UNIDADE DE ENSINO SUPERIOR VALE DO IGUAÇU
FACULDADES INTEGRADAS DO VALE DO IGUAÇU
CURSO DE SISTEMAS DE INFORMAÇÃO

CARLOS ALBERTO BARBOSA STENZEL

TEORIA E PRÁTICA DA COMPUTAÇÃO PARALELA PARA MELHOR DESEMEPENHO DE PROCESSAMENTO

União da Vitória – PR
2015

CARLOS ALBERTO BARBOSA STENZEL

USO DA COMPUTAÇÃO PARALELA PARA MELHOR DESEMEPENHO DE PROCESSAMENTO

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação da Faculdade de Ciências Exatas e Tecnológicas de União da Vitória da Unidade de Ensino Superior Vale do Iguaçu, como critério para obtenção de Grau de Bacharel em Sistemas de Informação.

Prof. Orientador: Cleverson Bússolo Klettenberg

União da Vitória – PR
2015

USO DA COMPUTAÇÃO PARALELA PARA MELHOR DESEMEPENHO DE PROCESSAMENTO
Por
Carlos Alberto Barbosa Stenzel

Trabalho de Conclusão de Curso aprovado com nota ____, para obtenção do grau de Bacharel em Sistemas de Informação, pela Banca examinadora formada por:

_____
Prof. M.Sc. Cleverson Bússolo Klettenberg
Orientador

_____
Prof. Andréa Tomko
Membro

_____
Prof. George Luiz Maluf
Membro

RESULTADO: _____

UNIGUAÇU: ____/____/_____

União da Vitória – PR
2015

DEDICATÓRIA

Dedico este trabalho para meus pais que me ajudaram e me incentivaram a realizar o curso de sistemas de informação apesar de todas as dificuldades encontradas no caminho.

AGRADECIMENTO

Agradeço a minha família, colegas de turma, professores e amigos que estavam comigo durante essa jornada de estudos. Fica neste trabalho minha eterna gratidão.

EPÍGRAFE

"O insucesso é apenas uma oportunidade para recomeçar de novo com mais inteligência." - *Henry Ford*

LISTA DE FIGURAS

Figuras

LISTA DE TABELAS

Tabelas

LISTA DE SIGLAS

API (*Interface de Programação de Aplicações*)

CEO (*Chief Executive Officer*)

CESDIS (*Center of Exellence in Space Data and Information Sciences*)

CPAR (Linguagem de programação paralela)

ENIAC (*Electronic Numerical Integrator and Computer*)

GRAM (*Globus Resource Allocation Manager*)

HPC (*High Performance Computing*)

Khz (quilohertz)

MDS (*Metacomputing Directory Service*)

MOSIX (*Multicomputer Operating System for UnIX*)

MPI (*Interface de Passagem de Mensagem*)

NASA (*National Aeronautics and Space Administration*)

NOW (*Networks of Workstations*)

OpenMP (*Open Multi-Processing*)

PVM (*Parallel Virtual Machine*)

RoD (*Resources on Demand*)

SMP (*Simultaneous Multiprocessing*)

SPC (*Scalable Parallel Computers*)

UNESP (*Universidade Estadual Paulista*)

VLSI (*Very Large Scale Integration*)

RESUMO

STENZEL, Carlos A. **Uso da computação paralela para melhor desempenho de processamento**. Trabalho de conclusão de Curso. Bacharelado em Sistemas de Informação, Faculdade do Vale do Iguaçu – União da Vitória – PR – 2015.

Desde o surgimento do primeiro computador digital eletrônico, a computação vem passado por um processo evolutivo intenso, tanto em nível de hardware quanto de software, sempre visando proporcionar um maior desempenho e tornar mais rápido a execução de processos. Para melhor uso de processando o uso da computação paralela é utilizando para melhorar o processo e obter um menor custo e uma flexibilidade maior na estrutura disponíveis nas empresas. Atualmente a computação paralela está presente em muitos locais, assim como bancos, onde o usuário final quer ter uma resposta rápida a ação executada, onde a não utilização da computação paralela ocasionaria em um maior tempo de resposta do comando executado. O uso da computação paralela permite que vários processos sejam executados ao mesmo tempo em paralelo, assim obtendo uma resposta mais rápida das funções executadas e sendo possível a utilização de Clusters e Grids podendo suportar ainda mais processos executando ao mesmo tempo, assim tendo um melhor proveito do poder do computador. Para que seja possível o uso dessa tecnologia é necessário o uso da programação paralela, existem diversas formas para utilizar, como a utilização de Forks, MPI, OpenMP, CPAR, entre muitos outros existes para facilitar a utilização da computação paralela, onde será abordado nesse trabalho o uso dos mesmo em um teste de desempenho comparativo.

Palavras-chaves: Computação Paralela, Desempenho, Processamento, Cluster.

ABSTRACT

STENZEL, Carlos A. **Use of parallel computing for better processing performance.** Completion of course work. Information Systems, Faculdade do Vale do Iguaçu – União da Vitória – PR – 2015.

Since the emergence of the first electronic digital computer, the computer has gone through an intense evolutionary process, both at the level of hardware and software, always aiming to provide a higher performance and make faster process execution. For better use of processing the use of parallel computation is using to improve the process and achieve a lower cost and greater flexibility in the structure available in enterprises. Currently, parallel computing is present in many places, as well as banks, where the end user wants to have a rapid response to action taken where non-use of parallel computing would result in a larger run command response time. The use of parallel computation allows multiple processes running simultaneously in parallel, thereby obtaining a faster response of the executed tasks and being possible to use Clusters and Grids may further bear processes running simultaneously, thus having a better use computer power. To be able to use this technology is necessary to use the parallel programming, there are many ways to use, and the use of Forks, MPI, OpenMP, CPAR, among many others exist to facilitate the use of parallel computing, which will be addressed in this work using the same in a comparative performance test.

Palavras-chaves: Parallel Computing, Performance, Processing, Cluster.

Sumário

1 – INTRODUÇÃO

Desde o surgimento do primeiro computador digital eletrônico, ENIAC (*Electronic Numerical Integrator and Computer*), em 1946, a computação vem passando por um processo evolutivo intenso, tanto em nível de hardware quanto de software, sempre visando proporcionar um maior desempenho e tornar mais rápido o processante de funções.

Na década de 90, redes de computadores tornaram-se mais rápidas e confiáveis, o que possibilitou a interligação de computadores de maneiras mais eficientes formando os sistemas distribuídos.

Sistemas distribuídos têm sido utilizados para a execução de programas paralelos, em substituição às arquiteturas paralelas, em virtude de seu menor custo e maior flexibilidade.

Atualmente a computação paralela está presente em muitas coisas que usamos, pode-se não perceber o uso, mais está presente, por exemplo, nos caixas eletrônicos dos bancos, nos aplicativos que são usados nos celulares ou computadores.

A computação paralela é utilizada para melhor desempenho em servidores para que o usuário final tenha uma resposta rápida ação solicitada, como por exemplo, retirar um extrato da sua conta em um caixa eletrônico, supondo que não fosse utilizado a computação paralela, então logo seus servidores não poderiam estar ligados em *cluster*, ou utilizando chips com múltiplos núcleos por exemplo que já estes mesmos são um exemplo do uso da computação paralela, então um servidor só pode executar uma função de cada vez, assim, quando um cliente solicitar um extrato, este pedido estará em uma fila para execução, então o usuário final, que nesse caso é um cliente de um banco que está tentando retirar seu extrato tem que esperar que todos os outros pedidos solicitados outros clientes que estejam na frente do seu pedido na fila de execução, sejam executados para que enfim o seu pedido de extrato solicitado seja processado e executado.

Este trabalho está divido em capítulos de forma a apresentar todo o conteúdo necessário para o entendimento. Na sequência são descritos cada capítulo que compõe o texto:

Capítulo 1: Introdução sobre o trabalho.

Capítulo 2: Revisão sobre os assuntos abordados e a evolução da computação.

Capítulo 3: Características da computação paralela e um teste de desempenho comparativo.

Capítulo 4: Conclusão obtida do trabalho apresentado.

1.1 – Justificativa

Cada vez mais que os sistemas computacionais se aproximam de seus limites físicos, mas importante é o desenvolvimento de algoritmos e capacidades eficientes de paralelização nos sistemas computacionais em múltiplos níveis. A incorporação de práticas de paralelização e o domínio de seus princípios são os componentes essenciais para dominar o tratamento da quantidade de dados produzidas nos dias atuais em tempo de utilizar seu potencial.

1.2 – Objetivos

1.2.1 – Objetivo geral

Mostrar como a utilização da computação paralela pode melhorar o desempenho de processamento em sistemas de grande porte que necessitem de rapidez e agilidade no processamento dos dados e funções.

1.2.2 – Objetivos específicos

- Apresentar os conceitos e o funcionamento da computação paralela.

- Mostrar de que forma paralelismo pode melhorar o tempo o desempenho de sistemas.

- Mostrar algumas técnicas utilizadas para a obtenção de maior desempenho.

- Efetuar uma comparação de alguns métodos utilizados na programação da computação paralela.

2 – REVISÃO LITERÁRIA

2.1 – Evolução dos Computadores

Segundo MARIMOTO (2000, p. 40), "[...] no final do século XIX, surgiram as primeiras válvulas. As válvulas foram usadas para criar os primeiros computadores eletrônicos, na década de 40."

Na década de 40 após o surgimento das válvulas surgiu o primeiro computador com propósito geral.

> O ENIAC (Computador e Integrador Numérico Eletrônico - Electroníc Numerícal 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. (STALLINGS W, 2002, p. 19).

O ENIAC ocupava muito espaço e consumia muita energia, e era composto segundo MARIMOTO (2000, p.40) "por nada menos do que 17.468 válvulas", seu gigantesco tamanho ocupa um galpão imenso, e era capaz de processar segundo MARIMOTO (2000, p. 40) "apenas 5.000 adições, 357 multiplicações e 38 divisões por segundo", nos dias atuais uma calculadora de bolso, por exemplo, tem um poder de processamento de muito mais operações.

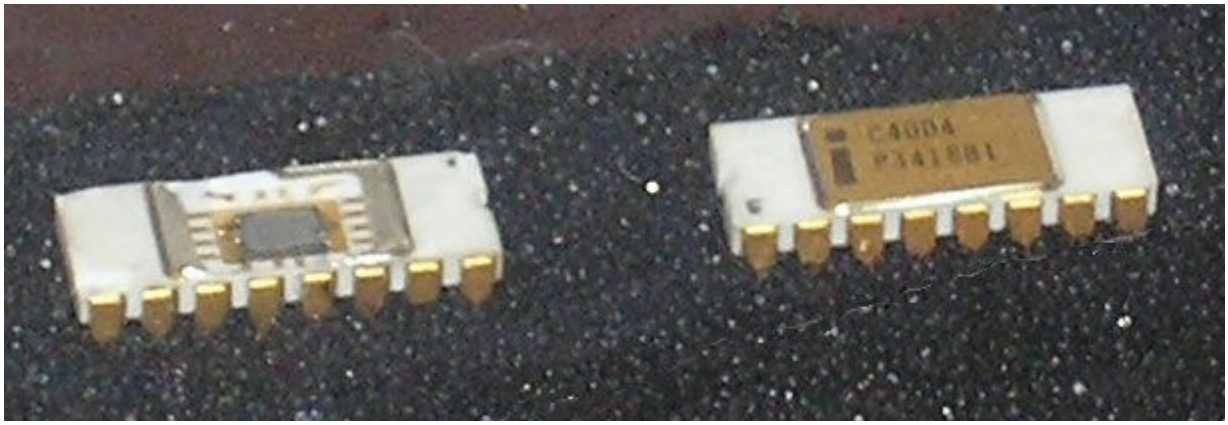
Com o desenvolvimento dos transistores entre 1952 e 1960, as válvulas se tornaram obsoletos.

> Com o tempo, os transistores passaram a ser a base da eletrônica, seguindo-se a construção de circuitos cada vez mais pequenos. Esta miniaturização permitiu que se tivesse a mesma capacidade de cálculo de um ENIAC na palma de uma mão. A diminuição do tamanho fez também diminuir a quantidade de energia necessária e o custo caiu com a produção em série dos novos processadores. (Jucimar Peruzzo, 2010, p. 301).

O grande salto ocorreu com o lançamento do microchip.

"O primeiro microchip comercial foi lançado pela Intel em 1971 e chamava-se 4004. Como o nome sugere, ela era um processador de apenas 4 bits que era composto por pouco mais de 2000 transístores." (MARIMOTO, 2000, p.43)

A imagem abaixo mostra o microchip 4004 sem tampa e com tampa, era muito eficiente para ser usado em calculadoras e dispositivos de controle. Esse primeiro processador tinha características únicas para seu tempo, como a velocidade do *clock*, que ultrapassava os 100 Khz (quilohertz).



Fonte: John Pilge – wikimidia.org

Apos o lançamento do 4004, diversos outros processadores foram lançados, cada processador lançado tinha um *clock* mais alto que o anterior, assim, tornou-se cada vez mais difícil resfriar os processadores *single core*.

"Chegou-se num ponto em que aumentar a velocidade dos processadores tornou-se bastante difícil. Um dos problemas foi o excesso de calor gerado, devido à grande densidades de transistores trabalhando em frequências elevadas." (jucimar peruzzo, 2008, p. 118).

2.2 – Processadores Multi-core

A solução para o problema de aquecimento que estavam enfrentando com os processadores com alto *clock*, foi o uso de dois ou mais núcleos, está tecnologia é conhecida como *multi-core*, segundo ANDRÁS (2011, p.3) "[...] múltiplos núcleo ou *multi-core* são usados para descrever arquiteturas com um número elevado de núcleos.", com essa tecnologia o processador reparte as tarefas entre os núcleos, o que faz que aumente a velocidade de processamento, segundo PERUZZO (2008, p.119) "isso possibilita que as instruções das aplicações sejam executadas em paralelo, numa menor frequência e produzindo menos calor."

> Uma arquitetura multicore é geralmente um multiprocessamento simétrico (Simultaneous Multiprocessing - SMP, em que múltiplos núcleos de processador tipicamente compartilham um segundo ou terceiro nível de cache comum e interconectado), implementado em um único circuito VLSI (Very Large Scale Integration). (Douglas Camargo Foster, p. 2, 2009).

Atualmente processares com arquitetura *multi-core* são encontrados na maioria dos computadores, celulares comercializados.

2.4 – Computação Distribuída

A necessidade de criar computadores que precisavam cada vez mais poder de processamento, fez surgiu a computação distribuída, a computação distribuída segundo COULOURIS (2011, p. 01) “é uma coleção de computadores autônomos interligados através de uma rede de computadores e equipados com software que permita o compartilhamento dos recursos do sistema: hardware, software e dado”.

Um problema é divido em várias tarefas, cada um é resolvido por um ou mais computadores, que estão conectados formando um sistema distribuído.

2.4.1 – Cluster

Cluster pode ser definido segundo STALLINGS (2002, p.15) “como um grupo de computadores completos interconectados”, essas máquinas que estão conectada por cabos de rede compartilham recursos, segundo STALLINGS (2002, p.15) “como um recurso de computação unificado, que cria a ilusão de construir uma única máquina”.

Os *clusters* podem ser classificados segundo BUYYA (1999) como:

Balanceamento de Caga: Pode ser visto como uma solução abrangente na utilização de grandes redes, porque ele ocasiona um aumento na capacidade, melhorando assim o desempenho, já que o balanceamento será feito por vários servidores, todos os servidores mantém uma cópia integral de todos os dados onde os servidores precisarão dessa cópia para ler as informações que chegarem até ele, um software de controle se encarrega de sincronizar todas as informações, caso haja falha em algum servidor, ou algum precise ser desligado, os demais continuam trabalhando normalmente, ao voltar, o software de controle sincroniza o servidor com os demais para voltar os serviços normalmente.

> Em clusters de balanceamento de carga, as tarefas de processamento são distribuídas o mais uniformemente possível entre os nós. O foco é fazer com que cada computador receba e atenda a uma requisição e não, necessariamente, que divide uma tarefa com outras máquinas. Imagine, por exemplo, que um grande site na internet receba por volta de mil visitas por segundo e que um cluster formado por 20 nós tenha sido desenvolvido para atender a esta demanda. Como se trata de uma solução de balanceamento de carga, estas requisições são distribuídas igualmente entre as 20 máquinas, de

> forma que cada uma receba e realize, em média, 50 atendimentos a cada segundo. (ALECRIM, E. 2013).

Para MARCOS PITANGA (2003) “este modelo distribui o tráfego entrante ou requisições de recursos provenientes dos nodos que executam os mesmos programas entre as máquinas que compõem o *cluster*”.Todos os nodos são responsáveis em controlar os pedidos. Se um nó falhar, as requisições são distribuídas entre os nós disponíveis no momento. Este tipo de solução é normalmente utilizado em fazendas de servidores web (*web farms*).

Um exemplo deste tipo de *cluster* é o MOSIX (*Multicomputer Operating System for UnIX*) . 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*.

Segundo o site do mesmo disponível em http://www.mosix.org 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.

Alta Disponibilidade: Sua principal função é manter os serviços ativos pelo maior tempo possível, onde é possível sua aplicação através de redundância de hardware e configuração de softwares específicos. Nesse caso os serviços não pertencem apenas a um computador, mas sim ao *cluster* como um todo. A desvantagem desse tipo de *cluster* é que caso um serviço venha a falhar, pode-se perder um pouco de tempo, pois seu principal objetivo é manter os serviços ativos. Diferente do balanceamento de carga que divide o processamento com os demais computadores interligados.

> Nos clusters de alta disponibilidade, o foco está em sempre manter a aplicação em pleno funcionamento: não é aceitável que o sistema pare de funcionar, mas se isso acontecer, a paralisação deve ser a menor possível, como é o caso de soluções de missão crítica que exigem disponibilidade de, pelo menos, 99,999% do tempo a cada ano. Em determinadas circunstâncias, é tolerável que o sistema apresente algum grau de perda de desempenho, especialmente quando esta situação é consequência de algum esforço para manter a aplicação em atividade. (ALECRIM, E. 2013).

Para MARCOS PITANGA (2003) “estes modelos de *cluster* são construídos para prover disponibilidade de serviços e recursos de forma ininterrupta através do uso de redundâncias implícitas ao sistema. A ideia geral é que se um nó falhar (*failover*), aplicações ou serviços possam estar disponíveis em outro nó. Estes tipos de *cluster* são utilizados para base de dados de missões críticas, correios, servidores de arquivos e aplicações.”

Alta Performance de Computação: Conhecido também por Cluster de Alto Desempenho, o objetivo desde tipo de *cluster* é desenvolver supercomputadores para processamento paralelo, trabalhando em conjunto de forma integrada, trazendo resultados satisfatórios, com serviços rápidos e confiáveis.

> Clusters de alto desempenho são direcionados a aplicações bastante exigentes no que diz respeito ao processamento. Sistemas utilizados em pesquisas científicas, por exemplo, podem se beneficiar deste tipo de cluster por necessitarem analisar um grande variedade de dados rapidamente e realizar cálculos bastante complexos. (ALECRIM, E. 2013).

Para MARCOS PITANGA (2003) este modelo de *cluster* aumenta a disponibilidade e performance para as aplicações, particularmente as grandes tarefas computacionais. Uma grande tarefa computacional pode ser dividida em pequenas tarefas que são distribuídas ao redor das estações (nodos), como se fosse um supercomputador massivamente paralelo. Estes *clusters* são usados para computação científica ou análises financeiras, tarefas típicas para exigência de alto poder de processamento.

Cluster Combo ou Combinação: 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, entre muitos outros.

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 as estações como se fossem um supercomputador massivamente paralelo. É comum associar este tipo de *cluster* ao projeto *Beowulf*.

Segundo PEREIRA (2003) no Brasil, um exemplo é o *cluster* desenvolvido no ano de

2003, por um aluno da Universidade Estadual Paulista (UNESP), baseado no tipo *Beowulf*, esse *cluster* ficou 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.

Em 1980, três tendências se convergiram:

- Microcomputadores de alta performance
- Redes de alta velocidade
- Ferramentas padronizadas para computação distribuída de alta velocidade

Essas convergências foram fundamentais para o aprofundamento nas técnicas de sistemas paralelos, devido ao fato de sempre existir a necessidade de alto poder de processamento, seja para aplicações científicas ou para outras aplicações que exijam um poder de processamento mais elevado.

Com a tecnologia de *cluster* obtém-se alto poder de processamento com baixo custo (TANEMBAUM, 2006)

No final de 1993, dois cientistas da NASA (*National Aeronautics and Space Administration*) – Donald Becker e Thomas Sterling – começaram a planejar um sistema de processamento distribuído com computadores Intel Desktop. Seu objetivo era fazer com que esses computadores conectados em rede se assemelhassem ao processamento de um supercomputador. E que teriam um menor custo.

> No início de 1994, já trabalhando no CESDIS – Center of Exellence in Space Data and Information Sciences, Centro de Excelência em Dados Espaciais e Ciência da Informação, criaram o projeto Beowulf. Com 16 computadores Intel 486 DX4 conectados em uma rede Ethernet com sistema operacional Linux fizeram esse primeiro cluster chegar a um desempenho de 60 Mflops. (TANEMBAUM, 2006).

Segundo TANEMBAUM (2007) computação em *cluster* pode ser considerado como uma subcategoria do Paralelismo.

3 – DESENVOLVIMENTO

3.1 Tipo de pesquisa

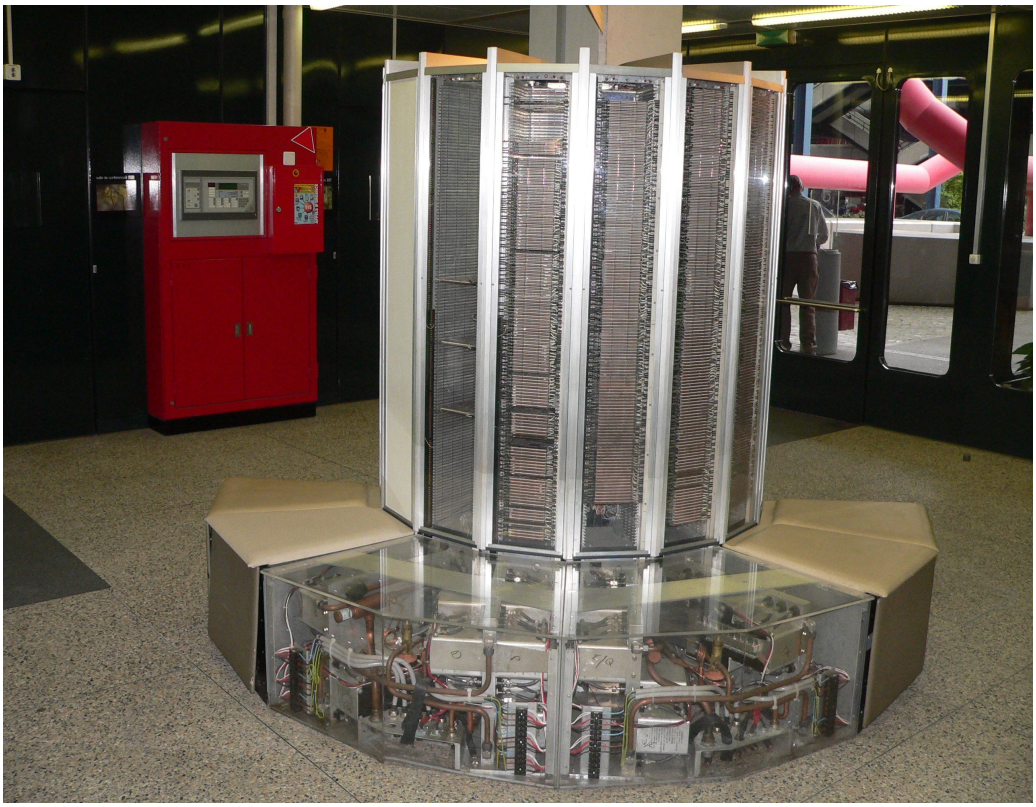
Esse trabalho pode se classificar quanto a sua natureza como uma pesquisa aplicada onde busca gerar conhecimento sobre o uso da computação paralela para melhor desempenho de processamento, quanto a classificação a forma e à abordagem do problema pode se definir como uma pesquisa quantitativa onde comparar técnicas de programação paralela para melhor desempenho, quanto ao ponto de vista de seus objetivos pode ser classificada como uma pesquisa explicativa onde visa aumentar o conhecimento sobre a computação paralela, quanto ao ponto de vista dos procedimentos técnicos pode-se classificar como uma pesquisa bibliográfica.

3.2 – Computação Paralela e de alto desempenho

Segundo STALLINGS (2002) os primeiros supercomputadores surgiram nos anos 70. E até aquele momento segundo TANENBAUM (2007) 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 para iniciar outra.

Com esses fatos, pode-se perceber que a divisão de tarefas traria avanço significativos para os computadores dessa época quanto a questão de desempenho computacional. A partir dessa época surgiu dois caminhos seguidos para solucionar a divisão das tarefas, a arquitetura paralela e sistema distribuído.

Nos meados de 1975 foi criado o ILLIAC IV, que possuía nada menos do que 64 processadores, considerado para aquela época o computador mais rápido que existia, mas a posição de computador mais rápido não durou muito, pois em 1976 surgiu o Cray-1, fabricado pela Cray Research. Na figura abaixo se pode ver como era o Cray-1.



Fonte: Cray - http://www.cray.com

Segundo FONSECA FILHO (2007, p.128) Cray-1 foi a primeira máquina pipeline, cujo processador executava uma instrução dividindo-a em partes, como em uma linha de montagens de carro por exemplo. 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 do Cray-1 era alto, e seu preço também era alto, segundo NULL e LOBUR (2011, p.58) algo em torno de 8 milhões de dólares, em 1976. Era capaz de processar até 133 *megaflops*, 133 milhões de operações aritméticas de ponto flutuante por segundo e suportava 8 megabytes de memória.

A tabela 1 mostra a relação dos 3 supercomputadores mais rápidos do mundo até novembro de 2014, disponibilizada pelo site TOP500 disponível através do endereço http://www.top500.org . Pode-se observar que a empresa Cray está em segundo lugar o supercomputador Titan, a empresa possui diversos com supercomputador entre os 500 mais rápidos do mundo. Até 2012 Titan era o mais rápido do mundo.

Rank	Fabricante	Computador	Localização	Cores	Desempenho TFlops
1	NUDT	**Tianhe-2 (MilkyWay-2)** - Cluster TH-IVB-FEP, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P	National Super Centro de Computação em Guangzhou - China	3120000	33,862.7
2	Cray Inc.	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gêmeos interconexão, NVIDIA K20x	DOE / SC / Oak Ridge National Laboratory - Estados Unidos	560640	17,590.0
3	IBM	**Sequoia** - BlueGene / Q, Poder BQC 16C 1,60 GHz, Personalizado	DOE / NNSA / LLNL Estados Unidos	1572864	17,173.2

Tabela 1: Relação dos 3 maiores supercomputador do mundo

Fonte: www.top500.org

Pode-se notar que a evolução foi significativa do Cray-1 para o Tianhe-2 e para o Titan, muito dessa evolução foi capaz a partir da utilização de processadores multicores oque fez a computação dar um salto gigantesco para melhor desempenho de processamento.

Como citado acima o Cray-1 era capaz de chegar a processar na casa dos *megaflops*, e atualmente já estamos na casa dos *teraflops*, isso em um espaço de tempo de 39 até os dias atualmente. Mas a visão mas clara da evolução pode ser visto que de 2012 do Titan para 2013 do Tianhe-2 quase dobrou a capacidade de processamento.

Um dos grandes desafios presentes atualmente na área de tecnologia da informação é viabilizar soluções computacionais que reduzam o tempo de processamento e forneçam respostas cada vez mais precisas.

Frequentemente surgem propostas com as mais diversas abordagens que exploram novas formas de resolver tais problemas ou tentam ainda melhorar as soluções existentes. Alguns desses problemas que podem ser resolvidos, ou utilização já essas tecnologias, que usam muito poder de processamento pode ser por exemplo, segundo QUINN (2003) “Simular eventos naturais como a previsão de tempo, dinâmica de fluidos, sequenciamento genético”, ainda segundo QUINN (2003) também pode ser usado em buscar novas drogas para a cura de doenças.

O supercomputador Titan segundo colocado, segundo o site https://www.olcf.ornl.gov/titan/ seu uso é para a ciência, onde são usados grandes quantidades de cálculos para solucionar ou simular experimentos em busca de descobertas na área da ciência.

Uma das grandes áreas que se dedica a propor tais melhorias é a computação paralela e de alto desempenho – HPC (*High Performance Computing*), que tem como base, várias subáreas da Ciência da Computação.

Não só no contexto científico é utilizado a computação de alto desempenho, pode-se observar que a uma explosão no volume de dados gerado a cada minuto.

Segundo TAURION (2013) "O volume de informações gerados pela sociedade é assustador. Por exemplo, o Twitter sozinho gera diariamente doze terabytes de tuítes.", ainda segundo TAURION (2013) "Em 2010, Eric Shimidt, então CEO (*Chief Executive Officer* - Diretor Executivo) do Google disse que a cada dois dias a sociedade já gerava tanta informação quanto gerou dos seus primórdios até 2003, ou sejam cinco exabytes"

Com a crescente geração de dados a utilização de supercomputador com cada vez mais poder de processamento é essencial para evolução dos supercomputadores.

Os *clusters* estão sendo usados em uma grande quantidade de aplicações, especialmente aquelas que exigem alto poder de processamento. 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.

Sistemas como o Facebook chegaram a armazenar 1 petabyte de informações no ano de 2008, registrando cerca de10 bilhões de fotos em seus servidores (WHITE, 2009, p. 1).

Até o ano de 2005 todos os computadores pessoais possuíam processadores com apenas um único núcleo físico. No mesmo ano, tanto Intel como a AMD lançaram processadores hoje denominados de processadores de múltiplos núcleos ou mais conhecidos como *multi-core*, os núcleos estão interconectados através de vias especiais de alta velocidade, e tem um cache e uma controladora de memória comum.

3.3 – Métricas em Computação Paralela

Algumas métricas são utilizadas para medir o desempenho do uso da computação

paralela em alguns sistemas, podemos dividi-las em 4 tipos:

- Tempo de execução
- Fator de aceleração ou *speedup*
- Eficiência
- Custo

3.3.1 – Tempo de execução

O tempo de execução de um sistema usando computação paralela para melhorar seu desempenho é definido como o tempo necessário utilizado para resolver um problema computacional. Segundo KUMAR (1994) o tempo compreendido entre o momento em que o processamento paralelo inicia a sua execução até o momento em que o último processamento termina a execução. O tempo de execução é medido contanto o número de passos consecutivos executado pelo algoritmo, no pior caso, do começo ao fim do processamento (AKL, 1997). 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, 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 (1995) o tempo de execução é formado por três componentes, são eles:

- O tempo de processamento
- O tempo de comunicação
- 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 ou por um conjunto de parâmetros. 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 a 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.

3.3.2 – Fator de aceleração

Segundo FOSTER (1995) a segunda medida de um sistema paralelo é o seu fator de aceleração ou *speedup*. O fator de aceleração é a informação de quanto se ganhou em desempenho pelo uso da computação paralela de uma aplicação que não usa.

A definição formal do fator de aceleração ou *speedup*, S, é a razão entre o tempo de execução seria do melhor algoritmo sequencial 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 aquele utilizado pelo algoritmo sequencial. Essa definição, formalizada por Gene Amdahl, ficou conhecida como Lei de Almdahl ou limites de *speedup* (FOSTER, 1995).

A argumentação de Amdahl procurava demostrar 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, paralelo e serial do programa.

3.3.3 – Eficiência

Segundo JAIN (1991) a eficiência é uma medida da fração do tempo em que o processador é ultimamente empregado, ela é definida como a razão entre o fator de aceleração e o número de processadores.

Em um sistema paralelo, 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 pelo sistema. Matematicamente é demostrada pela fórmula $E = \frac{S}{p}$.

A eficiência também pode ser expressa como a razão entre o tempo de execução do algoritmo sequencial para resolver o problema e o custo de resolver o mesmo problema em *p*

processadores.

3.3.4 – Custo

Segundo KUMAR (1994) o custo para solucionar um problema em um sistema paralelo é o produto do tempo de execução paralelo e o número de processadores utilizado. O custo reflete a soma de tempo que cada processador gasta resolvendo o problema.

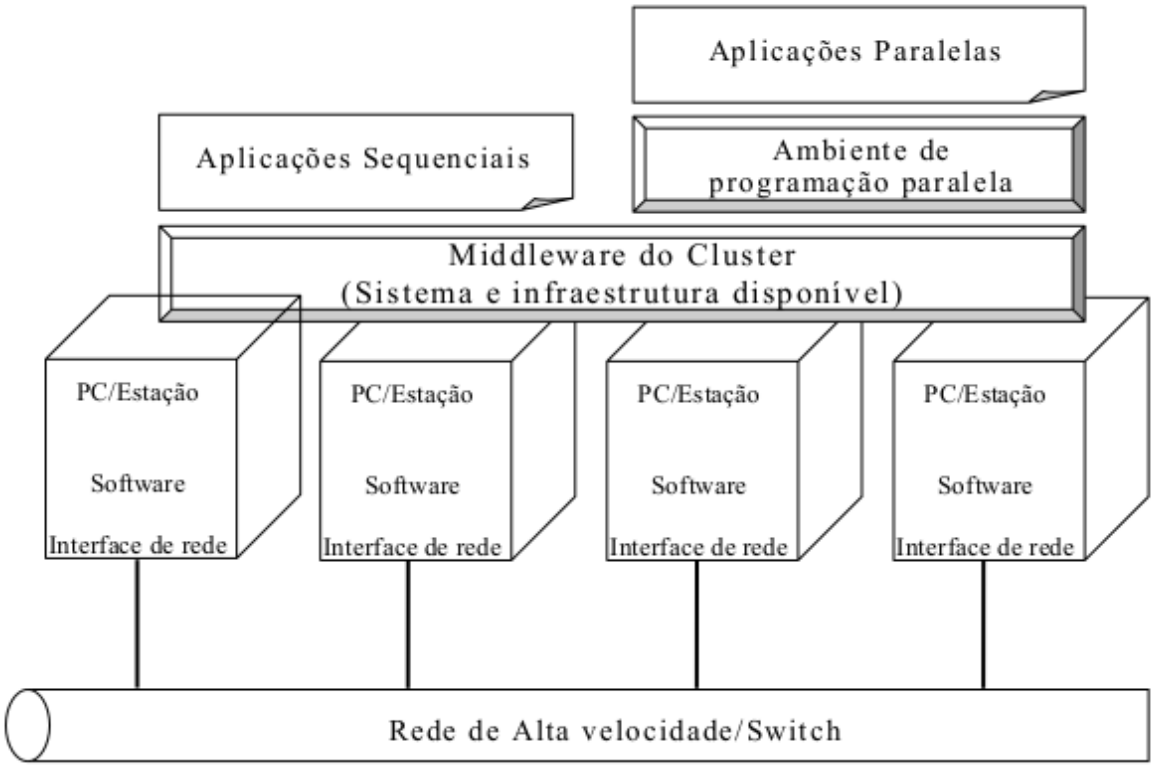
3.4 – Utilização de Clusters e Grades na Computação Paralela

3.4.1 – Clusters

Segundo BUYYA (1999) *cluster* consiste de um conjunto de computadores interconectados trabalhando em conjunto para executar aplicações e integrando recursos computacionais. 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 obtidas com a implementação de *clusters* são o aumento da confiabilidade, distribuição de carga e desempenho.

Também pode-se chamar cada computador no *cluster* de nó. Segundo BUYYA (1999) a arquitetura típica de um *cluster* pode-se ser mostrada na figura abaixo:



Fonte: BUYYA (1999) - *Arquitetura de um cluster de computadores.*

Na imagem acima, pode-se ver que o hardware da rede atua como um processador de comunicação, no qual é responsável pelo recebimento e transmissão de pacotes de dados entre os nós através da rede/*switch*.

O software de comunicação oferece um meio de comunicação rápido e confiável entre os nós e o ambiente externo. Os nós 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, incluído as bibliotecas de passagem de mensagens, depuradores e análise de desempenho das aplicações.

3.4.2 – MPI

Usado para a troca de mensagens entre os nós do *cluster*, sendo mais avançada que a PVM (*Parallel Virtual Machine*), pois pode trabalhar com mensagens para todos os computadores ou para apenas um determinado grupo.

Segundo OTTO (1998) a Interface de Passagem de Mensagem (MPI), é um paradigma de programação 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).

Segundo HUSS-LEDERMAN (1998) a MPI 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 de MPI-1, segundo OTTO (1998) 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, segundo HUSS-LEDERMAN (1998) foi disponibilizada em 1997.

No padrão MPI, uma aplicação é constituída por um ou mias processos que se comunicam, adicionando-se funções para envio e recebimento de mensagens entre os processos. Pode se observar na imagem abaixo a estrutura de um programa MPI, escrito na linguagem de programação C.

```
#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 ();
}
```

Estrutura de um programa MPI

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

3.4.3 – Grades

A grade computacional segundo FOSTER (1999) 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 segundo BERMAN (2003) 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, segundo YANG (2003) o fornecimento transparente de recursos da grade necessários as aplicações ou serviços que necessita de pouco ou nenhum atraso. A má qualidade de serviços na rede pode prejudicar significativamente o fornecimento eficiente de RoD.

Existem várias plataformas de computação em grade, como por exemplo o Globus, Condor, OurGrid, entre muitas outras.

3.4.3.1 – Globus

O Globus toolkit segundo KESSELMAN (1998) é um conjunto de serviços que facilita a computação em grade permitindo a submissão e o controle de aplicações, descobertas de recursos, movimentações de dados e segurança no ambiente da grade.

Com essas características tem 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. Pode-se se observar a arquitetura do Globus na imagem abaixo:

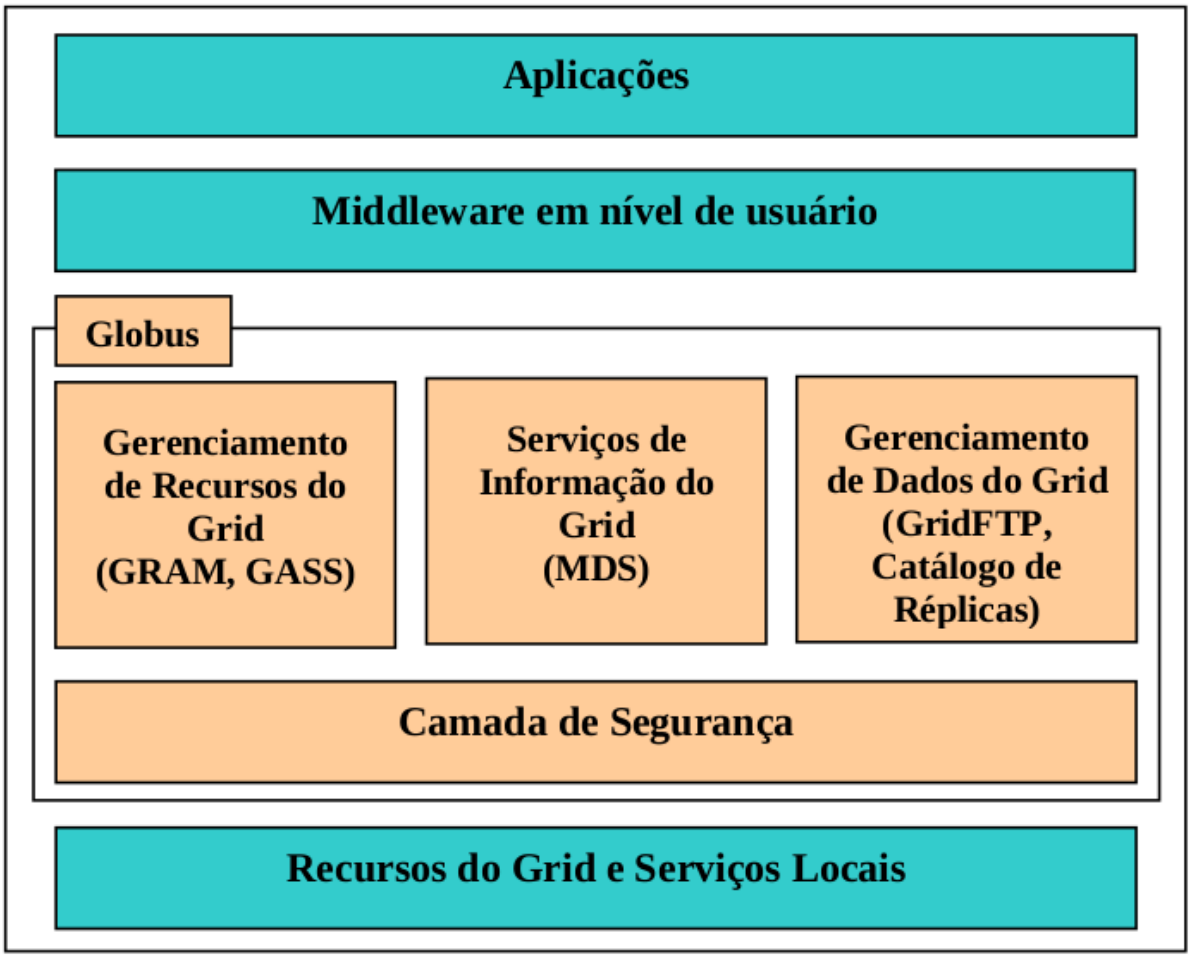


Figura 2: Arquiterura do Globus

Figura 1: Arquitetura Globus

A utilização dos serviços do Globus segundo KARONIS (1998) a instalação e configuração de uma considerável infraestrutura de suporte, onde cada recurso é gerenciando por uma instância do *Globus Resource Allocation Manager* (GRAM), que é responsável por instanciar, monitorar e reportar o estado das tarefas alocadas para o recurso.

Ainda segundo KARONIS (1998) 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 diretórios do Globus, o *Metacomputing Directory Service* (MDS).

3.4.3.2 – Condor

Segundo MUTKA(1998) o Condor surgiu 1984, ele é um *middleware* que aproveita os

ciclos ociosos de 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 de grade.

O Condor tem o objetivo segundo LITZKOW(1998) de fornecer grande quantidade de poder computacional a médio e longo prazo utilizando recursos ociosos na rede, e também a sua alta vazão (*high throunghput*) e não o alto desempenho sustentável (*high performance*), portanto o Condor visa fornecer desempenho sustentável a médio e longo prazo, mesmo que o desempenho instantâneo do sistema possa variar consideravelmente.

3.4.3.3 – OurGrid

O OurGrid segundo CIRNE(2006) 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.

3.5 – Programação Paralela

A programação paralela é uma forma de computação em que o processamento de um algoritmo é divido em partes e executado ao mesmo mesmo tempo, de forma a aumentar o desempenho de processo e deixando mais rápido a execução de softwares.

O OpenMP (*Open Multi-Processing*) segundo PITANGA(2008) é uma API (Interface de Programação de Aplicações) que visa o desenvolvimento de aplicações *multitheads* de uma forma mais fácil em ambientes de programação C/C++ e Fortran. Ainda segundo PINTANGA (2008) é composto por diretivas de compilação de bibliotecas para programação *multitheads*, suportando o modelo de paralelismo dos dados, permitindo o paralelismo incremental e combina partes de código escrito na forma serial e paralela em um único código fonte.

O CPAR (Linguagem de programação paralela) foi criado pela Professora Doutora Liria Matsumoto Sato, e segundo SATO (1995) CPAR é uma extensão paralela da linguagem C que utiliza alguns elementos das linguagens C e Ada, tal como um modelo de programação de multitarefas. Ainda segundo SATO (1995) ela foi projetada visando oferecer construções simples para a exploração do paralelismo em múltiplos nives, permitindo uma melhor

utilização da localidade de memória.

O CPAR oferece variáveis compartilhadas para o compartilhamento das informações entre os processos. A comunicação entre diversos elementos de processamento ocorre através das variáveis compartilhadas.

3.6 – Uso da Programação Paralela

Para testar o desempenho de processamento paralelo, sera usado um algoritmo de multiplicação de matrizes, por usar um grande esforço computacional.

3.6.1 – Algoritmo sequencial

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#define SIZE 1000

float A[SIZE][SIZE];
float B[SIZE][SIZE];
float C[SIZE][SIZE];

void inicializa_matriz(){
   int i,j;
   for (i=0;i < SIZE;i++){
      for(j=0;j<SIZE;j++){
         A[i][j]=3*i+j;
         B[i][j]=i+3*j;
         C[i][j]=0.0;
      }
    }
}
void multiplica_matriz(){
   int i,j,k;
   for (i=0;i < SIZE;i++){
      for(k=0;k<SIZE;k++){
         for(j=0;j<SIZE;j++){
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
         }
      }
   }
}
main(){
   inicializa_matriz();
   struct timeval inicio, final;
   int tempogasto;
   gettimeofday(&inicio, NULL);
   multiplica_matriz();
   gettimeofday(&final, NULL);
   tempogasto = (int) (1000 * (final.tv_sec - inicio.tv_sec) + (final.tv_usec - inicio.tv_usec) / 1000);
```

```
    printf("Tempo decorrido: %d\n", tempogasto);
    return 0;
}
```

3.6.2 – Implementação usando MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
#include <sys/time.h>

int main(int argc , char *argv[]){
    int tam = atoi(argv[1]);
    int size,rank,sum=0,i,j,k;
    int **a, **b, **c;
    struct timeval inicio, final;
    int tempogasto;
    gettimeofday(&inicio, NULL);
    MPI_Status status;

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

    // criando um array de ponteiros
    a =(int **)malloc(tam * sizeof(int*));
    b =(int **)malloc(tam * sizeof(int*));
    c=(int **)malloc(tam * sizeof(int*));

    // alocando memória para as matrizes
    for(i = 0; i < tam; i++){
            a[i]=(int *)malloc(tam * sizeof(int));
            c[i]=(int *)malloc(tam * sizeof(int));
            b[i]=(int *)malloc(tam * sizeof(int));
    }

    // inicializando as matrizes com valor 1
    for(i = 0; i < tam; i++){
            for(j = 0; j < tam; j++){
                a[i][j] = 1;
                b[i][j] = 1;
            }
    }

    // dividindo as tarefas nos vários processos
    for(i = rank; i < tam; i = i+size){
            for(j = 0; j < tam; j++){
                sum=0;
                for(k = 0; k < tam; k++){
                        sum = sum + a[i][k] * b[k][j];
                }
                c[i][j] = sum;
            }
    }
```

```
    // Se o rank for diferente de um, ele envia o resultado
    // para o rank principal
    if(rank != 0){
            for(i = rank; i < tam; i = i+size){
                MPI_Send(&c[i][0], tam, MPI_INT, 0, 10+i, MPI_COMM_WORLD);
            }
    }

    // Sendo o rank principal, ele vai receber todas as partes calculadas
    // de cada um dos outros processos
    if(rank == 0){
            for(j = 1; j < size; j++){
                for(i = j; i < tam; i = i+size){
                        MPI_Recv(&c[i][0], tam, MPI_INT, j, 10+i, MPI_COMM_WORLD, &status);
                }
            }
    }
    MPI_Barrier(MPI_COMM_WORLD);

    // Finaliza
    if(rank == 0){
            MPI_Finalize();
    }
    if(rank == argc){
            gettimeofday(&final, NULL);
        tempogasto = (int) (1000 * (final.tv_sec - inicio.tv_sec) + (final.tv_usec - inicio.tv_usec) / 1000);
        printf("Tempo decorrido: %d\n", tempogasto);
    }

    return 0;
}
```

3.6.3 – Implementação usando OpenMP

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#define SIZE 1000

int size;
float A[SIZE][SIZE];
float B[SIZE][SIZE];
float C[SIZE][SIZE];
main(){
    int nproc;
    int i, j, k, n;
    size = SIZE;
    /* num de proc */
    nproc =2;
    struct timeval inicio, final;
    int tempogasto;

    #pragma omp parallel shared(A, B, C, size) private(i,j, k)
    {
```

```
    #pragma omp for
        for(i=0; i <size; i++){
                for(j=0;j<size;j++){
                    A[i][j]=3*i+j;
                    B[i][j]=i+3*j;
                    C[i][j]=0.0;
                }
        }

        gettimeofday(&inicio, NULL);
        #pragma omp for
        for(i=0; i<size; i++){
                for(k=0;k<size;k++){
                    for(j=0;j<size;j++){
                        C[i][j]=C[i][j]+A[i][k]*B[k][j];
                    }
                }
        }

    }

    gettimeofday(&final, NULL);
    tempogasto = (int) (1000 * (final.tv_sec - inicio.tv_sec) + (final.tv_usec - inicio.tv_usec) / 1000);
    printf("Tempo decorrido: %d\n", tempogasto);

}
```

3.6.4 – Implementação usando forks

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// definindo as matrizes globais
int **a, **b, **c;
// definindo tamanho da matriz e num de threads
int tam, num_proc;

// função para gerar uma matriz
// que vai preenchendo somando 1
int **gera_matriz(){
    int i;
    int *valores, **temp;

    // alocando memória
    valores = (int *)malloc(tam * tam * sizeof(int));
    temp = (int **)malloc(tam * sizeof(int*));

    for (i = 0; i < tam; i++){
            temp[i] = &(valores[i * tam]);
    }
    // retorna para poder ocupar na main
```

```
    return temp;
}

void multiplica_matriz(int proc_id){
    int i, j, k;
    int inicio, final;

    float passo = ceil((float)tam/num_proc);
    inicio = proc_id * passo;
    final = (proc_id + 1)* passo - 1;
    if(final > tam){
            final = tam -1;
    }

    // multiplicação
    printf("Inicio %d => Final %d \n", inicio, final );
    for (i = inicio; i <= final; i++){
            for (j = 0; j < tam; j++){
                c[i][j] = 0;
                for ( k = 0; k < tam; k++){
                        c[i][j] += a[i][k]*b[k][j];
                }
       }
    }
}

int main(int argc, char* argv[]){
    int i,j;
    //tempo
    struct timeval iniciotmp, finaltmp;
    int tempogasto;

    // definindo o tamanho da matriz
    tam = atoi(argv[1]);

    // definindo tamanho da matriz
    num_proc = atoi(argv[2]);

    // Tratamento para evitar que o número de threads
    // seja maior que o tam da matriz
    if(num_proc > tam){
            printf("O número de threads é maior que o tamanho da matriz. Por favor selecione um número
menor de threads.\n");
            return 0;
    }

    // alocando matriz
    a = gera_matriz();
    b = gera_matriz();
    c = gera_matriz();

    // populando matriz com valor 1
    for(i=0;i<tam;i++){
            for(j=0;j<tam;j++){
               a[i][j] = 1;
```

```
            b[i][j] = 1;
        }
    }

    int k;
    int process[num_proc];
    gettimeofday(&iniciotmp, NULL);

    // Chamo o processo, ele faz o cálculo
    // e então mato para o próximo continuar
    for(k=0;k<num_proc;k++){
        process[k] = fork();
        if(process[k] == 0){
            exit(0);
        }
        multiplica_matriz(k);
        wait(NULL);
    }

    gettimeofday(&finaltmp, NULL);
        tempogasto = (int) (1000 * (finaltmp.tv_sec - iniciotmp.tv_sec) + (finaltmp.tv_usec - iniciotmp.tv_usec) / 1000);
    printf("Tempo decorrido: %d\n", tempogasto);
    return 0;
}
```

3.6.5 – Implementação usando threads

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

// definindo as matrizes globais
int **a, **b, **c;
// definindo tamanho da matriz e num de threads
int tam, num_thread;

// função para gerar uma matriz
// que vai preenchendo somando 1
int **gera_matriz(){
    int i;
    int *valores, **temp;

    // alocando memória
    valores = (int *)malloc(tam * tam * sizeof(int));
    temp = (int **)malloc(tam * sizeof(int*));

    for (i = 0; i < tam; i++){
        temp[i] = &(valores[i * tam]);
    }
    // retorna para poder ocupar na main
    return temp;
}

void multiplica_matriz(int thread_id){
```

```
    int i, j, k;
    int inicio, final;

    float passo = ceil((float)tam/num_thread);
    inicio = thread_id * passo;
    final = (thread_id + 1)* passo - 1;
    if(final > tam){
            final = tam -1;
    }

    // multiplicação
    printf("Inicio %d => Final %d \n", inicio, final );
    for (i = inicio; i <= final; i++){
            for (j = 0; j < tam; j++){
                c[i][j] = 0;
                for ( k = 0; k < tam; k++){
                        c[i][j] += a[i][k]*b[k][j];
                }
        }
    }
}

int main(int argc, char* argv[]){
    int i,j;
    pthread_t *threads;
    //tempo
    struct timeval iniciotmp, finaltmp;
    int tempogasto;


    // definindo o tamanho da matriz
    tam = atoi(argv[1]);

    // definindo tamanho da matriz
    num_thread = atoi(argv[2]);

    // Tratamento para evitar que o número de threads
    // seja maior que o tam da matriz
    if(num_thread > tam){
            printf("O número de threads é maior que o tamanho da matriz. Por favor selecione um número
menor de threads.\n");
            return 0;
    }

    // alocando matriz
    a = gera_matriz();
    b = gera_matriz();
    c = gera_matriz();

    // populando matriz com valor 1
    for(i=0;i<tam;i++){
            for(j=0;j<tam;j++){
              a[i][j] = 1;
              b[i][j] = 1;
            }
```

```
    }
    gettimeofday(&iniciotmp, NULL);
    // alocando as threads
    threads = (pthread_t *)malloc(num_thread * sizeof(pthread_t));

    // criando as threads e chamando multiplica
    for(i=0; i < num_thread; i++){
            pthread_create(&threads[i], NULL, multiplica_matriz, i);
    }

    for (i = 0; i < num_thread; i++){
            pthread_join(threads[i], NULL);
    }

    // libera as memórias alocadas
    free(threads);

    gettimeofday(&finaltmp, NULL);
    tempogasto = (int) (1000 * (finaltmp.tv_sec - iniciotmp.tv_sec) + (finaltmp.tv_usec - iniciotmp.tv_usec)
/ 1000);
    printf("Tempo decorrido: %d\n", tempogasto);
    return 0;
}
```

3.6.6 – Implementação usando CPAR

```
/*Multiplicação de matrizes cpar*/
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

shared int size;
shared float A[1000][1000];
shared float B[1000][1000];
shared float C[1000][1000];

task spec inicializa_matriz();
task body inicializa_matriz(){
    int i,j;
    size=1000;
    forall i=0 to size-1 {
      for(j=0;j<size;j++){
          A[i][j]=3*i+j;
          B[i][j]=i+3*j;
          C[i][j]=0.0;
      }
    }
}
task spec multiplica_matriz();
task body multiplica_matriz(){
    int i,j,k;
    size=1000;
    forall i=0 to size-1 {
      for(k=0;k<size;k++){
        for(j=0;j<size;j++){
```

```
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
          }
        }
      }
}

main(){
    int nprocs=4;
    //tempo
    struct timeval iniciotmp, finaltmp;
    int tempogasto;
    alloc_proc(nprocs);
    create nprocs, inicializa_matriz();
    wait_all();
    gettimeofday(&iniciotmp, NULL);
    create nprocs, multiplica_matriz();
    wait_all();
    gettimeofday(&finaltmp, NULL);
    tempogasto = (int) (1000 * (finaltmp.tv_sec - iniciotmp.tv_sec) + (finaltmp.tv_usec - iniciotmp.tv_usec)
/ 1000);
    printf("Tempo decorrido: %d\n", tempogasto);
    return 0;
}
```

3.7 - Resultados obtidos

Para realizar os testes foi utilizado um computador com Intel Core i3-2375M 4 x 1.50GHz utilizando Ubuntu 14.04 LTS 64 bits, executando sempre nas mesma condições os algoritmos.

Ao executar o algoritmo sequencial com o tamanho de 500 foi obtido o resultado de 1095 milissegundos.

Nº proc	Cpar	OpenMP	forks	threads	MPI
2	96	966	1780	1538	252
4	107	962	1742	820	1063
6	74	966	1738	807	1077
8	63	967	1767	801	1123

Tabela 2: Executando matriz de 500

Ao executar o algoritmo sequencial com o tamanho de 1000 foi obtido o resultado de 8728 milissegundos.

Nº proc	Cpar	OpenMP	forks	threads	MPI
2	707	7722	15229	8749	9075
4	712	7763	15370	7569	9663
6	684	7710	15525	7361	9456
8	670	7716	15270	7048	9599

Tabela 3: Executando matriz de 1000

Ao executar o algoritmo sequencial com o tamanho de 1500 foi obtido o resultado de 29484 milissegundos.

Nº proc	Cpar	OpenMP	forks	threads	MPI
2	2342	26096	75843	32212	33995
4	2368	25931	59995	25604	38719
6	2441	25914	60445	26664	39139
8	2406	26117	58181	26421	39193

Tabela 4: Executando matriz de 1500

4 – CONCLUSÃO

Um dos objetivos deste trabalho é apresentar os conceitos e funcionamento da computação paralela, o que demonstrou que é atualmente essencial para sistemas computacionais, onde o tempo otimizado de uma resposta de processamento torna-se cada vez mais uma exigência. Apesar da grande evolução do hardware que permite um melhor desempenho do processamento e de baixo custo, se for utilizado de forma errada, apesar do alto potencial do hardware, pode acabar por obter um tempo maior de resposta, e a utilização errada do paralelismo no sistema também pode ter um desempenho prejudicado.

A computação paralela tem como objetivo dividir os processos para que seja possível uma redução do tempo de processamento assim como o esforço da infraestrutura. Atualmente ficou mais fácil essa implantação em diversos nível de sistemas, pois com a evolução do hardware, pode-se obter a utilização dos processadores multicores, assim tornando a utilização do paralelismo essencial para um melhor aproveitamento da infraestrutura, assim como a utilização de *clusters* e *grids*.

A utilização de *cluster* e *grids* ajudam e muito a obtenção de um melhor desempenho, onde o MPI se destaca quando utilizada dessas tecnologias que interligam computadores, para poder dividir seus processos entre eles.

Através de um teste de desempenho comparativo entre as principais formas de utilização de paralelismo em algoritmos, onde apresentou um melhor desempenho a utilização do CPAR no ambiente de teste utilizando, ressaltando que a utilização do OpenMP e do MPI oferece um melhor desempenho em ambientes onde são utilizados *cluster* e *grids*, a utilização de *forks* foi o que resultou em um pior desempenho, onde seu tempo de processamento ficou acima da utilização do algoritmo sequencial, onde os demais apresentaram pouca diferença, dependendo do uso de processadores e do tamanho da matriz utilizada.

Para realizar a aplicação em qualquer sistema deve-se ser feito uma análise para que assim saiba qual é a melhor opção para o sistema e o ambiente usado, para que assim possa-se obter um resultado melhor, alcançando o objetivo de diminuir o tempo de resposta da aplicação proposto pelo paralelismo.

Assim pode-se concluir que, o uso da computação paralela em diversos áreas,

independente do tamanho do sistema e do uso, pode melhorar o desempenho do processamento, desde que seja feito um estudo para descobrir qual opção apresenta um melhor desempenho para o ambiente utilizado.

REFERÊNCIAS BIBLIOGRÁFICAS

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

TANENBAUM, A. S. Organização Estruturada de Computadores. 5. ed. Prentice Hall, 2007.

Morimoto, Carlos E. - Manual de Hardware Completo 3º ed, 2000.

Peruzzo, J. Física Quântica, Clube de Autores, 2010.

Peruzzo, J. Fronteiras Da Física, Clube de Autores, 2008.

András Vajda. Programming Many-Core Chips. Springer Science & Business Media, 2011

Douglas Camargo Foster1. Arquiteturas Multicore. Programa de Pós-Graduação em Informática (PPGI) – Universidade Federal de Santa Maria (UFSM), 2009.

Coulouris, George; Jean Dollimore; Tim Kindberg; Gordon Blair. Distributed Systems: Concepts and Design (5th Edition), 2011.

Quinn, Michael J. , Parallel Programming in C with MPI and OpenMP, Mc Graw Hill, 1o edição, 2003.

Taurion, Cezar. Big Data – Brasposrt, 2013.

Fonseca Filho, Cléuzio, História da computação: O Caminho do Pensamento e da Tecnologia, Edipucrs, 2007.

Null, Linda e Lobur – Princípios Básicos de Arquitetura e Organização de Computadores – Bookman, 2011.

TOP500.org – Acessado em 28 de março de 2015 – disponível através do endereço – http://www.top500.org.

Titan – Acessado em 28 de marco de 2015 – disponível através do endereço - https://www.olcf.ornl.gov/titan/

WHITE, Tom. Hadoop: The Definitive guide. O‟Reilly. 2009.

Tanembaum, Andrew S. and Steen, Maarten, “Distributed Systems: principies and paradigms”. 2a Edição, 2006.

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

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

AKL, S. G. Parallel Computation: models and methods. Prentice-Hall, 1997.

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

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

PITANGA, M. Computação em Cluste. Acessado em 06 de maio de 2015 – disponível através do endereço http://www.clubedohardware.com.br/artigos/153.

ALECRIM, E. Cluster: Conceitos e Características. Acessado em 06 de maio de 2015 – disponível através do endereço http://www.infowester.com/cluster.php.

Mosix Cluster and Grid Management. Acessado em 06 de maio de 2015 – disponível através do endereço http://www.mosix.org.

The Beowulf Cluster Acessado em 06 de maio de 2015 – disponível através do endereço http://www.beowulf.org.

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.

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

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

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

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

2003.

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.

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

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.

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.

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.

PITANGA, M. - Construindo Supercomputadores com Linux. Brasport. 2008

SATO, L. M. Ambientes de programação para sistemas paralelos e distribuídos. Tese de Livre Docência, Escola Politécnica da Universidade de São Paulo, 1995.