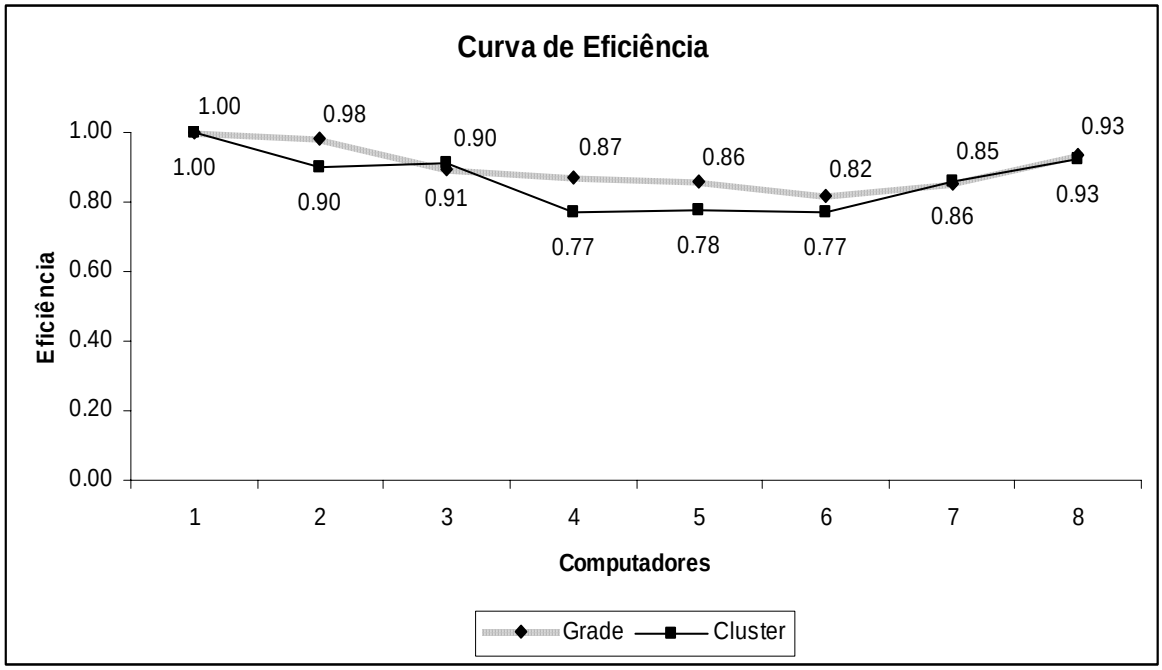


Figura 5.4 – *Curva de Eficiência considerando a tecnologia hyperthreading com 512MB de memória RAM.*

Tabela 5.2 *– Tempos de execução das tarefas na grade, com diferentes tamanhos de pacotes, considerando computadores hyperthreading com 512 MB de memória RAM. Os tempos estão expressos em minutos.*

Número de computadores processando as tarefas	**N=25**	**N=50**	**N=100**	**N=500**	**N=1000**
1	636	634	627	630	630
2	330	338	320	328	336
3	218	278	233	216	238
4	170	177	180	170	176
5	134	160	146	141	141
6	114	117	128	122	128
7	96	98	105	109	116
8	85	98	84	90	92

O tempo gasto no envio e recebimento dos pacotes é pequeno quando comparado com o tempo de execução das tarefas. Isso pode ser observado nos arquivos de log gerados pelas aplicações de cluster e grade contidos nos anexos a esta tese. O tempo médio gasto para processar

um pacote de tamanho 25 imagens é pequeno, menos de 60 segundos, considerando o envio e o recebimento dos pacotes a serem processados que fica abaixo de 1 segundo. Para um pacote de tamanho 100 o tempo de processamento é menor que 180 segundos com a transferência de arquivos entre 1 e 2 segundos e para um pacote de tamanho 1000 imagens, seu tempo de processamento fica em torno de 1800 segundos aproximadamente com tempo de transferência dos arquivos perto de 60 segundos.

Tabela 5.3 *– Tempos de execução das tarefas no cluster com diferentes tamanhos de pacotes, considerando computadores hyperthreading com 512 MB de memória RAM.*
Os tempos estão expressos em minutos.

Número de computadores processando as tarefas	**N=25**	**N=50**	**N=100**	**N=500**	**N=1000**
1	682	683	681	685	687
2	384	383	379	380	381
3	250	251	249	249	250
4	222	222	221	223	222
5	177	176	175	178	179
6	151	150	147	148	149
7	115	114	113	116	121
8	95	98	92	93	96

Esses resultados eram esperados uma vez que o tempo total de processamento de cada tarefa é bem maior que o tempo gasto para transmitir o pacote pela rede. Conseqüentemente a distribuição de carga não é afetada pelo tamanho das tarefas.

Outra observação aqui é que os computadores utilizados possuem processadores *multithreaded* [93], os quais gerenciam melhor a execução de tarefas simultâneas do que os processadores *single-threaded*, embora tal característica não tenha sido explicitamente utilizada pelas plataformas de cluster e grade quando executados os experimentos descritos anteriormente.

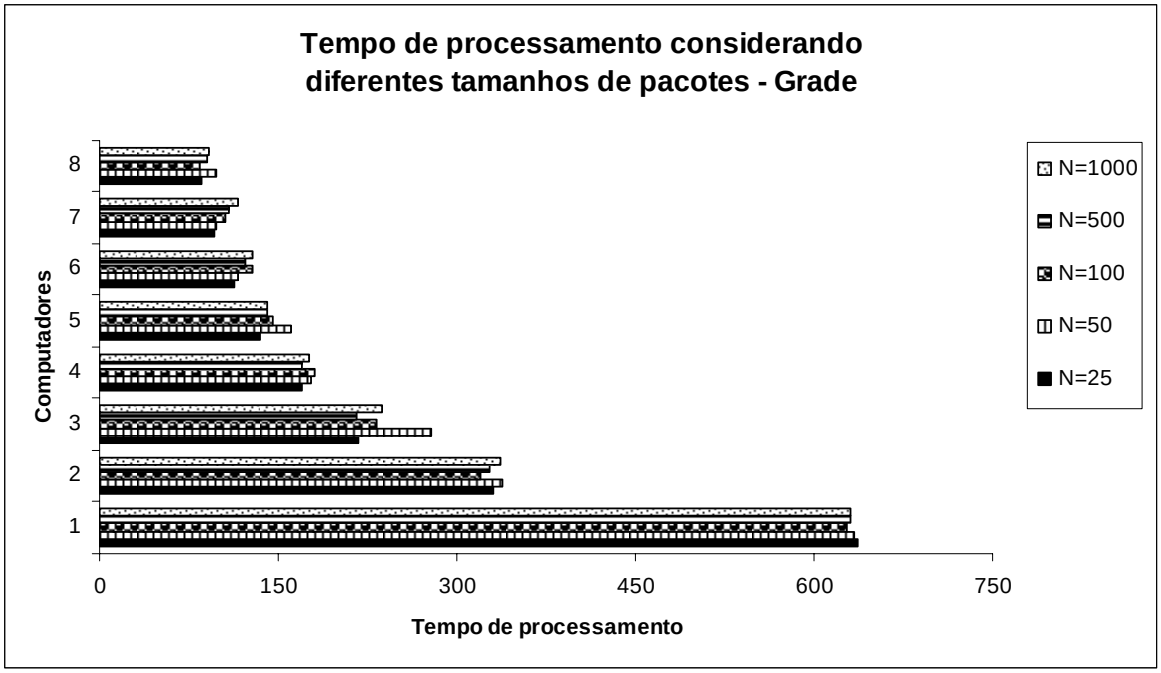


Figura 5.5 – *Diferentes valores para o balanceamento de carga considerando o ambiente de grade.*

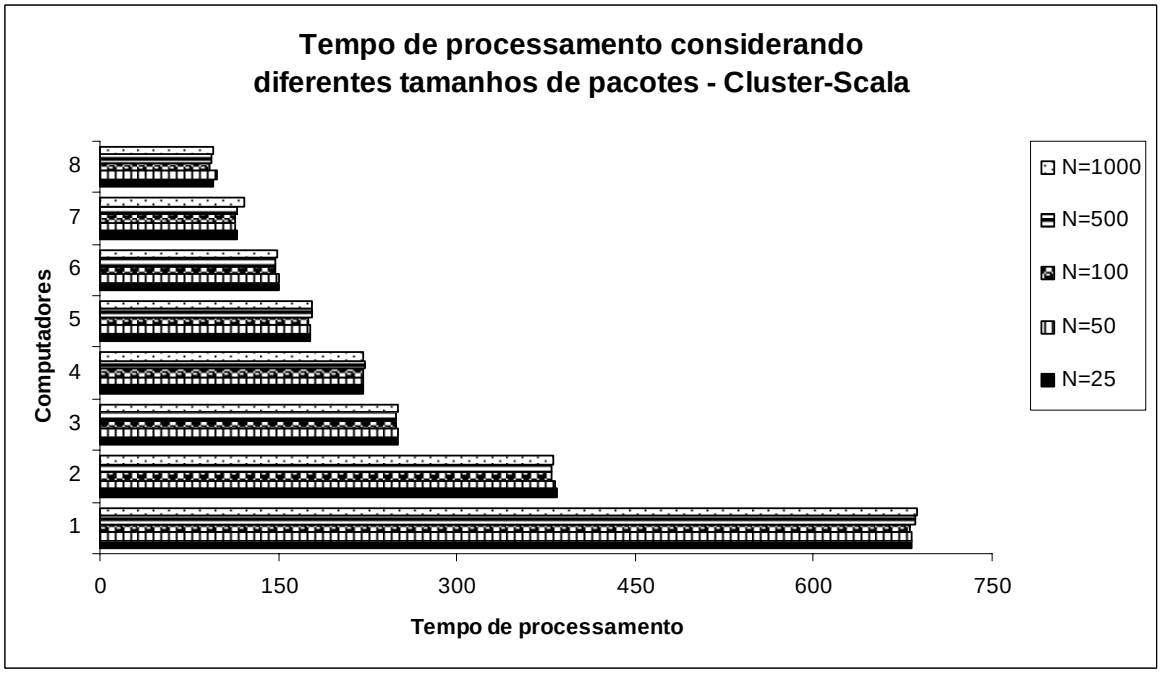


Figura 5.6 – *Diferentes valores para o balanceamento de carga considerando o ambiente de cluster-Scala.*

Com o intuito de verificar o comportamento das plataformas de grade e cluster frente à arquitetura *hyperthreading*, a Tabela 5.4 apresenta os tempos totais de execução das tarefas separadamente para a grade e o cluster, respectivamente. Os dados foram obtidos com a utilização de computadores *hyperthreading* com 1GB de memória RAM, considerando o agendamento de duas tarefas por nó no ambiente de cluster e pacotes com 100 imagens.

Os resultados obtidos na grade são bastante parecidos com os conseguidos no experimento anterior, apresentado na Tabela 5.1, evidenciando, assim, que a utilização de computadores com a arquitetura *hyperthreading* não proporcionam melhor desempenho no processamento das imagens de documentos uma vez que o agendamento de mais de uma tarefa por nó não é permitida. Isso é devido ao fato do software de grade escolhido, o OurGrid, não tratar essa característica das arquiteturas de computadores mais modernas.

Tabela 5.4 *– Tempos totais de execução das tarefas nas configurações de grade e cluster para N=100 imagens, considerando computadores hyperthreading com 1GB de memória RAM. Os tempos estão expressos em minutos.*

Número de computadores processando as tarefas	**Grade**			**Cluster-Scala**	
	Tempo com o mestre não dedicado	**Tempo com o mestre dedicado**	**Tempo com o mestre não dedicado e executando o DVD com vídeo**	**Tempo com o mestre não dedicado**	**Tempo com o mestre dedicado**
1	636	638	825	512	515
2	319	319	426	258	261
3	215	213	295	208	210
4	161	160	255	130	132
5	130	129	205	112	113
6	109	107	155	91	92
7	94	95	139	73	73
8	82	---	121	64	---

Os resultados mostram também que, tanto para a grade quanto para o cluster, dedicar um computador para atuar apenas como mestre, ou não, não impõe diferença nos tempos totais de execução das tarefas. Além disso, no caso da grade, a execução do vídeo juntamente com o processamento das tarefas aumentou o tempo total de execução, na média, em 30%.

O cluster permite o agendamento de duas tarefas por nó através da execução de duas instâncias do *BigBatch Client Module* nos nós clientes. Por esse motivo, quando as tarefas de processamento de imagens de documentos foram executadas no cluster tiveram uma redução de cerca de 25% no seu tempo total de processamento.

Quando as tarefas foram executadas na configuração em cluster, agendando apenas uma tarefa por nó, os resultados também foram bastante similares aos obtidos na configuração *hyperthreading* com 512 MB de memória RAM, indicando que o a quantidade de memória disponível não influenciou no processamento nesse caso.

A Figura 5.7 ilustra graficamente a diferença de desempenho das duas configurações de computadores *hyperthreading*, no ambiente de cluster, considerando o agendamento de duas tarefas por nó.

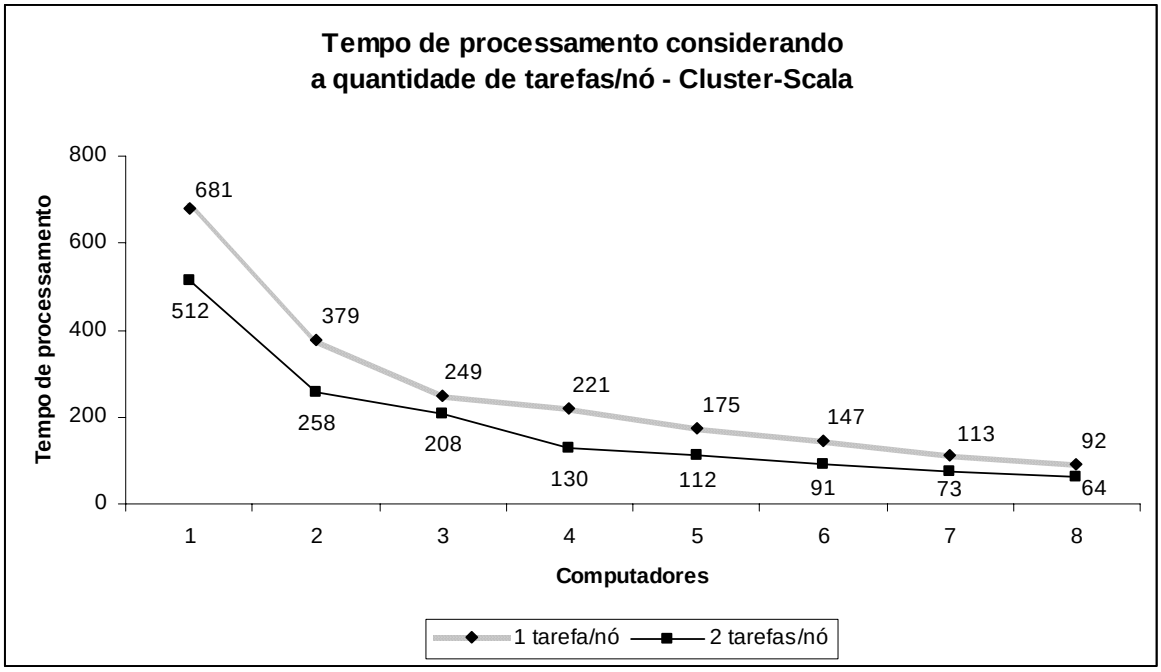


Figura 5.7 – *Tempo de processamento no cluster considerando a tecnologia hyperthreading e a variação de tarefas agendadas por nó (N=100).*

A Figura 5.8 e a Figura 5.9 mostram a curva de *speedup* e a curva de eficiência, respectivamente, para o ambiente de grade e de cluster. Os resultados mostram novamente que os dados têm comportamento próximo ao linear. A curva de eficiência fica bastante próxima do ideal, onde o valor considerado é igual a 1.

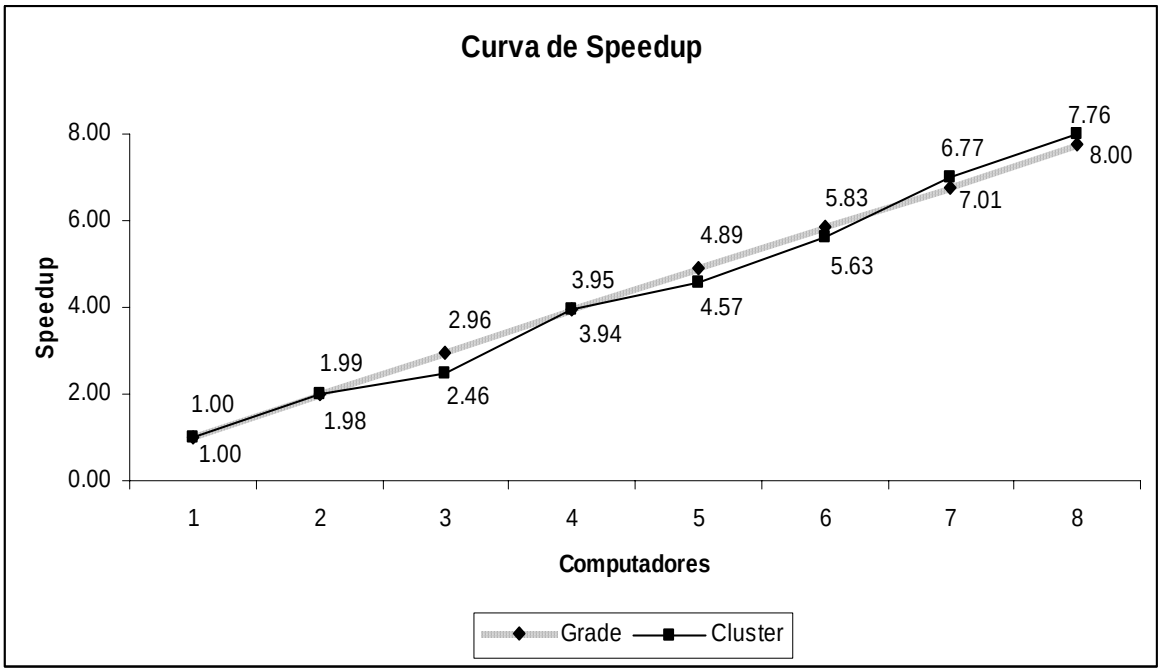


Figura 5.8 – *Curva de Speedup considerando a tecnologia hyperthreading com 1GB de memória RAM.*

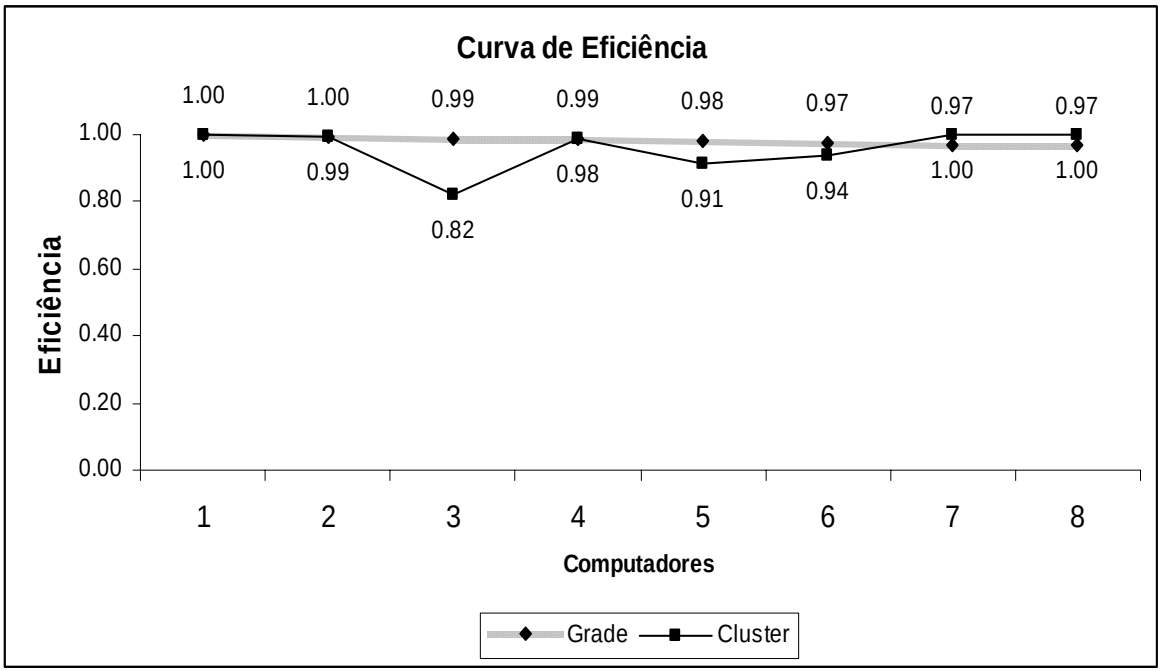


Figura 5.9 – *Curva de Eficiência considerando a tecnologia hyperthreading com 1GB de memória RAM.*

A Tabela 5.5 e a Tabela 5.6 referem-se aos resultados obtidos pela grade e pelo cluster, respectivamente, considerando a utilização da tecnologia *hyperthreading* para o cluster. Os dados

referem-se ainda à variação do número de imagens por pacote quando processadas todas as imagens.

Tabela 5.5 *– Tempos de execução das tarefas na grade com diferentes tamanhos de pacotes, considerando computadores hyperthreading com 1GB de memória RAM.*
Os tempos estão expressos em minutos.

Número de computadores processando as tarefas	**N=25**	**N=50**	**N=100**	**N=500**	**N=1000**
1	646	642	636	633	632
2	326	325	319	320	338
3	218	218	215	216	224
4	164	163	161	167	174
5	131	130	130	133	138
6	109	109	109	116	127
7	94	94	94	104	119
8	82	82	82	81	91

Os resultados não indicam variação significativa no tempo total de processamento das tarefas. Percebe-se uma adequação do balanceamento de carga adotado uma vez que os diferentes tamanhos de pacotes utilizados não causam impacto no tempo total de processamento da base completa de imagens de documentos.

O agrupamento das imagens em pacotes não levou em consideração nenhuma técnica específica, como a utilização do tamanho da imagem ou outra característica própria dela, para que as imagens fossem agrupadas. O agrupamento se fez na ordem em que elas foram armazenadas no disco rígido. Cabe ressaltar que os computadores utilizados são homogêneos contribuindo ainda mais para o balanceamento de carga adequado.

Com os resultados apresentados é possível também dizer que os diferentes tamanhos de pacotes adotados não causaram impacto no tráfego gerado na rede durante o envio e o recebimento dos arquivos necessários ao processamento das tarefas. Portanto, o tráfego de rede, nesse caso, é pequeno. A Figura 5.10 e a Figura 5.11 ilustram graficamente estes resultados obtidos confirmando a variação mínima existente no tempo de execução total das tarefas utilizando diferentes tamanhos de pacotes.

Tabela 5.6 *– Tempos de execução das tarefas no cluster com diferentes tamanhos de pacotes, considerando computadores hyperthreading com 1GB de memória RAM.*
Os tempos estão expressos em minutos.

Número de computadores processando as tarefas	**N=25**	**N=50**	**N=100**	**N=500**	**N=1000**
1	517	515	512	513	514
2	258	258	258	258	260
3	209	207	208	210	210
4	133	132	130	130	131
5	116	114	112	112	113
6	90	90	91	92	93
7	74	73	73	74	74
8	65	64	64	65	66

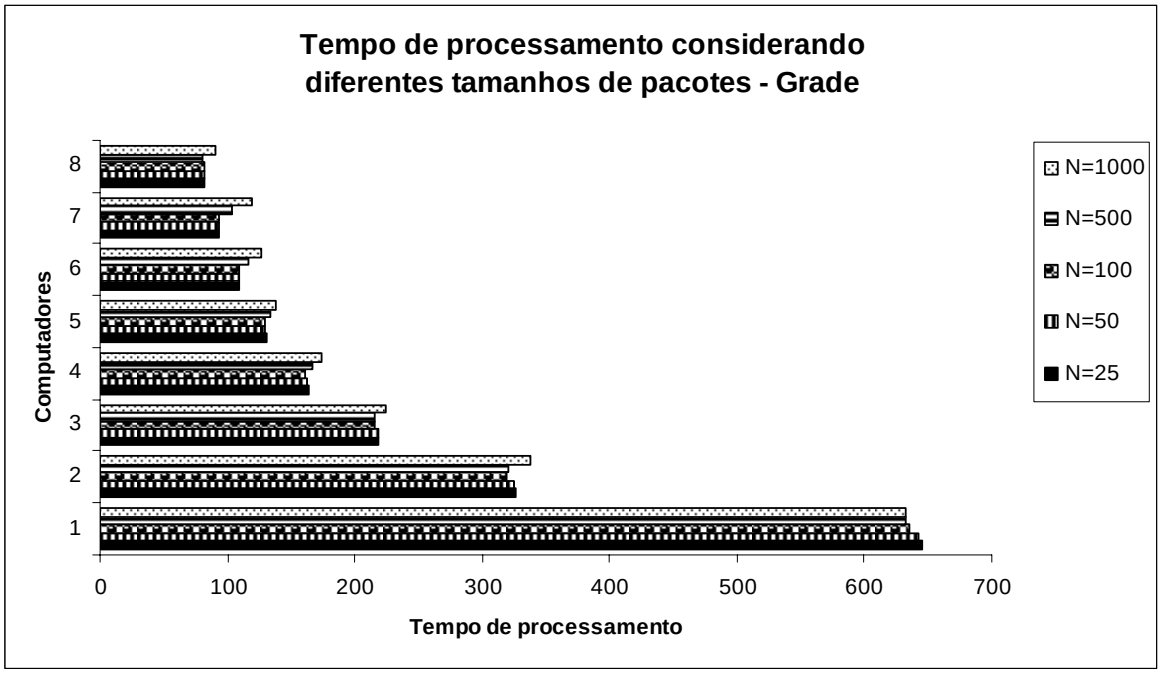


Figura 5.10 *– Tempo de processamento na grade considerando diferentes tamanhos de pacotes nos computadores hyperthreading com 1GB de memória RAM.*

Os resultados da execução do processamento das imagens na configuração em cluster, apresentados na Tabela 5.6, mostram que em relação à configuração em grade houve uma melhora

no tempo de processamento em torno de 20%. Isso se deve ao fato do software de cluster ser capaz de escalonar duas tarefas por nó, atingindo assim tempos melhores. Pode-se dizer ainda que o ganho de desempenho do cluster não foi maior devido à contenção de entrada/saída no acesso ao disco rígido, uma vez que ambas as tarefas no mesmo nó acessam extensivamente o disco rígido para realizar a leitura dos arquivos de imagem de entrada e escrita dos arquivos de imagem de saída.

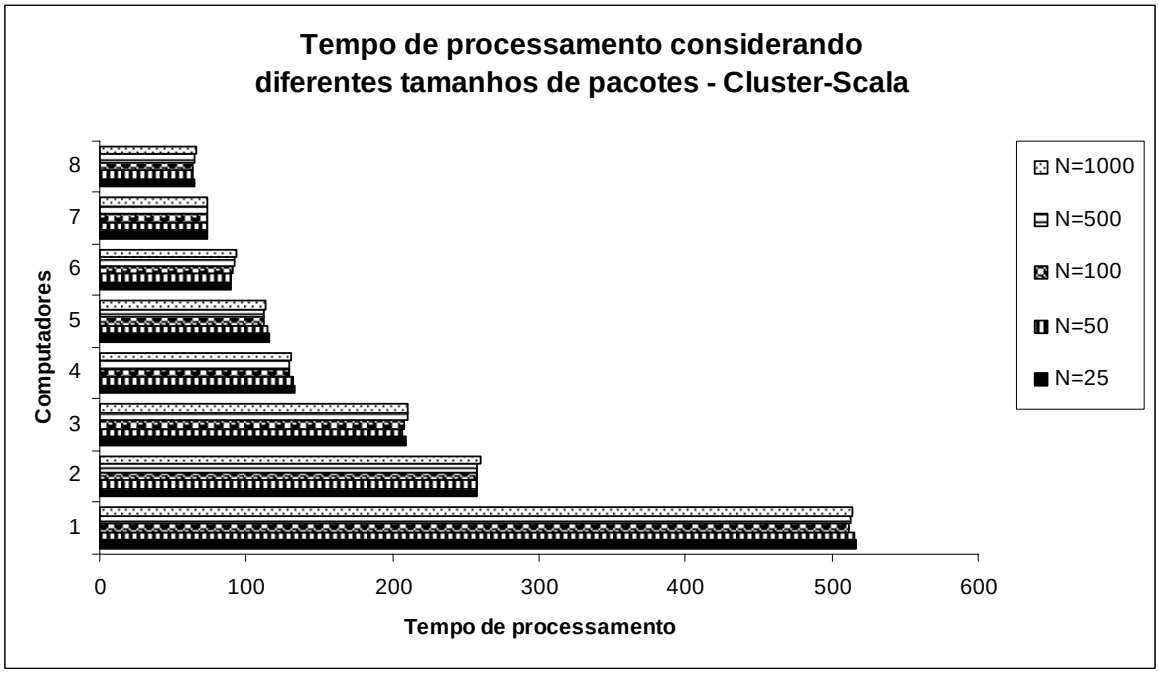


Figura 5.11 – *Tempo de processamento no cluster considerando diferentes tamanhos de pacotes nos computadores hyperthreading com 1GB de memória RAM.*

Vale ressaltar que os computadores utilizados incluem processadores *multithreaded* e dual core, que gerenciam melhor a execução de tarefas simultâneas que os tradicionais computadores *single-threaded*. Assim, espera-se que esses processadores possam tornar o uso das grades mais vantajoso para uma organização não apenas porque as tarefas da grade serão menos afetadas por cargas impostas pelos usuários, mas também as próprias aplicações dos usuários serão menos afetadas pelo fato do seu computador estar executando, ao mesmo tempo, as tarefas da grade. De fato, o uso difundido de multiprocessadores *single-chip* tende a aumentar a quantidade de potência computacional ociosa disponível, a qualquer tempo, em uma organização, e executar tarefas da grade pode ser um uso eficiente desta potência computacional.

A fim de comprovar a vantagem do uso de computadores com multiprocessadores *single-chip*, a Tabela 5.7 apresenta os tempos totais de execução das tarefas separadamente para grade e cluster, respectivamente, para o caso de pacotes com 100 imagens utilizando computadores Core 2

Duo. Os resultados mostram que os tempos obtidos na configuração em grade são muito similares aos obtidos nas configurações anteriores, as quais utilizaram a tecnologia *hyperthreading*. Como mencionado anteriormente, o software de grade escolhido para os experimentos, o OurGrid, não permite o agendamento de mais de uma tarefa por nó. Assim, o fato dos computadores possuírem processadores Core 2 Duo não interfere no tempo total de processamento das imagens de documentos.

***Tabela 5.7** – Tempos totais de execução das tarefas nas configurações de grade e cluster para N=100 imagens, executadas nos computadores Core 2 Duo.*
Os tempos estão expressos em minutos.

Número de computadores processando as tarefas	**Grade**			**Cluster-Scala**	
	Tempo com o mestre não dedicado	**Tempo com o mestre dedicado**	**Tempo com o mestre não dedicado e executando o DVD com vídeo**	**Tempo com o mestre não dedicado**	**Tempo com o mestre dedicado**
1	625	647	698	407	416
2	318	325	350	213	215
3	215	217	304	127	138
4	162	164	229	102	102
5	133	139	140	80	85
6	111	115	121	69	71
7	92	92	103	57	61
8	80	---	94	51	---

Dedicar um computador apenas para gerenciar os ambientes também não contribui para o melhoramento do tempo de processamento, pelo contrário, atrapalha. O fato do computador mestre ajudar no processamento melhora efetivamente a vazão do sistema o que indica que o trabalho de distribuição das tarefas é leve.

A adição de carga computacional extra no ambiente de grade teve seu processamento melhorado em relação aos tempos obtidos nos experimentos anteriores com a tecnologia *hyperthreading*. Isso é devido exatamente ao fato de estar sendo utilizado o processador Core 2 Duo que gerencia melhor a execução paralela das duas aplicações, o processamento de imagens e a execução do vídeo.

Os resultados obtidos pelo cluster foram em média 35% melhores que os obtidos pela grade, pois o cluster permite agendar duas tarefas em cada um dos seus nós, melhorando o desempenho global do sistema. Para mostrar esse ganho no desempenho conseguido pela configuração do cluster-Scala frente às arquiteturas de computadores utilizadas nos experimentos, os resultados obtidos são apresentados na Figura 5.12. São considerados o agendamento de apenas uma tarefa por nó, o agendamento de duas tarefas por nó e também o tempo de processamento nas duas arquiteturas de computadores utilizadas. HT refere-se à tecnologia *hyperthreading* e C2D a Core 2 Duo. Em relação às arquiteturas *hyperthreading* e dual core, utilizadas pelo cluster, o desempenho total do processamento das imagens de documentos foi melhorado em 23% na média.

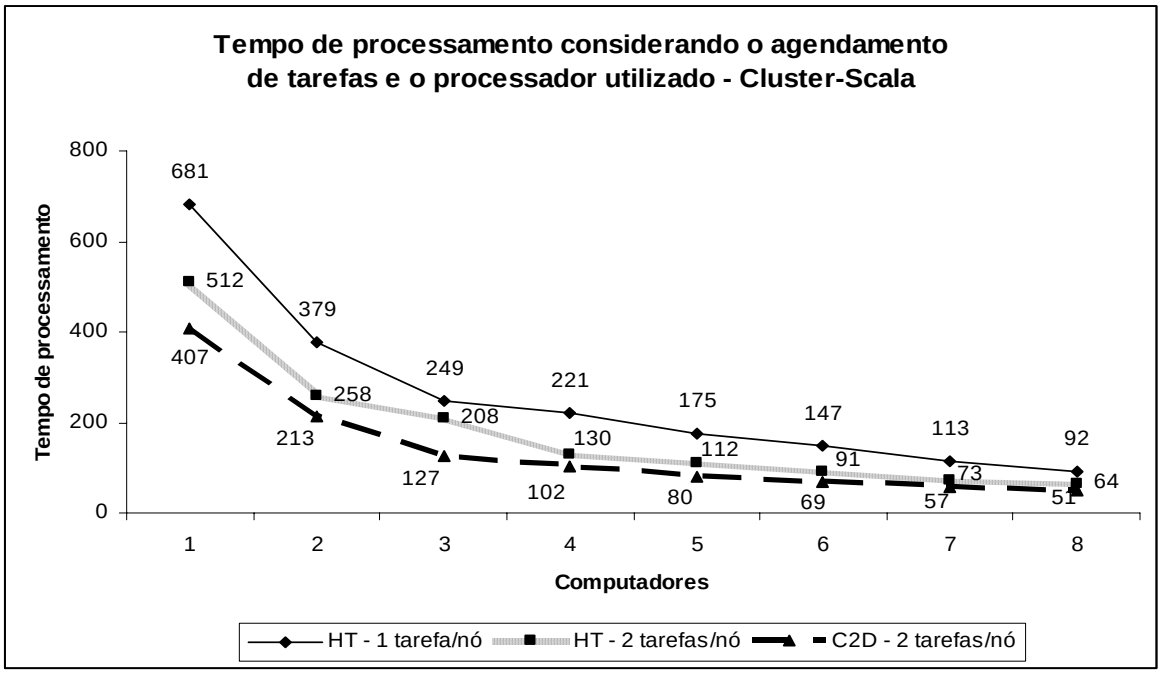


Figura 5.12 – *Desempenho obtido no cluster considerando as arquiteturas hyperthreading e dual core, além do escalonamento diferenciado de tarefas por nó.*

A Figura 5.13 e a Figura 5.14 ilustram mais uma vez a curva de *speedup* e a curva de eficiência, respectivamente, para os ambientes de grade e cluster considerando a utilização dos computadores com tecnologia Core 2 Duo. Os resultados confirmam novamente a proximidade linear dos dados além da curva de eficiência apresentar valor maiores que 1. Em um sistema paralelo ideal a eficiência é igual a 1.

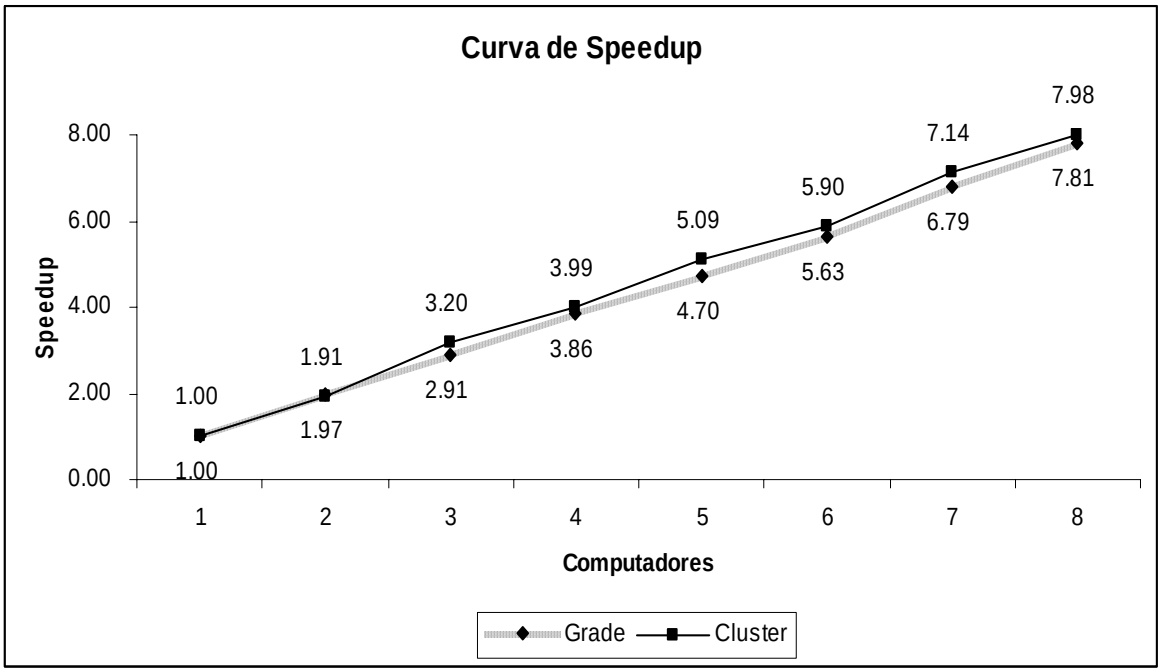


Figura 5.13 – *Curva de Speedup considerando a tecnologia Core 2 Duo com 2GB de memória RAM*

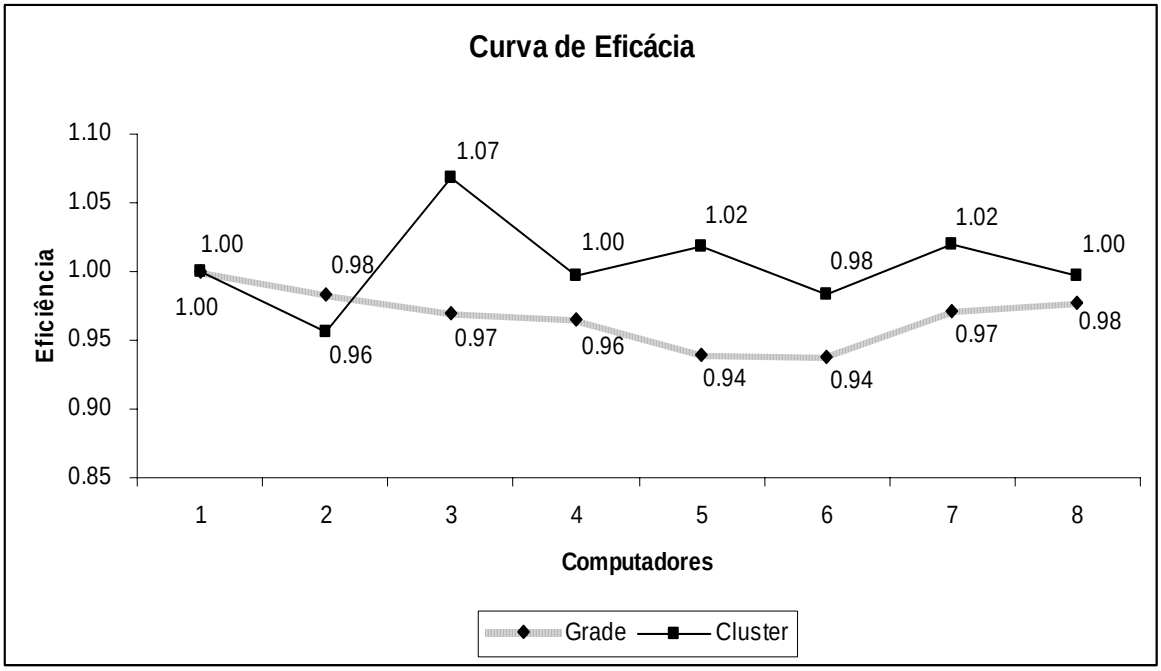


Figura 5.14 – *Curva de Eficiência considerando a tecnologia Core 2 Duo com 2GB de memória RAM*

A Tabela 5.8 e a Tabela 5.9 referem-se aos resultados obtidos pela grade e pelo cluster, respectivamente, nos computadores Core 2 Duo, considerando a variação no tamanho do pacote quando processadas todas as imagens. Os resultados indicam mais uma vez que a distribuição de carga é adequada, não sendo afetada pelos tamanhos dos pacotes, ou seja, não influencia no tempo total de execução das tarefas. Além disso, os resultados também sugerem que o tráfego de rede é baixo, uma vez que não existe variação nos tempos de execução com os diferentes tamanhos de pacotes utilizados.

Os resultados da execução do processamento das imagens na configuração em cluster, apresentados na Tabela 5.9, indicam uma melhora de 35%, em média, no tempo total de processamento quando comparada à configuração da grade. Novamente, esta melhora no desempenho se deve ao fato do cluster ser capaz de aproveitar os benefícios da tecnologia Core 2 Duo, com dois processadores, escalonando duas tarefas por nó e atingindo resultados melhores.

Tabela 5.8 *– Tempos de execução das tarefas na grade com diferentes tamanhos de pacotes, considerando computadores Core 2 Duo. Os tempos estão expressos em minutos.*

Número de computadores processando as tarefas	**N=25**	**N=50**	**N=100**	**N=500**	**N=1000**
1	635	631	625	624	624
2	324	322	318	320	330
3	218	218	215	218	220
4	170	164	162	169	175
5	129	130	133	129	131
6	107	108	111	110	111
7	91	93	92	95	97
8	80	80	80	80	82

A Figura 5.15 ilustra as variações nos tempos de processamento das imagens para as três configurações de computadores utilizadas, considerando o ambiente de grade. A Figura 5.16 refere-se às variações nos tempos de processamento das imagens no cluster considerando as três configurações de computadores utilizadas. Os dados indicam os tempos obtidos quando da simulação com pacotes de tamanho N=100. HT1 refere-se aos computadores *hyperthreading* com 512GB de RAM, HT2 *hyperthreading* com 1GB de RAM e C2D Core 2 Duo com 2GB de RAM.

Tabela 5.9 *– Tempos de execução das tarefas no cluster com diferentes tamanhos de pacotes, considerando computadores Core 2 Duo. Os tempos estão expressos em minutos.*

Número de computadores processando as tarefas	**N=25**	**N=50**	**N=100**	**N=500**	**N=1000**
1	413	409	407	406	406
2	219	215	213	213	215
3	137	136	137	137	139
4	98	100	102	104	107
5	84	82	80	82	85
6	72	68	69	67	72
7	59	58	57	59	61
8	50	49	51	52	50

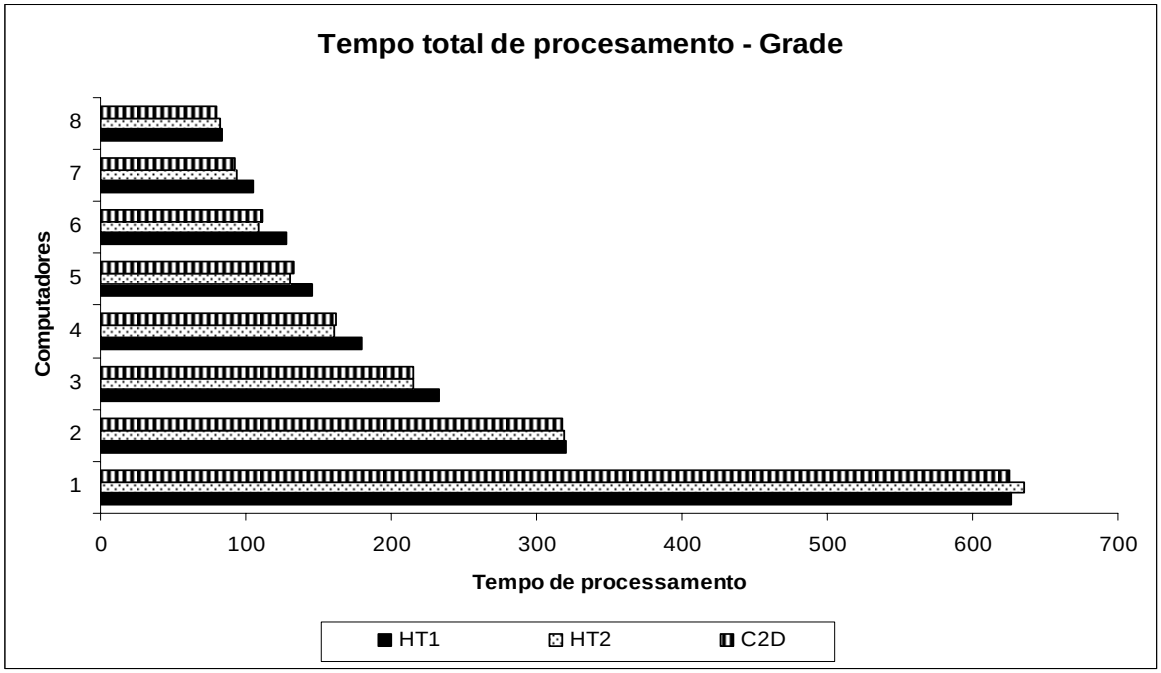


Figura 5.15 – *Tempo de processamento na grade considerando computadores hyperthreading e Dual Core.*

Outra observação a ser feita, a respeito do desempenho de clusters e grades, leva em consideração os protocolos de rede utilizados. Ambas as configurações usaram passagem de mensagem sobre a pilha de protocolos TCP/IP, enquanto as aplicações de cluster geralmente

utilizam implementações MPI que resultam em *overheads* de rede mais baixos. Isto foi considerado em outro experimento detalhado em capítulo mais adiante.

Outra forma de tornar o cluster mais eficiente é realizar o balanceamento de carga estaticamente, podendo levar vantagem do fato que a configuração da rede disponível para o cluster é conhecida antecipadamente.

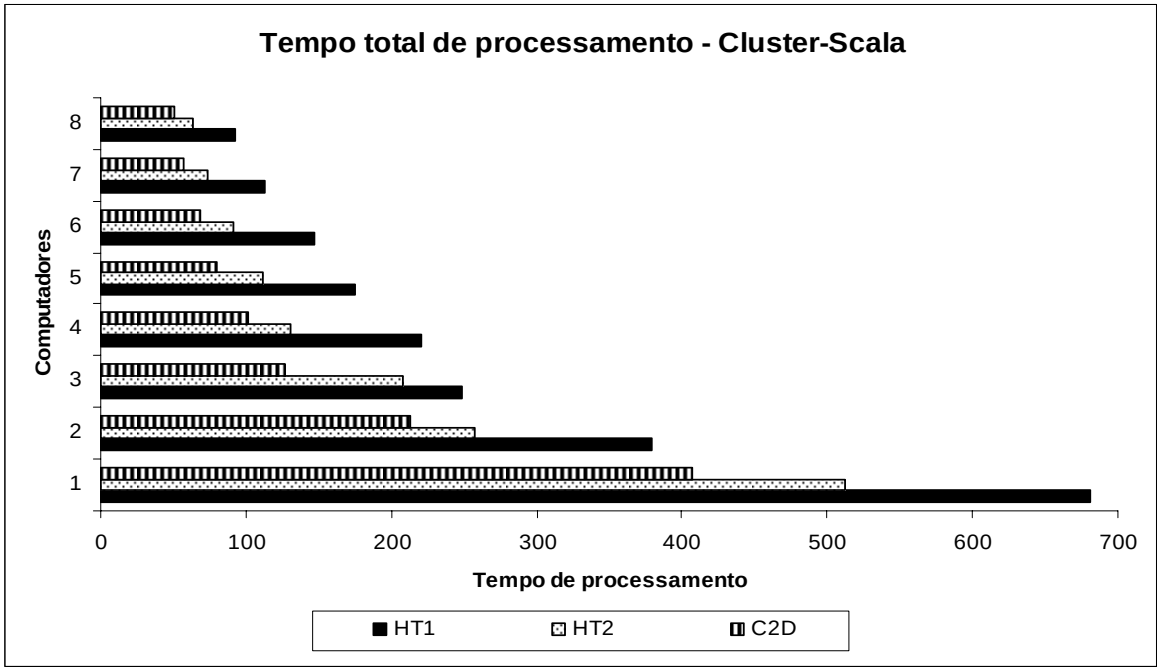


Figura 5.16 – *Tempo de processamento no cluster considerando computadores hyperthreading e Dual Core.*

5.2 Cluster-Scala versus Cluster-MPI

Comumente as aplicações executadas em ambientes de cluster são escritas explicitamente para ele levando em consideração a configuração do cluster a ser utilizado além das especificidades do problema a ser resolvido. Para isso, existe uma série de ferramentas, bibliotecas e linguagens que auxiliam na codificação. Dois exemplos são a biblioteca MPI (*Message Passing Interface* – Interface de Passagem de Mensagem) [57] e a linguagem de programação Scala [84].

A Scala é uma linguagem de programação funcional e orientada a objetos projetada para trabalhar com a plataforma Java, alavancando sua portabilidade e disponibilidade de outras bibliotecas. Possui ainda bom suporte para a programação distribuída utilizando *Actors* [89] para fazer a comunicação entre os nós do cluster.

O MPI tornou-se um padrão para comunicação de dados em computação paralela permitindo que informações sejam passadas entre os vários processadores ou nós de um cluster. MPI é

composta por um conjunto de sub-rotinas padronizadas de comunicação, desenvolvidas em linguagem de programação C que são utilizadas no desenvolvimento de programas paralelos. Pode ser utilizado em programas escritos nas linguagens de programação FORTRAN, C ou C++.

Várias versões do padrão MPI estão disponíveis para utilização como MPICH-1, MPICH-2 e MS-MPI. A versão MS-MPI foi primeiramente disponibilizada pela Microsoft através do Windows® Compute Cluster Server 2003 e atualmente através do Windows® HPC Server 2008, uma solução para computação de alto desempenho.

O cluster-MPI desenvolvido utiliza o ambiente de execução disponível no Windows® HPC Server 2008. O experimento com o cluster-MPI consistiu no processamento das imagens de documentos utilizando a ferramenta BigBatch. A distribuição das tarefas foi realizada através de um programa escrito em linguagem de programação C que utiliza chamadas MPI para coordenar e se comunicar com os nós.

O sistema operacional utilizado foi o Microsoft HPC Server 2008. Devido aos requisitos de hardware exigidos por esse sistema operacional, foi possível utilizar apenas os computadores Intel Core 2 Duo com clock de 2.66 GHz, 2 GB de memória RAM DDR2 com cache L1 de 64 KB e L2 de 4096 KB, HD SATA II de 250 GB, conectados em rede local Ethernet de 100Mbit/s através de um switch Ethernet 3COM, modelo OfficeConnect, com 08 portas mais uma porta de Uplink. A topologia de rede utilizada é aquela apresentada na Figura 4.15.

A ferramenta BigBatch e os pacotes com as imagens a serem processadas foram armazenados em um único computador, o nó cabeça. Os demais nós do cluster acessaram os arquivos através de um diretório compartilhado entre eles. Os pacotes foram disponibilizados em tamanhos diferentes, 25, 50, 100, 500 e 1000, de acordo com a quantidade de imagens a serem processadas em cada um deles. Em cada uma das condições observadas, o número de nós computacionais utilizados no cluster foram variados a fim de determinar a variação no tempo de execução da aplicação em relação ao número de nós disponíveis. A quantidade de computadores variou de um a oito.

A Tabela 5.10 apresenta os resultados da execução do processamento de imagens de documentos no cluster-MPI de acordo com a variação no tamanho do pacote utilizado. Duas tarefas por nó foram agendadas aproveitando de fato as potencialidades da tecnologia dual core.

Os resultados mostram que o tempo de processamento diminui à medida que novos nós do cluster ajudam no processamento das tarefas, confirmando que o paralelismo é adequado ao problema de processamento de imagens de documentos. Os resultados indicam ainda pouca variação no tempo de execução quando comparados entre eles, considerando os diferentes tamanhos e pacotes. Ou seja, o fato de utilizar cargas diferentes a serem processadas não influencia no resultado total do tempo de processamento. A distribuição de carga nesse caso é eficiente, além disso, os nós do cluster são homogêneos. Pode-se afirmar ainda que o tráfego de rede é baixo, uma

vez que não existe variação nos tempos de execução com diferentes tamanhos de pacotes. Essa afirmação é mostrada mais adiante em outra seção. A Figura 5.17 ilustra graficamente o desempenho do cluster-MPI frente aos diferentes tamanhos de pacotes.

Tabela 5.10 *– Tempos de execução das tarefas no cluster-MPI com diferentes tamanhos de pacotes, considerando computadores Core 2 Duo. Os tempos estão expressos em minutos.*

Número de computadores processando as tarefas	**N=25**	**N=50**	**N=100**	**N=500**	**N=1000**
1	253	252	251	253	253
2	111	111	112	113	113
3	78	78	78	79	77
4	63	63	64	59	58
5	51	49	50	51	52
6	42	42	42	41	40
7	32	36	36	32	36
8	30	28	25	25	29

Com esses resultados é possível comparar o desempenho do cluster-Scala com o cluster-MPI em relação às tarefas de processamento de imagens de documentos. A Figura 5.18 mostra graficamente os resultados obtidos nas duas configurações, cluster-Scala e cluster-MPI, quando executados os pacotes de tamanho N=100 imagens. É possível observar uma redução nos tempos de processamento do cluster-MPI de aproximadamente 40% em relação ao cluster-Scala. Em alguns casos essa redução chega a 50%.

A Figura 5.19 mostra os tempos totais de processamento considerando os ambientes de grade, cluster-Scala e cluster-MPI quando processados os pacotes de tamanho N=100. Em relação ao cluster-Scala os resultado foram reduzidos em 35% quando comparados aos obtidos pela grade. Já o cluster-MPI teve uma redução média de 62% em relação à grade. Essas reduções são explicadas principalmente pelo fato de que o OurGrid não permite que sejam escalonadas mais de uma tarefa por nó, ao contrário do cluster. Assim, a utilização de computadores com processador dual core não faz diferença para a grade. A nova versão do OurGrid ainda não faz distinção entre computadores com mais de um processador.

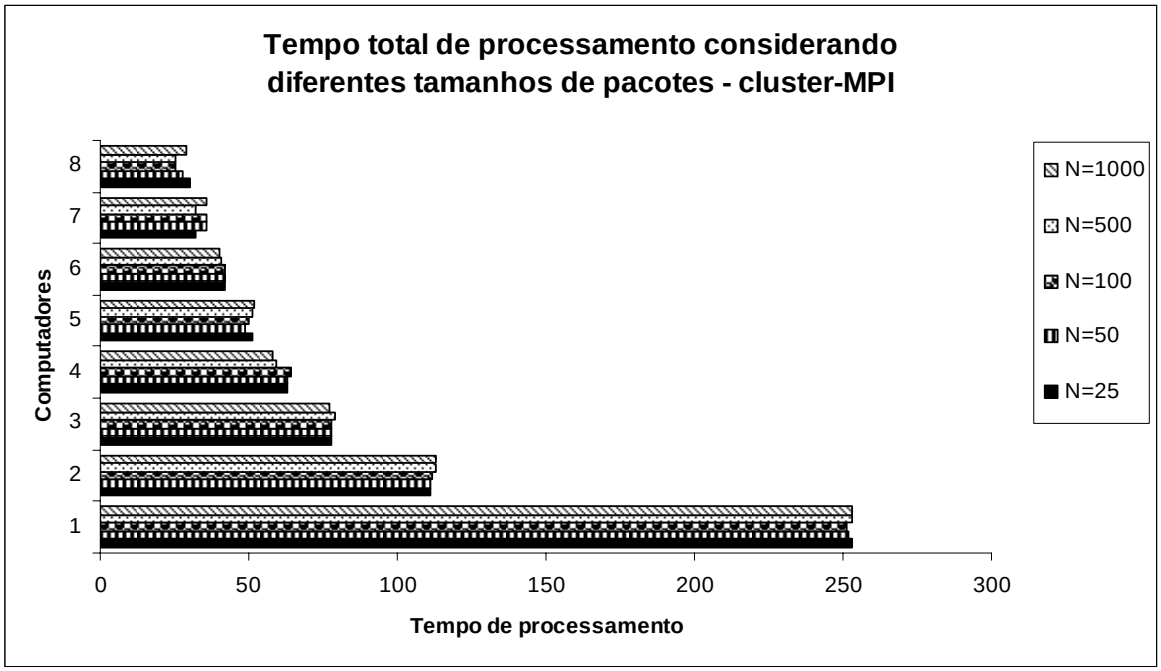


Figura 5.17 – *Tempos de processamento obtidos no cluster-MPI frente aos diferentes tamanhos de pacotes.*

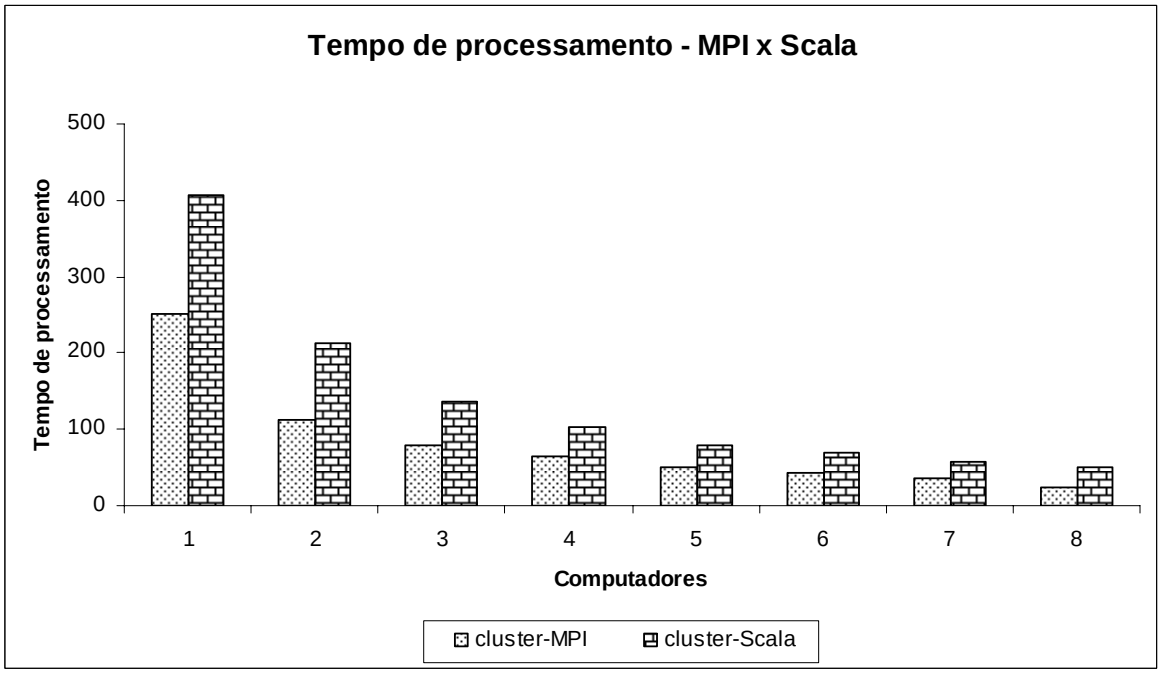


Figura 5.18 – *Cluster-MPI x cluster-Scala considerando pacotes de tamanho N=100 imagens.*

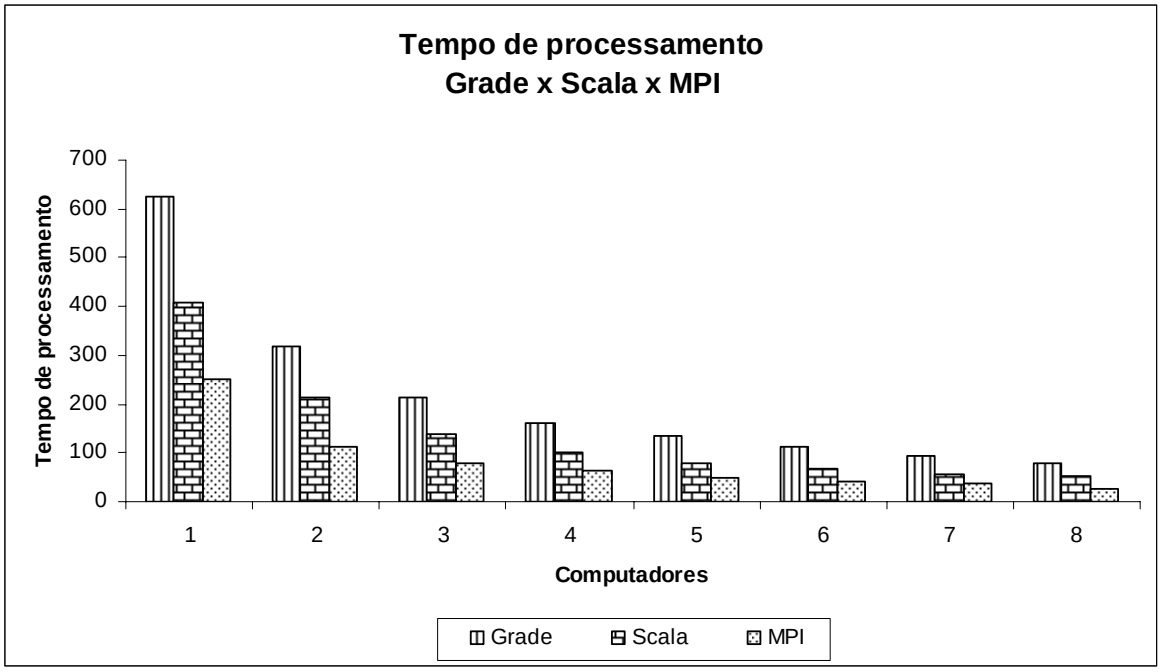


Figura 5.19 – *Grade x Cluster-Scala x Cluster-MPI considerando pacotes de tamanho N=100 imagens.*

5.3 Considerações Sobre a Utilização dos Ambientes de Grade e Clusters

Os clusters estão geralmente organizados em redes locais, com recursos dedicados e homogêneos, enquanto as grades são utilizadas sobre a Internet, não dedicadas e heterogêneas. Essa maneira de dividir as duas arquiteturas permitiu que fosse associada à idéia que o custo computacional de uma grade seria maior do que um cluster, devido a vários fatores como a latência na comunicação, o roteamento inerentemente mais complexo, a complexidade e a robustez do software, etc. Por isso, escolher entre uma ou outra arquitetura não é uma tarefa trivial visto que ainda não há estudos disponíveis consolidados que indiquem os altos custos da utilização da arquitetura grade em relação à arquitetura cluster.

O desempenho é o principal fator de comparação que os usuários levam em consideração, no entanto existem outros aspectos a serem considerados, e muitas vezes não mensuráveis. Pode-se destacar primeiramente a facilidade de instalação e configuração do ambiente. O cluster-Scala exige a instalação da solução BigBatch no nó mestre e o Módulo BigBatch Cliente nos nós clientes. A grade exige que sejam instalados no nó mestre a solução BigBatch e os componentes MyGrid e Peer do OurGrid e o componente UserAgent nos nós clientes. Como ambas as soluções são baseadas na plataforma Java é preciso instalar o módulo *Java Runtime* em todos os nós envolvidos na execução das tarefas. Na configuração do cluster-MPI é preciso ter a biblioteca MPI instalada em todos os nós, com acesso a um diretório compartilhado da rede. O mais importante aqui é o fato

de que a grade depende do software utilizado, o OurGrid, em comparação com o cluster. Isso pode gerar custos elevados de manutenção em longo prazo, devido a necessidade de mão-de-obra especializada para a manutenção do sistema de grade.

Outro ponto a ser destacado é a escalabilidade do sistema. Pela forma como as grades foram concebidas, elas levam vantagem nesse ponto, pois foram idealizadas tendo isso em mente. É possível incluir nós a partir de diferentes domínios da mesma organização, ou mesmo obter nós de outros domínios na Internet, disponibilizando alguns nós internos da organização para que usuários externos da grade também possam utilizá-los nos seus processamentos. No caso de utilizar domínios externos, o tempo de processamento total das tarefas pode ser prejudicado pela heterogeneidade dos nós e também pelo tráfego de rede, podendo tornar o tempo de transmissão maior que o tempo de processamento, degradando o desempenho geral. Mas para tal afirmação é preciso uma investigação mais aprofundada.

A possibilidade de utilizar os nós clientes na realização de outras tarefas simultaneamente é outro ponto a ser considerado. Essa é uma característica própria das grades, já o cluster costuma ter seus nós dedicados ao processamento de uma tarefa. Porém, o Módulo BigBatch Cliente não faz restrições sobre a execução de outros aplicativos conjuntamente no mesmo computador.

A grade permite ainda que o processamento das tarefas seja executado apenas quando os nós estiverem ociosos e não estão sendo utilizados por uma pessoa – atualmente a ociosidade é detectada através da execução de uma proteção de tela. Esse ponto, juntamente com a facilidade de incluir outros nós na grade, faz com que a grade reúna um número muito maior de nós clientes do que o cluster. No entanto, é importante destacar o fato de que o envio de documentos para serem processados em nós fora da organização pode acarretar em um problema de privacidade e deve ser fortemente considerado.

Os resultados apresentados nesse capítulo permitem que sejam feitas algumas afirmações quanto à utilização de uma arquitetura em relação à outra. Quando utilizados computadores com um único processador, os resultados obtidos nas duas arquiteturas são bastante semelhantes. Embora os resultados obtidos na grade sejam um pouco melhores, essas diferenças não são significativas. Porém, quando utilizados computadores com mais de um processador, onde o cluster permite o agendamento de mais de uma tarefa por nós, aí sim as diferenças entre os resultados são expressivas. Os resultados obtidos no cluster-Scala são em média 35% melhores que os obtidos na grade, lembrando que o OurGrid não permite que sejam agendadas mais de uma tarefa por nó. Além disso, a utilização do padrão MPI no ambiente de cluster apresentou o melhor desempenho no processamento das imagens de documento. A melhora foi em torno de 40%. A tecnologia MPI é um padrão de fato eficiente para o processamento de alto desempenho.

5.4 Trabalhos Relacionados

Em geral, a escolha de uma ou outra arquitetura depende das características intrínsecas do problema e dos computadores disponíveis; os clusters são normalmente organizados em redes locais, com recursos dedicados e homogêneos, enquanto as grades geralmente são utilizadas em WANs e na Internet, redes não dedicadas e heterogêneas. Tal maneira de dividir as arquiteturas trouxe associada à preconceituosa idéia que o custo computacional de uma grade seria maior do que um cluster, devido à latência na comunicação, o roteamento inerentemente mais complexo, etc. Entretanto, até o presente trabalho não foi encontrada na literatura qualquer referência que realmente justificasse tal preconceito e mesmo quantificasse os *overheads* do uso da arquitetura grade em relação a cluster.

A seguir estão relatados dois trabalhos que comparam o desempenho das plataformas de cluster e grade, porém a grade utilizada é heterogênea e executada sobre a Internet.

Em [90] são apresentados dados do processamento de grandes volumes de imagens hiperespectral obtidas pelo projeto APEX – *Airborne Prism EXperiment* [91] nos ambientes de cluster e grade. São executados dois módulos de processamento hiperespectral em dois ambientes diferentes de computação distribuída, um cluster Beowulf homogêneo e uma infraestrutura heterogênea de grade própria.

No cluster, o modelo de programação utilizado é baseado no padrão MPI e um único programa é executado em todos os nós simultaneamente. As tarefas são altamente independentes, a baixa latência de comunicação entre os nós não é necessária, mas a alta largura de banda de rede sim uma vez que 3MB de dados são devolvidos no processamento. Os resultados mostram que uma tarefa em um cluster com 8 nós esperou cerca de 15 minutos na fila para ser executada em 20 minutos. A mesma tarefa, submetida em um cluster com 64 nós, esperou por mais de 14 dias apesar de que provavelmente seria executada em menos de 5 minutos. Segundo os autores o tempo de execução de uma tarefa em um cluster geralmente está entre os melhores, pois o cluster geralmente possui nós atualizados, bons sistemas de I/O, sistemas de comunicação com maior largura de banda/baixa latência e melhores compiladores e bibliotecas de desenvolvimento que normalmente não estão disponíveis em outra plataforma. Se uma tarefa é relativamente grande e exige grande comunicação entre os nós, a execução em um cluster pode ser a única possibilidade viável. No entanto, devido à competição por um recurso compartilhado com alta demanda, o acesso consistente e confiável ao cluster pode não ser garantido.

A grade utilizada no protótipo, Grid APEX Framework, é composta de máquinas de um departamento da universidade, uma empresa média de desenvolvimento de software e uma rede privada doméstica. A grade utiliza o protocolo HTTP (*Hypertext Transfer Protocol* – Protocolo de Transferência de Hipertexto) para fazer a comunicação entre as instituições. Há um escalonador de tarefas alimentando os nós da grade através de pedidos HTTP no lado do cliente que,

periodicamente, requisitam as tarefas ao servidor. O componente cliente é um pequeno programa que faz uma solicitação HTTP para um servidor *web* pedindo por uma tarefa, executa o processamento e, em seguida, envia de volta os resultados. Quando o servidor *web* recebe os resultados, ele encaminha para o escalonador de tarefas que mantém as estatísticas. Uma vez que as tarefas são independentes, não há comunicação entre os nós e, portanto, as tarefas são executadas em um nó umas após as outras de maneira assíncrona para as tarefas sendo executadas em outros nós (ao contrário do cluster). Em uma rede heterogênea, esta é uma questão vital uma vez que o nó mais rápido na rede pode executar mais tarefas que o nó mais lento. Os resultados apresentados são referentes ao processamento em uma grade com 82 nós onde uma pequena fração de nós realizou a maior quantidade do trabalho. A adição de nós lentos não melhora o tempo de execução e pode degradar o tempo de processamento total. Devido ao tipo de tarefa utilizada, há uma penalidade no tempo de execução ao ser usada uma máquina remota, em comparação com uma máquina local.

Assim, os autores [90] concluem que cada ambiente tem vantagens e desvantagens sendo ambos viáveis. Eles conseguiram uma diminuição no tempo de processamento de cerca de metade de um dia para aproximadamente meia hora. Devido ao sistema de filas utilizado no cluster disponível, a utilização de 8 nós fornece melhor eficiência devido ao longo tempo de espera na fila quando utilizados muitos nós no cluster, embora 522 nós estivessem disponíveis. Coincidentemente, um número relativamente baixo de nós na grade, 12, também ofereceu a melhor eficiência devido à velocidade relativa desses nodos quando comparada à velocidade dos 70 nós, bem mais lentos, utilizado.

Uma extensão do trabalho [90] descrito anteriormente é apresentada em [92]. De acordo com os autores, uma dificuldade na geração de dados da observação da Terra utilizando imagens aéreas ou obtidas por satélites é a influência de dados atmosféricos nas imagens coletadas. Para auxiliar no processamento é utilizados um programa científico de modelagem chamado MODTRAN e que se tornou de fato uma base para tal processamento. Outro software amplamente utilizado na correção de parâmetros atmosféricos é o ATCOR que emprega uma estratégia de pré-processamento de grandes volumes de dados que servem de base de dados para o MODTRAN. O cálculo desse pré-processamento tem levado semanas para ser realizado e a fim de diminuir esse tempo de processamento os autores utilizam as plataformas de cluster e grade para acelerar a produção dos dados, variando o número de computadores em potência de dois até 64. A plataforma de cluster utilizada é o Matterhorn [93], constituído com 522 computadores AMD Opteron 244 (1,8 MHz) conectados por meio de Ethernet 100Mbit/s ou Myrinet. A plataforma de grade utilizada é o Condor, composto por 99 computadores com as configurações Intel x86 compatível, AMD x64, ambas com o sistema operacional Linux, Apple G5 com o sistema operacional MacOSX, e Sparc Solaris.

Os resultados apresentados mostram que a grade fornece tempos de resposta mais rápidos do que o cluster embora esse seja mais bem equipado. Ambas as implementações de grade e cluster podem ser usadas para reduzir o tempo de execução de dez dias em uma única CPU, para menos de dois dias usando recursos modestos no processamento. Ao medir apenas o tempo de processamento no cluster observa-se um comportamento linear dos resultados. Já na grade que utiliza o HTTP sobre a Internet, a escalabilidade do tempo de execução não é linear, no entanto o custo adicional de I/O pode ser limitado pelo bom uso do software de servidor *web* e a utilização de compressão de dados em grandes arquivos de entrada e/ou saída.

6. CONSIDERAÇÕES SOBRE O TRÁFEGO DE REDE

Wireshark [94] é um software analisador de protocolos de rede bastante utilizado que permite monitorar, controlar os dados que trafegam na rede e analisar os pacotes recebidos e transmitidos por qualquer interface de rede. A análise dos pacotes capturados pode ser feita em tempo real ou através de um arquivo de captura, armazenado no disco rígido, o qual permite visualizar o conteúdo e informações detalhadas de cada pacote. Há uma centena de protocolos (TCP, UDP, ICMP, etc.) que podem ser filtrados pelo *Wireshark* a fim de facilitar a análise. É um software de código aberto e em versões anteriores era chamado de *Ethereal*. Pode ser executado tanto no ambiente Linux quanto no ambiente Windows.

O Gerenciador de Tarefas do Windows é um utilitário disponibilizado juntamente com o sistema operacional, que mostra informações atualizadas do sistema como os aplicativos que estão em execução, o uso do processador e da memória, a utilização da rede e etc. A guia Rede exibe uma representação gráfica do desempenho da rede, fornecendo informações estatísticas como a utilização da rede, velocidade, taxa de transmissão, status entre outros.

O Gerenciador do Cluster HPC [95] permite coletar e exibir informações sobre o cluster através de gráficos e relatórios que mostram estatísticas da utilização da CPU, do disco rígido e da rede. Esses gráficos e relatórios possibilitam acompanhar a utilização dos recursos além de ajudar no diagnóstico de possíveis pontos de estrangulamento dos recursos.

Por suas funcionalidades, o *Wireshark* foi utilizado como software de análise do tráfego de rede. O Gerenciador de Tarefas do Windows e o Gerenciador do Cluster HPC foram usados para acompanhar graficamente, em tempo real, a utilização da rede durante o processamento das imagens de documentos com a ferramenta BigBatch.

Dois protocolos recebem atenção especial na análise feita porque foram aqueles que mais apareceram nos pacotes coletados pelo *Wireshark* durante a execução das tarefas de processamento de imagens. São eles o SMB e o TCP. O protocolo SMB (*Server Message Block* – Bloco de Mensagem do Servidor) é um protocolo padrão de Internet usado pelo Windows para compartilhar arquivos, impressoras, portas seriais e para a comunicação com outros computadores. Em um ambiente de rede, os servidores disponibilizam os sistemas de arquivos e recursos para os clientes, os clientes fazem solicitações SMB para os recursos e os servidores fornecem respostas SMB. Por isso o SMB é caracterizado como um protocolo de solicitações e respostas cliente-servidor. Ele utiliza o protocolo TCP como protocolo de transporte.

O protocolo TCP (*Transmission Control Protocol* – Protocolo de Controle de Transmissão) tem como principal objetivo realizar a comunicação entre aplicações de dois computadores diferentes. O TCP garante que todos os pacotes serão enviados com sucesso, pois realiza transmissões orientadas à conexão, a aplicação envia um pedido de conexão para o destino, usa a conexão para transferir os dados e ao final da transmissão a conexão é encerrada.

De acordo com o tempo de execução do processamento das tarefas nos ambientes de grade e cluster, o tempo de captura dos pacotes na grade, considerando os oito computadores disponíveis, ficou em torno de 01 hora e 30 minutos, no cluster-Scala esse tempo foi diminuído para 01 hora, considerando o agendamento de duas tarefas por nó e no cluster MPI 30 minutos em média.

Devido à quantidade de pacotes capturados durante a execução do processamento das imagens, os pacotes foram divididos em vários arquivos e posteriormente analisados separadamente. Assim, os resultados apresentados referem-se à média dos valores.

6.1 Grade

A Figura 6.1 apresenta os resultados sobre o tráfego de rede gerado quando da execução do processamento dos pacotes de tamanho 25 imagens no ambiente de grade. O tráfego gerado é 99% pertencente ao protocolo TCP e 1% a outros protocolos. Embora existam muitos pacotes de tamanhos variados sendo transmitidos, a taxa de transmissão na rede foi considerada baixa, conforme ilustra a Figura 6.2, em média 8MBit/s.

Ao processar os pacotes de tamanho 100 imagens, percebe-se que não há uma uniformidade no tráfego gerado na rede. Há períodos de atividade intensa na rede quando os pacotes são transmitidos, mas há também momentos de ociosidade na rede. A Figura 6.3 mostra essa percepção. Na Figura 6.4 estão apresentadas as estatísticas do tráfego de rede gerado, mostrando que a taxa de transmissão da rede, nesse caso, fica em torno de 7MBit/s. Logo, o fato de variar a carga a ser processada não altera o comportamento da rede. Essas observações podem ser confirmadas pela Figura 6.5 e Figura 6.6 que mostram os resultados para o processamento dos pacotes de tamanho 1000 imagens. Os pacotes processados são bastante grandes, a maioria chegando perto de 150MB, acarretando períodos de grande ociosidade da rede, onde a taxa de transmissão é menor que 3MBit/s, combinados com períodos de tráfego mais intenso e aí a taxa de transmissão chega a 21MBit/s. Porém, considerando que a taxa de transmissão máxima na rede utilizada é de 100MBit/s, esses valores obtidos podem ser considerados baixos.

A partir desses resultados confirma-se o fato de que processar tarefas de alta granularidade ou fina granularidade não causa impacto no sistema, do ponto de vista de utilização da rede. Outro fator a ser considerado é que os computadores utilizados são homogêneos tanto no cluster quanto na grade.

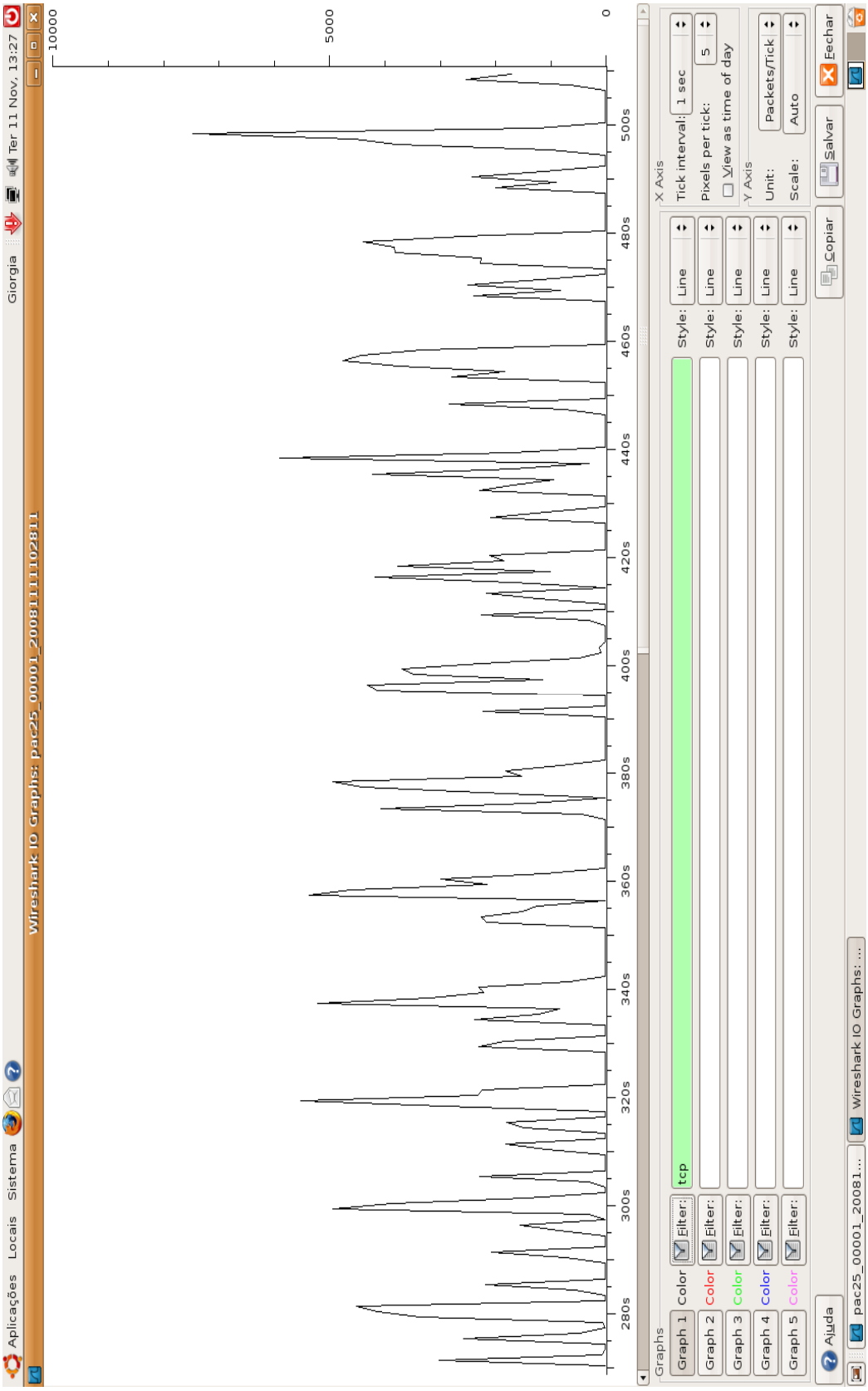


Figura 6.1 – *Wireshark e o tráfego de rede gerado no ambiente de grade quando processados os pacotes de tamanho 25 imagens.*

Figura 6.2 – *Wireshark e as estatísticas de rede obtidas no ambiente de grade quando processados os pacotes de tamanho 25 imagens.*

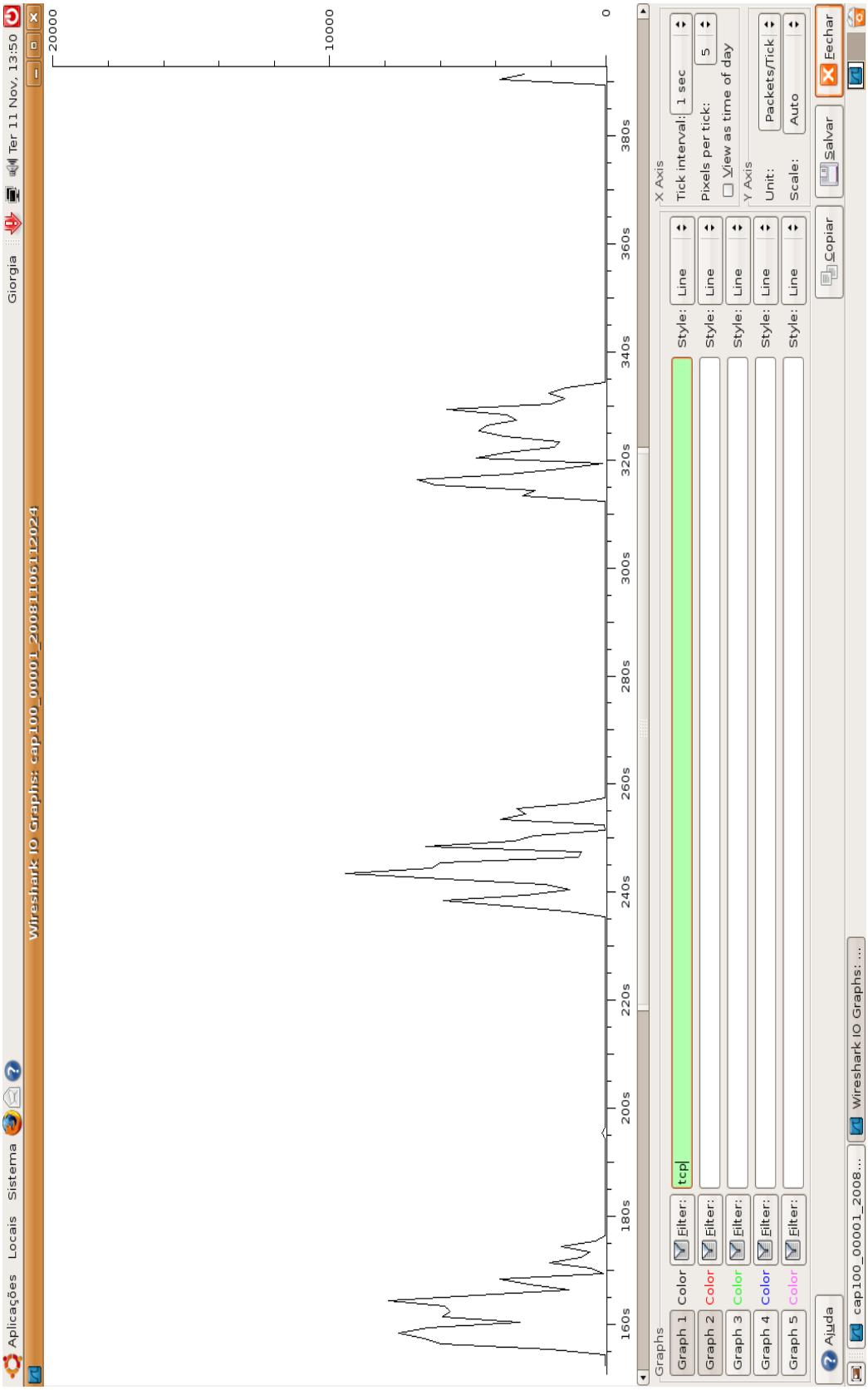


Figura 6.3 – *Wireshark e o tráfego de rede gerado no ambiente de grade quando processados os pacotes de tamanho 100 imagens.*

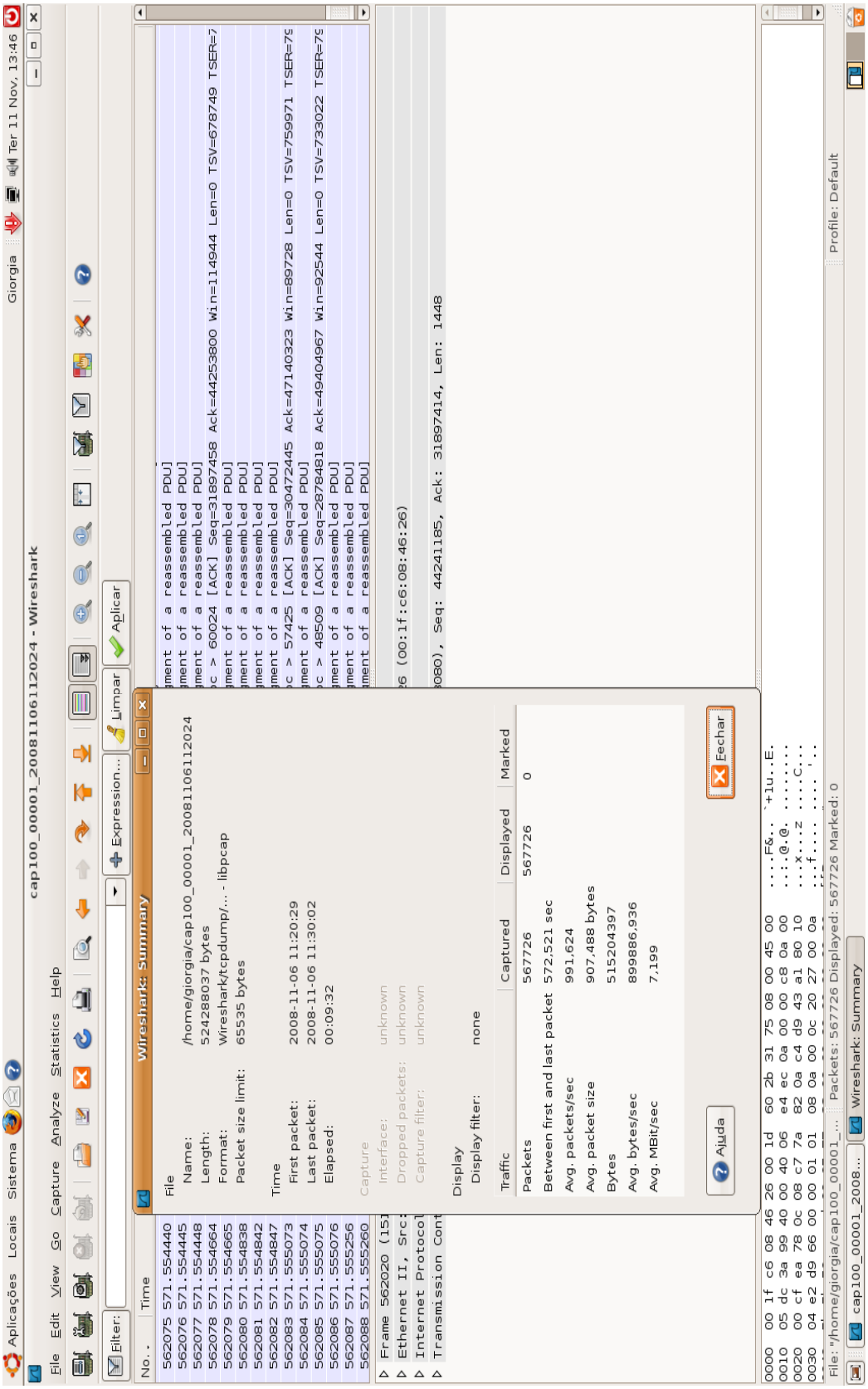


Figura 6.4 – *Wireshark e as estatísticas de rede obtidas no ambiente de grade quando processados os pacotes de tamanho 100 imagens.*

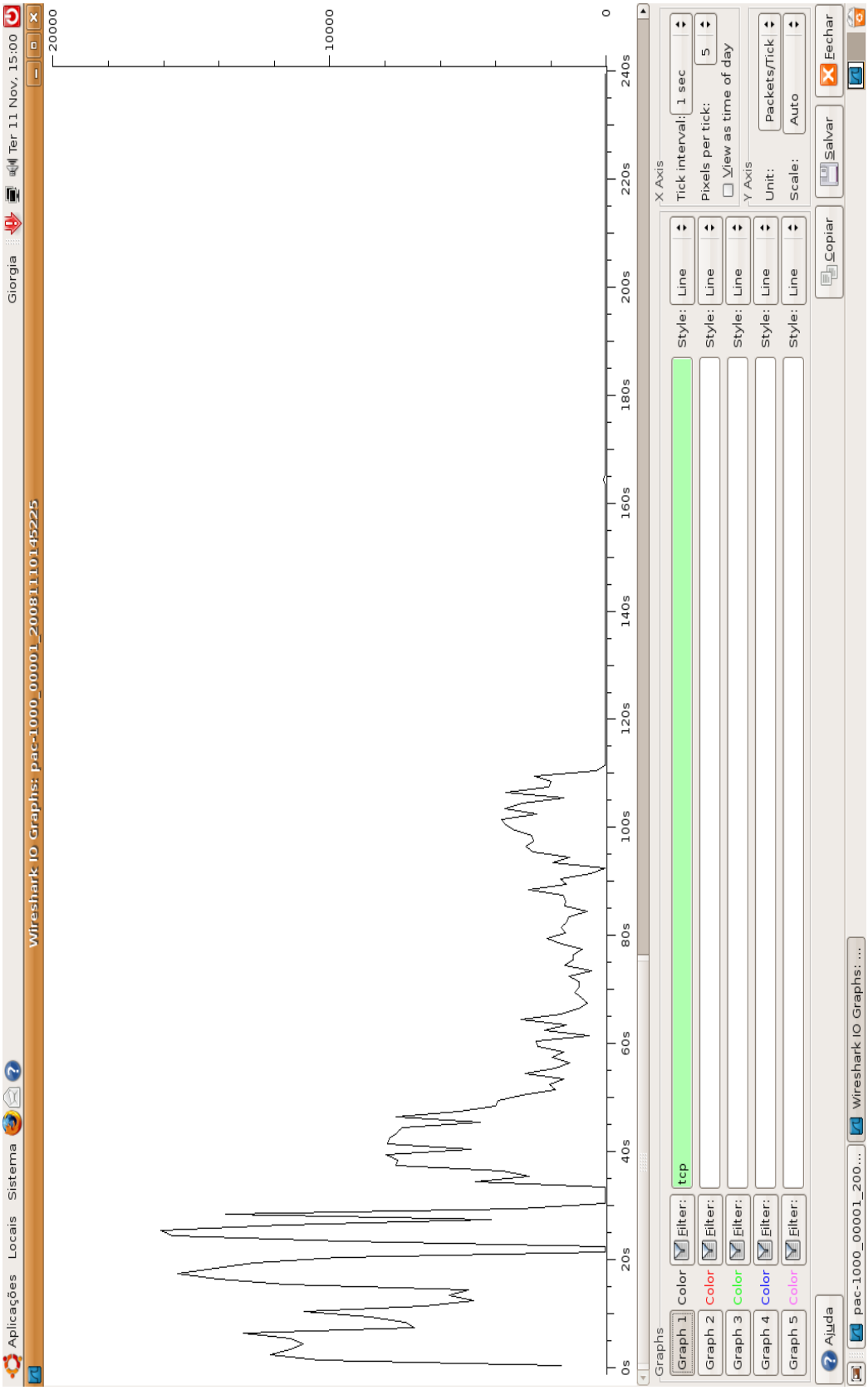


Figura 6.5 – *Wireshark e as estatísticas de rede obtidas no ambiente de grade quando processados os pacotes de tamanho 1000 imagens.*

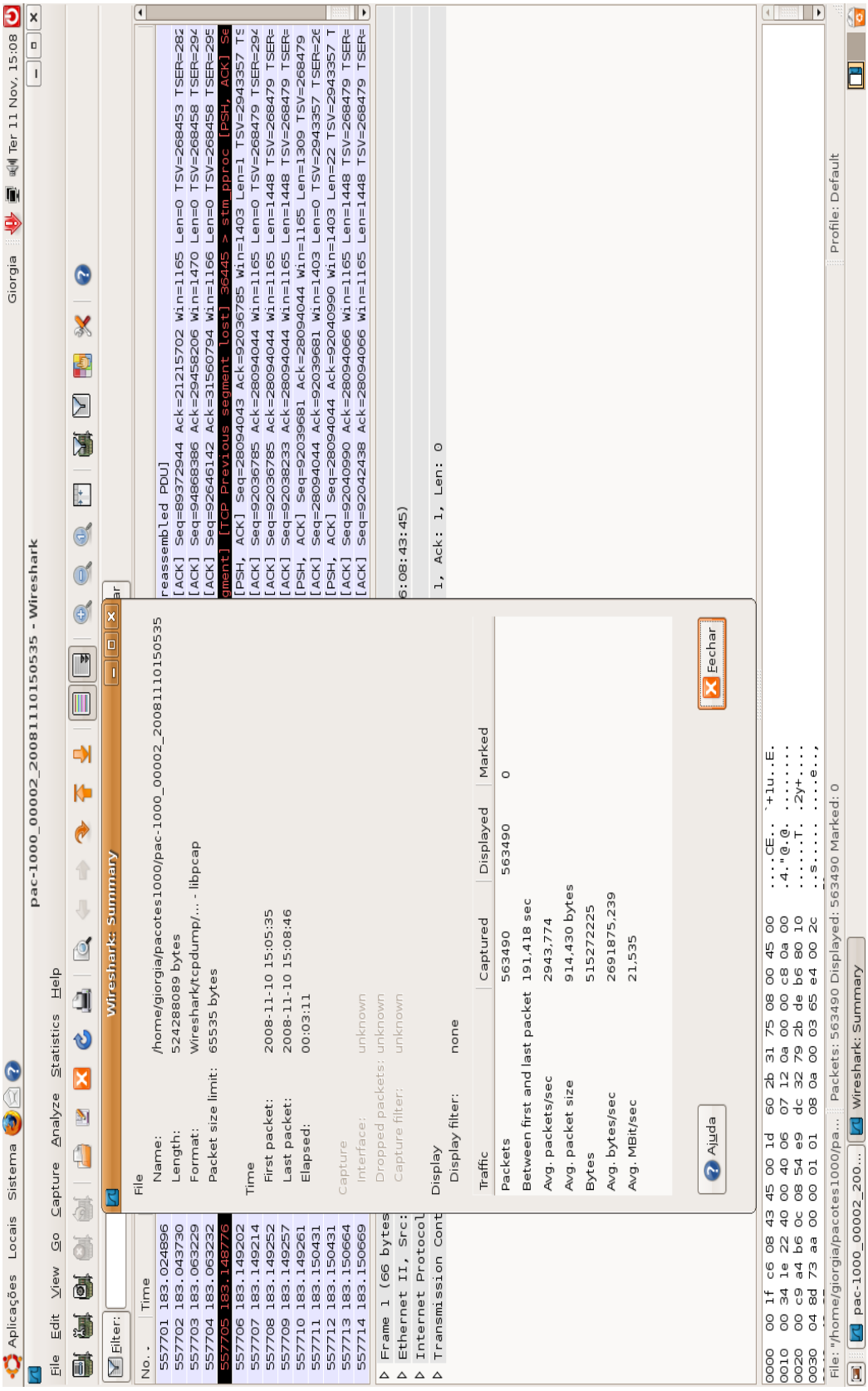


Figura 6.6 – *Wireshark e o tráfego de rede gerado no ambiente de grade quando processados os pacotes de tamanho 1000 imagens.*

6.2 Cluster-Scala

O comportamento do tráfego de rede no cluster-Scala durante o processamento dos pacotes de imagens de documentos é bastante similar ao comportamento da grade. O tráfego presente é, em sua maioria, pertencente ao protocolo TCP com 99% dos pacotes capturados.

Durante o período de processamento dos pacotes de tamanho 25 imagens é possível observar a presença de tráfego mais constante na rede porque esse pacotes são numerosos porém pequenos, demoram pouco tempo para serem processados e transmitidos de volta ao computador mestre. Logo, a ociosidade da rede é baixa e a taxa de transmissão atingida também pode ser considerada assim. Essas observações podem ser vistas na Figura 6.7 que ilustra o tráfego de rede capturado durante o processamento desses pacotes e na Figura 6.8 que apresenta as estatísticas do tráfego de rede.

No processamento dos pacotes de tamanho 100 imagens, observa-se uma ociosidade maior na rede em alguns momentos devido ao tamanho maior dos pacotes sendo processados, porém o tempo de transmissão é baixo, poucos segundos. A Figura 6.9 ilustra o tráfego gerado durante o processamento desses pacotes. E a Figura 6.10 ilustra as estatísticas do tráfego de rede gerado.

O tamanho dos pacotes processados com 1000 imagens cada pode chegar a 200MB em alguns casos. O tempo de processamento desses pacotes é maior, chegando a alguns minutos e conseqüentemente o tempo gasto para transmiti-los pela rede também é grande. Alguns pacotes podem levar quase 40 minutos para serem transmitidos. Assim, no tráfego na rede gerado durante o processamento desses pacotes de 1000 imagens percebe-se grandes períodos de ociosidade da rede e portanto a taxa de transmissão não é alta. A Figura 6.11 e a Figura 6.12 ilustram os resultados obtidos com o processamento desses pacotes.

Com base nos resultados obtidos sobre o tráfego de rede durante o processamento das tarefas no cluster-Scala, observa-se que a taxa de utilização da rede é baixa e portanto processar tarefas de diversos tamanhos não impacta o desempenho global do processamento das imagens.

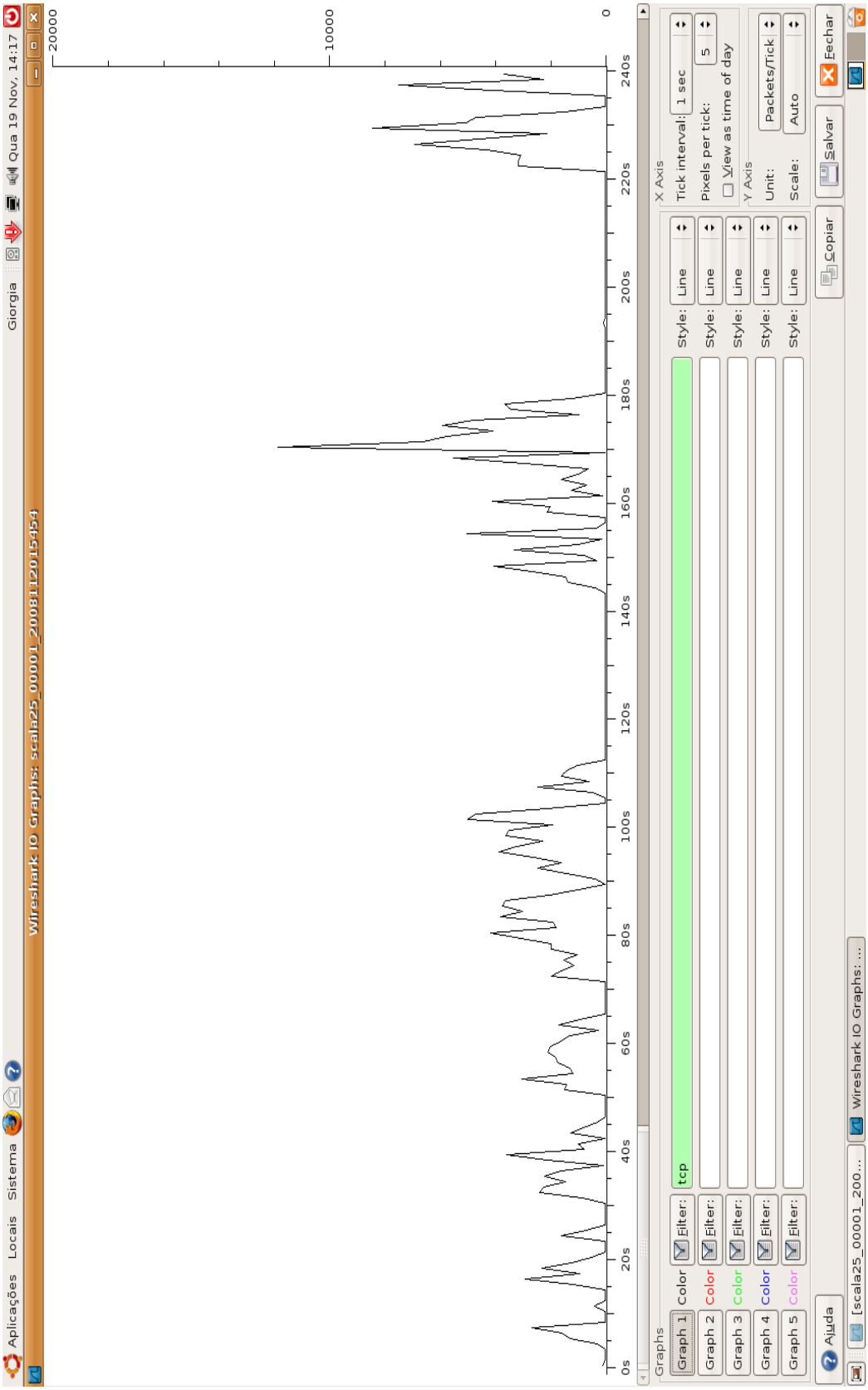


Figura 6.7 – *Wireshark e o tráfego de rede gerado no cluster-Scala quando processados os pacotes de tamanho 25 imagens.*

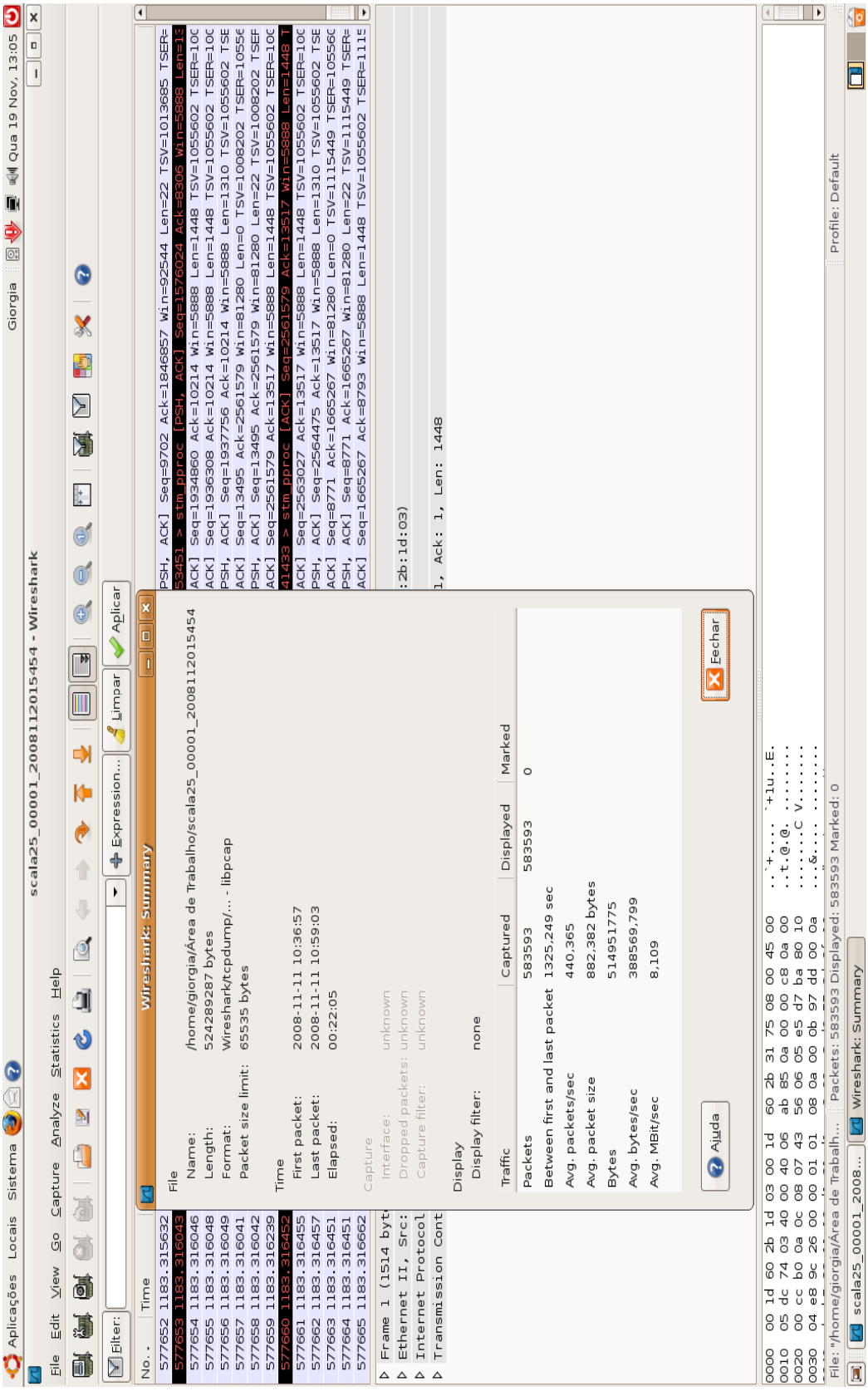


Figura 6.8– *Wireshark e as estatísticas do tráfego de rede quando processados os pacotes de tamanho 25 imagens no ambiente cluster-Scala.*

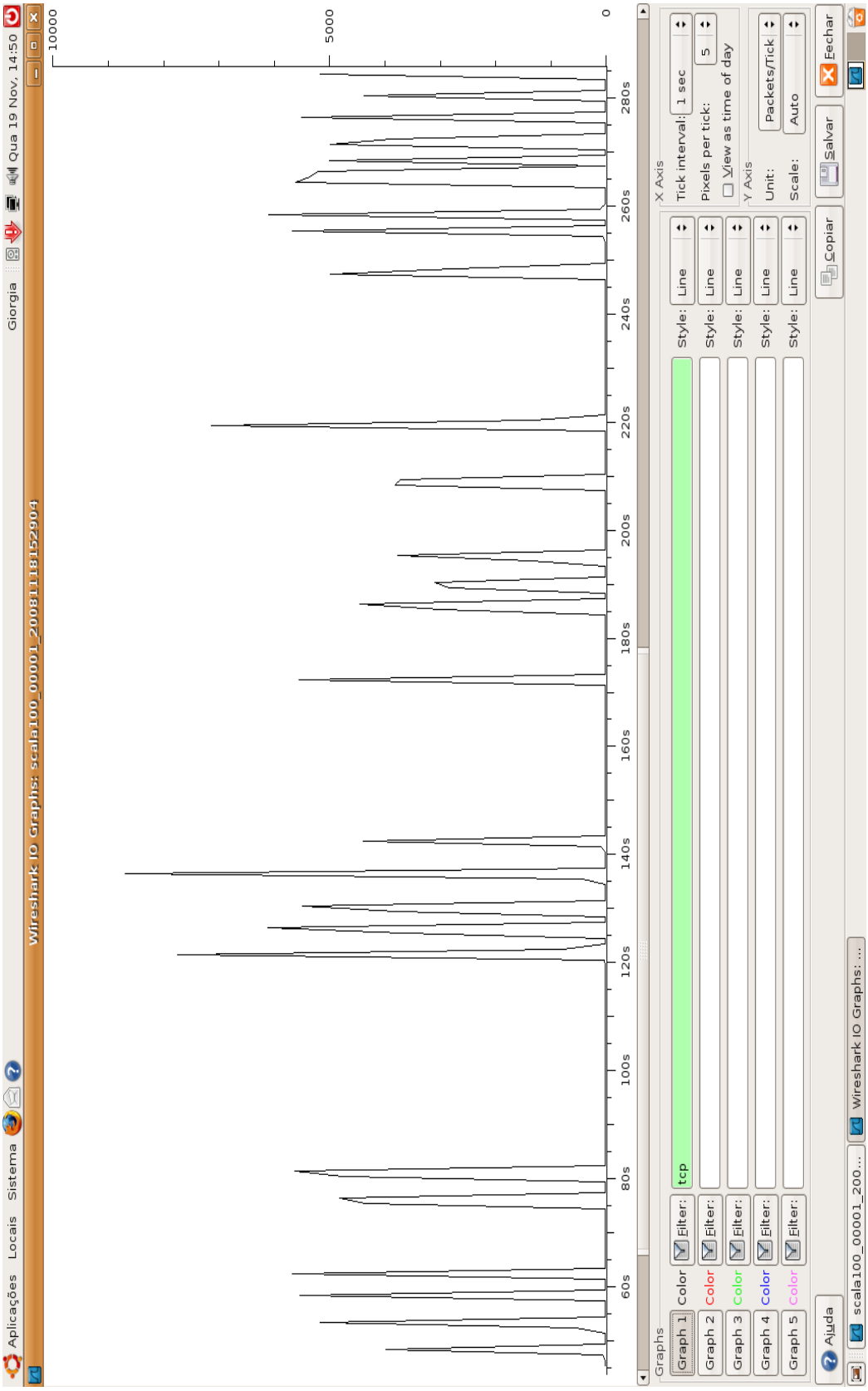


Figura 6.9– *Wireshark e o tráfego de rede gerado no cluster-Scala quando processados os pacotes de tamanho 100 imagens.*

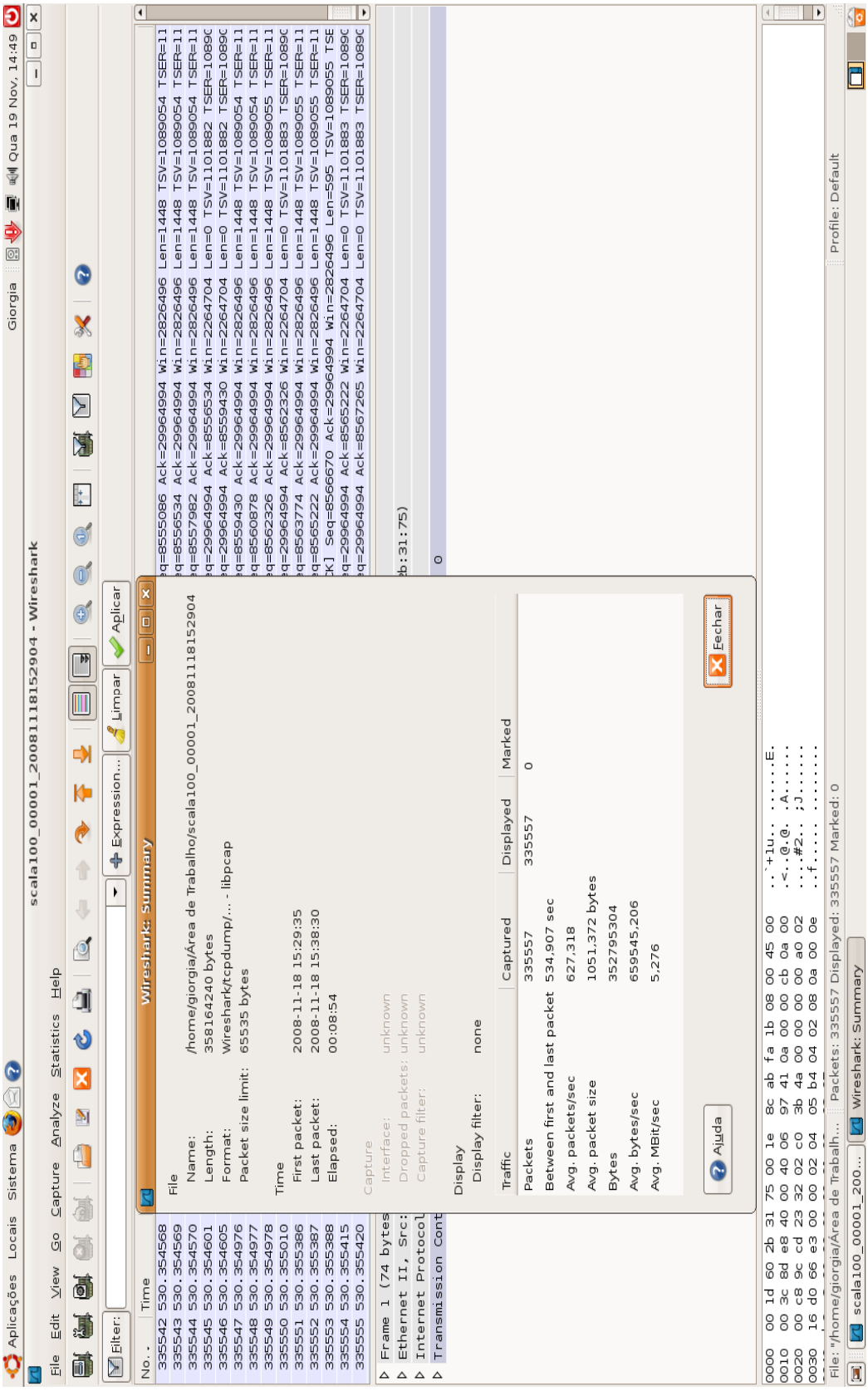


Figura 6.10 – *Wireshark e as estatísticas do tráfego de rede quando processados os pacotes de tamanho 100 imagens no ambiente cluster-Scala.*

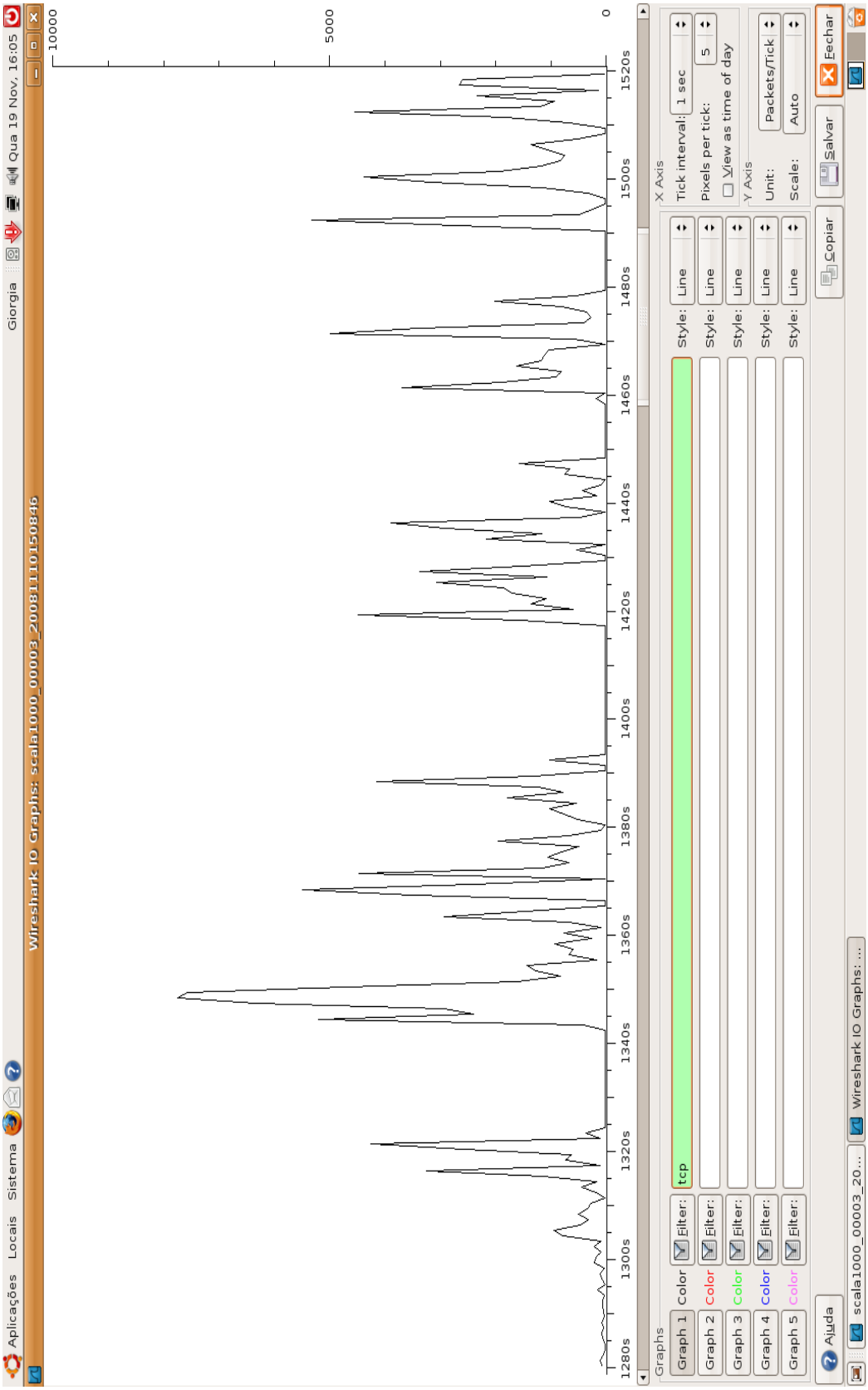


Figura 6.11– *Wireshark e o tráfego de rede gerado no cluster-Scala quando processados os pacotes de tamanho 1000 imagens.*

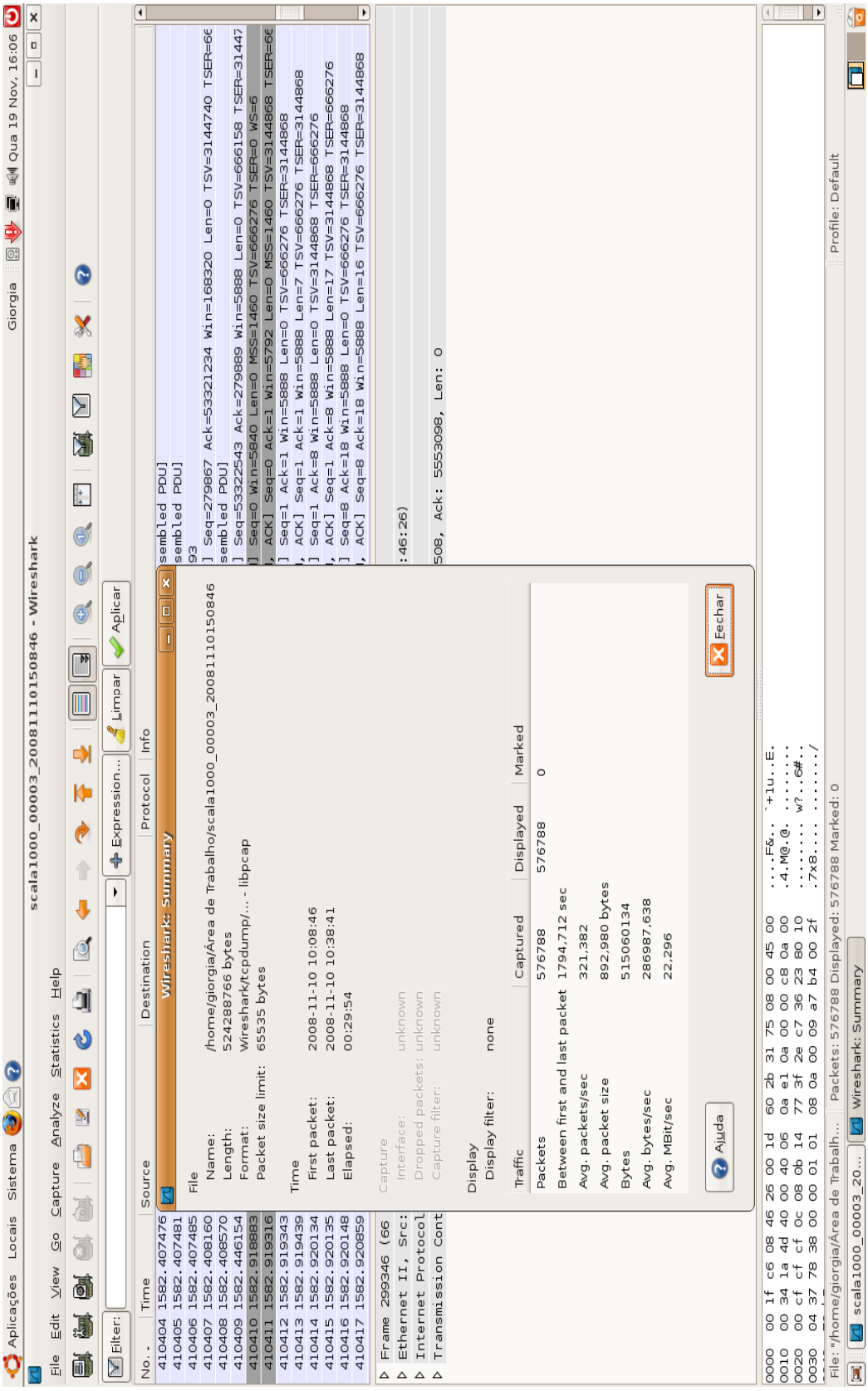


Figura 6.12 – *Wireshark e as estatísticas do tráfego de rede quando processados os pacotes de tamanho 1000 imagens no ambiente cluster-Scala.*

6.3 Cluster MPI

A Figura 6.13 ilustra a ocorrência dos pacotes SMB e TCP durante o processamento das imagens de documentos com pacotes de tamanho 25 imagens. Com a utilização da ferramenta *Wireshark* foi possível constatar a predominância do tráfego TCP durante o processamento dos diferentes tamanhos de pacotes. Em média 95% do tráfego gerado na rede correspondem ao protocolo TCP, 4% ao protocolo SMB e 1% a outros protocolos como, por exemplo, o UDP, ACK e outros. A linha vermelha indica a ocorrência do protocolo TCP e a linha preta a ocorrência do protocolo SMB.

Na Figura 6.14 são mostradas estatísticas sobre o tráfego da rede como o tamanho médio dos pacotes, a média de pacotes capturados por segundos, a taxa média de transferência dos dados, entre outros. O tráfego médio atingido foi em torno de 15MBit/s. Observando a Figura 6.15, é possível acompanhar a utilização da rede durante o processamento das tarefas, em torno de 30 minutos, através do Gerenciador do Cluster HPC.

A Figura 6.16, obtida pela utilização do Gerenciador de Tarefas do Windows, ilustra graficamente o tráfego gerado na rede durante o processamento efetivo dos pacotes de imagens de documentos com tamanho 25 imagens. A linha na cor vermelha refere-se aos bytes enviados, a linha amarela refere-se aos bytes recebidos e a linha verde refere-se ao total de bytes trafegando na rede. A taxa de utilização da rede durante o período do processamento fica em 27%.

Considerando o processamento dos pacotes de imagens de documentos de tamanho 100 imagens, os resultados são apresentados na Figura 6.17, Figura 6.18, Figura 6.19 e Figura 6.20. A Figura 6.17 ilustra a ocorrência dos pacotes TCP e SMB capturados durante o processamento das tarefas. A maioria do tráfego gerado é correspondente ao protocolo TCP, 95%, seguido do protocolo SMB, 4%, e outros, 1%. A linha preta refere-se à ocorrência do protocolo TCP e a linha vermelha ao protocolo SMB.

Na Figura 6.18 são apresentadas as estatísticas do tráfego de rede gerado durante o processamento dos pacotes onde a taxa de transferência média ficou em 17Mbit/s. A Figura 6.19 permite acompanhar o período completo de processamento das tarefas e também observar o comportamento da rede através do Gerenciador do Cluster HPC. E a Figura 6.20 ilustra os resultados através do Gerenciador de Tarefas do Windows mostrando as estatísticas da rede obtidas em tempo de execução das tarefas.

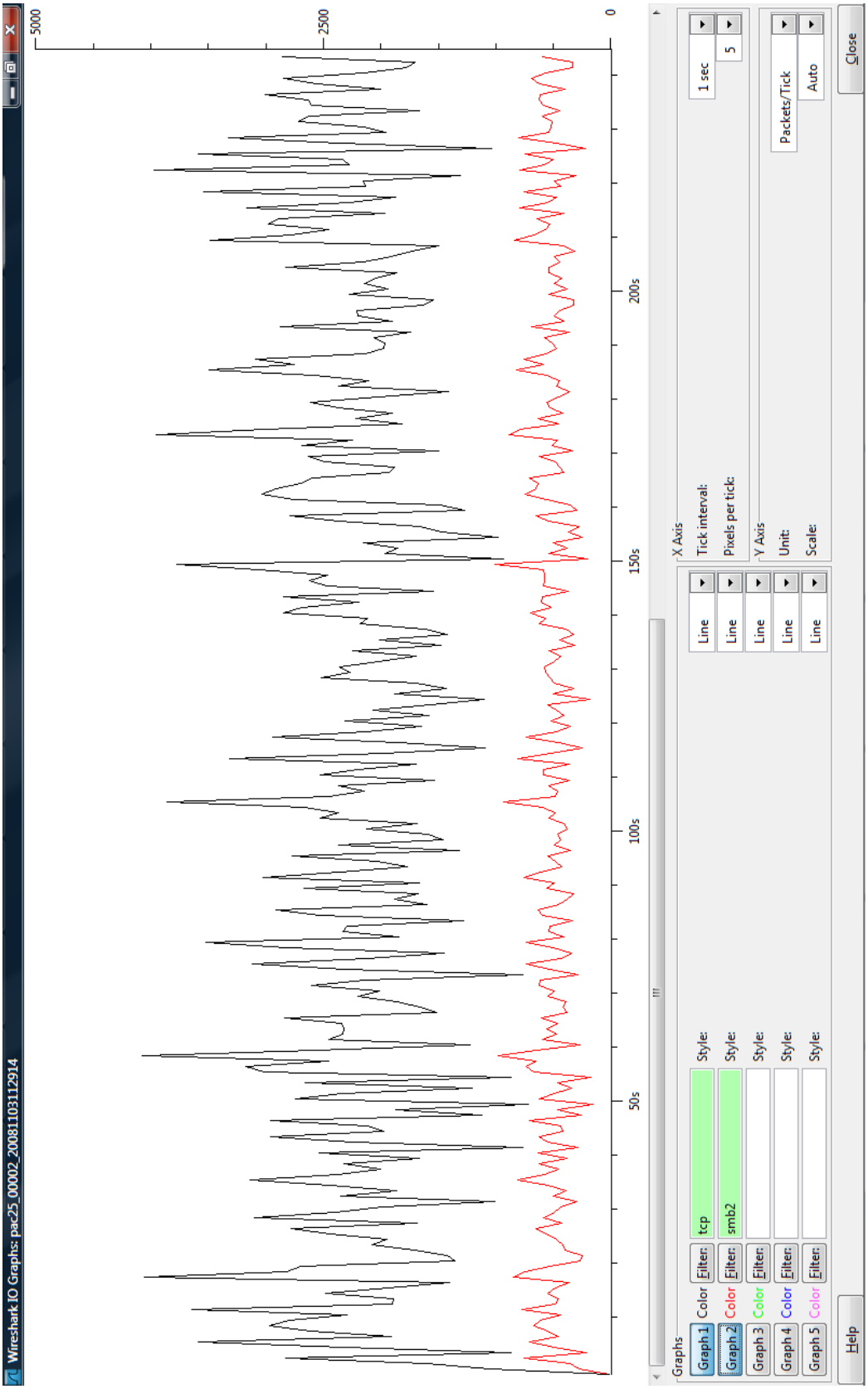


Figura 6.13 – *Wireshark no cluster MPI: gráfico ilustrando a ocorrências dos pacotes TCP e SMB durante o processamento dos pacotes de tamanho 25 imagens.*

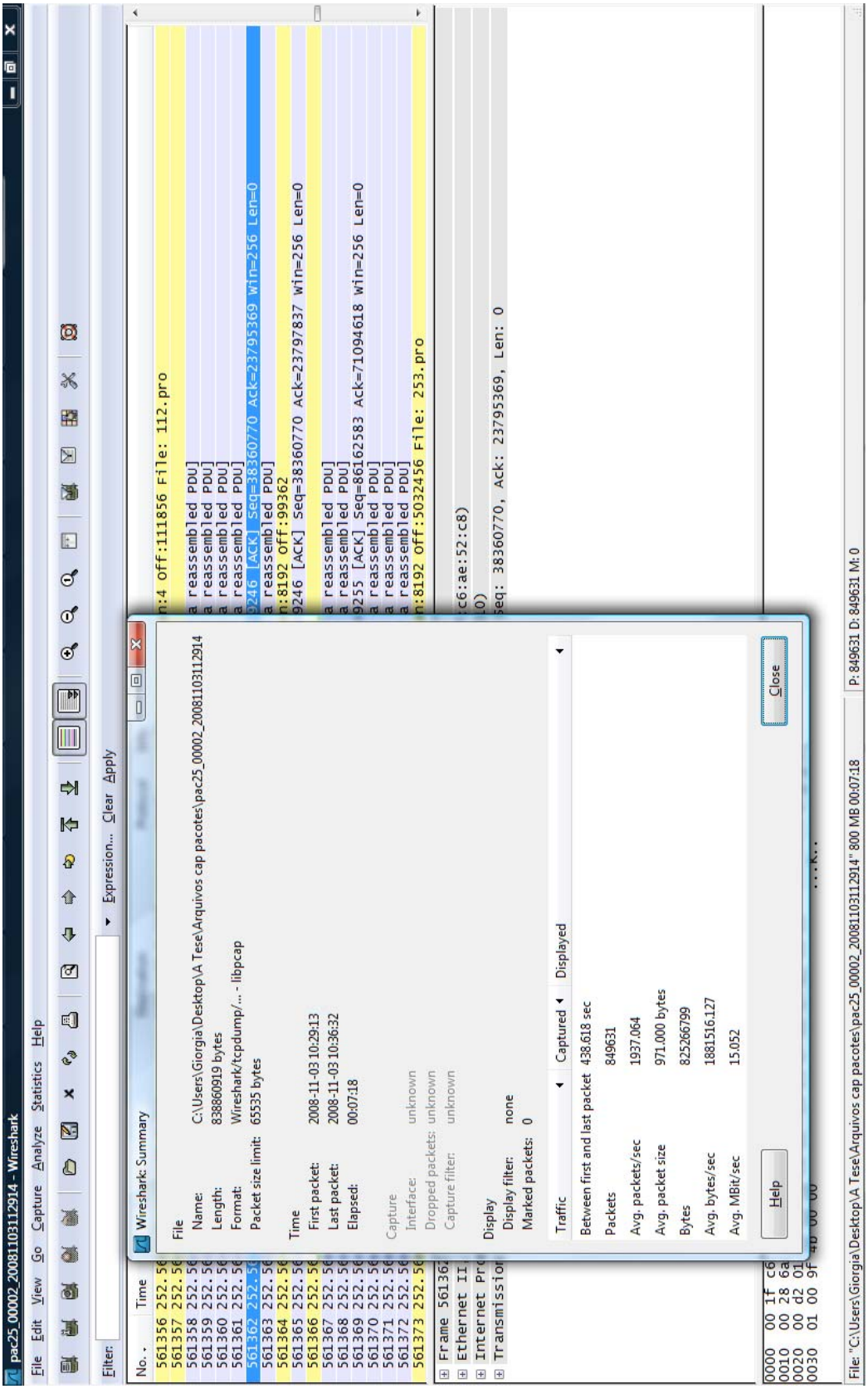


Figura 6.14 – *Wireshark no cluster MPI: estatísticas sobre o tráfego de rede durante o processamento dos pacotes de tamanho 25 imagens.*

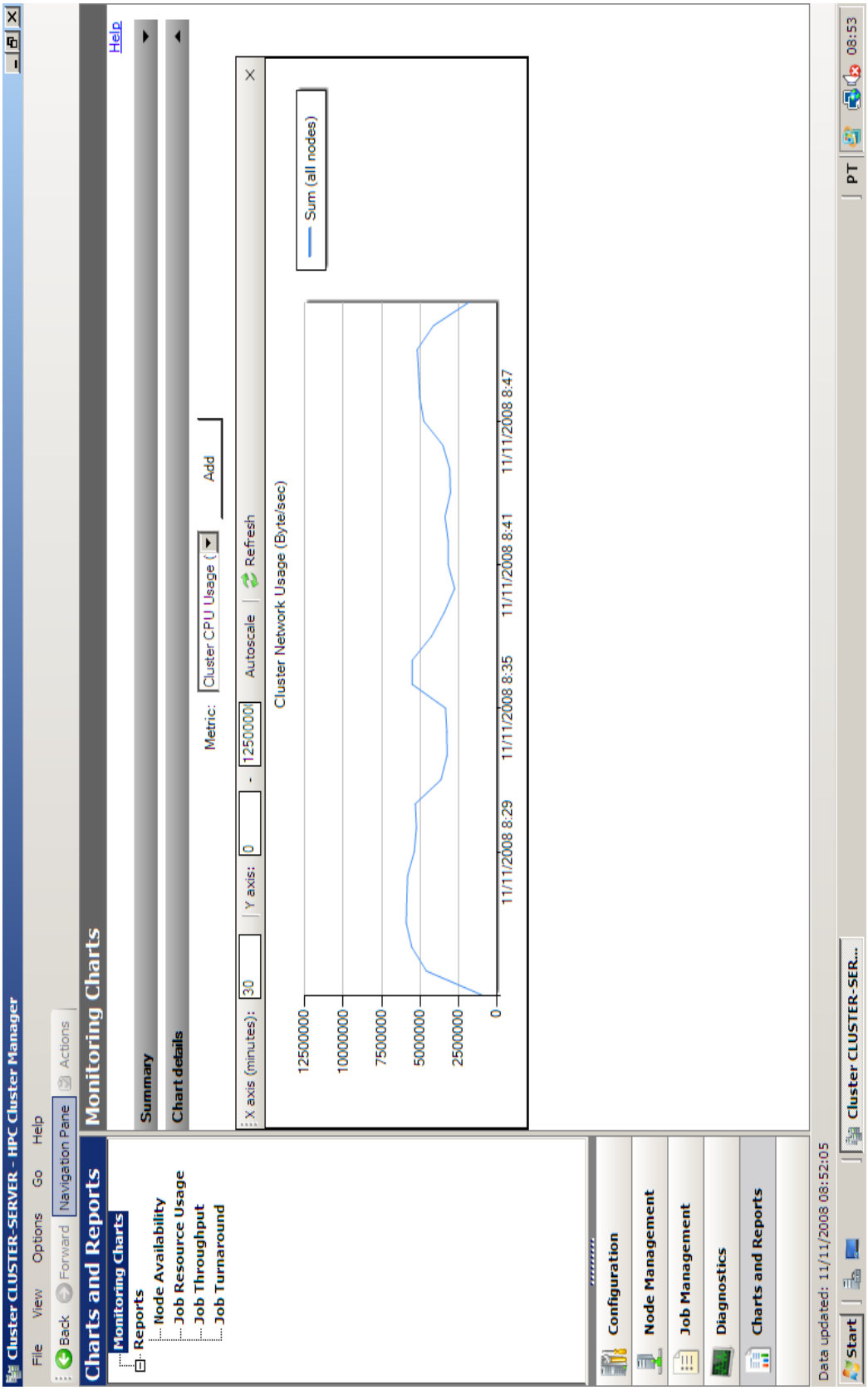


Figura 6.15 – *Gerenciador do Cluster HPC: gráfico ilustrando a utilização da rede durante o processamento dos pacotes de tamanho 25 imagens.*

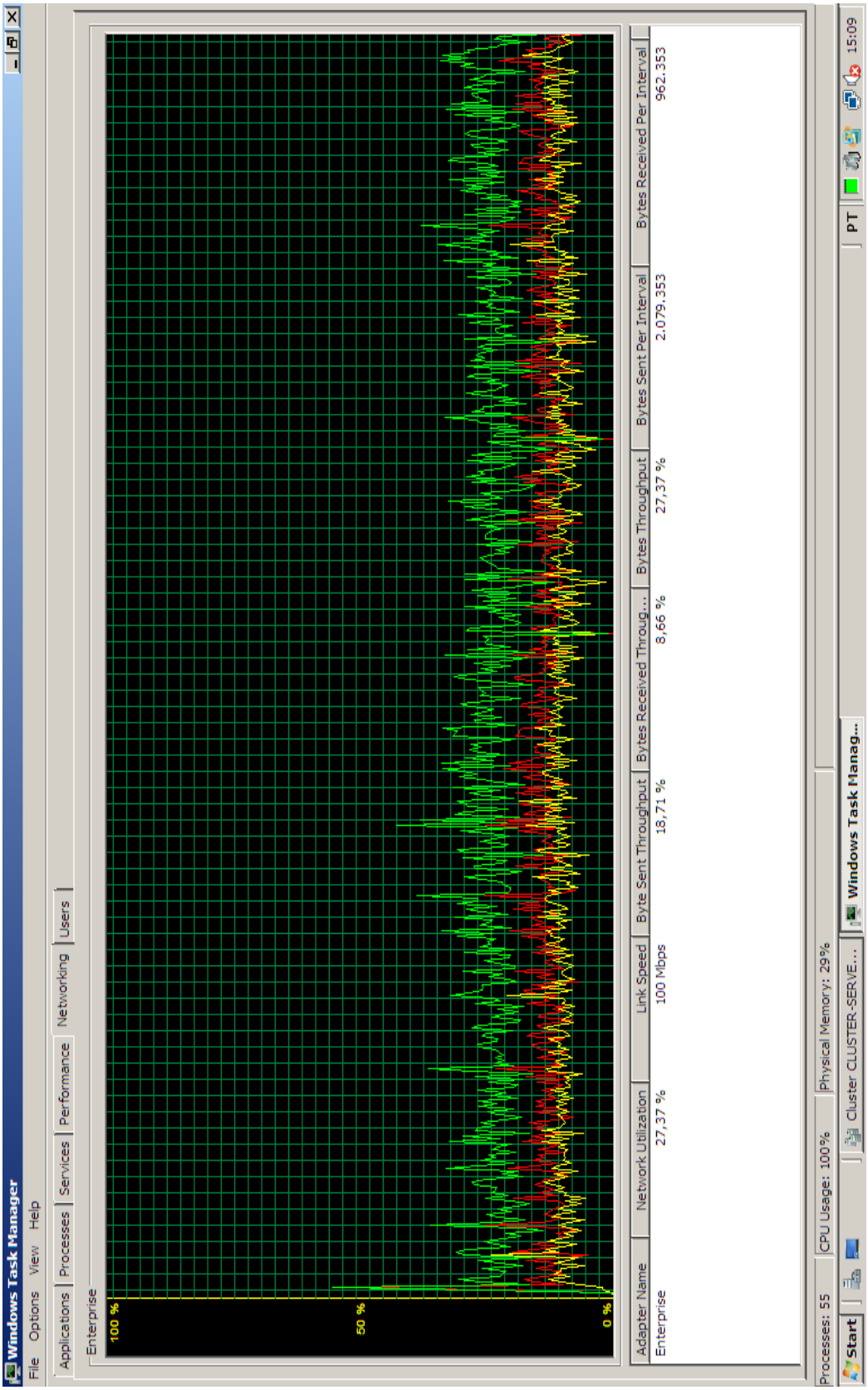


Figura 6.16 – *Gerenciador de Tarefas: estatísticas sobre a utilização da rede durante o processamento dos pacotes de tamanho 25 imagens.*

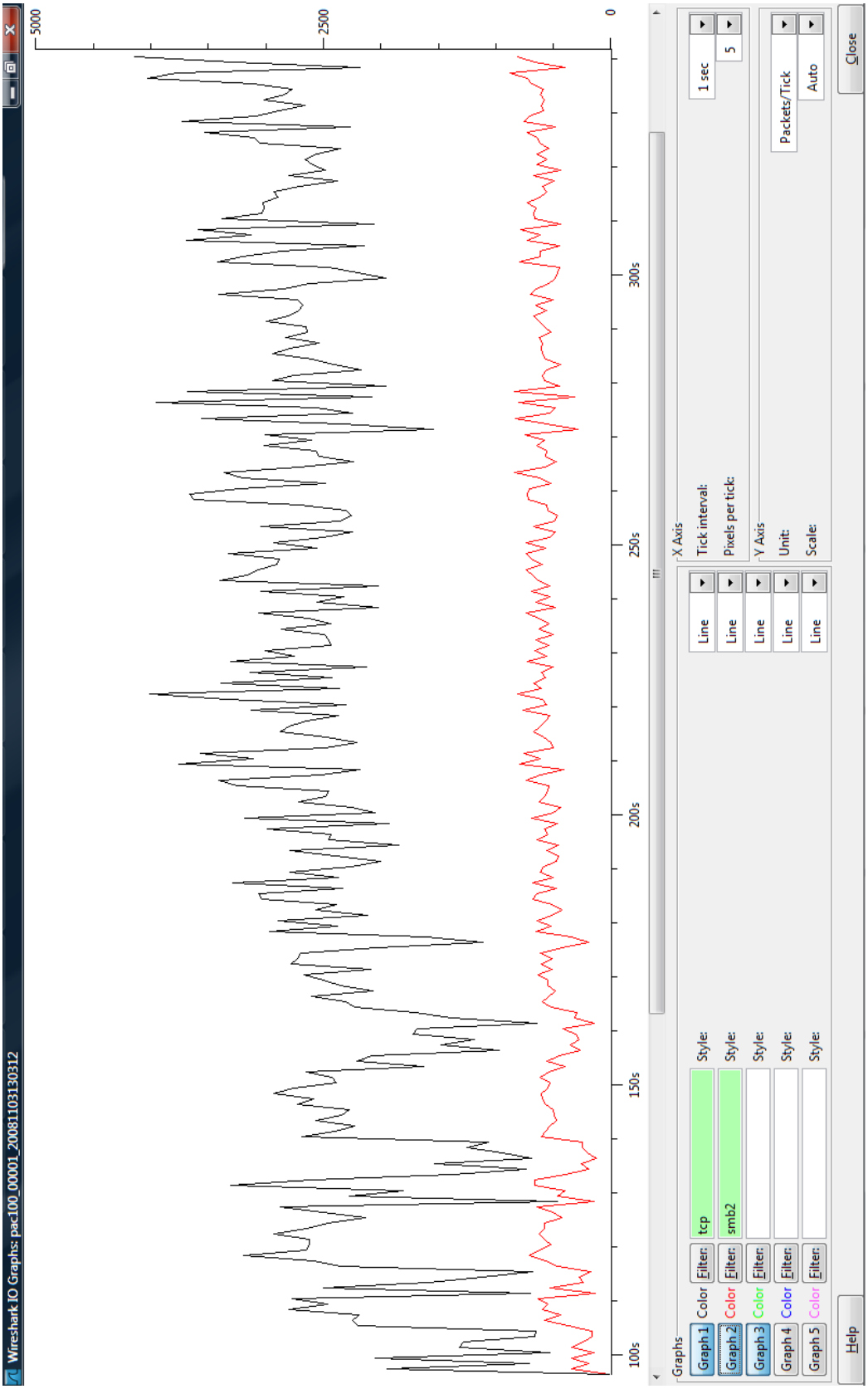


Figura 6.17 – *Wireshark no cluster MPI: gráfico ilustrando a ocorrências dos pacotes TCP e SMB durante o processamento dos pacotes de tamanho 100 imagens.*

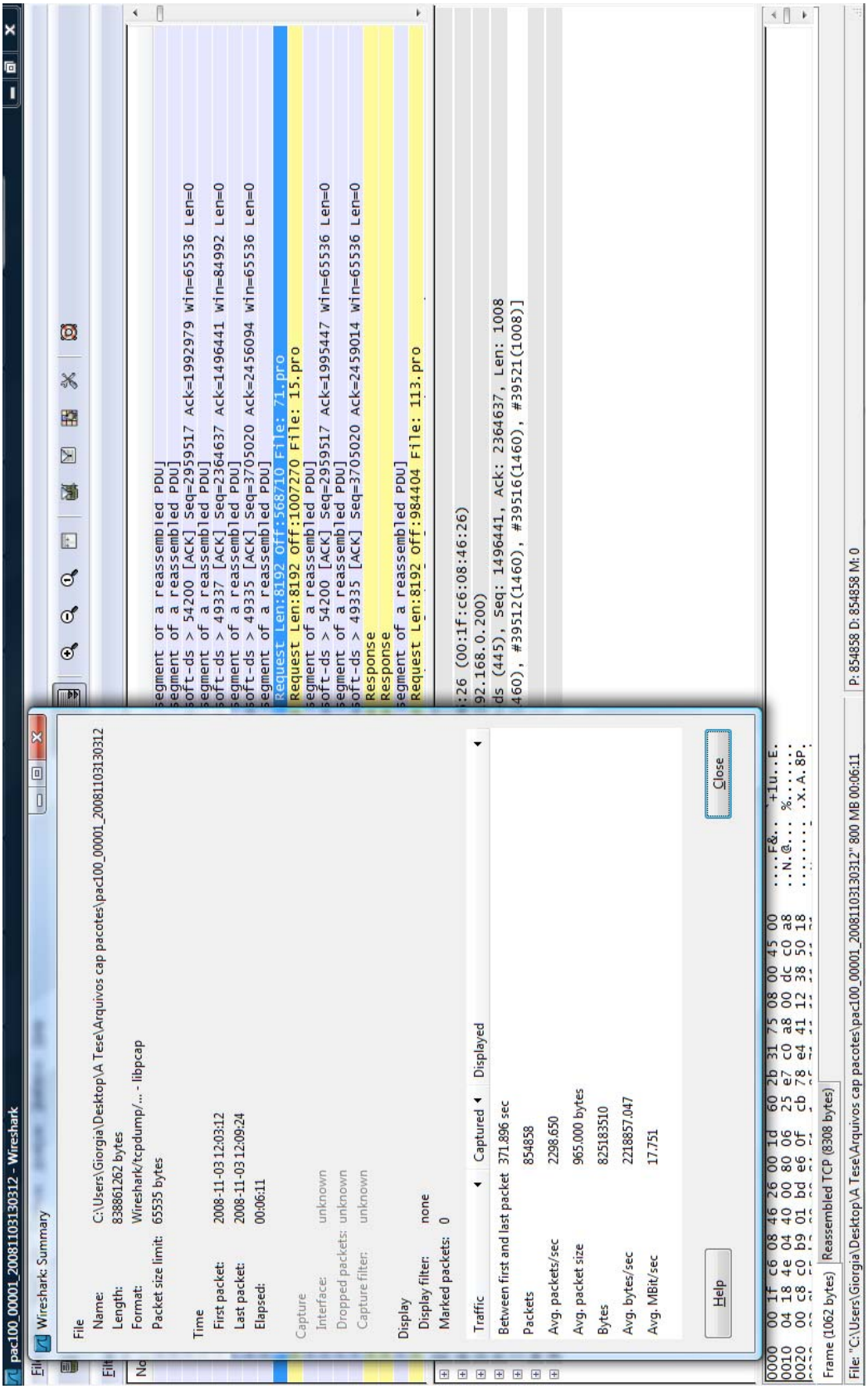


Figura 6.18 – *Wireshark no cluster MPI: estatísticas sobre o tráfego de rede durante o processamento dos pacotes de tamanho 100 imagens.*

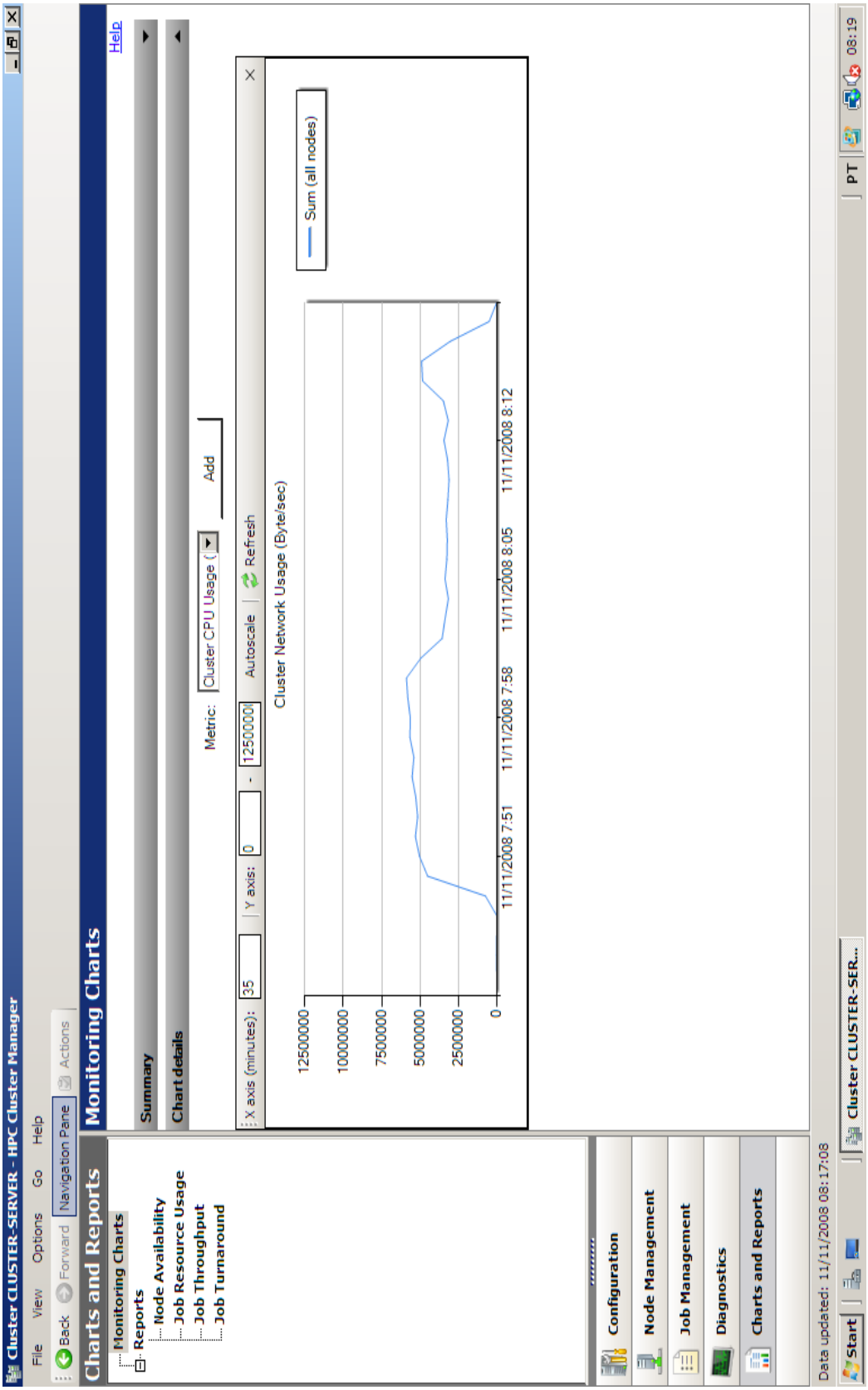


Figura 6.19 – *Gerenciador do Cluster HPC: Utilização da rede durante o processamento dos pacotes de tamanho 100 imagens.*

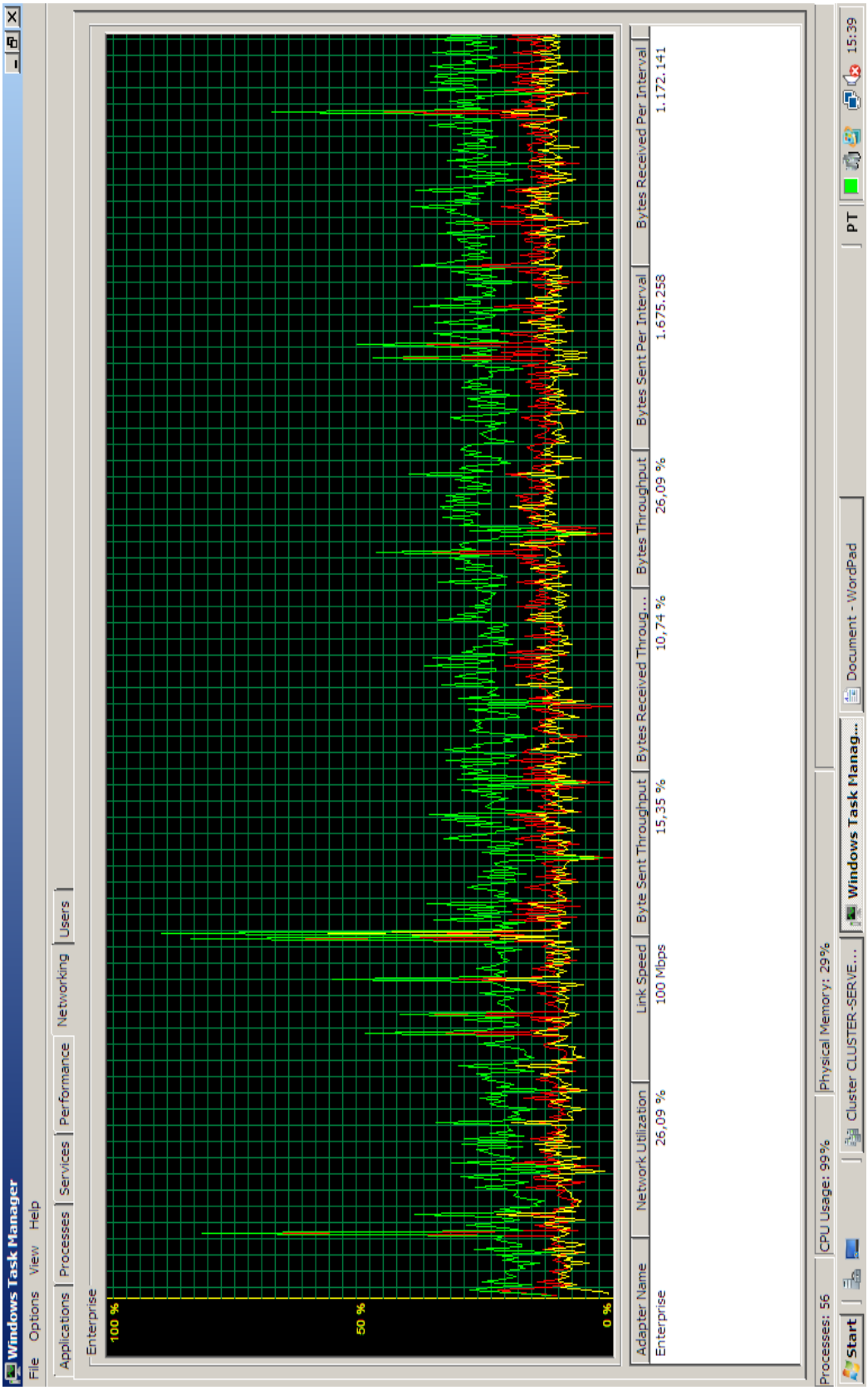


Figura 6.20 – *Gerenciador de Tarefas: estatísticas sobre a utilização da rede durante o processamento dos pacotes de tamanho 100 imagens.*

Para as tarefas de tamanho 1000 imagens, os resultados são apresentados nas figuras que seguem. A Figura 6.21 refere-se à ocorrência dos pacotes TCP e SMB durante o processamento das tarefas. Os resultados foram obtidos com a utilização da ferramenta *Wireshark*; a linha preta refere-se ao protocolo TCP, 95%, e a linha vermelha refere-se à ocorrência do protocolo SMB, 4%. Os demais protocolos somam 1%.

Na Figura 6.22 são apresentadas as estatísticas do tráfego de rede gerado durante a execução das tarefas de tamanho 1000 imagens. A taxa de transferência média é de 14MBit/s.

A Figura 6.23 ilustra a utilização da rede durante o processamento das tarefas de tamanho 1000 imagens, sendo possível acompanhar o comportamento da rede durante todo o tempo de processamento, quase 38 minutos.

Finalmente, a Figura 6.24 obtida pelo Gerenciador de Tarefas do Windows, ilustra graficamente o tráfego gerado na rede durante o processamento efetivo dos pacotes de imagens de documentos com tamanho 1000 imagens. A taxa de utilização da rede durante o período de processamento das tarefas fica em 14%.

Esses resultados permitem confirmar que o tráfego de rede gerado durante o processamento dos pacotes de diferentes tamanhos é baixo, considerando os 100MBit/s disponíveis. Permite confirmar ainda que o fato de utilizar pacotes com diferentes tamanhos, gerando tráfegos diferentes na rede nos momentos de transferências dos pacotes não causa impacto no tempo total de execução das tarefas. Os resultados apresentados não mostram diferenças que mereçam ser consideradas, visto que os valores são bastante próximos. Assim, o balanceamento de carga é eficiente para o problema abordado.

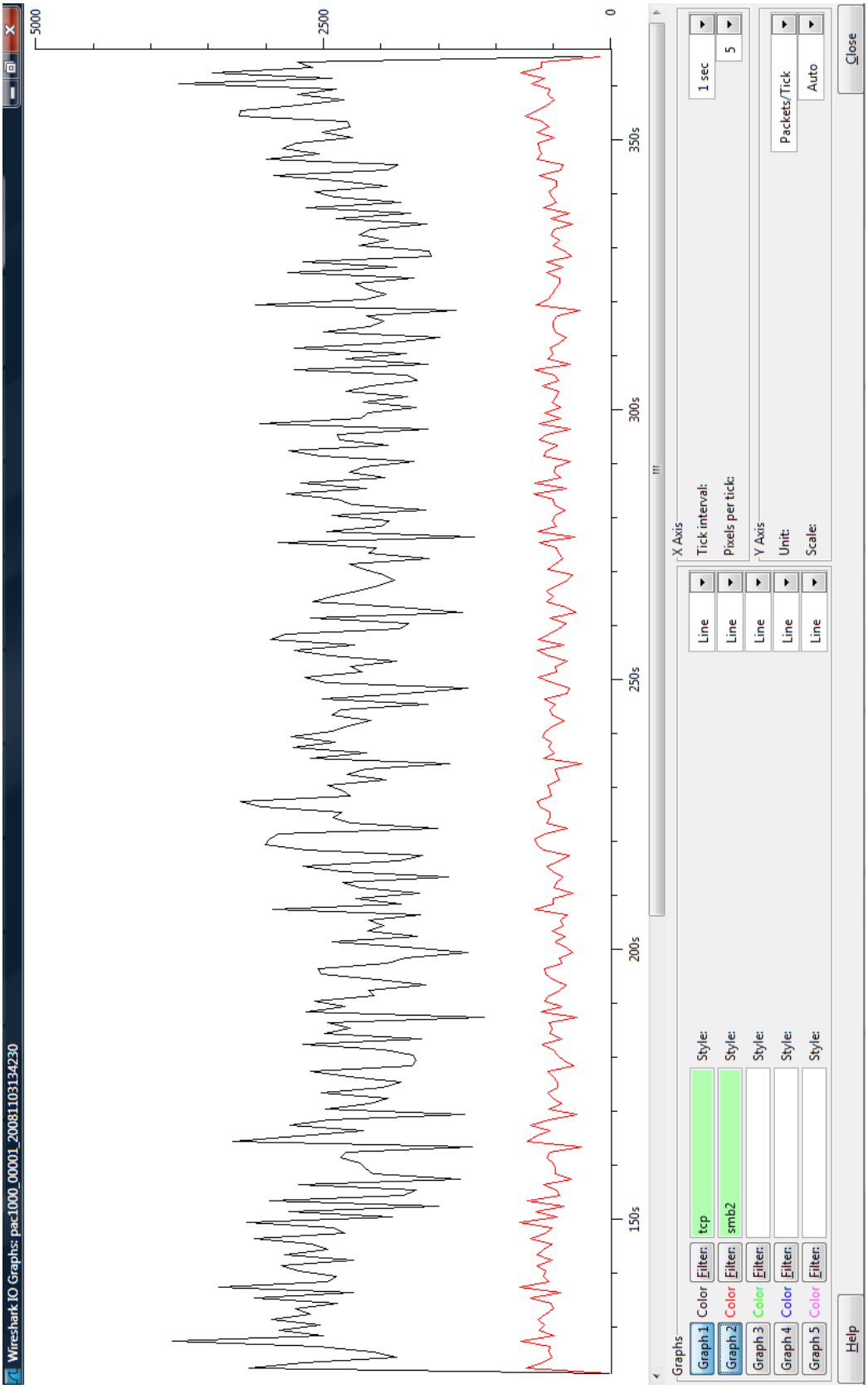


Figura 6.21 – *Wireshark no cluster MPI: gráfico ilustrando a ocorrências dos pacotes TCP e SMB durante o processamento dos pacotes de tamanho 1000 imagens.*

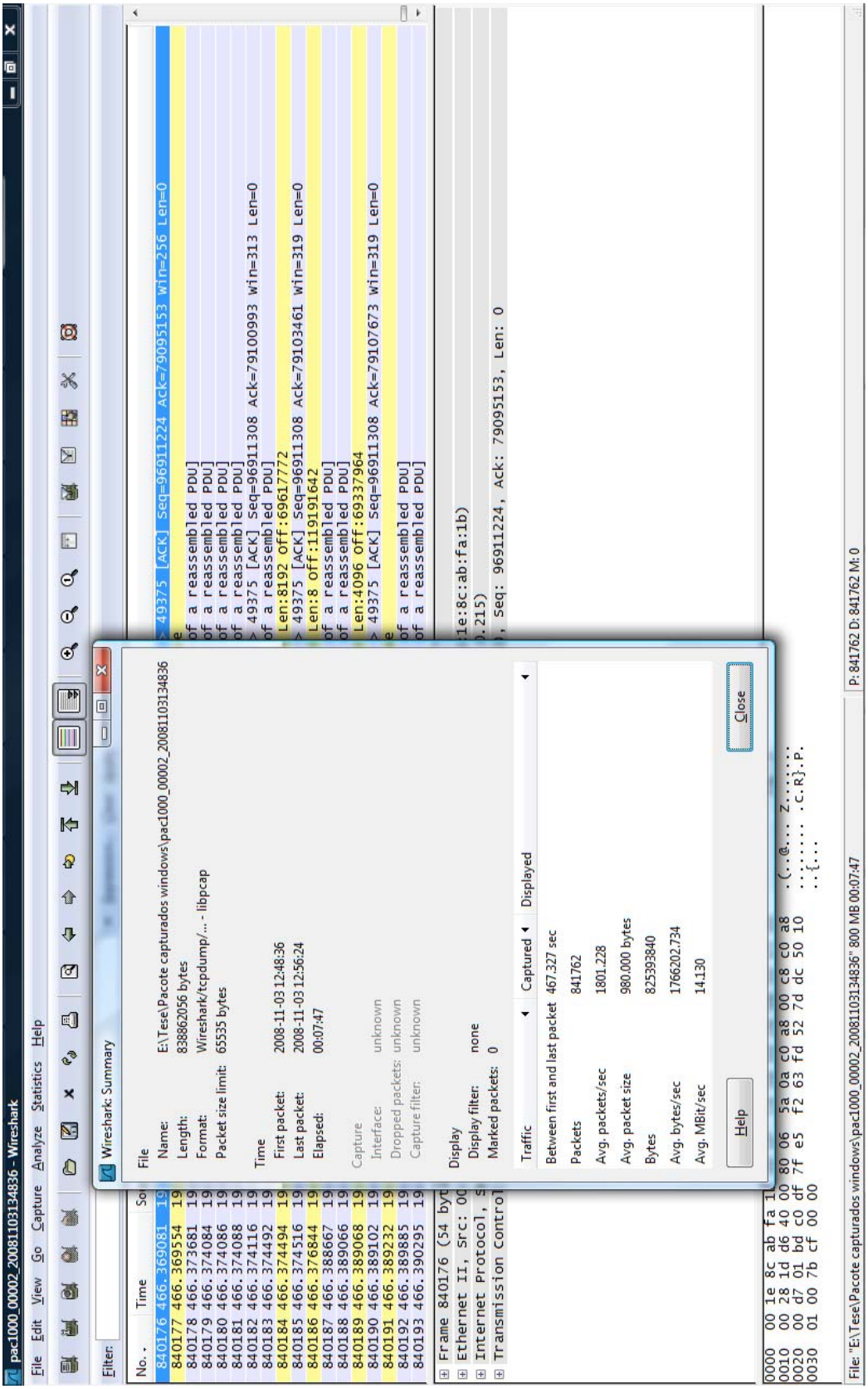


Figura 6.22 – *Wireshark no cluster MPI: estatísticas sobre o tráfego de rede durante o processamento dos pacotes de tamanho 1000 imagens.*

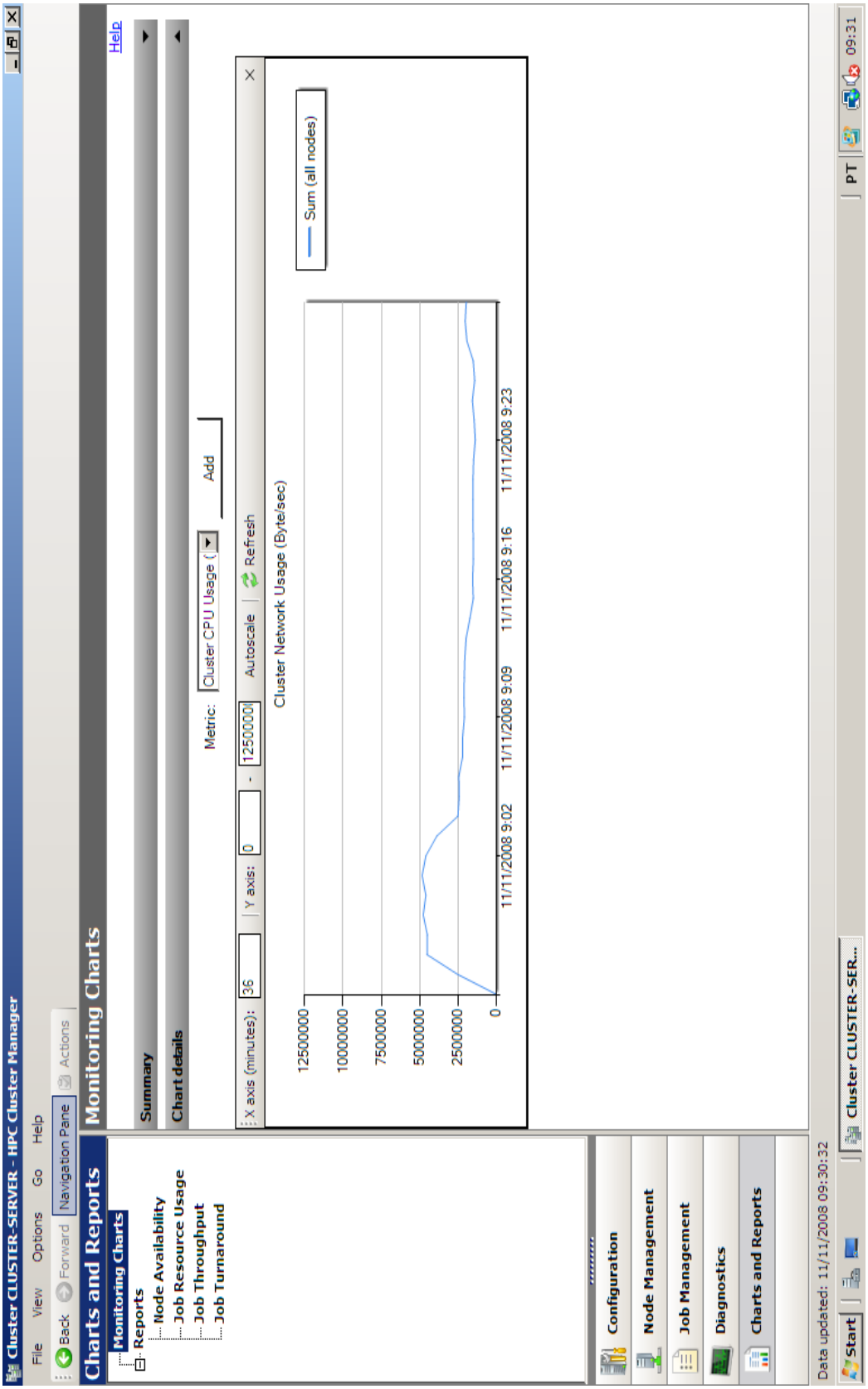


Figura 6.23 – *Gerenciador do Cluster HPC: Utilização da rede durante o processamento dos pacotes de tamanho 1000 imagens.*

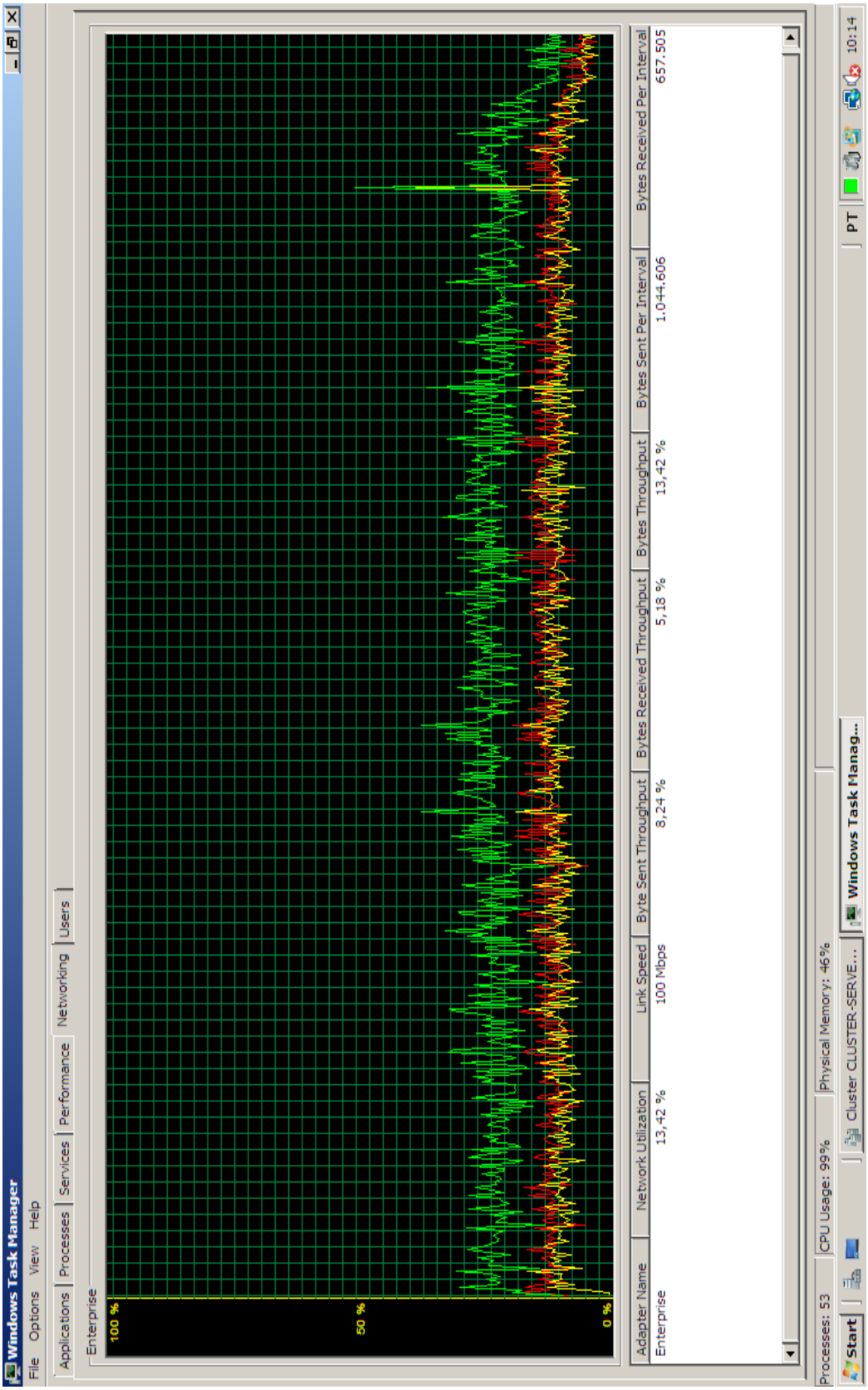


Figura 6.24 – *Gerenciador de Tarefas: estatísticas sobre a utilização da rede durante o processamento dos pacotes de tamanho 1000 imagens.*

7. CONCLUSÕES

Esta tese apresentou aspectos de desempenho das plataformas de cluster e grade, considerando a execução da aplicação BigBatch. BigBatch é um ambiente de processamento de imagens de documentos projetado para processar lotes de milhares de documentos monocromáticos. Uma das flexibilidades e pioneirismo da ferramenta BigBatch é a possibilidade de trabalhar em ambientes distribuídos como clusters e grades. Dessa maneira é possível analisar comparativamente clusters e grades quando executam as mesmas tarefas.

Para tal análise foi processado um lote de 21.200 imagens reais de documentos monocromáticos com a ferramenta BigBatch. A base de imagens processadas corresponde à carga produzida em 1 dia de trabalho (8 horas) de um scanner de linha de produção com alimentação automática. As imagens foram organizadas em pacotes de diferentes tamanhos, 25, 50, 100, 500 e 1000 imagens, com o objetivo de medir o quanto cargas de diferentes tamanhos afetam o desempenho das duas plataformas. Os experimentos foram realizados sobre diversos cenários como a variação no tamanho do pacote, a participação ou não do computador mestre no processamento das tarefas e a adição de uma carga computacional extra.

A infraestrutura física e de software adotada nos experimentos consistiu de computadores com a tecnologia *hyperthreading* em seus processadores e também computadores *dual core*. Com relação ao software de grade e cluster, foi utilizado o OurGrid e duas aplicações para o cluster, uma utilizando a linguagem de programação Scala e a outra utilizando a biblioteca padrão em computação paralela a MPI.

Os resultados obtidos mostraram que embora o software de grade seja considerado, na literatura, mais robusto e complexo que o de cluster, ele pode ser utilizado em redes locais para o processamento deste tipo de tarefa, uma vez que o seu desempenho foi ligeiramente melhor em relação ao cluster quando utilizados computadores com a tecnologia *hyperthreading* e agendada apenas uma tarefa por nó. Porém, quando utilizada a tecnologia *hyperthreading* e *dual core* no cluster, que permite o escalonamento de duas tarefas por nó, aproveitando realmente as potencialidades das arquiteturas de computadores disponíveis, o desempenho obtido é significativamente melhor que a grade. Essa configuração não é permitida no OurGrid, sendo indiferente a arquitetura de computadores utilizada. Comparando os dois ambientes de cluster desenvolvidos, o cluster-MPI é expressivamente melhor que o cluster-Scala.

Outro ponto analisado foi em relação ao tráfego de rede gerado durante a execução do processamento da base de imagens. A taxa de utilização da rede é baixa ficando em torno de 25%

da sua capacidade, confirmando a variação inexistente nos tempos de processamento dos diferentes tamanhos de pacotes. O tráfego predominante é do protocolo TCP contando ainda com SMB, UDP e outros protocolos em proporções bem menores.

Para a situação estudada é possível concluir que o ambiente de grade é mais vantajoso que o ambiente de cluster somente quando considerado o agendamento de apenas uma tarefa por nó. Já o cluster se comporta melhor quando utilizado o padrão MPI e o agendamento, sempre que possível, de mais de uma tarefa por nó.

7.1 Trabalhos Futuros

A tese apresentada não cobre todas as possibilidades de comparação, havendo outros pontos interessantes a serem observados. Um desses pontos diz respeito ao escalonamento das tarefas; as aplicações em ambientes de cluster geralmente não realizam o balanceamento de carga dinâmico, sendo feito estaticamente através da divisão das tarefas a serem processadas entre os nós do cluster antes da sua execução. Essa possibilidade elimina o escalonamento das tarefas e a transferência de arquivos entre os nós do cluster podendo diminuir o tempo de processamento, uma vez que a configuração do cluster é conhecida. A grade, por utilizar software genérico projetado para trabalhar sobre ambientes heterogêneos sempre faz o balanceamento de carga dinamicamente.

Outro ponto a ser estudado é a utilização das propriedades das imagens para agrupá-las em pacotes, na tentativa de conseguir balanceamento de carga melhor. Os pacotes utilizados nos experimentos foram agrupados aleatoriamente, na ordem em que as imagens foram armazenadas no disco rígido. Logo, alguns pacotes são muito maiores do que outros, não sendo, claramente, a distribuição de imagens por pacote utilizada a mais adequada. Assim, a investigação sobre a melhor forma de agrupar as imagens em pacotes depende das propriedades e características de cada imagem, sendo necessária a criação de modelos que relacionem esses fatores com o tempo de processamento delas. Um modelo simples seria agrupá-las levando em conta o tamanho do pacote e não a quantidade de imagens, por exemplo.

Durante os experimentos realizados na grade, tentou-se simular uma carga computacional extra ao processamento das imagens com a execução de um vídeo. Porém, seria mais real utilizar essas cargas randomicamente, simulando, de fato, a utilização do computador pelo seu dono e mostrando o comportamento da grade em situações onde os usuários estão executando outras tarefas simultaneamente. Seguindo essa linha, a utilização de nós externos obtidos de outras grades na estrutura OurGrid também é um ponto a ser explorado. Não está claro se o aumento na latência da rede e a diminuição da largura de banda causariam impacto na utilização de nós externos sobre a Internet.

ANEXO 1 – CONFIGURAÇÃO DO OURGRID 3.3

Arquivos de Configuração e Log

A – Componente Peer

<peer.properties>

```
#+++++++++++++++++++++++++++++++++++++++
# Mandatory (there is no default values)
#+++++++++++++++++++++++++++++++++++++++

#address used by the peer to publish it's rmi services
peer.name = 10.0.0.221

#The port number that will be used by Ourgrid peer
peer.port = 3081

#The peer administrator e-mail
peer.email = giorgiamattos@uol.com.br


#+++++++++++++++++++++++++++++++++++++++++++++++++++++
# Optional (default values listed below will be used)
#+++++++++++++++++++++++++++++++++++++++++++++++++++++

#indicates if the peer should join the community, becoming available to
#other peers
peer.joincommunity = yes


#+++++++++++++++++++++++++++++++++++++++++++++++++++++
# Optional (no default values will be used)
#+++++++++++++++++++++++++++++++++++++++++++++++++++++

#address used by other peers to contact this peer
#peer.externalname =

#A symbolic peer name
#peer.label = PEER LACRI

#a short description about the peer site
#peer.description =

#peer.corepeername = corepeer.lsd.ufcg.edu.br
#peer.corepeerport = 3087

#A default SDF file to automatically set the GuMs on the Peer
#initialization
peer.default.sdf = /home/giorgia/peer/mysite.sdf

#The local latitude
#peer.latitude =

#The local longitude
#peer.longitude =
```

```
#Security options, ALL must be set for security to work. These options
#are OFF be default
#peer.security = no
#peer.secureport =
#peer.keystorefilename =
#peer.keystorepassword =
#peer.truststorefilename =
#peer.truststorepassword =
```

<mysite.sdf>

```
gumdefaults:
    site: mysite.com
    type: ualinux
    port: 3080
    os: linux
    remExec : ssh -x $machine $command
    copyFrom : scp $machine:$remotefile $localfile
    copyTo : scp $localfile $machine:$remotefile

gum:
    name: 10.0.0.221
gum:
    name: 10.0.0.222
gum:
    name: 10.0.0.223
gum:
    name: 10.0.0.224
gum:
    name: 10.0.0.225
gum:
    name: 10.0.0.226
gum:
    name: 10.0.0.227
gum:
    name: 10.0.0.228
```

B – Componente MyGrid

<mg.properties>

```
File 'mg.properties' was not found at '/home/giorgia/.mygrid'!
Creating a new one...
1/7: Home machine name: [cluster-01]
2/7: RMI port number: [1099]
3/7: Scheduling heuristic (workqueue/storageaffinity): [storageaffinity]
4/7: How many replicas of a task should Mygrid try to run simultaneously?
[1]
5/7: How many times should Mygrid try to execute a given task before
considering it has failed? [1]
6/7: How many times may a machine fail within a job before it is
blacklisted? [1]
7/7: Save Data Discovery history (for use with Storage Affinity only)?
[yes]
```

<mypeer.gdf>

```
peer:
    name: 10.0.0.221
    label: PEER LACRI
    port: 3081
```

<mygrid.log>

```
[2008/02/20 16:56:30:514] DEBUG
org.ourgrid.mygrid.scheduler.EBSchedulerFacade.<init> ==> The
EBSchedulerFacade has been successfully created.
[2008/02/20 16:56:30:617] DEBUG
org.ourgrid.mygrid.scheduler.BlackListManager.<init> ==> Black List
Manager created
[2008/02/20 16:56:30:618] DEBUG
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.setMaxReplicas ==>
Max replicas was set to '1'
[2008/02/20 16:56:30:619] DEBUG
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.setMaxFails ==> Max
fails was set to '1'
[2008/02/20 16:56:30:621] DEBUG
org.ourgrid.mygrid.scheduler.EBSchedulingHeuristicFactory.getHeuristic
==> Loading StorageAffinity scheduler...
[2008/02/20 16:56:30:653] DEBUG
org.ourgrid.mygrid.scheduler.SchedulerEventEngine.startProcessing ==> The
SchedulerEventEngine was successfully started.
[2008/02/20 16:56:30:666] INFO
org.ourgrid.mygrid.ui.MyGridUIManager.startMyGridService ==> MyGrid 3.3.1
is up and running.
[2008/02/20 16:57:58:534] DEBUG
org.ourgrid.common.spec.main.CommonCompiler.run ==> Compilation started
for source "/home/giorgia/mygrid/mypeer.gdf"
```

```
[2008/02/20 16:57:58:556] DEBUG
org.ourgrid.common.spec.main.CommonCompiler.compile ==> Compilation
finished successfully!
[2008/02/20 16:57:58:561] DEBUG
org.ourgrid.mygrid.ui.MyGridUIManager.initRMIRegistry ==> Cannot start
RMI registry at port: 1099
[2008/02/20 16:57:58:575] DEBUG
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realSetPeers ==>
Peer Spec [rmi://192.168.0.117:3081/LOCAL_ACCESS,PEER III TELEMATICA] has
been set.
[2008/02/20 16:57:58:576] ERROR
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realSetPeers ==>
The Peer labeled PEER III TELEMATICA at
rmi://192.168.0.117:3081/LOCAL_ACCESS is not UP now, but MyGrid will be
trying to contact it
[2008/02/20 16:57:58:621] INFO
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realPeerAlive ==>
Peer 192.168.0.117 is rejoined at URL
rmi://192.168.0.117:3081/LOCAL_ACCESS
[2008/02/20 16:59:40:931] DEBUG
org.ourgrid.common.spec.main.CommonCompiler.run ==> Compilation started
for source "/home/giorgia/pacotes100/filtros-100.jdf"
[2008/02/20 16:59:41:312] DEBUG
org.ourgrid.common.spec.main.CommonCompiler.compile ==> Compilation
finished successfully!
[2008/02/20 16:59:41:576] DEBUG
org.ourgrid.mygrid.scheduler.BlackListManager.initiateBlackListForJob ==>
Black List has been initiated for the job <2> with 212 tasks
[2008/02/20 16:59:41:579] DEBUG
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realNeedGums ==>
Asking 212 GuMs to Proxy[Gump,RemoteObjectInvocationHandler[UnicastRef
[liveRef:
[endpoint:[192.168.0.117:3081](remote),objID:[6556b1d5:118386c1dfb:-8000,
2]]]]]
[2008/02/20 16:59:41:627] DEBUG
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realNeedGums ==>
Request 3780962955580717057 successfully done to peer
Proxy[Gump,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[192.168.0.117:3081](remote),objID:[6556b1d5:118386c1dfb:-8000,
2]]]]]
[2008/02/20 16:59:41:630] INFO
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realAddJob ==> Job
'2' was added.
[2008/02/20 16:59:41:870] DEBUG
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realHereIsGum ==>
The EBGridManager has received a new Gum
(192.168.0.117:3080@192.168.0.117:3081 ) from Peer
rmi://192.168.0.117:3081/LOCAL_ACCESS for request 3780962955580717057
[2008/02/20 16:59:41:935] DEBUG
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realCreateNewReplica
==> Replica '2.1.1' was created
[2008/02/20 16:59:42:012] INFO
org.ourgrid.mygrid.replicaexecutor.EBReplicaManager.executeReplica ==>
About to start 2.1.1
[2008/02/20 16:59:42:015] INFO
org.ourgrid.mygrid.scheduler.StorageAffinity.schedule ==>
192.168.0.117:3080@192.168.0.117:3081 was assigned to execute replica
2.1.1
```

```
[2008/02/20 16:59:42:080] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.1.1 init phase execution started.
[2008/02/20 16:59:42:086] DEBUG
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.1.1 about put file '/home/giorgia/pacotes100/filtros' to
'filtros'
[2008/02/20 16:59:42:087] INFO
org.ourgrid.gridmachine.useragent.UserAgentClient.putFile ==> Sending the
file filtros to /tmp/mygrid-1163491837423576064/filtros at
192.168.0.117:3080@192.168.0.117:3081
[2008/02/20 16:59:42:422] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.1.1 put file /home/giorgia/pacotes100/filtros to /tmp/mygrid-
1163491837423576064/filtros sucessfully.
[2008/02/20 16:59:42:423] DEBUG
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.1.1 about put file '/home/giorgia/pacotes100/1.org' to '1.org'
[2008/02/20 16:59:42:423] INFO
org.ourgrid.gridmachine.useragent.UserAgentClient.putFile ==> Sending the
file 1.org to /tmp/mygrid-1163491837423576064/1.org at
192.168.0.117:3080@192.168.0.117:3081
[2008/02/20 16:59:44:384] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.1.1 put file /home/giorgia/pacotes100/1.org to /tmp/mygrid-
1163491837423576064/1.org sucessfully.
[2008/02/20 16:59:44:385] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.1.1 init phase execution completed: ExitValue: 0 StdOut: Init
phase OK. StdErr: null
[2008/02/20 16:59:44:385] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.remotePhase ==>
Replica 2.1.1 remote phase execution started.
[2008/02/20 17:01:12:821] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.remotePhase ==>
Replica 2.1.1 remote phase execution completed:  ExitValue: 0 StdOut:
StdErr:
[2008/02/20 17:01:12:823] DEBUG
org.ourgrid.mygrid.replicaexecutor.PermissionManager.requestPermission
==> Permission request from replica 2.1.1
[2008/02/20 17:01:12:828] DEBUG
org.ourgrid.mygrid.replicaexecutor.PermissionManager.requestPermission
==> Permission granted to replica 2.1.1
[2008/02/20 17:01:12:828] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.finalPhase ==>
Replica 2.1.1 final phase execution started.
[2008/02/20 17:01:12:858] INFO
org.ourgrid.gridmachine.useragent.UserAgentClient.getFile ==> Sending
file /tmp/mygrid-1163491837423576064/1.pro
[2008/02/20 17:01:14:346] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.finalPhase ==>
Replica 2.1.1 final phase execution completed: ExitValue: 0 StdOut: final
phase OK. StdErr: null
[2008/02/20 17:01:14:346] INFO
org.ourgrid.common.util.TimeDataGenerator.report ==> Replica 2.1.1 Init
phase duration:  2304 ms
[2008/02/20 17:01:14:347] INFO
org.ourgrid.common.util.TimeDataGenerator.report ==> Replica 2.1.1
assigned to 192.168.0.117:3080@192.168.0.117:3081 Remote phase duration:
88435 ms
```

```
[2008/02/20 17:01:14:347] INFO
org.ourgrid.common.util.TimeDataGenerator.report ==> Replica 2.1.1 Final
phase duration:  1517 ms
[2008/02/20 17:01:14:357] DEBUG
org.ourgrid.mygrid.replicaexecutor.EBReplicaManager.replicaFinished ==>
Removing thread of finished replica:2.1.1
[2008/02/20 17:01:14:357] DEBUG
org.ourgrid.mygrid.replicaexecutor.EBReplicaManager.replicaFinished ==>
Attepting to abort the other replicas of task 2.1
[2008/02/20 17:01:14:358] INFO
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realReplicaFinished
==> Replica 2.1.1 finished on '192.168.0.117:3080@192.168.0.117:3081'
with status 'FINISHED'
[2008/02/20 17:01:14:358] DEBUG
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realReplicaFinished
==>
Init result: ExitValue: 0 StdOut: Init phase OK. StdErr: null
Remote result: ExitValue: 0 StdOut:  StdErr:
Final result: ExitValue: 0 StdOut: final phase OK. StdErr: null
[2008/02/20 17:01:14:386] INFO
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realGumIsReady ==>
The GuM '192.168.0.117:3080@192.168.0.117:3081' was released
[2008/02/20 17:01:14:463] DEBUG
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realCreateNewReplica
==> Replica '2.2.1' was created
[2008/02/20 17:01:14:577] INFO
org.ourgrid.mygrid.replicaexecutor.EBReplicaManager.executeReplica ==>
About to start 2.2.1
[2008/02/20 17:01:14:584] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.2.1 init phase execution started.
[2008/02/20 17:01:14:584] DEBUG
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.2.1 about put file '/home/giorgia/pacotes100/filtros' to
'filtros'
[2008/02/20 17:01:14:585] INFO
org.ourgrid.gridmachine.useragent.UserAgentClient.putFile ==> Sending the
file filtros to /tmp/mygrid-6550886404051515392/filtros at
192.168.0.117:3080@192.168.0.117:3081
[2008/02/20 17:01:14:615] INFO
org.ourgrid.mygrid.scheduler.StorageAffinity.schedule ==>
192.168.0.117:3080@192.168.0.117:3081 was assigned to execute replica
2.2.1
[2008/02/20 17:01:14:759] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.2.1 put file /home/giorgia/pacotes100/filtros to /tmp/mygrid-
6550886404051515392/filtros sucessfully.
[2008/02/20 17:01:14:760] DEBUG
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.2.1 about put file '/home/giorgia/pacotes100/2.org' to '2.org'
[2008/02/20 17:01:14:760] INFO
org.ourgrid.gridmachine.useragent.UserAgentClient.putFile ==> Sending the
file 2.org to /tmp/mygrid-6550886404051515392/2.org at
192.168.0.117:3080@192.168.0.117:3081
[2008/02/20 17:01:15:933] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
Replica 2.2.1 put file /home/giorgia/pacotes100/2.org to /tmp/mygrid-
6550886404051515392/2.org sucessfully.
[2008/02/20 17:01:15:933] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.initPhase ==>
```

```
Replica 2.2.1 init phase execution completed: ExitValue: 0 StdOut: Init
phase OK. StdErr: null
[2008/02/20 17:01:15:934] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.remotePhase ==>
Replica 2.2.1 remote phase execution started.
[2008/02/20 17:02:42:959] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.remotePhase ==>
Replica 2.2.1 remote phase execution completed:  ExitValue: 0 StdOut:
StdErr:
[2008/02/20 17:02:42:959] DEBUG
org.ourgrid.mygrid.replicaexecutor.PermissionManager.requestPermission
==> Permission request from replica 2.2.1
[2008/02/20 17:02:42:960] DEBUG
org.ourgrid.mygrid.replicaexecutor.PermissionManager.requestPermission
==> Permission granted to replica 2.2.1
[2008/02/20 17:02:42:960] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.finalPhase ==>
Replica 2.2.1 final phase execution started.
[2008/02/20 17:02:42:978] INFO
org.ourgrid.gridmachine.useragent.UserAgentClient.getFile ==> Sending
file /tmp/mygrid-6550886404051515392/2.pro
[2008/02/20 17:02:44:128] INFO
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThread.finalPhase ==>
Replica 2.2.1 final phase execution completed: ExitValue: 0 StdOut: final
phase OK. StdErr: null
[2008/02/20 17:02:44:128] INFO
org.ourgrid.common.util.TimeDataGenerator.report ==> Replica 2.2.1 Init
phase duration:  1349 ms
[2008/02/20 17:02:44:129] INFO
org.ourgrid.common.util.TimeDataGenerator.report ==> Replica 2.2.1
assigned to 192.168.0.117:3080@192.168.0.117:3081 Remote phase duration:
87025 ms
[2008/02/20 17:02:44:129] INFO
org.ourgrid.common.util.TimeDataGenerator.report ==> Replica 2.2.1 Final
phase duration:  1168 ms
[2008/02/20 17:02:44:135] DEBUG
org.ourgrid.mygrid.replicaexecutor.EBReplicaManager.replicaFinished ==>
Removing thread of finished replica:2.2.1
[2008/02/20 17:02:44:136] DEBUG
org.ourgrid.mygrid.replicaexecutor.EBReplicaManager.replicaFinished ==>
Attepting to abort the other replicas of task 2.2
[2008/02/20 17:02:44:136] INFO
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realReplicaFinished
==> Replica 2.2.1 finished on '192.168.0.117:3080@192.168.0.117:3081'
with status 'FINISHED'
[2008/02/20 17:02:44:136] DEBUG
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realReplicaFinished
==>
Init result: ExitValue: 0 StdOut: Init phase OK. StdErr: null
Remote result: ExitValue: 0 StdOut:  StdErr:
Final result: ExitValue: 0 StdOut: final phase OK. StdErr: null
[2008/02/20 17:02:44:147] INFO
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.realGumIsReady ==>
The GuM '192.168.0.117:3080@192.168.0.117:3081' was released
[2008/02/20 17:02:44:196] DEBUG
org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager.realCreateNewReplica
==> Replica '2.3.1' was created
.
.
.
```

```
[2008/02/21 05:09:22:958] INFO
org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager.cancelRequests ==>
All requests and Gums related to job '2' were removed.
[2008/02/21 07:41:35:181] DEBUG
org.ourgrid.mygrid.ui.MyGridUIManager.initRMIRegistry ==> Cannot start
RMI registry at port: 1099
[2008/02/21 07:41:35:425] INFO
org.ourgrid.mygrid.ui.command.Stopper.run ==> Shutting down MyGrid 3.3.1
[2008/02/21 07:41:35:536] DEBUG
org.ourgrid.mygrid.scheduler.SchedulerEventEngine.unbindAndUnexportRMIObj
ects ==> rmi://ubuntu04:1099/GRID_MANAGER is already unbound:GRID_MANAGER
[2008/02/21 07:41:35:565] DEBUG
org.ourgrid.mygrid.replicaexecutor.EBReplicaManager.shutdown ==> The
EBReplicaManager is going down
[2008/02/21 07:41:35:566] DEBUG
org.ourgrid.mygrid.replicaexecutor.ReplicaExecutorThreadManager.shutdown
==> The ThreadManager is going down
[2008/02/21 07:41:35:567] DEBUG
org.ourgrid.mygrid.replicaexecutor.PermissionManager.shutdown ==> The
PermissionManager is going down
[2008/02/21 07:41:36:137] INFO
org.ourgrid.mygrid.ui.command.Stopper.run ==> MyGrid was successfully
stopped!
```

C – Componente UserAgent

<ua.properties>

```
#the grid machine fully qualified name
ua.name = 10.0.0.221

#Public address by which the machine can be contacted
#ua.externalname =

#the port number where the user agent is listen requests. Must be equals
#to port atribute from the gum description
ua.port = 3080

#the root directory where playpens are created
ua.playpenroot = /tmp

#the playpen size
ua.playpensize = NO_LIMIT

#Directory used for storage
ua.storagedir = .mgstorage

#the storage size
ua.storagesize = NO_LIMIT

#it turns on the security feature
#(options: yes or no)
#ua.security = no

#tells if the daemon that checks the computer idleness have to be
#initialized with the UA
#(options: yes or no)
#ua.idlenessdetector = no

#How many time the computer shall be idle to initialize the UA services
#in seconds
#ua.idlenesstime = 1800

#Security options, ALL must be set for security to work. These options
#are OFF be default
#ua.security = no
#ua.secureport =
#ua.keystorefilename =
#ua.keystorepassword =
#ua.truststorefilename =
#ua.truststorepassword =
```

ANEXO 2 – CONFIGURAÇÃO DO CLUSTER-SCALA

Arquivos Fonte, de Configuração e Log

A – Componente Mestre

<Server.scala>

```
//
// Server.scala
// The BigBatch Cluster Server
//
// Andrei de A. Formiga, 2007-03-05
//

package br.ufpe.ee.bigbatch.cluster.server;

import scala.actors.Actor;
import scala.actors.Actor._;
import scala.actors.remote.RemoteActor;
import scala.actors.remote.Node;
import scala.collection.mutable.Queue;
import scala.collection.mutable.HashMap;
import org.apache.log4j.Logger;
import org.apache.log4j.Level;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.FileAppender;
import org.apache.log4j.PatternLayout;


// Messages sent to the scheduler
case class ClientAvailable(id: int)
case object Schedule

// The Server object
object Server
{
  val name = "BigBatchClusterServer"
  val port = 9010

  private val freeClients = new Queue[int]
  private val clientMap = new HashMap[int, ClientInfo]
  private val logger = Logger.getLogger("BigBatch.Cluster.Server")

  // next id value for clients
  private var nextId = 0

  val handler = actor
  {
    RemoteActor.alive(ClusterParameters.serverPort)
    RemoteActor.register(Symbol(name), self)

    // message loop
    while (true)
    {
      receive
```

```
{
  case Test(m) => System.out.println(m)
  case Info(m) => logger.info(m)

  case ToClient(m, cid) =>
    logger.info("Sending message to client")
    try
    {
      val c = clientMap(cid)
      c.actor ! Test(m)
    }
    catch
    {
      case e: java.io.IOException =>
        logger.error("Error sending to client: " + e.getMessage())
    }

  case Register(n, a) =>
    logger.info("Registered: " + n + " - id: " + nextId)
    val clientActor = RemoteActor.select(Node(a,
                                  ClusterParameters.clientPort),
                                  Symbol(n))
    clientMap(nextId) = new ClientInfo(n, nextId, clientActor)
    clientActor ! Registered(n, nextId)

    self ! ClientAvailable(nextId)
    nextId += 1
    self ! Schedule

  case File(name, bytes, tid) =>
    FileMessageBuilder.fileFromMessage(name, bytes)
  case TaskCompleted(tid, cid) =>
    val d = new java.util.Date()
    logger.info("Client " + cid + " finished task " + tid)
    self ! ClientAvailable(cid)
    self ! Schedule

  case Terminate =>
    logger.info("BigBatch Cluster Server terminated")
    System.exit(0)

  // scheduling messages
  case ClientAvailable(id) => freeClients += id

  case FileReceived(cid, tid) =>
    logger.info("Client " + cid + " confirmed reception of file for
                 task " + tid)
    val c = clientMap(cid)
    val t = BigBatchTasks.getTask(tid)
    c.actor ! (t.toMessage)
    logger.info("Sent task " + tid + " to client " + cid)

  case Schedule =>
    while (!freeClients.isEmpty)
    {
      val cid = freeClients.dequeue
      val c = clientMap(cid)
      val tid = BigBatchTasks.getNext
      logger.info("Task " + tid + " assigned to client " + cid)
      try
```

```
          {
            val f = BigBatchTasks.fileMessage(tid)
            logger.info("Sending file for task " + tid + " to client "
                        + cid)
            c.actor ! f  // TODO: what to do if file is never received?
            logger.info("Sent file for task " + tid)             }
          catch
          {
            case e: java.io.IOException =>
              logger.error("Could not assign task " + tid +
                           ". Message: " + e.getMessage())
            // TODO : what action to take when the task is not created?
          }
        }
    }
  }
}

// TODO: delete the scheduler actor
val scheduler = actor
{
  while (true)
  {
    receive
    {
      case ClientAvailable(id) => freeClients += id
      case Schedule =>
        logger.info("Scheduler called")
        if (!freeClients.isEmpty)     // TODO: this should be a while
        {
          val cid = freeClients.dequeue
          val c = clientMap(cid)
          val tid = BigBatchTasks.getNext
          //val actor = RemoteActor.select(Node(c.addr,
                                    ClusterParameters.clientPort),
          //                        Symbol(c.name))
          logger.info("Task " + tid + " assigned to client " + cid)
          try
          {
            sender ! ToClient("Testing...", cid)
            //BigBatchTasks.assignTask(tid, c.actor)
          }
          catch
          {
            case e: java.io.IOException =>
              logger.error("Could not assign task " + tid +
                           ". Message: " + e.getMessage())
            // TODO : what action to take when the task is not created?
          }
        }
    }
  }
}

def main(args: Array[String]) = {
  BasicConfigurator.configure()

  val appender = new FileAppender(
                 new PatternLayout("[%d{dd MMM yyyy HH:mm:ss,SSS}] %-5p
                                    %c - %m - %r%n"), "bbcserver.log")
```

```
    logger.addAppender(appender)
    logger.setLevel(Level.INFO)
    logger.info("BigBatch Cluster Server started")
  }
}
```

<TerminateServer.scala>

```
package br.ufpe.ee.bigbatch.cluster.server;

import scala.actors.Actor;
import scala.actors.Actor._;
import scala.actors.remote._;


object TerminateServer
{
  def main(args: Array[String]) = {
    val server = RemoteActor.select(Node("127.0.0.1", Server.port),
                                          Symbol(Server.name))
    System.out.println("OK")
    server ! Terminate
    System.exit(0)
  }
}
```

B – Componente Cliente

<ClienteAgent.scala>

```
//
// ClientAgent.scala
// The BigBatch Cluster client agent
//
// Andrei de A. Formiga, 2007-03-05
//

package br.ufpe.ee.bigbatch.cluster.client;

import scala.actors.Actor;
import scala.actors.Actor._;
import scala.actors.remote._;
import java.net.InetAddress
import org.apache.log4j.Logger;
import org.apache.log4j.Level;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.FileAppender;
import org.apache.log4j.PatternLayout;


class ClientAgent(name: String, serverNode: Node)
{
  private var id = -1

  val logger = Logger.getLogger("BigBatch.Cluster.Client")

  val agent = actor
  {
    // logger initialization
    val appender = new FileAppender(
                   new PatternLayout("[%d{dd MMM yyyy HH:mm:ss,SSS}] %-5p
                                      %c - %m - %r%n"), "bbcclient.log")
    logger.addAppender(appender)
    logger.setLevel(Level.INFO)
    logger.info("Cluster client " + name + " started")

    // remote actor initialization
    RemoteActor.alive(ClusterParameters.clientPort)
    RemoteActor.register(Symbol(name), self)

    // server communication initialization
    val addr = InetAddress.getLocalHost().getHostAddress()
    val server = RemoteActor.select(serverNode,
                                    Symbol("BigBatchClusterServer"))
    server ! Register(name, addr)

    while (true) {
      receive {
        case Test(m) => System.out.println(m)
        case Info(m) => logger.info(m)
        case Registered(n, cid) =>
          logger.info("Client registered with id " + cid)
          id = cid
        case Terminate =>
          logger.info("Client " + name + " (" + id + ") terminated")
```

```
          // TODO: unregister client
          System.exit(0)
        case File(name, bytes, tid) =>
          FileMessageBuilder.fileFromMessage(name, bytes)
          logger.info("Received file " + name + " from server")
          server ! FileReceived(id, tid)
        case TaskAssignment(tid, args) =>
          val t = new Task(tid, args)
          logger.info("Task " + tid + " received, execing: " +
                      t.getCommandLine)
          try
          {
            t.exec()
            logger.info("Task " + tid + " completed, sending result...")
            val f = FileMessageBuilder.messageFromFile(
                    BigBatchTasks.resFileFromPack(tid), tid)
            server ! f
            server ! TaskCompleted(tid, id)
            logger.info("File transfer for task " + tid + " completed")
          }
          catch
          {
            case e: java.io.IOException =>
              logger.error("Error executing task " + tid +
                           ". Message: " + e.getMessage())
            // TODO : what action to take when the task is not created?
          }

        case x => logger.error("Unrecognized message received by client :
                               " + x)
      }
    }
  }
}

object Main
{
  def main(args: Array[String]) = {
    if (args.length < 1)
    {
      System.err.println("Parâmetros: <nome> <servidor>")
      System.exit(1)
    }

    BasicConfigurator.configure()

    val name = args(0)
    val agent = new ClientAgent(name, Node(args(1),
                                ClusterParameters.serverPort))
    System.out.println("---")
  }
}
```

<TerminateClient.scala>

```
package br.ufpe.ee.bigbatch.cluster.client;

import scala.actors.Actor;
import scala.actors.Actor._;
import scala.actors.remote._;


object TerminateClient
{
  def main(args: Array[String]) = {
    if (args.length < 1)
    {
      System.err.println("Especifique o cliente")
      System.exit(1)
    }

    val agent = RemoteActor.select(Node("127.0.0.1",
                                        ClusterParameters.clientPort),
                                        Symbol(args(0)))
    System.out.println("OK")
    agent ! Terminate
    System.exit(0)
  }
}
```

C – Arquivos de configuração e log

<BigBatchTasks.scala>

```
package br.ufpe.ee.bigbatch.cluster;

import scala.actors.Actor;

object BigBatchTasks {
  val states = new Array[boolean](214);

  private val program = "./filtros"
  private val packSuffix = "00"
  private val packFileSuffix = "00.org"
  private val resFileSuffix = "00.pro"

  def packFileFromPack(pack: int) = pack + packFileSuffix

  def resFileFromPack(pack: int) = pack + resFileSuffix

  def getNext = {
    // disable task 75
    states(75) = true

    var i = 1
    while (states(i))
      i += 1
    states(i) = true
    i
  }

  def remoteFileName(srvAddr: String, filePath: String) =
                     srvAddr + ":" + filePath

  def fileMessage(pack: int) =
  {
    val packFile = packFileFromPack(pack)
    val resFile = resFileFromPack(pack)
    FileMessageBuilder.messageFromFile(packFile, pack)
  }

  def taskCmdArgs(pack: int) = Array(program, pack + packSuffix)

  def getTask(pack: int) = new Task(pack, taskCmdArgs(pack))
}
```

<ClientInfo.scala>

```
package br.ufpe.ee.bigbatch.cluster;

import scala.actors.Actor;

case class ClientInfo(name: String, id: int, actor: Actor);
```

<ClusterParameters.scala>

```
package br.ufpe.ee.bigbatch.cluster;

object ClusterParameters
{
  val serverPort = 9010
  val clientPort = 9011
}
```

<Message.scala>

```
// Message.scala
// Definitions for messages exchanged between client and server

package br.ufpe.ee.bigbatch.cluster;

import java.io.IOException;

abstract class Message

case class  Test(message: String) extends Message
case class  ToClient(message: String, cid: int) extends Message
case class  Info(message: String) extends Message
case class  Register(name: String, addr: String) extends Message
case class  Registered(name: String, id: int) extends Message
case object Terminate extends Message
case class  TaskAssignment(id: int, cmd: Array[String]) extends Message
case class  TaskAck(id: int, task: Task) extends Message
case class  TaskCompleted(tid: int, cid: int) extends Message
case class  File(name: String, bytes: Array[byte], tid: int) extends
                 Message
case class  FileReceived(cid: int, tid: int) extends Message

object FileMessageBuilder
{
  def messageFromFile(name: String, tid: int) = {
    //System.out.println("Iniciando mensagem")
    var in : java.io.FileInputStream = null
    try {
      in = new java.io.FileInputStream(name)
      val bytes = new Array[byte](in.getChannel().size().toInt)
      // TODO: verify overflow?
      in.read(bytes)
      in.close()
      //System.out.println("Mensagem criada")
      File(name, bytes, tid)
    }
    catch
    {
      case e: IOException => throw e
    }
    finally {
      if (in != null)
        in.close()
    }
  }
  def fileFromMessage(f: File) = {
```

```
    val out = new java.io.FileOutputStream(f.name)
    out.write(f.bytes)
    out.close()
  }
  def fileFromMessage(name: String, bytes: Array[byte]) = {
    val out = new java.io.FileOutputStream(name)
    out.write(bytes)
    out.close()
  }
}
```

<Task.scala>

```
package br.ufpe.ee.bigbatch.cluster;

class Task(id: int, cmd: Array[String])
{
  def exec() : unit = {
    try
    {
      val builder = new ProcessBuilder(cmd)
      builder.directory(new java.io.File(System.getProperty("user.dir")))
      val proc = builder.start()
      proc.waitFor()
    }
    catch
    {
      case e: java.io.IOException => throw e
    }
  }

  def toMessage = TaskAssignment(id, cmd)

  def getCommandLine = cmd.toString()
}

class FileTask(id: int, orig: String, dest: String)
extends Task(id, Array("scp", orig, dest))
{
}

class BigBatchTask(id: int, putT: FileTask, getT: FileTask, task: Task)
{
}

object TestTask
{
  val task = new Task(01, Array("shlomo"))
}
```

<bbcserver.log>

```
[01 Mai 2007 21:49:54,801] INFO  BigBatch.Cluster.Server - BigBatch
Cluster Server started - 0
[01 Mai 2007 21:50:01,339] INFO  BigBatch.Cluster.Server - Registered:
cluster1 - id: 0 - 6538
[01 Mai 2007 21:50:01,356] INFO  BigBatch.Cluster.Server - Task 1
assigned to client 0 - 6555
[01 Mai 2007 21:50:01,434] INFO  BigBatch.Cluster.Server - Sending file
for task 1 to client 0 - 6633
[01 Mai 2007 21:50:02,102] INFO  BigBatch.Cluster.Server - Sent file for
task 1 - 7301
[01 Mai 2007 21:50:02,297] INFO  BigBatch.Cluster.Server - Client 0
confirmed reception of file for task 1 - 7496
[01 Mai 2007 21:50:02,307] INFO  BigBatch.Cluster.Server - Sent task 1 to
client 0 - 7506
[01 Mai 2007 21:52:33,833] INFO  BigBatch.Cluster.Server - Client 0
finished task 1 - 159032
[01 Mai 2007 21:52:33,833] INFO  BigBatch.Cluster.Server - Task 2
assigned to client 0 - 159032
[01 Mai 2007 21:52:34,852] INFO  BigBatch.Cluster.Server - Sending file
for task 2 to client 0 - 160051
[01 Mai 2007 21:52:35,512] INFO  BigBatch.Cluster.Server - Sent file for
task 2 - 160711
[01 Mai 2007 21:52:35,807] INFO  BigBatch.Cluster.Server - Client 0
confirmed reception of file for task 2 - 161006
[01 Mai 2007 21:52:35,809] INFO  BigBatch.Cluster.Server - Sent task 2 to
client 0 - 161008
[01 Mai 2007 21:55:11,375] INFO  BigBatch.Cluster.Server - Client 0
finished task 2 - 316574
[01 Mai 2007 21:55:11,375] INFO  BigBatch.Cluster.Server - Task 3
assigned to client 0 - 316574
[01 Mai 2007 21:55:11,438] INFO  BigBatch.Cluster.Server - Sending file
for task 3 to client 0 - 316637
[01 Mai 2007 21:55:12,120] INFO  BigBatch.Cluster.Server - Sent file for
task 3 - 317319
[01 Mai 2007 21:55:12,284] INFO  BigBatch.Cluster.Server - Client 0
confirmed reception of file for task 3 - 317483
[01 Mai 2007 21:55:12,288] INFO  BigBatch.Cluster.Server - Sent task 3 to
client 0 - 317487
[01 Mai 2007 21:57:41,645] INFO  BigBatch.Cluster.Server - Client 0
finished task 3 – 466844
.
.
.

[02 Mai 2007 00:20:07,767] INFO  BigBatch.Cluster.Server - Task 60
assigned to client 0 - 9012966
[02 Mai 2007 00:20:08,204] INFO  BigBatch.Cluster.Server - Sending file
for task 60 to client 0 - 9013403
[02 Mai 2007 00:20:09,238] INFO  BigBatch.Cluster.Server - Sent file for
task 60 - 9014437
[02 Mai 2007 00:20:09,350] INFO  BigBatch.Cluster.Server - Client 0
confirmed reception of file for task 60 - 9014549
[02 Mai 2007 00:20:09,352] INFO  BigBatch.Cluster.Server - Sent task 60
to client 0 - 9014551
[02 Mai 2007 00:22:51,812] INFO  BigBatch.Cluster.Server - Client 0
finished task 60 - 9177011
```

ANEXO 3 – CONFIGURAÇÃO DO CLUSTER MPI

Arquivos Fonte e Log

A – Componente cluster-MPI

```
/**
 * BigBatch
 * Processamento de Documentos em clusters usando MPI
 *
 * Andrei de A. Formiga, 08-2008
 * Giorgia de Oliveira Mattos, 08-2008
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include "mpi.h"


/*** constantes ****************************************/

#define TRUE                    1
#define FALSE                   0
#define MAX_LINE_LENGTH         60
#define EXE_NAME                "filtros.exe"


/*** variaveis globais *********************************/

/* valor para tag da mensagem */
int tag = 0;


/*** funcoes auxiliares ********************************/
int content_pres(char *line)
{
  char *c;
  int len = strlen(line);

  for (c = line; *c; ++c)
    if (!isspace(*c))
      return TRUE;

  return FALSE;
}

int content_present(char *trimmed_line)
{
  return (trimmed_line[0] != '\0');
}

char *trim_line(char *line)
{
```

```
  static char trimmed_line[MAX_LINE_LENGTH];
  char *l, *tl;

  for (l = line, tl = trimmed_line; *l; ++l)
  {
    if (!isspace(*l))
       *tl++ = *l;
  }

  *tl = '\0';

  return trimmed_line;
}

obtem_horario ()
{
    time_t     now;
    struct tm  ts;
    char       buf[80];

    // Obtem o horário corrente
    time(&now);

    // Formata e imprime o horario, "ddd yyyy-mm-dd hh:mm:ss"
    ts = *localtime(&now);
    strftime(buf, sizeof(buf), "%a %Y-%m-%d %H:%M:%S", &ts);
    printf("%s\n", buf);
}

void process_file_list(char *flist_name)
{
  FILE  *flist;
  char  line[MAX_LINE_LENGTH];
  char  *tline;
  int   npacks = 0;
  char  cmdline[MAX_LINE_LENGTH];

  flist = fopen(flist_name, "r");
  if (flist == NULL)
  {
    fprintf(stderr, "Erro abrindo arquivo de lista de processamento:
                     %s\n", flist_name);
    exit(2);
  }

  while (fgets(line, MAX_LINE_LENGTH, flist) != NULL)
  {
     tline = trim_line(line);
    if (content_present(tline))
    {
       printf("Processando: %s... \n", tline);
       printf ("Inicio: "); obtem_horario();
       sprintf(cmdline, "%s %s", EXE_NAME, tline);
       if (system(cmdline) == 0)
       {
        printf("Completo!\n");
        printf("Termino: "); obtem_horario();
        npacks++;
       }
       else
```

```
          printf("Erro!!!\n");
    }
  }

  MPI_Send(&npacks, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);

  fclose(flist);
}


/*** main ************************************************/
int main(int argc, char **argv)
{
  int          rank;         /* rank do processo          */
  int          np;           /* no de processadores       */
  int          nrecv = 0;    /* no de mensagens recebidas */
  int          npacks;       /* no de pacotes processados */
  MPI_Status   status;       /* status de recebimento MPI */
  char         *crank = "";

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  if (rank == 0)
  {
    // Processo Mestre
    printf("BigBatch: processando em %d nos...\n", (np-1));
    printf("Esperando finalizacao das tarefas...\n");

    while (nrecv < np - 1)
    {
      MPI_Recv(&npacks, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
               &status);
      printf("*** No %d processou %d pacotes\n", status.MPI_SOURCE,
              npacks);
      nrecv++;
    }

    printf("Todos os nos concluidos\n");
  }
  else // Processo Escravo
  {
    if (argc < 2)
      fprintf(stderr, "Especifique o arquivo de lista de
                       processamento\n");
    else
    {
       itoa(rank,crank,10);
       argv[1] = strcat(argv[1],crank);
       process_file_list(argv[1]);
    }

  }

  MPI_Finalize();

  return 0;
}
```

B – Log

```
BigBatch: processando em 16 nos...
Esperando finalizacao das tarefas...
*** No 4 processou 1 pacotes
*** No 3 processou 1 pacotes
*** No 5 processou 1 pacotes
*** No 2 processou 1 pacotes
*** No 1 processou 1 pacotes
*** No 6 processou 1 pacotes
*** No 8 processou 1 pacotes
*** No 9 processou 1 pacotes
*** No 7 processou 1 pacotes
*** No 15 processou 2 pacotes
*** No 12 processou 2 pacotes
*** No 14 processou 2 pacotes
*** No 13 processou 2 pacotes
*** No 11 processou 2 pacotes
*** No 10 processou 2 pacotes
Processando: 1...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:27:21
Processando: 2...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:27:18
Processando: 5...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:27:16
Processando: 3...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:27:11
Processando: 4...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:27:01
Processando: 12...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:37:28
Processando: 13...
Inicio: Thu 2008-10-09 17:37:28
Completo!
Termino: Thu 2008-10-09 17:53:48
Processando: 10...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:37:36
Processando: 11...
Inicio: Thu 2008-10-09 17:37:36
Completo!
Termino: Thu 2008-10-09 17:54:06
Processando: 8...
Inicio: Thu 2008-10-09 17:20:30
Completo!
```

```
Termino: Thu 2008-10-09 17:35:57
Processando: 16...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:37:13
Processando: 17...
Inicio: Thu 2008-10-09 17:37:13
Completo!
Termino: Thu 2008-10-09 17:53:34
Processando: 9...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:37:28
Processando: 14...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:37:01
Processando: 15...
Inicio: Thu 2008-10-09 17:37:01
Completo!
Termino: Thu 2008-10-09 17:53:08
Processando: 6...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:27:26
Processando: 7...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:39:49
Processando: 20...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:36:09
Processando: 21...
Inicio: Thu 2008-10-09 17:36:09
Completo!
Termino: Thu 2008-10-09 17:51:56
Processando: 18...
Inicio: Thu 2008-10-09 17:20:30
Completo!
Termino: Thu 2008-10-09 17:36:13
Processando: 19...
Inicio: Thu 2008-10-09 17:36:13
Completo!
Termino: Thu 2008-10-09 17:53:16

Todos os nos concluídos!
```

GLOSSÁRIO

Avatar:	Representação gráfica de um utilizador em realidade virtual, podendo variar desde um sofisticado modelo 3D até uma simples imagem. São normalmente pequenos, aproximadamente 100 pixels de altura por 100 pixels de largura, para que não ocupem demasiado espaço na interface, deixando espaço livre para a função principal do site, programa ou jogo que se está utilizando.
Balanceamento de carga:	Consiste em distribuir de maneira uniforme a carga de trabalho a ser processada de forma que todos os elementos envolvidos no processamento trabalhem de forma equilibrada.
Cluster:	Conjunto de computadores interligados entre si, através de uma rede local, que compartilham recursos com o objetivo de resolver um determinado problema.
Gigaflops:	Bilhões de operações aritméticas de ponto flutuante por segundo.
Grade:	Conjunto de recursos amplamente dispersos e conectados entre si, através de redes de longa distância, com o objetivo de resolver um determinado problema quando estes recursos encontram-se ociosos.
Latência de comunicação:	A latência de um sistema de comunicação é o tempo mínimo necessário para se transmitir um objeto, incluindo qualquer *overhead* necessário para transmissão e recepção.
Megaflops:	Milhões de operações aritméticas de ponto flutuante por segundo.
Multimídia:	Integração de diferentes modalidades de mídia (gráficos, imagens, textos, áudio, animação e vídeo) na representação de dados.
Operações aritméticas em ponto flutuante:	Operações aritméticas de adição, subtração, multiplicação e divisão com números reais.
Petaflops:	Quadrilhões de operações aritméticas de ponto flutuante por segundo.
Ponto flutuante:	Formato de representação digital do sistema de números reais da matemática usado nos computadores.
Processamento de imagens:	Qualquer forma de processamento de dados onde os dados de entrada e saída são imagens tais como documentos escaneados, fotografias entre outros.
Overhead:	Qualquer processamento ou armazenamento em excesso, seja de tempo de processamento, de memória, de largura de banda ou qualquer outro recurso que seja requerido para ser utilizado ou gasto para executar uma determinada tarefa. Como consequência pode

	piorar o desempenho do recurso que sofreu o overhead.
Teraflops:	Trilhões de operações aritméticas de ponto flutuante por segundo.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] KUMAR, V., GRAMA, A. et al. *Introduction to Parallel Computing: design and analysis of parallel algorithms.* The Benjamin/Cummings Publishing Company, 1994.

[2] TANENBAUM, A. S. *Organização Estruturada de Computadores*. 5. ed. Prentice Hall, 2007.

[3] STALLINGS W. *Arquitetura e Organização de Computadores: projeto para o desempenho*. 5. ed. Prentice Hall, 2002.

[4] MONTEIRO, A. M. *Introdução à Organização de Computadores*. 5. ed. LTC Editora, 2007.

[5] LAKNER, G. *IBM System Blue Gene Solution: Blue Gene/P System Administration*. 2. ed. . IBM Redbooks, 2008. Disponível em: http://www.redbooks.ibm.com/abstracts/sg247417.html. Acesso em: 23/10/2008.

[6] LASCU, O., ALLSOPP, N., VEZOLLE, P. et al. *Unfolding the IBM e-server Blue Gene Solution*. 1. ed. IBM Redbooks, 2005. Disponível em: http://www.redbooks.ibm.com/redbooks/pdfs/sg246686.pdf. Acesso em: 23/10/2008.

[7] CORRADO, M. Fact Sheet & Background: Roadrunner Smashes the Petaflop Barrier. IBM Press Releases, 2008. Disponível em: http://www-03.ibm.com/press/us/en/pressrelease/24405.wss. Acesso em: 23/10/2008.

[8] BUYYA, R. *High Performance Cluster Computing: architectures and systems, Volume 1*. Prentice Hall, 1999.

[9] FOSTER, I. KESSELMAN, C. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[10] FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, v.15, n. 3, p. 200-222.

[11] FOSTER, I.; KESSELMAN, C. The Globus project: A Status Report. In: 7th HETEROGENEOUS COMPUTING WORKSHOP (Mar. 1998 : Orlando). *Proceedings.* Orlando. p. 4-18.

[12] CHERVENAK, A.; FOSTER, I.; KESSELMAN, C.; et al. The DataGrid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, v. 23, n. 3, p. 187-200.

[13] The DataGrid Project. Disponível em: http://eu-datagrid.web.cern.ch/eu-datagrid. Acesso em: 08/11/2007.

[14] BIRN – Biomedical Informatics Research Network. Disponível em: http://www.nbirn.net. Acesso em: 13/11/2007.

[15] MAMMOGRID. European Federated Mammogram Database Implemented on a GRID Structure. Disponível em: http://www.mammogrid.com. Acesso em: 08/11/2007.

[16] SPRACE – São Paulo Regional Analysis Center. Disponível em: http://www.sprace.org.br. Acesso em: 13/11/2007.

[17] ClearImage 5. Inlite Res. Inc. Disponível em: http://www.inliteresearch.com. Acesso em: 10/02/2008.

[18] Leadtools 13. Leadtools Inc. Disponível em: http://www.leadtools.com. Acesso em: 10/02/2008.

[19] ScanFix Bitonal Image Optimizer 4.21. TMS Sequoia. Disponível em: http://www.tmsinc.com. Acesso em: 15/01/2008.

[20] Skyline Tools Corporate Suite 7. Skyline Tools Imaging. Disponível em: http://www.skylinetools.com. Acesso em: 17/03/2008.

[21] LINS, R. D.; ÁVILA, B. T.; FORMIGA, A. A. BigBatch: An Environment for Processing Monochromatic Documents. In: INTERNATIONAL CONFERENCE ON IMAGE ANALYSIS AND RECOGNITION (Sep. 2006 : Póvoa de Varzim). *Proceedings*. Póvoa de Varzim. p. 886-896.

[22] UNGER, S. A Computer Oriented Toward Spatial Problems. In: WESTERN JOINT COMPUTER CONFERENCE: CONTRASTS IN COMPUTERS (May. 1958 : Los Angeles). *Proceedings*. Los Angeles. p. 234-239.

[23] CANTONI, V.; FERRETTI, M.; LEVIALDI, S.; et al. A Pyramid Project Using Integrated Technology. In: _____.. *Integrated Technology for Parallel Image Processing*, Academic Press, 1985. p. 121–132.

[24] MÉRIGOT, A.; CLERMONT, P.; MÉHAT, J.; et al. A Pyramidal System for Image Processing, In: _____. *Pyramidal Systems for Image Processing and Computer Vision*, NATO ASI Series, Springer-Verlag, 1986. p. 109–124.

[25] CANTONI, V.; FERRETTI, M. Pyramidal Architectures for Computer Vision, Plenum Press, New York, 1994.

[26] HAMEY, L. G. C.; WEBB, J. A.; WU, I-C. An Architecture Independent Programming Language for low-level Vision. *Computer Vision, Graphics, and Image Processing*, San Diego. v. 48, n. 2, p. 246-264, Nov. 1989.

[27 WEBB, J.A. Steps Toward Architecture Independent Image Processing. *Computer*, California. v. 25, n. 2, p. 21–31, Feb. 1992.

[28] MORROW, P. J.; CROOKES, D.; MCPARLAND, P.J. Ial: A Parallel Image Processing Programming Language, *IEE Proceedings I*, v. 137, n. 3, p. 176–182, Jun. 1990.

[29] BROWN, J.; CROOKES, D. A High Level Language for Parallel Image Processing. *Image and Vision Computing*. v. 12, n. 2, p. 67–79, Marc. 1994.

[30] STEELE, J.A. *An Abstract Machine Approach to Environments for Image Interpretation on Transputers*. Irlanda, 1994. Tese (Doutorado em Ciências) – Fac. Science, Queens University of Belfast.

[31] NICK, J. M.; FOSTER, I.; KESSELMAN, C. et al. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In: _____. *Grid Computing: Making the Global Infrastructure a Reality*. JohnWiley& Sons : Inglaterra. 2003.

[32] MedGrid. http://www.creatis.insa-lyon.fr/MEDIGRID. Acessado em 20/11/2008.

[33] MONTAGNAT, J.; BRETON, V.; MAGNIN, I. Using Grid Technologies to Face Medical Image Analysis Challenges. In: 1st INTERNATIONAL WORKSHOP ON BIOMEDICAL COMPUTATIONS ON THE GRID. (May. 2003 : Tokyo). *Proceedings*. Tokyo, Japan. p. 588-59.

[34] KAO, O. On Parallel Image Retrieval With Dynamically Extracted Features. *PARALLEL COMPUTING: Systems & Applications.* v. 34, n. 12, p. 700–709, Dec. 2008.

[35] OH, M.; AIZAWA, K. Large-scale Image Sensing by a Group of Smart Image Sensors. *PARALLEL COMPUTING: Systems & Applications.* v. 34, n. 12, p.710–717, Dec. 2008.

[36] COLOMBO, C.; DEL BIMBO, A.; VALLI, A. A Real-time Full Body Tracking and Humanoid Animation System. *PARALLEL COMPUTING: Systems & Applications.* v. 34, n. 12, p. 718–726, Dec. 2008.

[37] ISGRÒ, F.; TEGOLO, D. A Distributed Genetic Algorithm for Restoration of Vertical Line Scratches. *PARALLEL COMPUTING: Systems & Applications.* v. 34, n. 12, p. 727–734, Dec. 2008.

[38] JONKER, P.P.; OLK, J.G.E.; NICOLESCU, C. Distributed Bucket Processing: A Paradigm Embedded in a Framework for the Parallel Processing of Pixel Sets. *PARALLEL COMPUTING: Systems & Applications.* v. 34, n. 12, p. 735–746, Dec. 2008.

[39] FREY, J.; TANNENBAUM, T.; LIVNY, M.; et al. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Journal of Cluster Computing*, Amsterdam, v. 5, n. 3, p. 237-246, Jul. 2002.

[40] CIRNE, W.; BRASILEIRO, F.; ANDRADE, N.; et al. Labs of the World, Unite!!!. *Journal of Grid Computing*, Amsterdam, v. 4, n. 3, p.225-246. Sept. 2006.

[41] Ourgrid. Disponível em: http://www.ourgrid.org. Acesso em: 25/10/2007.

[42] openMosix, an Open Source Linux Cluster Project. Disponível em: http://openmosix.sourceforge.net. Accesso em: 23/05/2007.

[43] LITZKOW, M.; LIVNY, M.; MUTKA, M. Condor – a hunter of idle workstations. In: 8th INTERNATIONAL CONFERENCE OF DISTRIBUTED COMPUTING SYSTEMS (Jun. 1998 : San Jose). *Proceedings*. San Jose. p. 104-111.

[44] MICROSOFT. Windows HPC Server 2008: Technical Overview of Windows HPC Server 2008. September 2008. Disponível em: http://www.microsoft.com/downloads/details.aspx?FamilyId=7A4544F0-81F2-4778-8A59-35C43BA49875&displaylang=en. Acesso em: 06/10/2008.

[45] MICROSOFT. Implantação e gerenciamento do Microsoft Windows Compute Cluster Server 2003. Disponível em: http://www.microsoft.com/brasil/technet/prodtechnol/windowsserver/library/9330fdf8-c680-425f-8583-c46ee7730698.mspx. Acesso em 01/09/2007.

[46] SNIR, M., OTTO, S., HUSS-LEDERMAN, S. et al. *MPI – The complete reference, Volume 1*. 2. ed. The MIT Press, 1998.

[47] CHANDRA, R., MENON, R. et al. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[48] AKL, S. G. *Parallel Computation: models and methods.* Prentice-Hall, 1997.

[49] FOSTER, I. *Designing and Building Parallel Programs: concepts and tools for parallel software engineering*. Addison-Wesley Publishing Company, 1995.

[50] FLYNN, M. J.; Rudd K.W.; Parallel Architectures. *ACM Computing Surveys*. v. 28, n. 1, p. 67-70, Mar. 1996.

[51] NAVAUX, P. O. A., DE ROSE, C. A. F. *Arquiteturas Paralelas*. Série Livros Didáticos nº 15. Editora Sagra-Luzzato, 2008.

[52] JAIN, R. *The Art of Computer Systems – Performance Analysis: Techniques for experimental design measurement, simulation and modeling*. John Wiley & Sons Inc, 1991.

[53] GROPP, W., HUSS-LEDERMAN, S., LUMSDAINE, A., et al. *MPI – The complete reference. Volume 2*. 2. ed. The MIT Press, 1998.

[54] Mosix Cluster and Grid Management. Disponível em: http://www.mosix.org. Acesso em: 13/11/2007.

[55] The Beowulf Cluster Site. Disponível em: http://www.beowulf.org. Acesso em: 08/11/2007.

[56] PEREIRA, J. H., CANDURI, F., OLIVEIRA. J. S. et. al. Structural bioinformatics study of EPSP synthase from Mycobacterium tuberculosis. *Biochemical and Biophysical Research Communications*. v. 312, n. 3, p. 608-614, Dec. 2003.

[57] Message Passing Interface (MPI) – Tutorial. Disponível em: http://www.llnl.gov/computing/tutorials/mpi. Acesso em: 17/09/2007.

[58] MPICH2 Home Page. Disponível em: http://www-unix.mcs.anl.gov/mpi/mpich2. Acesso em: 29/10/2007.

[59] MPI Fórum. Disponível em: http://www.mpi-forum.org. Acesso em: 29/10/2007.

[60] BERMAN, F., FOX, G. C., HEY, A. J. G. The open grid services architecture and data grids. In: _____. *Grid Computing: Making The Global Infrastructure a Reality*. JohnWiley& Sons : Inglaterra. 2003.

[61] YANG, K.; GUO, X.; GALIS, A. et al. Towards efficient resource on-demand in Grid Computing. *ACM SIGOPS Operating Systems Review*. v. 37, n. 2, p. 37-43, Apr. 2003.

[62] YANG, L. T.; GUO, M. Global Grids and Software Toolkits: A Study of Four Grid Middleware Technologies. ASADZADEH, P.; BUYYA, R.; KEI, C. L. et al. *High Performance Computing: Paradigm and Infrastructure.* John Wiley & Sons, Inc. 2006. p. 431-458.

[63] CZAJKOWSKI, K.; FOSTER, I.; KARONIS, N.; et al. A resource management architecture for metacomputing systems. In: IPPS/SPDP WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING (Mar. 1998 : Orlando). *Proceedings*. Orlando. p.62-82.

[64] THAIN, D.; TANNENBAUM, T.; LIVNY, M. Condor and the grid. In: _____. *Grid Computing: Making the Global Infrastructure a Reality*. JohnWiley& Sons : Inglaterra. 2003.

[65] BASNEY, J.; LIVNY, M. *Deploying a High Throughput Computing Cluster.* vol. 1. High Performance Cluster Computing. Prentice Hall. 1999.

[66] EPEMA, D., LIVNY, M., VAN DANTZIG, R., EVERS, X., AND PRUYNE, J. A worldwide flock of Condor: Load sharing among workstation clusters. *Future Generation Computer Systems. v. 12*, p. 53-65, May 1996.

[67] CIRNE, W.; MARZULLO, K. Open Grid: A user-centric approach for grid computing. In: 13th SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (Sept. 2001 : Pirenópolis). *Proceedings*. Pirenópolis.

[68] CIRNE, W.; PARANHOS, D.; COSTA, L. et al. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. 32nd INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (Oct. 2003 : Kaohsiung). *Proceedings*. Kaohsiung. p. 407-416.

[69] PARANHOS, D.; CIRNE, W.; BRASILEIRO, F. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. *Euro-Par 2003 Parallel Processing – Lecture Notes in Computer Science*, v. 2790/2004, pp. 169-180, June 2004.

[70] SANTOS-NETO, E.; CIRNE, W.; BRASILEIRO, F. et al. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In: *Job Scheduling Strategies for Parallel Processing*, v. 3277/2005, pp. 210-232, May 2005.

[71] ANDERSON, D.; COBB, J.; KORPELA, E. SETI@home: An Experiment in Public-Resource Computing. *Communication of the ACM*, New York, v. 45, n. 11, p. 56-61, Nov. 2002.

[72] Data Mining Tools and Services for Grid Computing Environments. Disponível em: http://www.datamininggrid.org/downloads.htm. Acesso em: 20/01/2008.

[73] ABRAMSON, D., GIDDY, J.; KOTLER, L. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In: IPDPS INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (May. 2000 : Cancun). *Proceedings*. Orlando. p. 520-528.

[74] STILES, J. R.; BARTOL, T. M.; SALPETER, E. E.; et. al. Monte Carlo Simulation of Neuro-transmitter Release Using MCell, a General Simulator of Cellular Physiological Processes. In: 6th ANNUAL CONFERENCE ON COMPUTATIONAL NEUROSCIENCE (Jul. 1998 : Big Sky). Proceedings. Big Sky. p. 279-284.

[75] SMALLEN, S.; CASANOVA, H.; BERMAN, F. Applying Scheduling and Tuning to On-line Parallel Tomography. In: CONFERENCE ON HIGH PERFORMANCE NETWORKING AND COMPUTING (Nov. 2001 : Denver). *Proceedings*. Denver. p. 12-30.

[76] SMALLEN, S.; CIRNE, W.; FREY, J.; et al. Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. In: 9th HETEROGENEOUS COMPUTING WORKSHOP (May 2000 : Cancun). *Proceedings*. Cancun. p. 241-252.

[77] Kodak Digital Science Scanner 1500. Disponível em: http://www.kodak.com. Acesso em: 09/03/2006.

[78] Leadtools 13. Leadtools Inc. Disponível em: http://www.leadtools.com. Acesso em: 10/02/2008.

[79] MATTOS, G. O.; FORMIGA, A. A.; LINS, R. D.; MARTINS, F. M. J. BigBatch: A Document Processing Platform for Clusters and Grids. In: 23rd ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING (Mar. 2008 : Fortaleza). *Proceedings*. New York : ACM Press, 2008. v. I. p. 434-441.

[80] MATTOS, G. O; FORMIGA, A. A, LINS, R. D.; CARVALHO JUNIOR, F. H. A Comparison of Cluster and Grid Configurations Executing Image Processing Tasks in a Local Network. In: THE SEVENTH INTERNATIONAL CONFERENCE ON NETWORKING (Apr. 2008 : Cancun). *Proceedings*. New York : IEEE Press, 2008. p. 408-414.

[81] MICROSOFT. Using Windows Compute Cluster Server 2003 Job Scheduler. Disponível em: http://technet2.microsoft.com/WindowsServer/en/Library/4fb21c96-759c-493a-bb29-f14bd491160d1033.mspx. Acesso em: 01/09/2007.

[82] MICROSOFT. Using Windows HPC Server 2008 Job Scheduler. Jun. 2008. Microsoft Corporation. Disponível em: http://www.microsoft.com/downloads/details.aspx?FamilyId=DAB977D5-2311-4B80-9257-477838C0EB6F&displaylang=en. Acesso em: 30/10/2008.

[83] MICROSOFT. Using Microsoft Message Passing Interface. Disponível em: http://technet2.microsoft.com/windowsserver/en/library/4cb68e33-024b-4677-af36-28a1ebe9368f1033.mspx?mfr=true. Acesso em: 17/09/2007.

[84] ODERSKY, M. Scalable Component Abstractions. In: 20th ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS. (Oct. 2005 : San Diego). p. 41-57.

[85] HALLER, P., and ODERSKY, M. *Event-Based Programming without Inversion of Control*. LNCS. v. 4228, p. 4-22, Sep. 2006.

[86] MICROSOFT. Microsoft TechNet. Introducing Windows 2000 Server. Microsoft Press, 2000. Disponível em: http://technet.microsoft.com/en-us/library/bb742424.aspx. Acesso em: 01/11/2008.

[87] LANTZ, E. Using Microsoft Message Passing Interface (MS MPI). Microsoft Corporation, 2008. Disponível em: http://www.microsoft.com/downloads/details.aspx?FamilyId=D5838485-507C-45CD-8AA6-6D6B383D1071&displaylang=en. Acesso em: 08/09/2008.

[88] PARIKH, S. MARTINEZ, T. E. Dual Processors, Hyper-Threading Technology, and Multi-Core Systems. Disponível em: http://www.intel.com/cd/ids/developer/asmo-na/eng/200677.htm. Acesso em 01/05/2008.

[89] Ubuntu Home Page. Disponível em: http://www.ubuntu.com. Acesso em: 01/10/2007.

[90] BRAZILE, J., SCHAEPMAN, M. E., SCHLÄPFER, D. et al. Cluster versus grid for large-volume hyperspectral image preprocessing. In: SPIE END-TO-END ATMOSPHERIC PROCESSING. v. 5548, pp.48-58. Aug. 2004. Denver, USA.

[91] APEX - Airborne Prism EXperiment. Disponível em: http://www.apex-esa.org/apex/htdocs/modules/APEX. Acesso em: 01/12/2007.

[92] BRAZILE, J., SCHAEPMAN, M. E., SCHLÄPFER, D. et al. Cluster versus grid for operational generation of ATCOR's MODTRAN-based look up tables. *Parallel Computing*. v. 34, n. 1, p. 32-46, Jan. 2008.

[93] GODKNECHT, A.J., BOLLIGER, C., University of Zurich Matterhorn cluster. Disponível em : http://www.matterhorn.uzh.ch. Acesso em: 01/12/2007.

[94] WIRESHARK Network Protocol Analyser. Version 0.99.6a (SVN Rev 22276). Disponível em: http://www.wireshark.org/. Acesso em: 09/10/2008.

[95] MICROSOFT. HPC Cluster Manager. Disponível em: http://technet.microsoft.com/en-us/library/cc972800.aspx Acesso em: 30/10/2008.