FABIO VALLE REGO GORINO

BALANCEAMENTO DE CARGA EM CLUSTERS DE ALTO DESEMPENHO: UMA EXTENSÃO PARA A LAM/MPI

MARINGÁ

2006

FABIO VALLE REGO GORINO

BALANCEAMENTO DE CARGA EM CLUSTERS DE ALTO DESEMPENHO: UMA EXTENSÃO PARA A LAM/MPI

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Ronaldo Augusto de Lara Gonçalves

MARINGÁ

2006

Ficha catalográfica elaborada pelo setor de processos técnicos BICEN/UEPG.

Gorino, Fabio Valle Rego
G669 Balanceamento de carga em clusters de alto desempenho: uma extensão para LAM/MPI. Maringá, 2006.
91f. : il.

Dissertação (mestrado) - Universidade Estadual de Maringá – Pr.

Orientador: Prof. Dr. Ronaldo augusto de Lara Gonçalves

1- Balanceamento de carga. 2- Clusters Beowulf. 3 – Plataforma LAM/PI.

CDD 004.35

FABIO VALLE REGO GORINO

BALANCEAMENTO DE CARGA EM CLUSTERS DE ALTO DESEMPENHO: UMA EXTENSÃO PARA A LAM/MPI

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em

BANCA EXAMINADORA

Prof. Dr. Ronaldo Augusto de Lara Gonçalves
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. João Angelo Martini
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Marcio Augusto de Souza
Universidade Estadual de Ponta Grossa – Deinfo/UEPG

AGRADECIMENTOS

Ao Prof. Ronaldo A. L. Gonçalves, meu orientador, que recebe adequadamente este título, pelas correções e adequações realizadas neste trabalho e por, diversas vezes, ter se disposto a resolver problemas nos equipamentos, me ajudando quando eu não podia estar em Maringá.

A minha mãe, pela preocupação e todo o apoio possível durante o período em que morei em Maringá, assim como aos meus irmãos que me ajudaram como puderam quando precisei.

A Deus, que acredito, muitas vezes foi solicitado pelas pessoas citadas para zelar por mim, e sempre me fez relembrar das minhas capacidades, me levando para frente.

EPÍGRAFE

"O fracasso jamais me surpreenderá se minha decisão de vencer for suficientemente forte." (Og Mandino).

RESUMO

Com a popularização dos clusters Beowulf, desde 1994, e a difusão da filosofia "faça você mesmo" um cluster pessoal, cada vez é mais comum encontrar clusters heterogêneos constituídos por equipamentos convencionais que, muitas vezes, não satisfazem mais as necessidades dos usuários quando usados isoladamente. Normalmente, esses equipamentos são utilizados por meio de uma plataforma de software que provê suporte ao desenvolvimento e execução de aplicações distribuídas, tal como a plataforma LAM/MPI.

Entretanto, a simples conexão destes recursos computacionais em um ambiente de execução paralela e distribuída não garante alto desempenho para as aplicações. O potencial de desempenho de um cluster não será bem explorado se os recursos disponíveis não forem utilizados de forma equilibrada e de acordo com o poder computacional de cada computador conectado. Para maximizar o desempenho dos clusters heterogêneos, técnicas de balanceamento de carga podem ser usadas para distribuir a carga de acordo com os recursos, permitindo torná-los mais eficientes e adiando o estado de obsoleto dos recursos mais antigos.

Apesar disto, uma das bibliotecas de programação paralela existentes, bem conhecida e muito usada atualmente, a LAM/MPI, não utiliza técnicas sofisticadas de balanceamento de carga e distribui carga segundo o método *round-robin*, em função da sua facilidade de implementação, não explorando assim a heterogeneidade do cluster eficientemente.

O presente trabalho contextualiza o balanceamento de carga, investigando diversos trabalhos relatados e realiza experimentações reais sob diferentes situações de carga. Os experimentos reais foram realizados em uma seqüência de complexidade progressiva que permitiram propor e avaliar o uso de alguns algoritmos de balanceamento de carga para a

plataforma LAM/MPI, melhorando o seu desempenho.

Os algoritmos propostos foram avaliados em um cluster de 8 nodos na execução de diferentes aplicações. A primeira versão foi testada sobre as aplicações de multiplicação matricial e cálculo integral, mostrando resultados superiores a 50%, com pico de 59% no melhor caso. Duas outras versões foram testadas em uma aplicação paralela de reconhecimento de seqüências de DNA. Os resultados também mostram um desempenho superior a 57% sobre a plataforma original no melhor caso.

Estes resultados comprovam a importância do uso de algoritmos de balanceamento mais elaborados e mostram que a extensão da LAM/MPI provê resultados satisfatórios para diferentes tipos de aplicações. Uma outra questão importante é que as aplicações não precisam ser modificadas para se beneficiar do balanceamento de carga, o qual é transparente para o usuário da biblioteca LAM/MPI.

Palavras-chave: Balanceamento de Carga, Cluster Beowulf, Plataforma LAM/MPI.

ABSTRACT

In view of the popularization of Beowulf clusters since 1994 and the spread of "do-it-yourself" philosophy, heterogeneous clusters have become more and more common to find which are constituted by conventional equipments that, many times, no more satisfy the necessities of the users when running alone. Normally, these equipments are used by means of a software platform that provides support for developing and executing distributed applications, such as the LAM/MPI platform.

However, the simple connection among these computational resources in an environment for parallel and distributed execution does not assure high performance for the applications. The performance potential of a cluster will not be well explored if the available resources are not used in an equalized mode and according to the computational power of each connected computer. To maximize the performance of heterogeneous clusters, load balancing techniques can be used to allocate the load according to the resources available in each node, which helps recover the cluster effectiveness and spare the obsolete status of the computational resource.

In spite of this, one of the libraries for parallel programming, well-known and most used actually, the LAM/MPI, does not use sophisticated techniques for load balancing and distributes the load in a round-robin fashion, because of its implementation easiness, what not explore heterogeneity of the cluster efficiently.

The present work shows an overview about load balancing, investigating several related works and makes real experimentations under different load situations. The real experiments were made in a sequence of progressive complexity that allowed proposing and evaluating the use of some load balancing algorithms for the LAM/MPI, improving its performance.

The proposed algorithm was evaluated on 8-nodes cluster in an execution of a parallel application of genetic sequences recognition. The results showed a very significant performance improvement over the original platform. Other applications, matrix multiplication and integral calculation, were also evaluated and presented similar performance gain, it indicates that the algorithm implemented is not aimed specifically for a group of applications. Another important point is that none of the applications have been modified in order to fit the load balancing, resulting in a transparent method for the LAM/MPI library user.

Keywords: Load Balancing, Beowulf Cluster, LAM/MPI Platform.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Após a divulgação do cluster Beowulf, construído por Thomas Sterling e Donald Becker em 1994, os clusters começaram a se popularizar em virtude do custo de aquisição destes ser inferior ao custo de um supercomputador vetorial. Em 2000 já existiam milhares de clusters Beowulf operando nos ambientes mais variados. Em junho de 2001, 28 clusters Beowulf estavam entre os 500 sistemas mais rápidos do mundo (BELL & GRAY, 2002).

A aplicabilidade dos clusters também cresceu (meteorologia, geografia, genética, física quântica, computação gráfica), juntamente com a quantidade de comunidades científicas e comerciais que perceberam o potencial de processamento destes sistemas a um custo relativamente acessível. A partir de então, os grupos de pesquisas do ensino superior puderam adquirir e montar seus clusters utilizando a filosofia do "faça você mesmo" (MEREDITH, 2003). Entretanto, conforme os clusters conquistavam novas áreas de aplicação, se tornaram maiores e começaram a apresentar dificuldades, tais como falta de escalabilidade, baixa disponibilidade e desempenho insatisfatório, entre outras.

Estas dificuldades estão sendo superadas com o uso de técnicas de tolerância a falhas, balanceamento de carga, otimização de códigos e bibliotecas paralelas, inclusive com a combinação destas e de outras soluções existentes para resolver uma ampla gama de problemas. Estas soluções podem ser implementadas pela própria aplicação paralela, pela biblioteca que suporta a programação paralela ou em camadas intermediárias, objetivando sempre tirar máximo proveito dos recursos do cluster.

Em clusters heterogêneos, que conectam nodos com recursos de tipos e/ou capacidades distintas, o baixo desempenho pode surgir se as diferenças no poder de processamento entre os nodos não forem observadas e tratadas de forma correta. Atualmente, devido ao rápido avanço da tecnologia de processadores e a necessidade de aproveitar os

equipamentos já existentes, os clusters tendem a se tornarem heterogêneos (BEVILACQUA, 1999, BOHN & LAMONT, 1999).

Um cluster destinado a aplicações paralelas de alto desempenho, formado por nodos heterogêneos, pode apresentar baixo desempenho se todos os nodos forem tratados indiscriminadamente. Este baixo desempenho tende a ser mais expressivo quanto maior for a diferença entre o poder computacional dos nodos (BOHN & LAMONT, 1999). Buscando aumentar o desempenho, técnicas de balanceamento de carga podem ser aplicadas ao cluster, fazendo com que os nodos recebam carga de acordo com as capacidades de seus recursos computacionais (DROZDOWSKI & WOLNIEWICZ, 2003).

Apesar disto, uma das bibliotecas de programação paralela mais conhecida e usada atualmente, a LAM/MPI, se limita a distribuir carga segundo o método *round-robin*, em função da sua facilidade de implementação, e assim não explora de forma eficiente as variações de carga do cluster. Neste contexto, este trabalho investiga as questões mais relevantes na área de balanceamento de carga com o objetivo de aumentar o desempenho de clusters baseados na plataforma LAM/MPI.

Com base nesta investigação, vários experimentos práticos foram realizados em um cluster heterogêneo de 8 nodos, com o objetivo de avaliar diferentes configurações, desvendar as situações em que o desempenho pode ser maior, combinar idéias e propor uma extensão para a biblioteca LAM/MPI. Esta extensão incorpora os algoritmos experimentados de forma a possibilitar que a LAM/MPI realize o balanceamento de carga.

O presente trabalho está organizado conforme segue. O capítulo 2 apresenta uma revisão de conceitos e de trabalhos relacionados ao estudo de cluster de computadores. O capítulo 3 apresenta uma revisão de conceitos e de trabalhos relacionados ao estado da arte em balanceamento de carga. O capítulo 4 apresenta o padrão MPI e a plataforma LAM/MPI, focando principalmente a lógica de funcionamento, implementação do *mpirun* e mostrando os

pontos do código que devem ser alterados para incluir o balanceamento de carga. No capítulo 5 são discutidas questões de implementação, assim como são apresentados os resultados dos experimentos da extensão elaborada. No capítulo 6 são apresentadas as conclusões e algumas sugestões para trabalhos futuros. As referências bibliográficas são apresentadas no final.

2 CLUSTERS DE COMPUTADORES.

Segundo Sterling, um dos pioneiros na área, um cluster é um sistema de computação composto por computadores de varejo (*off the shelf*), denominados nodos, conectados entre si por uma rede local e frequentemente gerenciados como uma única máquina, sendo então denominado de cluster Beowulf (STERLING, 2000, HAWICK et al., 1999). O cluster Beowulf surgiu como um supercomputador de baixo custo, para resolver uma grande variedade de problemas e deve permitir uma boa escalabilidade das aplicações para que apresente um custo-benefício aceitável (UNDERWOOD et al., 2001).

Os clusters são projetados para sustentar alto desempenho sobre um único problema, prover alta taxa de saída para um conjunto de aplicações diferentes (*throughput*), manter alta disponibilidade dos nodos e permitir amplo acesso aos discos e canais de E/S (MEREDITH et al., 2003). As características de disponibilidade e escalabilidade são muitas vezes exploradas comercialmente (PFISTER, 1996), mas a característica mais importante do cluster Beowulf é a possibilidade de que as aplicações continuem funcionando por diversas gerações de computadores, protegendo os investimentos realizados. O cluster Beowulf é uma alternativa aos supercomputadores vetoriais e aos clusters proprietários, normalmente encontrados nos supercentros de computação (BELL & GRAY, 2002).

2.1 Aplicabilidade dos Clusters.

O uso de clusters vem sendo explorado nas mais diferentes comunidades devido a tal solução oferecer um equilíbrio entre o desempenho desejado e o custo do sistema. Mas os clusters não são utilizados somente em sistemas em que se busca alto desempenho, sua aplicabilidade se estende a outras áreas, e pode ser dividida em quatro categorias básicas quanto à funcionalidade: alta disponibilidade, balanceamento de carga e computação de alto desempenho (BAKER, 2000).

É possível também classificar um cluster quanto a sua estrutura: pilhas de PCs, clusters de workstation e clusters distribuídos (grids). Um cluster pode ainda ser chamado de homogêneo ou heterogêneo. Um cluster é dito homogêneo quando todos os seus nodos são formados por máquinas semelhantes e utiliza uma única tecnologia de comunicação entre os nodos. Um cluster heterogêneo pode ser formado por diferentes máquinas e diferentes tecnologias de comunicação entre seus nodos (BOHN & LAMONT, 1999, HAASE et al., 2005). Um motivo para esta característica, é a adição de nodos mais modernos ao conjunto de nodos que já faziam parte do cluster, resultando em um cluster heterogêneo.

2.1.1 Cluster de Alta Disponibilidade.

Este tipo de cluster deve ser utilizado quando, como o próprio nome diz, o sistema precisa estar disponível o maior tempo possível, ou seja, respondendo as solicitações recebidas em 99,9% dos casos, ou mais. A forma para se atingir a alta disponibilidade é fazendo-se uso de técnicas de redundância, em que se busca eliminar os pontos únicos de falha do sistema (SPOF – *Single Point Of Failure*). Para isso, criam-se réplicas dos serviços ou duplicam-se componentes de hardware como interfaces de rede, discos rígidos, fontes de alimentação e etc. (JALOTE, 1994).

Essa redundância permite que em caso de falha de um componente (software ou hardware) duplicado, o mesmo possa ser substituído por um componente secundário. Uma abordagem é manter os componentes secundários sempre atualizados e ativos, mas em um estado de espera. Assim que o componente principal é detectado como falho, a réplica toma o seu lugar.

2.1.2 Cluster para Balanceamento de Carga.

Este tipo de cluster é utilizado quando um serviço é requisitado por muitos usuários e existem diversas máquinas responsáveis por responder pelo mesmo serviço. Para

garantir que todas as máquinas recebam uma carga de trabalho equivalente, *daemons* (programas de monitoramento) estão ativos em cada servidor, monitorando a utilização da CPU e outros recursos. Com o resultado da monitoração, os servidores são capazes de realizar uma redistribuição da carga de tarefas, balanceando a utilização de todos os servidores. Se um nodo falhar, as requisições são redistribuídas entre os nodos disponíveis.

Um cluster com balanceamento de carga possui uma alta escalabilidade e pode ser facilmente expandido com a inclusão de novos nodos.

2.1.3 Cluster para Computação de Alto Desempenho.

Quando a palavra cluster é pronunciada, a primeira coisa que passa pela cabeça é alto desempenho. De fato, foi buscando alto desempenho que Sterling classificou seu conjunto de computadores de cluster. Desde então, cluster e alto desempenho têm se apresentado em conjunto. Esse tipo de cluster é o mais comum entre as comunidades científicas, sistemas de previsão e simulação, tarefas típicas que exigem alto poder de processamento. Sua função é conceitualmente simples: dado um problema complexo e identificado como “paralelizável”, um servidor (mestre) deve ser responsável por dividir este problema em inúmeros pedaços a serem processados em nodos escravos (nodos dedicados ao processamento). Assim que cada nodo encontrar a sua solução, ele a envia ao servidor para que a solução completa do problema possa ser montada. O grande desafio é encontrar uma solução arquitetural universal para esse tipo de problema, já que cada caso exige um modo de divisão próprio e uma implementação de software de processamento paralelo específica.

2.2 Estruturas de Cluster.

Existem três formas de se ter um cluster organizado estruturalmente, a diferença básica entre as estruturas é se o recurso computacional está dedicado ou não exclusivamente para ao processamento do cluster e a localização física dos nodos.

- Pilha de PCs: Esta é a estrutura comumente adotada por comunidades científicas, em que geralmente o uso do cluster é destinado a aplicações de alto desempenho. Basicamente um cluster desta categoria pode ser descrito como um empilhamento de gabinetes (escravos) que são gerenciados por um computador mestre (servidor) através da rede. O escravo não possui monitor, teclado ou mouse, por estes motivos os nodos escravos podem ser empilhados formando os *Pile of PCs* (PoPCs) (CARNS et al., 1999). Todo o cluster é configurado e operado pelo servidor, que pode também ser responsável por dividir as tarefas e enviá-las para os escravos. Assim que a tarefa é concluída pelo escravo, ele a envia de volta para o mestre para que a solução completa seja montada e visualizada.
- Cluster de *workstations*: Diferente da pilha de PCs, aqui todos os nodos são computadores completos, com monitor, teclado, mouse e ainda outros dispositivos. Neste caso, os nodos do cluster são utilizados pelos usuários, para navegar na Internet, processamento de texto e todo o tipo de trabalho que uma *workstation* pode realizar. O cluster, efetivamente, só é utilizado em períodos em que as estações estão ociosas, por exemplo, durante a noite. O nodo mestre, normalmente se encontra em local físico diferente e protegido por mecanismos de segurança. Neste tipo de cluster é comum o uso de agendamento de tarefas. O usuário submete a tarefa escrita em lote, as tarefas são armazenadas em uma fila e serão executadas uma a uma quando o cluster estiver funcionando. Os resultados são colhidos e devolvidos para cada usuário.
- Cluster distribuído ou Grid: O cluster distribuído, como o próprio nome sugere é um sistema paralelo e distribuído, composto por uma coleção de

recursos computacionais (clusters) interconectados, que se comunicam através de redes não locais (WAN). Os clusters que compõem o grid podem estar estruturados tanto em pilhas de PCs como em cluster de workstations.

2.3 Trabalhos Relacionados a Cluster de Computadores.

Devido ao elevado potencial de processamento paralelo e distribuído que os clusters oferecem, muitos grupos de pesquisa e desenvolvimento têm buscado soluções que permitam o máximo aproveitamento destes (CHIOLA, 1998). A seguir são apresentadas diferentes técnicas, envolvendo modelos analíticos, simuladores, ferramentas e experimentos reais, grande parte baseada em MPI.

Hoganson (HOGANSON, 1999) desenvolveu um modelo analítico para análise de desempenho de clusters interconectados, o qual explora diferentes estratégias de alocação de processos a processadores de um ambiente multiprogramado (*multi tasks*). Fazendo uso de heurísticas, o modelo apresenta a melhor relação entre o tamanho do cluster e a fração da aplicação paralela dedicada a cada segmento do cluster, de forma a aumentar o paralelismo e reduzir a sobrecarga na comunicação. Nguyen e Peirre (NGUYEN & PIERRE, 2001) também criaram um modelo analítico e o experimentaram em um simulador de um sistema de arquivos paralelos, com o intuito de estudar a escalabilidade dos clusters. Como resultado, mostrou que o crescimento de um cluster depende do grau de saturação da rede de comunicação. A saturação da rede de comunicação limita a escalabilidade do cluster.

Andresen (ANDRESEN et al., 2003) apresentou um sistema de monitoração da comunicação em clusters Beowulf. O sistema pode ser utilizado no nível de processo, com baixo consumo de recursos, e foi batizado de DISTOP. Outra ferramenta deste tipo foi elaborada por Agarwala (AGARWALA et al., 2003), chamada DPROC, a qual é capaz de capturar os dados e comunicações do sistema operacional e exportá-los para as aplicações no formato XML. Ching-Li (LI et al., 2005) propôs uma ferramenta gráfica para a visualização

de programas paralelos MPI que visa facilitar a depuração da execução e o balanceamento manual de carga. Outro estudo realizou uma comparação entre diversas ferramentas de configuração e manutenção de cluster (MEREDITH et al., 2003).

Nupairoj (NUPAIROJ & NI, 1994) investigou várias características e funcionalidades de quatro implementações MPI: LAM, MPICH, Chimp e Unify, concluindo que a sobrecarga de todas elas é alta e precisa ser reduzida. Entretanto, elegeu a LAM como a melhor delas em termos de estabilidade e facilidade de uso. Nguyen (NGUYEN & LE, 2003) também avaliou três implementações MPI: MPICH, LAM e MPI-Madeleine, afirmando que o MPI é o padrão mais popular e que as diferentes implementações possuem desempenhos e funcionalidades semelhantes, embora MPI-Madeleine se mostrou mais eficiente. Outro trabalho (LUSK, 2001) também analisou o uso do padrão MPI, focando a evolução de hardware, software, facilidades de uso e a escalabilidade. Diferentes implementações foram brevemente descritas.

Avresky (AVRESKY & NATCHEV, 2005) trabalhou em um algoritmo de reconfiguração dinâmica de redes para tolerar falhas em múltiplos nodos e links, especialmente em redes de alta velocidade com topologia arbitrária. O objeto de estudo de outro trabalho foi o SDVM (HAASE et al., 2005) um cluster de computadores heterogêneos, que permite operar com qualquer topologia de rede e com capacidade de expansão ou retração do cluster sem interferência no fluxo de programas. Kolen e Hutcheson (KOLEN & HUTCHESON, 2001) apresentaram uma topologia organizacional para clusters de 50 processadores ou mais, de baixo custo, a qual utiliza grupos de quatro placas-mãe de mesmo fabricante e grupos de oito placas-mãe compartilhando a mesma fonte de energia, de forma a minimizar o consumo de energia, economizar fiação e facilitar a manutenção.

3 BALANCEAMENTO DE CARGA.

O balanceamento de carga tenta distribuir a carga de trabalho de acordo com a disponibilidade de processamento e recursos de cada máquina no sistema computacional. Esta distribuição visa maximizar a utilização dos recursos, possibilitando um melhor desempenho do sistema. O balanceamento de carga deve ser aplicado em pontos que podem se tornar um gargalo no sistema, caso contrário não irá contribuir com a melhoria de desempenho, e dependendo da estratégia adotada, pode até prejudicar o desempenho do sistema devido a possíveis sobrecargas (*overheads*) do próprio mecanismo de balanceamento de carga (WILLIAMS,1991).

3.1 Terminologia para Balanceamento de Carga.

As terminologias utilizadas para o balanceamento de carga são variadas e conflitantes. É comum a discussão de características do balanceamento de carga a respeito do gerenciamento ser centralizado ou distribuído, se as decisões do balanceamento são estáticas (independe da modificação de estado do sistema) ou dinâmicas (depende da modificação de estado do sistema). Essa última característica deve também ser classificada como determinística ou probabilística, adaptável ou não adaptável. Todas as características anteriores são importantes para a comparação entre estratégias de balanceamento de carga e para indicar o potencial de cada estratégia.

Casavant e Kuhl publicaram um trabalho na tentativa de unificar as diferentes notações utilizadas para se classificar o balanceamento de carga (CASAVANT & KUHL, 1988). Baumgartner e Wah utilizam uma notação de balanceamento de carga não exclusivamente baseada em Casavant e Kuhl por considerarem que a notação destes mistura características da estratégia de balanceamento de carga com os requisitos do escalonador

(BAUMGARTNER & WAH, 1988). No presente trabalho, serão adotadas notações com base nos trabalhos de Casavant & Kuhl e Baumgartner & Wah.

Segundo Casavant e Kuhl, o primeiro passo é classificar o escalonamento como local ou global. O escalonamento local envolve a atribuição de um processo a uma fatia de tempo de um único processador. Este é o trabalho de um escalonador de processos em um sistema operacional mono-processado (SILBERSCHATZ et al, 2001). Neste trabalho o foco está no escalonamento global, que precisa decidir onde (em qual nodo) o processo será atribuído, e a tarefa de escalonar o processo localmente é deixada para o sistema operacional. Tal abordagem permite que os processadores do sistema multi-processado (cluster) aumentem a sua autonomia enquanto reduzem a responsabilidade (e conseqüente sobrecarga) do mecanismo de escalonamento global (balanceamento de carga). É bom ressaltar que não existe a obrigatoriedade do balanceamento de carga ser centralizado, podendo o mesmo ser distribuído entre os nodos que compõem o sistema multi-processado.

Seguindo a classificação hierárquica de Casavant e Kuhl, o próximo nível classifica o balanceamento como estático ou dinâmico. Mas nesse nível hierárquico, adotou-se a definição de Baumgartner e Wah, que reflete a flexibilidade das regras de escalonamento de reagirem ao estado de carga do sistema. E não a uma designação estática ou dinâmica para a atribuição dos processos, como sugere Casavant e Kuhl. Portanto, o balanceamento estático considera que o estado do sistema não se altera ou se altera muito pouco, enquanto o balanceamento dinâmico, também chamado de dependente de estado, detecta alterações no estado de carga do sistema e gerencia os processos com base no estado atual, podendo realizar uma redistribuição de carga se necessário (BAUMGARTNER & WAH, 1988).

Uma característica que pode colocar o algoritmo de balanceamento em risco é considerar onde será tomada a decisão do escalonamento. Uma estratégia de balanceamento com a tomada de decisão centralizada é mais simples, porém pode ocasionar o surgimento de

um gargalo no nodo responsável pela decisão, limitando a escalabilidade do sistema, além de apresentar um ponto crítico de falha. Agora se a tomada de decisão for distribuída, a sobrecarga para transmitir as informações de status do sistema pode ser tão alta que chegue a reduzir o benefício do balanceamento de carga. Estudo realizado por Zhou, aponta que a sobrecarga da comunicação é importante para as duas abordagens (ZHOU, 1986)[1]. Outra pesquisa indica que informações de status de carga em excesso não são somente desnecessárias, como podem ser prejudiciais (BAUMGARTNER & WAH, 1988).

Outra característica para se classificar uma estratégia de balanceamento de carga pode considerar de onde parte a iniciativa para a distribuição da carga (WANG & MORRIS, 1985)[1]. Wang e Morris propuseram esse critério em que o nodo carregado procura por outro nodo que possa receber parte da sua carga, ou seja, a distribuição da carga foi iniciada pela origem da sobrecarga. Outra forma é um nodo subutilizado estar frequentemente procurando tirar carga dos nodos que estejam mais sobrecarregados que ele próprio, estratégia conhecida como iniciada pelo destino. É possível trabalhar com as duas abordagens simultaneamente. Porém, estudos revelam que a estratégia de balanceamento de carga iniciada pelo destino apresenta maior potencial para melhorar o desempenho do que a estratégia iniciada pela origem (BAUMGARTNER & WAH, 1988).

O algoritmo de balanceamento de carga pode ser classificado também como adaptativo ou não. Essa característica considera que o algoritmo leva em consideração muitos parâmetros para a sua tomada de decisão. Em resposta às mudanças no sistema, o escalonador pode passar a ignorar ou reduzir a importância de algum parâmetro, se o mesmo considerar que as informações repassadas por tal parâmetro são inconsistentes com o restante dos parâmetros ou não está fornecendo informações a respeito das mudanças do sistema. Em

[1] Conforme consta em (BAUMGARTNER & WAH, 1988).

contra-partida, um algoritmo não adaptativo é aquele que não modifica a sua base de controle de acordo com um histórico de atividade do sistema (CASAVANT & KUHL, 1988).

Lling classifica os seus algoritmos de balanceamento de carga de acordo com duas características que podem ser agregadas às características já apresentadas. A primeira leva em consideração de onde obter informações para a tomada de decisão, sobre manter ou migrar um processo. A decisão pode ser baseada em informações locais ou globais. Uma decisão baseada em informações locais considera apenas o status de carga do próprio nodo ou no máximo dos nodos imediatamente adjacentes. Uma decisão baseada em informações globais considera o status de carga de um subconjunto ou de todos os nodos do sistema. A segunda se preocupa com a migração da carga (processo) para um outro processador com o intuito de reduzir o desbalanceamento de carga. Se a migração ocorre apenas para os vizinhos diretos, então o espaço de migração é local. Caso contrário o espaço de migração é considerado global (LLING et al, 1991).

Em resumo, a abordagem de balanceamento de carga implementada no presente trabalho pode ser classificada como “escalonamento global sub-ótimo adaptativo” (CASAVANT & KUHL, 1988). Global porque trabalha com o escalonamento de processos de todo o cluster. Sub-ótimo porque pode não trazer a melhor solução para o problema de escalonamento, por se tratar de um problema NP-completo (FEITELSON, 1997), para o qual, nenhuma estratégia de balanceamento de carga é ótima em todos os casos (BLAZEWICZ et al., 2002, KACER & TVRDÍK, 2002). E adaptativo porque dependendo do nível de carga do cluster alguns parâmetros recebem mais atenção para a decisão da distribuição de carga.

3.2 Trabalhos Relacionados ao Balanceamento de Carga.

Como o balanceamento de carga é o esforço de manter todos os processadores do cluster realizando algum trabalho produtivo (GEORGE, 1999), é compreensível a grande quantidade de trabalhos realizados a este respeito. Dentre muitos, Decker apresentou a

ferramenta VDS, que distribui automaticamente os pacotes gerados por aplicações paralelas e balanceia a carga entre os nodos de uma rede (DECKER, 1997). Em outro trabalho, com o objetivo de reduzir o tempo de execução do próprio algoritmo de distribuição de carga, Attiya propôs um algoritmo ótimo, executado em duas fases, sendo cada fase baseada em uma heurística diferente: *Simulated Annealing* e *Branch-and-Bound* (ATTIYA & HAMAM, 2004).

Choi propôs uma métrica de carga baseada no número de tarefas que efetivamente afetam o desempenho do sistema. Os resultados de simulações mostraram um ganho de 11% no tempo de execução (CHOI et al., 2003). Outra técnica de balanceamento de carga foi proposta por Chau, baseada em arquiteturas hipercubo, com o objetivo de reduzir o tráfego de mensagens (CHAU & FU, 2003). Já Ibrahim, avaliou o desempenho de sistemas paralelos que usam o balanceamento de carga dinâmico em termos de tempo de execução, velocidade e eficiência na execução de uma versão paralela do algoritmo de busca em profundidade (*depth-first*), sobre a plataforma MPI (IBRAHIM & XINDA, 2002).

Bohn pesquisou técnicas para realizar o balanceamento assimétrico em um cluster heterogêneo e mostrou que quando a diferença do poder de processamento entre os nodos do cluster é pequena, os benefícios alcançados com o balanceamento assimétrico são pequenos, caso contrário, é possível obter um excelente ganho de desempenho evitando os processadores mais lentos (BOHN & LAMONT, 1999). Ainda trabalhando com um cluster formado por nodos heterogêneos, Wong utilizou o *benchmark* paralelo NAS em um cluster com nodos interconectados por redes de comunicação Gigabit Ethernet e concluiu que os processos MPI podem ser escalonados mais facilmente que *threads* OpenMP (WONG & JIN & BECKER, 2004).

Com clusters heterogêneos também trabalhou Bevilacqua, que propôs um método eficiente de balanceamento de carga em cluster de *workstations*. Basicamente um

gerente é responsável por enviar carga para o nodo ocioso toda vez que este solicitar. Quando o gerente não estiver atendendo aos pedidos, este deve trabalhar nos dados que ele armazena. Os resultados experimentais alcançaram uma eficiência superior a 90% (BEVILACQUA, 1999). Mudando de foco, Aversa sugere que os clusters de servidores Web podem tirar grande proveito do balanceamento de carga (AVERSA & BESTAVROS, 2000). Em seu trabalho, Aversa propôs e avaliou a implementação de um protótipo de servidor Web distribuído, obtendo resultados positivos quanto a escalabilidade e inclusive custo, quando da aplicação de sua técnica em sistemas de pequeno porte.

Já o principal objetivo de Kacer foi verificar a adequabilidade do balanceamento de carga para processos curtos com uso intenso do processador, por exemplo, compiladores e utilitários de compressão, entre outros. Como resultado do estudo, uma técnica de execução remota de processos foi proposta e comparada com a estratégia de balanceamento adotada no Mosix. Com os testes realizados, os resultados apontaram que a técnica proposta é superior à migração de processos utilizada no Mosix em muitos casos, e apresenta resultados semelhantes nos demais (KACER & TVRDÍK, 2002). Também no trabalho de Shen, a preocupação estava em políticas de balanceamento de carga para serviços de rede de granularidade fina (processos curtos). O trabalho concluiu que a política de *polling* randômico é adequada para esses casos (SHEN & YANG & CHU, 2002).

Ainda trabalhando com processos curtos, George focou na distribuição de carga baseada no tempo de resposta dos nodos quando da submissão de programas pequenos e rápidos (GEORGE, 1999), mas o método de decisão adotado é bastante específico para problemas que representam seus dados em estruturas de dados agregadas (vetores).

Savvas propôs um algoritmo chamado PSTS, cuja idéia principal é dividir recursivamente o cluster em subespaços e encontrar a dimensão que provenha o melhor desempenho. A proposta se mostrou eficiente para sistemas que permanecem desbalanceados

por longo tempo (SAVVAS & KECHADI, 2004). Também o modelo de carga divisível proposto por DrozDowski, pode ser utilizado quando as aplicações permitem dividir a computação em partes de tamanho arbitrário. A principal contribuição do seu trabalho foi a modelagem matemática do problema de carga divisível utilizando hierarquias de memória (DROZDOWSKI & WOLNIEWICZ, 2003).

Legrand trabalhou com algoritmos iterativos, os quais operam sobre uma grande matriz retangular de dados. Essa matriz é divida em segmentos verticais que são alocados a recursos computacionais (processadores). A cada passo do algoritmo, os segmentos são atualizados por cada nodo e sua informação de fronteira é trocada entre os vizinhos. Legrand projetou uma heurística que provê o mapeamento do anel de comunicação e um esquema de distribuição de dados eficiente (LEGRAND et al., 2004).

4 PLATAFORMA LAM/MPI.

A programação paralela utilizando a biblioteca MPI pode ser vista como uma coleção de processos que executam aplicações, escritas em uma linguagem seqüencial comum, e realizam chamadas a rotinas da biblioteca para enviar e receber mensagens (FOSTER, 1995). Na maioria das implementações com MPI, um conjunto fixo de processos é criado no início da execução da aplicação. Mas pode acontecer de cada processo executar uma aplicação diferente. Quando isso acontece, o paradigma MPI pode ser referenciado como um modelo MPMD (*multiple program multiple data*) para diferenciar do SPMD (*single program multiple data*) em que cada processo executa a mesma aplicação. Para este trabalho o modelo adotado é o SPMD.

Neste trabalho, o MPI foi escolhido inicialmente, para fornecer suporte para o envio e recebimento de mensagens. A justificativa para esta escolha foi devido ao MPI ser o padrão “de facto” para troca de mensagens (*message-passing*) (FOSTER, 1995), além de outras vantagens que são apresentadas neste texto.

Quando se trata de desenvolvimento de aplicações, um fator importante é a facilidade para se depurar o código. O paradigma de troca de mensagens é por si só considerado muito mais fácil de depurar do que outros tipos de programação paralela (Fox, 1994). Oferecer ao usuário a possibilidade de executar a aplicação passo-a-passo, é de grande importância para se identificar o ponto onde as coisas não estão funcionando.

No paradigma de troca de mensagens, falhas na troca de mensagens normalmente estão relacionadas com falhas humanas. Além disso, os problemas são normalmente repetíveis, visto que na maioria das vezes estão relacionados com erros lógicos no código. O que nem sempre acontece com os paradigmas de multi-processamento (*multiprocessing*) e memória compartilhada (*shared-memory*), já que as falhas tendem a depender do momento relativo de vários eventos no código. Como resultado, a depuração

pode fazer com que o problema apareça em diferentes lugares ou até mesmo, não aconteça. Tal comportamento pode ser melhor explicado pelas condições de corrida (*race conditions*) em (SILBERSCHATZ et al, 2001).

Toda aplicação paralela realiza comunicação entre os processos, nem que seja apenas o envio dos dados iniciais e o recebimento dos resultados. Já foi comentado que o MPI é o responsável por realizar a comunicação entre os processos, mas o que a biblioteca permite? Os processos podem realizar comunicação ponto-a-ponto, enviando uma mensagem de um processo para o outro através de identificadores (*rank*). Esta opção pode ser utilizada para comunicação local ou não estruturada.

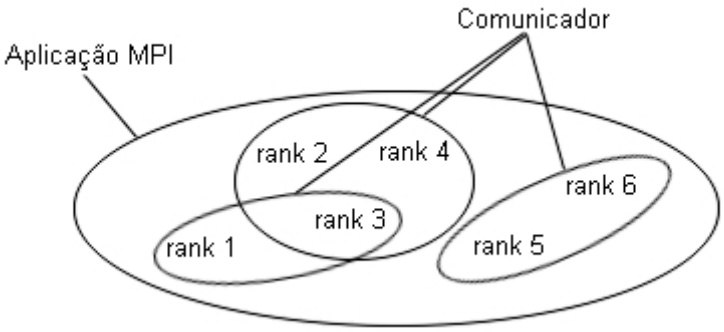

FIGURA 1: Criação de grupos comunicadores dentro de uma aplicação MPI.

O MPI permite também a criação de grupos de processos, que podem realizar operações de comunicação coletiva como o *broadcast*. A criação destes grupos faz com que o MPI ofereça uma característica muito importante para o desenvolvimento de software, que é a programação modular. Um mecanismo chamado comunicador (*communicator*) permite ao programador definir módulos que encapsulam estruturas de comunicação interna. Cada comunicador criado engloba certo número de processos da aplicação MPI, permitindo que os dados sejam tratados sobre diferentes contextos na mesma aplicação. Isso possibilita que mensagens de difusão (*broadcast*) sejam recebidas apenas pelos processos que pertencem ao mesmo comunicador, como representado na FIGURA 1. Esta característica do MPI permite

um aumento de produtividade por parte do programador e eficiência na execução dos algoritmos (FOX, 1994).

O MPI trabalha com seus próprios tipos, como MPI_CHAR, MPI_SHORT, MPI_INT, MPI_FLOAT e outros. Mas nem sempre os dados que o programador pretende utilizar estão definidos pelo MPI. Outro caso que permite ganho de produtividade é quando se deseja enviar uma seqüência de dados e seria interessante enviar todos os dados em uma única mensagem. O MPI provê mecanismos para a criação de tipos derivados, o que previne operações de cópia de dados e reduz o número de mensagens transmitidas. Por exemplo, enviar uma linha de uma matriz bidimensional que está armazenada em colunas exige que estes elementos, não contíguos, sejam enviados um em cada mensagem diferente, o que gera desperdício com os cabeçalhos que serão utilizados em cada mensagem. Mas esse desperdício pode ser evitado, se o tipo de dado adequado for utilizado, transmitindo a linha inteira em uma única mensagem.

E este é só um resumo das características que justificam a escolha da biblioteca MPI, muitas outras características também podem ser apresentadas, como:

- Tipos heterogêneos: permite que diferentes tipos de dados sejam encapsulados em um único tipo derivado, permitindo assim, a comunicação de mensagens heterogêneas.
- Aquisição de informações do ambiente: Uma aplicação pode obter informações a respeito do ambiente no qual ele está executando (nome da máquina, tempo decorrido desde uma data do passado, etc.), inclusive informações que podem ser usadas para melhorar o desempenho do algoritmo.
- Operações globais adicionais: Estas operações suportam comunicação de todos para todos e contribuições de tamanho variável vindo de diferentes

processos. Operações de redução podem ser utilizadas para localizar valores mínimos e máximos e executar reduções com funções definidas pelo usuário.

- Modos de comunicação especializados: Normalmente a comunicação no MPI é assíncrona, mas modo síncrono também é suportado. Neste modo o emissor fica bloqueado até que o dado correspondente seja recebido. Permite também comunicação com uso de *buffer*, o qual possibilita que o programador defina uma capacidade para o mesmo, buscando garantir que os recursos do sistema não se esgotem durante a comunicação (*overflow*).

A implementação LAM/MPI (*Local Area Multicomputer MPI*) (NGUYEN & LE, 2003 e LUSK, 2001), adequada para ambientes heterogêneos, é um software livre e de código aberto. Além disso, ela é usada mundialmente, possui uma ativa comunidade de usuários e o seu grupo de desenvolvedores está disponível para contato, auxiliando na solução de dúvidas. Sua característica de projeto modular permite organizar as pesquisas, levando o software a atender uma grande variedade de plataformas de multicomputadores, computadores multiprocessados e ambientes distribuídos (ONG, 2000). Um programa na LAM/MPI pode obter informações específicas do ambiente de execução, que podem ajudar no desenvolvimento de algoritmos de balanceamento de carga.

A LAM/MPI utiliza um pequeno *daemon* de nível de usuário para o controle de processos, redirecionamento de mensagens e comunicação entre processos. Este *daemon* (*lamd*) é carregado no início de uma sessão, através do aplicativo *lamboot* que pode utilizar diversas tecnologias de execução de comandos remotos como *rsh/ssh*, TM (OpenPBS / PBS Pro), SLURM ou BProc, para realizar a operação de carga. O *daemon* também auxilia na execução da aplicação, que é feita com o comando *mpirun*. Por ser a base do ambiente de

execução da LAM/MPI, é indispensável executar o *daemon lamd* em cada nó do cluster (MARTINS, 2005).

4.1 Lógica de Funcionamento do *mpirun*.

Nesta seção é apresentada e discutida a seqüência de passos seguidos pelo aplicativo *mpirun*, parte integrante da biblioteca LAM/MPI. Entretanto, os passos aqui apresentados não representam na totalidade os passos existentes no *mpirun*. O código fonte do aplicativo é formado por diversos arquivos (módulos), e a conexão entre estes módulos é bastante complexa. Portanto, foi adotada uma simplificação, e os passos aqui apresentados foram escolhidos, após estudo do código fonte, devido a sua representatividade para a pesquisa. Cada passo individualmente não representa, necessariamente, um procedimento ou função dentro do código fonte. Um passo pode representar uma estrutura de controle ou um conjunto de módulos que resulta no processamento do passo.

Esta seção está dividida em duas subseções, a primeira apresenta a estrutura lógica padrão do aplicativo *mpirun*, que é parte integrante da biblioteca LAM/MPI, e de agora em diante será conhecido como *mpirun* ORIG, para indicar que corresponde ao *mpirun* original. Na segunda seção, são apresentadas as modificações realizadas no *mpirun* ORIG para a realização do balanceamento de carga. Esta versão modificada do *mpirun*, a partir deste ponto, será tratada como *mpirun* BALX, indicando a versão balanceada do *mpirun*. O X, da sigla BALX, é um número natural e representa a versão da extensão que está sendo mencionada naquele momento.

4.1.1 Estrutura Lógica do *mpirun* ORIG.

A FIGURA 2 mostra o fluxograma simplificado do *mpirun*. O primeiro passo do *mpirun* é verificar se o usuário que está executando-o possui privilégios de super-usuário. A

segurança do sistema computacional é o principal motivo para esta verificação. Um aplicativo executado por um super-usuário possui permissões que podem colocar em risco o sistema computacional se a aplicação executada pelo *mpirun* possuir código malicioso. Por este motivo, a execução da aplicação por um super-usuário é abortada e uma mensagem de erro é exibida.

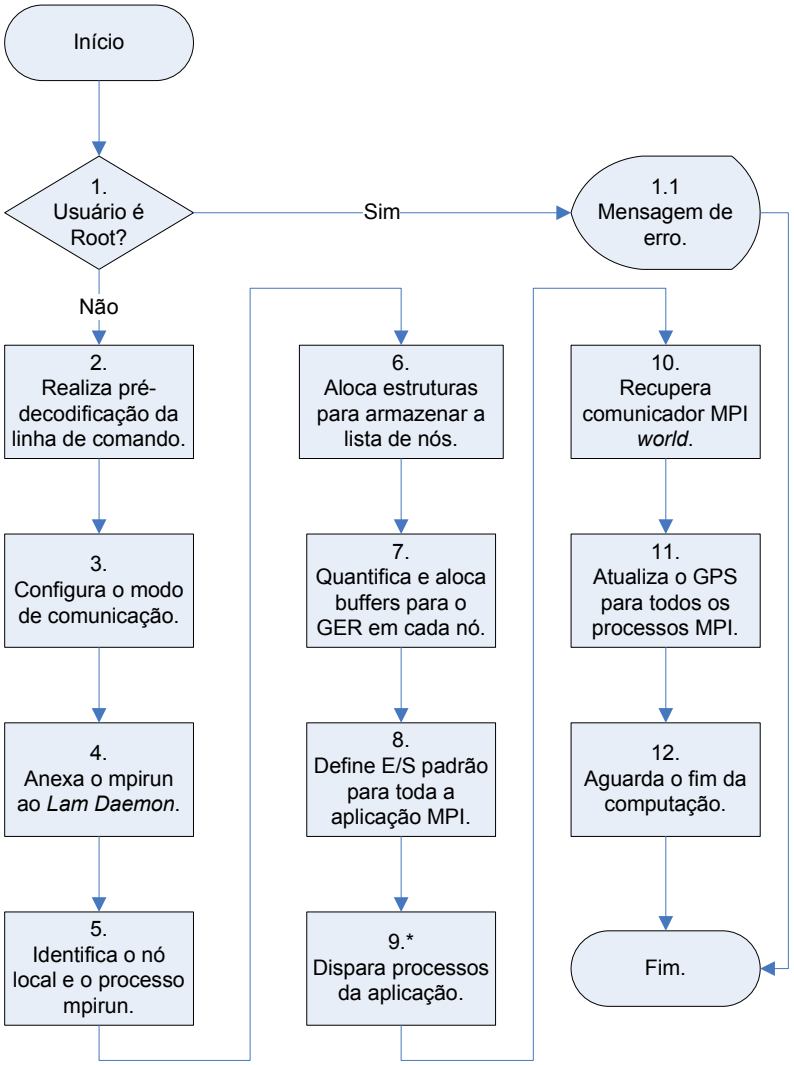

FIGURA 2: Fluxograma simplificado do MPIRUN (ORIG).

Confirmando-se o fato de que o usuário que solicitou a execução do *mpirun* não possui privilégios de super-usuário, o algoritmo realiza uma pré-decodificação da linha de comando no passo 2. Ainda no mesmo passo, uma análise da linha de comando é realizada para verificar a sua sintaxe. A biblioteca LAM/MPI dispõe de uma série de funções que

auxiliam na realização dessa análise. Mais adiante esse assunto é retomado, já que foram criados novos parâmetros na versão do *mpirun* BAL.

No passo 6, as informações com respeito ao número de processos e os nodos que irão executá-los são recuperadas da linha de comando, e uma estrutura de dados agregada (vetor) é criada com uma posição para cada processo solicitado pelo usuário. Essa estrutura passa a ser conhecida como GPS, pois cada posição do vetor é uma estrutura de dados composta (registro) contendo informações necessárias para localizar, no cluster, cada processo pertencente à aplicação. Os campos do registro armazenam informações referentes ao identificador do nodo, identificadores do processo (*pid* e outros) e o *rank* do processo executando naquele nodo.

No caso de existirem mais processos do que o número de nodos, o campo com o identificador do nodo terá o mesmo valor, mas os campos com identificadores dos processos e o *rank* serão distintos. Lembre-se de que o GPS possui uma entrada para cada processo.

Determinado o número de processos, o *mpirun* pode preparar algumas condições para a comunicação entre os mesmos. Esse dispõe de uma propriedade que procura evitar perdas de pacotes provocadas por *overflow* durante a comunicação entre um par de processos. Essa propriedade é conhecida por GER (*Guaranteed Envelope Resources*), que cria e gerencia *buffers* que serão utilizados para a comunicação entre os pares (LAM/MPI, 2005). O GER também é responsável por notificar o processo causador do *overflow* sobre o esgotamento dos recursos. Os *buffers* em cada nodo, para o GER, são alocados no passo 7 da FIGURA 2, podendo o GER ser desabilitado a critério do usuário. Por padrão, a biblioteca LAM/MPI utiliza o GER com uma configuração mínima (BURNS & DAOUD, 1995).

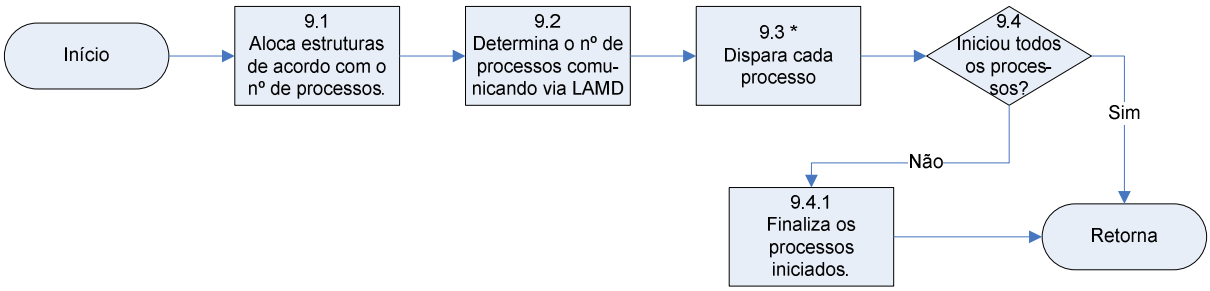

FIGURA 3: Detalhamento do passo 9 pertencente ao fluxograma do MPIRUN (ORIG).

Definidos os meios de comunicação entre os processos e as interfaces de E/S, é possível iniciar os processos (Passo 9). Devido a importância do passo 9 para a realização do balanceamento de carga, este foi expandido para uma discussão mais detalhada na FIGURA 3. No passo 9.1 são criadas estruturas, de acordo com o número de processos, que irão armazenar informações, que depois serão repassadas para o devido preenchimento do GPS. O passo 9.3 trata de uma estrutura de repetição que dispara cada processo em seu respectivo nodo. Esse passo é abordado em mais detalhes a seguir.

Ao término da estrutura de repetição do passo 9.3, uma verificação é realizada para averiguar se todos os processos solicitados pelo usuário estão no estado de pronto. Caso algum processo, por qualquer motivo, não consiga ser alocado e iniciado, todos os demais processos serão finalizados e um erro será reportado ao usuário.

O passo 9.3 é onde tudo efetivamente acontece, e para uma melhor explanação este também foi expandido, FIGURA 4. Neste ponto já existe uma estrutura contendo informações sobre os nodos do cluster e uma estrutura a ser preenchida com informações sobre os processos (GPS). Basta agora começar a atribuir os processos aos nodos. A estrutura contendo as informações dos nodos é uma lista circular, que é percorrida do início para o fim, retornando ao início caso o número de processos seja maior que o número de nodos.

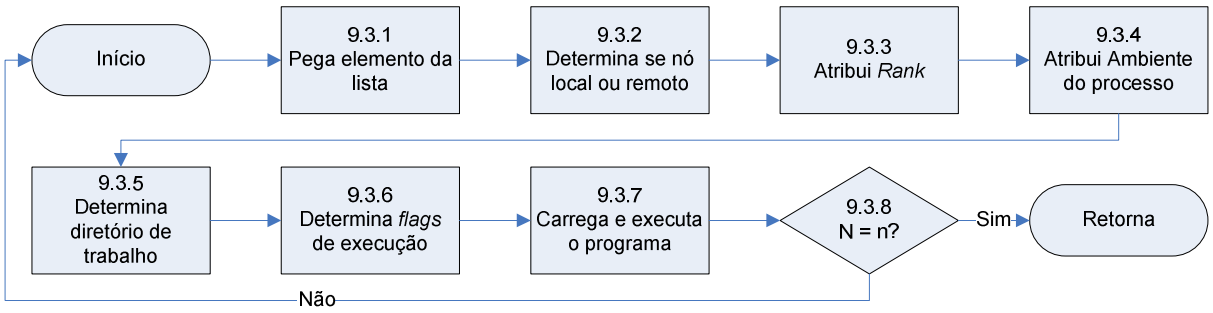

FIGURA 4: Detalhamento do passo 9.3 pertencente ao fluxograma do passo 9 (ORIG).

Determinada a posição atual da lista, informações sobre o respectivo nodo são recuperadas. Verifica-se se o nodo recuperado é local ou remoto à instância do *mpirun*. Tal verificação faz-se necessária devido a transferência do executável e para atribuir as interfaces de E/S. Em seguida, um valor seqüencial iniciado em zero é atribuído ao processo (*rank*) que será disparado logo em seguida. A condição de parada da repetição ocorre quando o número de processos iniciados (n) for igual ao número de processos solicitados pelo usuário (N).

Uma vez que N = n, o fluxo do programa retorna para o passo 9.4 na FIGURA 3, onde cada processo disparado é verificado para determinar se está no estado de pronto. Confirmada tal situação, o fluxo do programa retorna para o passo 10 na FIGURA 2, onde todos os processos são conhecidos, permitindo preencher os campos do GPS para cada um dos processos.

Com o GPS completamente preenchido, o mesmo precisa ser repassado para todos os processos. Essa tarefa é executada pelo passo 11, garantindo que cada processo seja capaz de localizar qualquer outro processo da mesma aplicação. Como passo final, o passo 12 aguarda uma mensagem de concluído (*MPI_Finalize*) de cada um dos processos iniciados. Quando todas as mensagens forem recebidas, o *mpirun* encerra a sua execução.

4.1.2 Modificações que Caracterizam o *mpirun* BAL.

Para que o *mpirun* realize o balanceamento da carga, um passo adicional foi criado (FIGURA 5) e outro modificado (FIGURA 6), como é apresentado nos fluxogramas a

seguir. Antes de apresentar as modificações cruciais do *mpirun*, é bom ressaltar que um parâmetro deve ser informado para que este realize o balanceamento de carga. Para a execução do *mpirun* BAL, a seguinte linha de comando deve ser utilizada:

```
$ mpirun -cpu C teste
```

Com o parâmetro *–cpu* informado na linha comando, o novo passo inserido entre os passos 2 e 3 da FIGURA 5 é executado e oferece condições para que ocorra o balanceamento de carga. A definição deste novo parâmetro pode ser visualizada no ANEXO II.

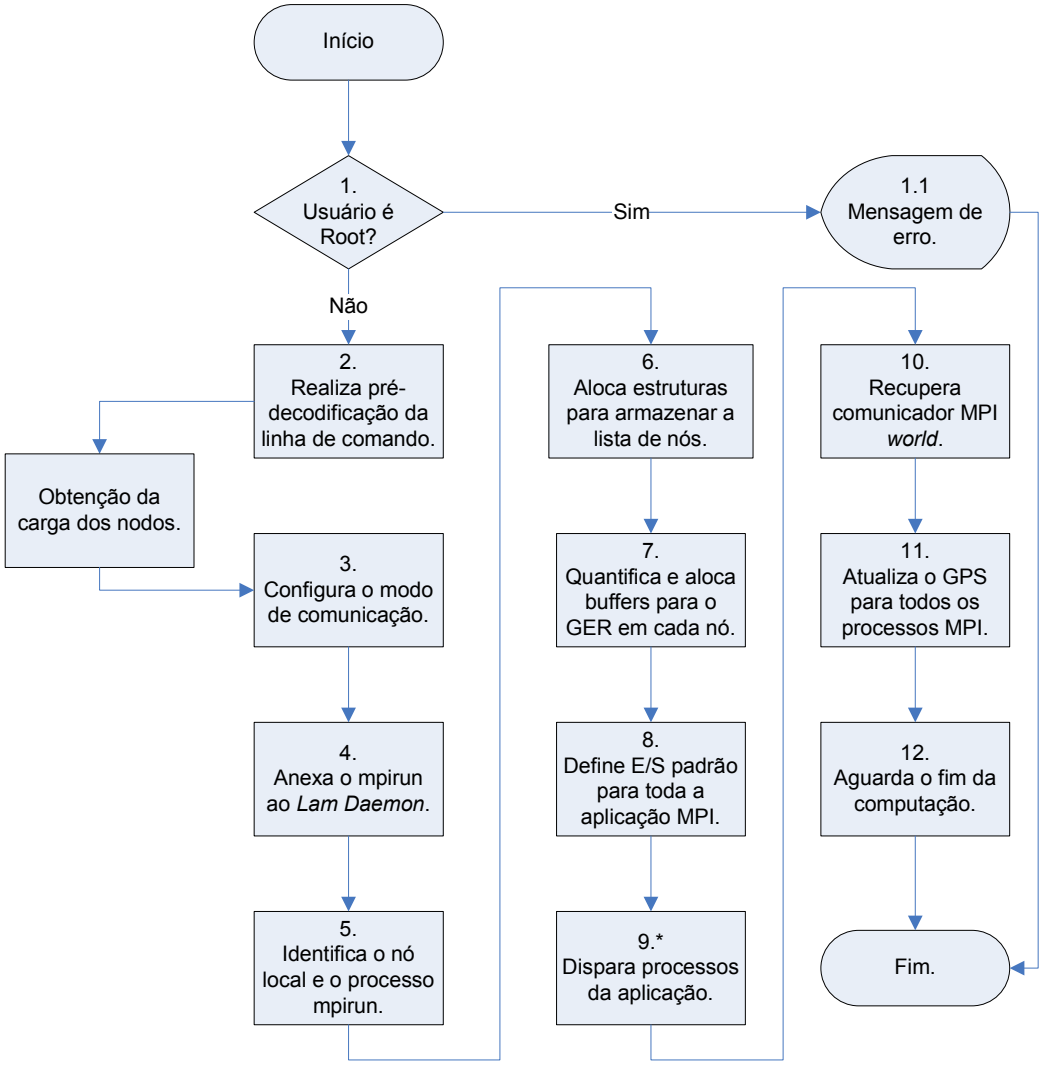

FIGURA 5: Fluxograma simplificado do MPIRUN (BAL).

Este passo tem por finalidade obter informações referentes a carga e ao poder computacional de cada nodo que compõe o cluster. Para atingir essa meta, foi criado um

módulo chamado *MPI_Libt*, que quando disparado pelo *mpirun*, faz uso da própria estrutura disponibilizada pela biblioteca LAM-MPI para obter as informações requisitadas. Ou seja, o módulo *MPI_Libt* pode ser classificado como uma aplicação MPI, visto que ele dispara uma segunda instância do *mpirun*, utilizando todo o cluster e solicitando que cada nodo informe a sua carga atual e o seu poder computacional. Mais detalhes sobre este módulo podem ser consultados no ANEXO I.

Ao término da execução, o módulo *MPI_Libt* disponibiliza para os demais passos do *mpirun* as informações requisitas já ordenadas. As informações estão organizadas em um vetor de registros com dupla ordenação. O vetor está ordenado em ordem decrescente de poder computacional dos nodos, e os nodos de mesmo poder computacional estão ordenados em forma crescente de carga de CPU, como ilustra a Tabela 1.

Tabela 1: Vetor duplamente ordenado, disponibilizado pelo módulo *MPI_Libt*.

	0	**1**	**2**	**3**
Poder Computacional	5000	5000	3000	2000
Carga	0.0	1.4	0.1	0.3

A partir do passo 3, a execução continua o fluxo normal, já descrito pelo *mpirun* ORIG até o passo 9, onde o passo 9.3.1, na FIGURA 6, foi modificado para que os nodos tivessem novos processos alocados de acordo com regras. As regras, para um nodo receber um novo processo, são discutidas no capitulo 5. Os detalhes desta modificação podem ser consultados no ANEXO III.

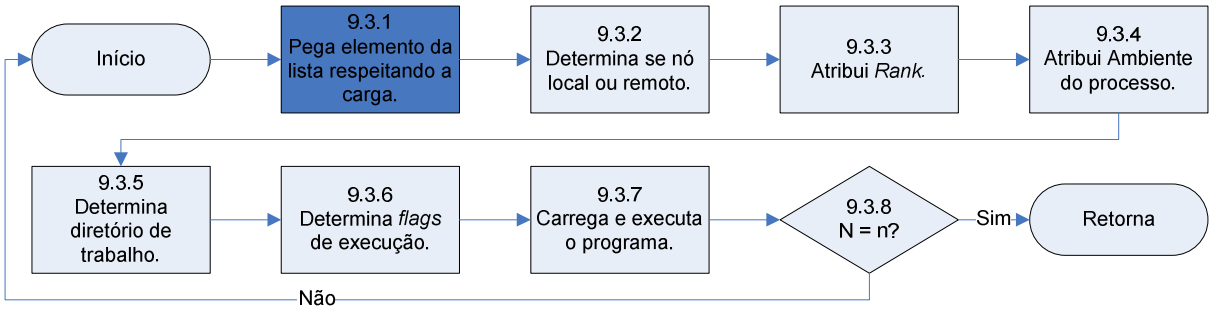

FIGURA 6: Detalhamento do passo 9.3 com o passo 9.3.1 modificado para o *mpirun* BAL.

É possível perceber que as modificações realizadas no *mpirun*, para que o mesmo realize o balanceamento de carga, geram sobrecarga toda vez que uma nova aplicação MPI é disparada. Mas os experimentos mostram que os benefícios são maiores que o prejuízo. Principalmente quando o *mpirun* BAL é aplicado a um cluster heterogêneo, em que o poder computacional dos nodos é melhor aproveitado, evitando que recursos fiquem ociosos.

5 BALANCEAMENTO DE CARGA NA LAM/MPI

Neste capítulo são apresentados os experimentos que surgiram durante o desenvolvimento deste trabalho, resultando na avaliação de três versões para o balanceamento de carga na LAM/MPI. Cada versão foi testada, avaliada e com base no comportamento e nos resultados, uma nova versão foi proposta com melhorias sobre a anterior, resultando em uma seqüência progressiva no desenvolvimento de soluções cada vez mais eficientes. No final, a última versão é apresentada como a melhor opção.

Todos os experimentos deste trabalho foram realizados no cluster heterogêneo HPPCA/DIN/UEM, o qual é formado por oito (8) nodos, sendo quatro equipados com processador Pentium IV HT de 3.0 GHz e mais quatro nodos equipados com processador Pentium IV de 1.8 GHz, todos os nodos com 512 MB de memória. Esses nodos estão conectados por um *switch* 3COM padrão *fast ethernet* e são utilizados como escravos. Um processador ATHLON de 1.0 GHz é usado como mestre. Os códigos foram escritos em linguagem C ANSI e compilados em gcc 3.2.2. O ambiente utilizado foi o *Linux Red Hat* com *kernel* 2.4.29.

A versão da biblioteca de troca de mensagens utilizada foi a LAM/MPI 7.1.1 e a extensão é baseada nesta versão. Durante os experimentos, três aplicações foram utilizadas na avaliação de desempenho: multiplicação matricial, cálculo integral e reconhecimento de DNA (seqüências genéticas), sob diferentes configurações quanto ao tamanho da entrada de dados. Para forçar o desbalanceamento da carga no cluster, cargas artificiais foram disparadas de forma controlada e de acordo com o propósito desejado, variando em número e local de ativação. As cargas artificiais são formadas por aplicações de cálculo de integral com um número muito grande de trapézios, o que resulta em um elevado tempo de execução. As cargas artificiais são iniciadas antes do início das simulações e terminam após a execução destas.

5.1 Investigações Preliminares.

Antes que os experimentos para avaliar o balanceamento de carga na LAM/MPI sejam iniciados, alguns experimentos preliminares foram realizados. Primeiramente, foi realizada uma caracterização dos nodos componentes do cluster para se determinar o impacto do crescente aumento no número de cargas artificiais. Posteriormente, foi investigado como obter e como usar diferentes índices de cargas. As próximas seções detalham estes experimentos.

5.1.1 Caracterizando os Nodos do Cluster.

O desempenho dos nodos do cluster HPPCA foi testado com a imposição de N cargas artificiais por nodo, compostas de aplicações que realizam o cálculo de integral, com $0 < N \leq 8$. Os resultados foram divididos em dois grupos, de acordo com o poder computacional, o grupo das estações rápidas (3.0 GHz) e o grupo das estações lentas (1.8 GHz). O experimento buscou descobrir a diferença nos tempos de execução entre os grupos, bem como avaliar de que forma do crescimento da carga afeta o desempenho dos nodos.

Na FIGURA 7 está representada as linhas de crescimento do tempo de execução, que consiste na média do tempo de execução das N aplicações em cada ponto, para os dois grupos. Como é esperado, o grupo das estações lentas apresenta um tempo de execução superior. No ponto 8 a diferença com as estações rápidas já alcança 25%. Note a linearidade no crescimento do tempo de execução, em que a diferença do tempo de execução entre cada ponto simulado é aproximadamente o tempo de execução da carga no ponto 1. Esse comportamento permite afirmar que quanto maior é a carga nas estações lentas, menos elas devem ser consideradas pela distribuição de carga. Os pontos 9 e 10 são previsões, considerando a linearidade do crescimento do tempo de execução.

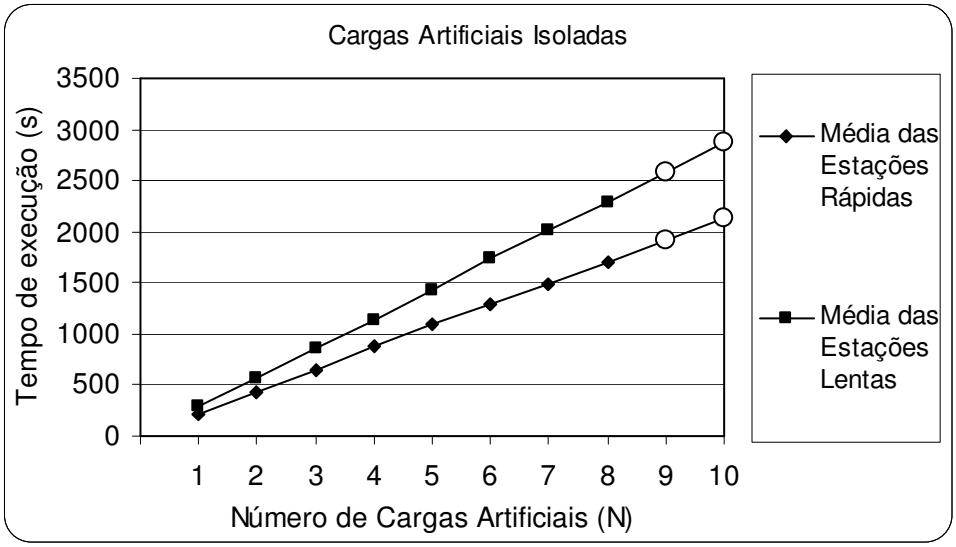

FIGURA 7: Avaliação da diferença de desempenho dos nodos do cluster.

Comumente, uma aplicação na LAM/MPI é composta por diversos processos, os quais são distribuídos a computadores chamados de escravos e os resultados são centralizados em um computador chamado de mestre. Neste sistema, o balanceamento de carga visa uma distribuição de processos de forma mais coerente com a disponibilidade de recursos de cada nodo do cluster. Sem o balanceamento de carga, o tempo que o usuário deve aguardar para finalizar a sua aplicação será diretamente proporcional ao tempo dos nodos mais lentos.

Diversos módulos da biblioteca LAM/MPI foram modificados e outros inseridos, de forma a permitir o balanceamento de carga (durante a distribuição de processos). Mais precisamente, o algoritmo de balanceamento foi inserido dentro do aplicativo *mpirun* (utilizado pelo usuário para iniciar a execução paralela), o qual é o responsável pela alocação de processos aos nodos do cluster. A implementação original da biblioteca LAM/MPI aloca os processos usando o escalonamento *round-robin*, fazendo a distribuição de carga baseada em uma estrutura de lista circular que contém informações sobre os nodos onde serão encaminhados os processos da aplicação para a respectiva execução. Entretanto, nenhum tipo de ordenação especial é imposto a esta estrutura.

5.1.2 Obtendo o Índice de Carga de CPU.

Para a obtenção de informações sobre a carga dos nodos do cluster foi utilizada a biblioteca LibGTop. Essa biblioteca é destinada a obter informações do sistema operacional, como uso de CPU, memória e informações acerca dos processos em execução. Em sistemas operacionais como Solaris ou o Digital Unix, em que é preciso ter privilégio de super-usuário para obter estas informações, a biblioteca LibGTop provê solução através de chamadas *setuid* e/ou *setgid*.

Mesmo sabendo que a LibGTop é parte do ambiente GNOME, sua interface de acesso é completamente independente de qualquer ambiente, o que permite que seja utilizada como uma biblioteca em qualquer software GPL (OPEN SOURCE, 2005). É grande o leque das plataformas suportadas, praticamente todos os sistemas baseados em Unix suportam a biblioteca LibGTop, inclusive o Linux que é foco deste trabalho.

A biblioteca oferece uma ampla variedade de funções, e cada função normalmente retorna mais de um tipo de informação sobre o sistema. A biblioteca não é responsável por gerar as informações que ela fornece, servindo apenas como meio para recuperar as informações mantidas pelo próprio sistema operacional, sendo este, o responsável pela precisão e atualização. Neste trabalho, iniciamos nossos experimentos usando o número de processos como índice de carga, mas logo este foi abandonado por informações relativas à média da carga de cpu em um intervalo de tempo. As razões serão expostas na seção 5.1.3.

A função que retorna as informações sobre o número de processos é a *glibtop_get_proclist (glibtop proclist * buf, int64_t which, int64_t arg)*. O *buf* é uma estrutura que armazena os dados retornados pela função. No argumento constante *which* são especificados os processos sobre os quais a função deve retornar informações e o argumento *arg* é utilizado em conjunto com a escolha do *which*, podendo ou não ser utilizado. Por

exemplo, para obter informações sobre todos os processos, o *arg* é ignorado, mas para verificar se o processo com o PID Y ainda existe, o valor de Y deve ser informado ao *arg*.

A estrutura *glibtop proclist* consiste de três campos, *number* que informa o número de entradas na lista de processos, *size* com o tamanho de cada entrada da lista e *total* que é o valor calculado de *number * size*. Os campos *size* e *total* podem ser usados para alocar memória no caso de se recuperar a lista com todos os processos.

A função que retorna a média da carga de tarefas é a *glibtop_get_loadavg (glibtop loadavg * buf)*. Esta função retorna as informações em uma estrutura de dados composta (uma *struct* em C, por exemplo). Um dos campos desta estrutura é o vetor de três posições *loadavg*. Cada uma das posições armazena a média de carga para os intervalos de um, cinco e quinze últimos minutos. Os demais campos da estrutura devem ser considerados obsoletos e serão removidos[2] nas futuras versões. O uso desta função pode ser consultado no ANEXO I.

5.1.3 Análise do Impacto do Número de Processos.

Verificar a influência que o número de processos como índice de carga do cluster tornou-se um objeto de investigação preliminar neste trabalho. A intenção é verificar se custo da preempção de todos os processos pode impactar na execução de um único processo. É aceitável que os resultados sejam representativos apenas quando realizados em um ambiente onde o número de processos prontos para execução supere o número de CPUs disponíveis (SILBERSCHATZ et al, 2001), pois o processador fica multiplexado entre os processos que o requisitam.

Para isso, testes consistindo de uma bateria de simulações em que uma aplicação "A" é disparada em conjunto com *x* instâncias de um mesmo tipo de programa foram

---

[2] Segundo especificações da biblioteca LibGTop versão estável 1.0.12

realizados. A aplicação "A" executa o cálculo de uma integral. Em cada simulação, as *x* instâncias adicionais executam um mesmo tipo de programa (1, 2, 3 ou 4). Os valores escolhidos para *x* variam entre 10, 20 e 300. Os tipos de programa estão resumidamente apresentados como segue:

- Tipo 1: Processo consiste em um laço de repetição que executa um incremento de uma variável e entra em espera por um tempo superior ao tempo de execução da aplicação "A";
- Tipo 2: Processo solicita entrada de dados para o usuário (*prompt*) e fica neste estado até ser encerrado;
- Tipo 3: Idêntico ao Tipo 1, mas com espera de 1/10 do tempo de execução da aplicação "A";
- Tipo 4: Semelhante ao Tipo 3, mas no lugar do incremento ele armazena os últimos 10 *timestamp*[3].

Segundo (LLING et al., 1991, FEITELSON, 2001), para reduzir o efeito de anomalias durante os testes recomenda-se a realização de uma série de repetições para que o cálculo da média dos resultados seja mais representativo. Por esse motivo, cada simulação foi composta por uma bateria de repetições de execução da aplicação "A", tendo como resultado a média dos tempos obtidos. Os resultados podem ser visualizados na FIGURA 8, os quais foram utilizados para avaliar do impacto da preempção de processos.

Observando o gráfico, pode-se perceber que o impacto provocado pela preempção dos processos é desprezível. Esperava-se que quanto maior o número de processos, maior seria o impacto da preempção no tempo de execução da aplicação "A". No entanto, a variação no tempo de execução é no máximo 0,054 segundos. Isto equivale a 3,2% do tempo de execução da aplicação "A", sem a existência de processos adicionais. Tal

[3] O número de segundos decorridos desde as 00:00 horas de 1 de janeiro de 1970 (UTC).

constatação inviabilizou que a preempção pudesse justificar a adoção do número de processos como índice de carga.

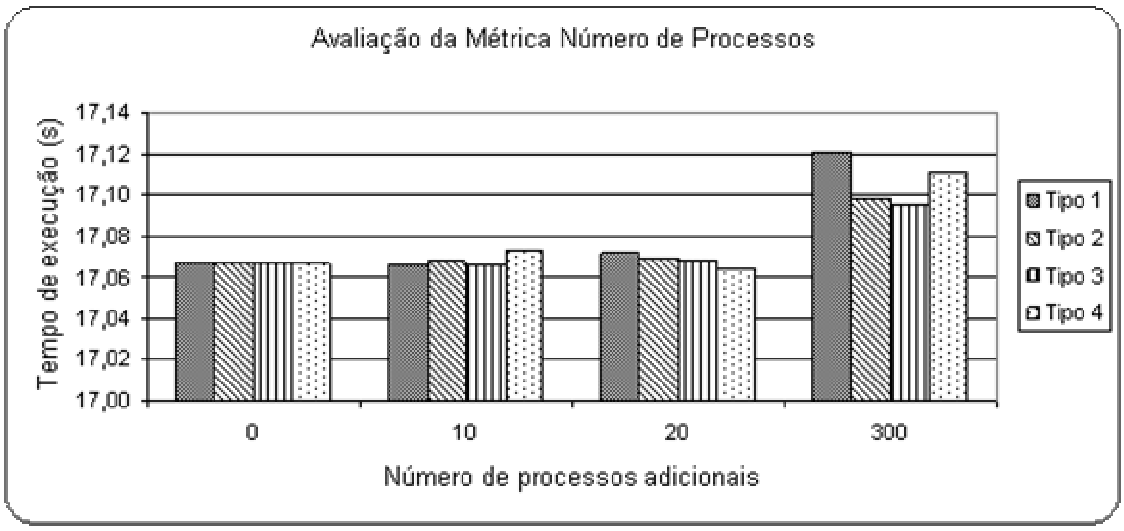

FIGURA 8: Avaliação do impacto do número de processos no cluster.

Sem alterar a prioridade dos processos, e considerando processos que rapidamente liberam a CPU (baixa utilização de CPU), pode-se concluir que o número de processos prontos é uma métrica desprezível para o balanceamento de carga. Ao contrário, se forem analisados os processos com alta utilização de CPU, basta um único destes, para sinalizar a carga do sistema. Assim sendo, partiu-se para a análise do índice de carga de CPU.

5.1.4 Análise do Impacto da Carga de CPU.

O algoritmo de escalonamento de processos sempre utiliza um ou mais parâmetros para decidir qual processo irá ocupar a CPU na próxima distribuição. Alguns parâmetros de decisão podem ser: prioridade de cada processo, ordem de chegada, tempo de espera na fila de prontos, entre outros, os quais podem ser combinados em um mesmo algoritmo, beneficiando uma ou outra classe de processos (SILBERSCHATZ et al, 2001).

Para um algoritmo de balanceamento de carga, estes parâmetros não são de grande importância, visto que os mesmos são utilizados pelo escalonamento local, que busca maximizar a utilização de uma CPU. O balanceamento de carga possui uma visão global do

sistema e deve utilizar como parâmetro de decisão algo que represente a carga de cada componente de forma a maximizar a utilização do sistema como um todo.

Para medir a carga de cada nodo do cluster uma opção é usar a média de uso de CPU. Tal média é recuperada do próprio sistema operacional através da já mencionada, biblioteca LibGTop. Como já foi apresentado, o sistema operacional é o responsável por manter e atualizar estes valores, que estão divididos em três intervalos de tempo. Cada intervalo de tempo representa a carga média de tarefas para o último intervalo de um, cinco e quinze minutos. A carga média é representada por um valor real que pode variar de 0 a $+\infty$, em que um valor próximo de 1,00, em qualquer dos intervalos escolhidos, significa que, no tempo que o intervalo representa, um processo esteve em execução usando 100% de CPU. Quando o intervalo de quinze minutos possuir um valor próximo de 1,00, com certeza os intervalos de um e cinco minutos também o terão.

5.1.5 Escolha do Intervalo para a Média de Carga.

Uma vez definida como métrica de carga a média de uso de CPU, torna-se necessário decidir como trabalhar com os intervalos (um, cinco e quinze minutos) retornados pela biblioteca LibGTop. Assim, deste momento em diante, o intervalo será considerado como média de carga. Sendo assim, a média de carga de quinze minutos representará a média de uso de CPU nos últimos quinze minutos. O intervalo de quinze minutos foi inicialmente considerado por ser adequado ao tempo disponível aos experimentos práticos.

Analisando os possíveis comportamentos da média dentro do intervalo, detectou-se que utilizar um intervalo muito grande, gera uma reação à carga muito lenta. Ou seja, a métrica apresenta uma reação lenta para representar as novas cargas, o que pode gerar uma sobrecarga dos nodos. Ao mesmo tempo, também reage lentamente nos casos em que o nodo está disponível para receber novas cargas, permitindo que os nodos fiquem ociosos.

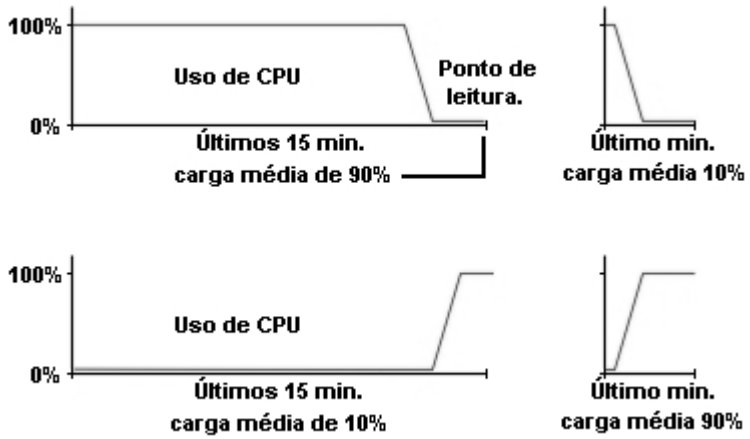

FIGURA 9: Alguns dos possíveis comportamentos da média de carga no intervalo de um (1) e quinze (15) minutos, em comparação com o uso de CPU instantâneo.

Como a intenção do balanceamento de carga é maximizar o uso dos recursos (BEVILACQUA, 1999), decidiu-se abrir mão da média de carga de um intervalo maior e adotar o intervalo de um minuto, visto que este oferece uma melhor utilização dos recursos por reagir mais rapidamente às variações de carga, como ilustra a FIGURA 9. A experimentação efetiva deste índice de carga pode ser vista na próxima seção.

5.2 Testando um Balanceamento Simples: Primeira Versão.

Com o objetivo de estimar o ganho de desempenho proporcionado por uma estratégia simples de balanceamento de carga, experimentos foram realizados na execução das aplicações de cálculo integral e multiplicação matricial usando a carga de CPU como índice de carga. A primeira aplicação divide a área sob a curva de uma função *f(x)* em um conjunto de trapézios e distribui estes trapézios entre os nodos escravos. O processo mestre finaliza o cálculo somando os resultados parciais recebidos de cada nodo escravo. A segunda atua sobre duas matrizes A e B. Neste caso, cada nodo escravo recebe uma quantidade de linhas da matriz A e a matriz B completa. O processo mestre finaliza a multiplicação colocando em C os resultados parciais recebidos de cada nodo escravo. Estes experimentos também fizeram uso da mesma aplicação de carga artificial conforme já definida.

O escalonamento de carga utilizado passa a respeitar as informações de carga de cada nodo. Estas informações são coletadas pelo módulo *MPI_Libt* que foi implementado utilizando a biblioteca *LibGTop* para extrair as informações de carga dos nodos. Esta estrutura de dados passa a ser ordenada de acordo com as informações sobre as cargas do cluster da menor para a maior.

Assim, durante a distribuição *round-robin* original, os nodos menos sobrecarregados são privilegiados sobre os mais sobrecarregados, sempre distribuindo um processo por vez ao mesmo tempo em que a estrutura é percorrida. Além disso, o balanceamento considera que para os nodos com carga de uso da CPU superior a certo fator limitante de carga, a distribuição dos processos para aquele nodo não será realizada, a não ser que não haja outra possibilidade de escolha de nodo com carga inferior ao fator limitante. O fator limitante de carga, fixado empiricamente, em nossos experimentos foi de 0,8 processo/tempo, mas pode ser reajustado dependendo da taxa média de carga de uso da CPU do cluster.

A aplicação que calcula integral foi executada inicialmente apenas nos quatro nodos mais rápidos do cluster, sendo paralelizada primeiramente em quatro processos e depois em oito processos, conforme mostram as FIGURA 10 e 11, respectivamente, as quais comparam a versão original (ORIG) com a versão estendida com o balanceamento (BAL1). De uma forma geral, pode-se observar que o balanceamento de carga melhora o desempenho desta aplicação. No melhor caso para quatro processos, o balanceamento reduziu o tempo de execução da aplicação em torno de 54% e para oito processos 42%.

Quando a quantidade de nodos sobrecarregados é menor, o balanceamento provê maior benefício, que diminui na medida em que um maior número de nodos fica sobrecarregado. Obviamente, a situação em que os quatro nodos estão sobrecarregados é a mesma em que não existe o desbalanceamento e, nesse caso, a LAM/MPI estendida é pouco

pior pela sobrecarga (*overhead*) causado pela função adicional de obter as taxas de carga dos nodos no início da computação, além daquele causado pela carga artificial. Mas este fato não é preocupante, pois este *overhead* é sempre constante para um mesmo cluster, independente da aplicação, sendo até desprezível para aplicações com tempo de execução a partir de minutos.

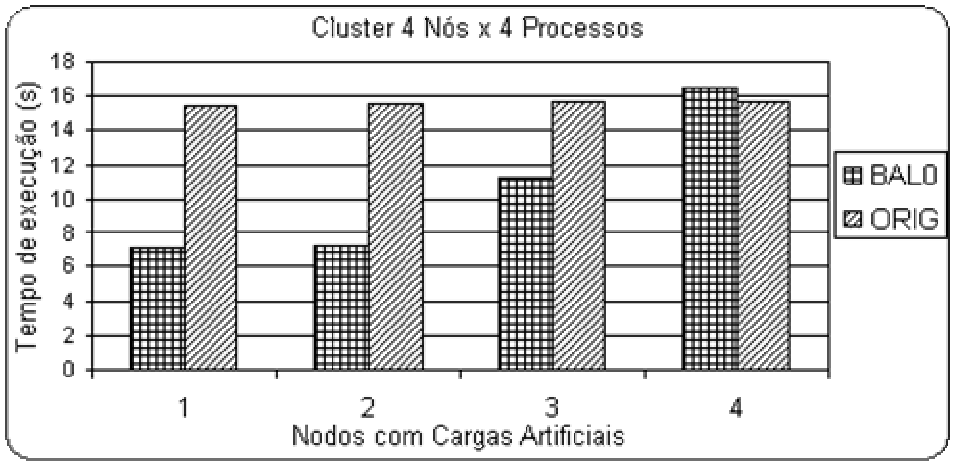

FIGURA 10: Cálculo de Integral: LAM/MPI Balanceada versus Original 4x4.

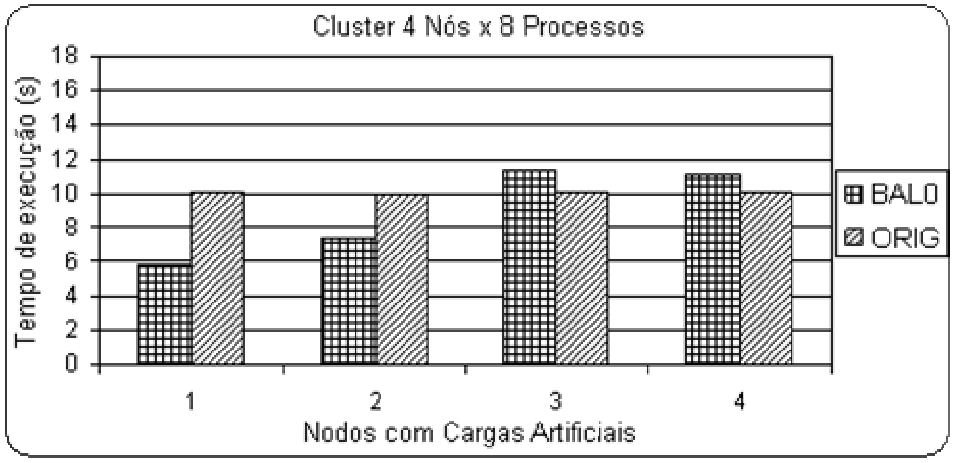

FIGURA 11: Cálculo de Integral: LAM/MPI Balanceada versus Original 4x8.

De qualquer forma, o fato da aplicação ter sido dividida em oito processos fez o benefício relativo do modelo BAL1 sobre o modelo ORIG diminuir, do que aquele obtido quando a aplicação foi dividida em quatro processos. Este fato se deve, em parte, porque a carga artificial causou menos impacto na aplicação. Note que, quando a aplicação é dividida em quatro processos, um nodo sobrecarregado prejudica ¼ da aplicação em um único

processo, mas quando a aplicação é dividida em oito processos, o nodo sobrecarregado prejudica ¼ da aplicação em dois processos, ou seja, o impacto é "amortizado".

De fato, nas experimentações realizadas durante a pré-investigação, o benefício do modelo BAL1 sobre o modelo ORIG é diretamente proporcional ao tamanho da carga artificial imposta no sistema. Quanto maior o número de aplicações de carga artificial maior o benefício, já que o balanceamento de carga busca os nodos ociosos enquanto for possível. Nas experimentações apresentadas nas FIGURA 10 e 11, foram injetadas quatro carga artificiais em cada nodo sobrecarregado. A FIGURA 12 mostra os resultados obtidos utilizando-se os oito nodos do cluster. Nestes experimentos foram usadas apenas duas cargas artificiais. No melhor caso, o balanceamento estático reduziu o tempo de execução da aplicação em torno de 51%.

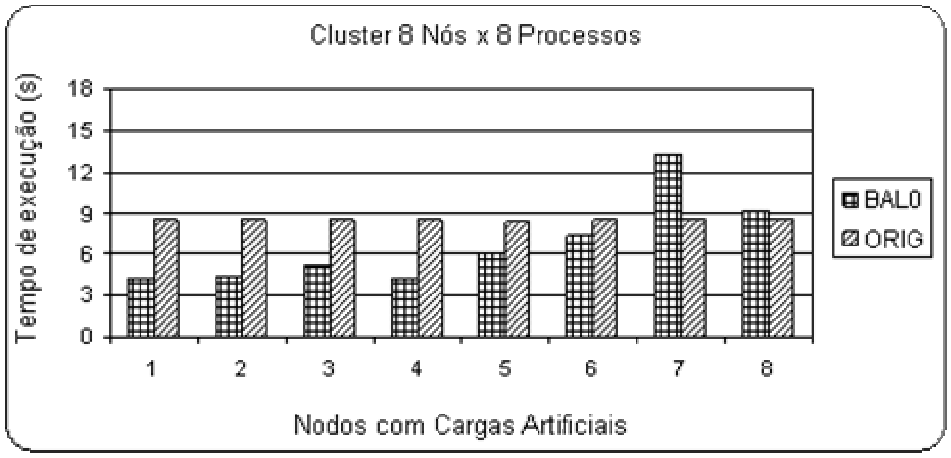

FIGURA 12: Cálculo de Integral: LAM/MPI Balanceada versus Original 8x8.

Uma observação importante que ainda não foi mencionada é que o modelo ORIG quase não se afeta pelo desbalanceamento provocado pela carga artificial. Este fato ocorre porque o tempo de duração de uma aplicação distribuída é basicamente imposto pelo processo que demora mais, pois a conclusão da aplicação deve necessariamente esperá-lo. Neste sentido, é indiferente se a aplicação tem que esperar por um, dois ou todos os processos terminarem, desde que ela necessariamente já tenha que esperar por um processo mais demorado.

Com o objetivo de certificar de que os resultados obtidos não são específicos para a aplicação de cálculo integral, experimentos com a aplicação de multiplicação matricial, também foram realizados e os resultados são mostrados na FIGURA 13. Neste caso, o balanceamento também se mostrou bastante favorável e no melhor caso, a redução do tempo de execução da aplicação atinge aproximadamente 59%. Convém ressaltar que neste caso foi injetada apenas uma aplicação de carga artificial, indicando que o benefício poderia ser ainda maior.

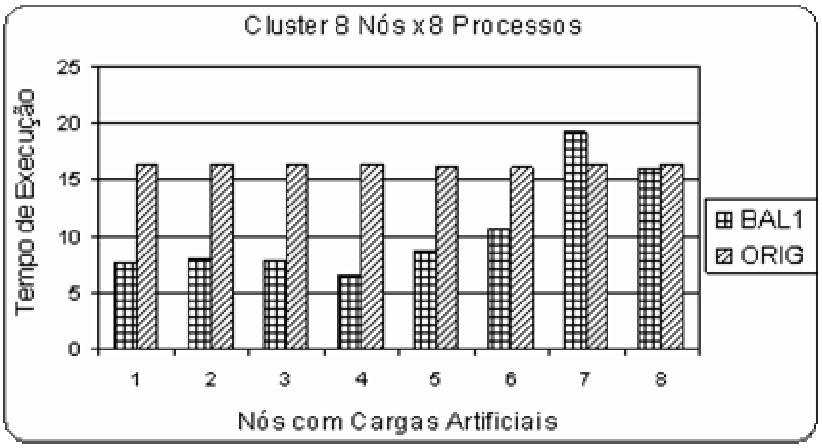

FIGURA 13: Multiplicação Matricial: LAM/MPI Balanceada versus Original 8x8.

Nas FIGURA 12 e 13 pode-se observar um prejuízo no desempenho quando sete nodos são sobrecarregados com carga artificial, sendo bem mais expressivo na aplicação de cálculo integral. Após a análise da situação, constatou-se que a justificativa para o ocorrido é que estando apenas um nodo com pouca carga, uma quantidade muito maior de processos é alocada sobre este nodo, se comparado com qualquer situação anterior. O ganho médio entre ambas as aplicações sobre a aplicação original pode ser visto na FIGURA 14.

Certifica-se nesta figura que o maior ganho ocorre quando quatro nodos estão sobrecarregados e a partir daí o ganho passa a diminuir tornando-se até negativo conforme já mencionado. Este fato está relacionado com a forma pela qual é feita a sobrecarga artificial do sistema. Em nossos experimentos, aplicamos as sobrecargas primeiramente nas estações

lentas (1.8GHz) e posteriormente nas estações rápidas (3.0GHz). Assim, quando cinco ou mais estações são sobrecarregadas, a sobrecarga artificial passa a atingir os nodos de maior capacidade de processamento, gerando a queda no desempenho.

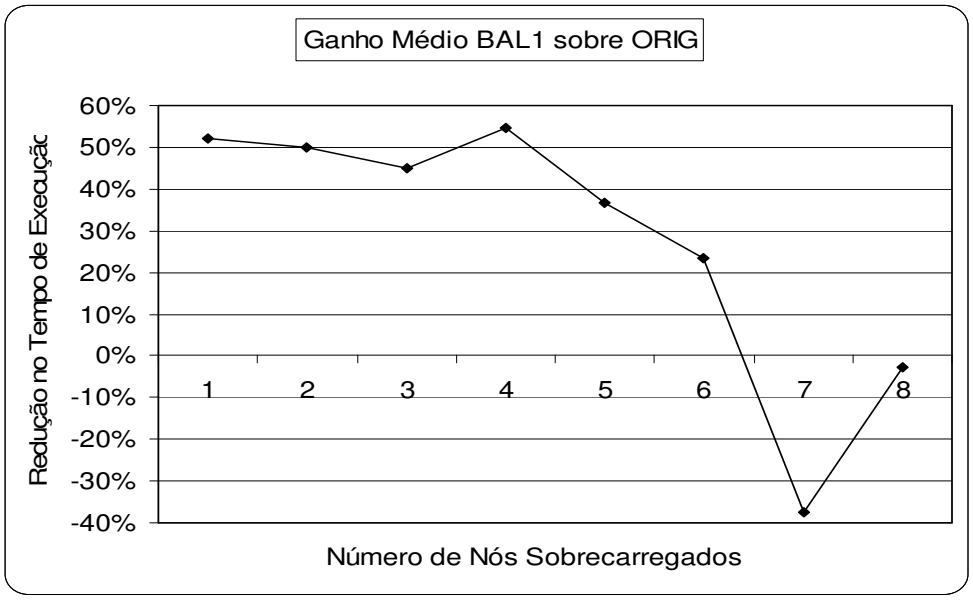

FIGURA 14: Ganho Médio Sobre as Aplicações 8x8.

5.3 Enriquecendo os Experimentos.

Apesar de proverem aumentos significativos no desempenho da execução das aplicações distribuídas, os experimentos relacionados nas seções anteriores foram bastante simples, tanto no que se refere à técnica de balanceamento, quanto ao índice de carga utilizados. A experimentação de algoritmos e índices de cargas mais elaborados sugere a obtenção de maior desempenho. Com isso surgem questões de projeto, que são pontos da pesquisa em que a decisão por uma ou outra solução torna o algoritmo mais ou menos complexo para o projetista, com melhor ou pior desempenho, com maior ou menor facilidade de uso, entre outras características.

Uma preocupação que surgiu após os primeiros experimentos foi a de obter informações mais realistas, diretamente do sistema, que pudessem refletir de forma mais adequada as variações da carga no sistema. Sabe-se que o sistema operacional gerencia e

armazena informações sobre os recursos e dispositivos que controla (SILBERSCHATZ et al, 2001). Além disso, o uso de outro tipo de aplicação paralela, a de cálculo de similaridade de DNA, foi utilizada por se tratar de uma aplicação bastante usada atualmente. As próximas seções discutem estas questões.

5.3.1 Aplicação de Reconhecimento de Genoma.

Esta aplicação realiza a comparação das seqüências genéticas de dois indivíduos e retorna um escore que representa o grau de semelhança entre os mesmos. Os genomas são codificados como seqüências de caracteres. O algoritmo simplificado da aplicação (MARTINS, 2005) para o cálculo da similaridade entre duas seqüências de DNA é apresentado no QUADRO I.

Esse algoritmo paralelo calcula a similaridade entre duas seqüências X = [1..m] e Y = [1..n]. A similaridade entre X e Y é obtida alinhando-se as duas seqüências e comparando as colunas correspondentes. Cada comparação resulta em um valor que representa o grau de igualdade da coluna. A soma resultante dos valores das colunas produz o escore.

O algoritmo funciona da seguinte forma. Considere cada coluna i. Se X[i] = Y[i], a coluna irá receber um valor p(X[i] ,Y[i]) > 0, e se X[i] ≠ Y[i], a coluna irá receber um valor p(X[i] ,Y[i]) ≤ 0. Finalmente, a coluna com um espaço recebe um valor –d, onde d Є N. Então, procura-se pelo alinhamento (ótimo) que representa o escore máximo. Esse escore máximo é chamado similaridade e pode ser denotado sim(X, Y).

A solução para o problema de determinar o escore máximo entre duas seqüências arbitrárias pode ser obtida calculando-se todos os escores, iniciando-se por seqüências pequenas e utilizando os resultados previamente calculados para determinar o resultado para seqüências maiores. Assim, existem m+1 seqüências possíveis para X e n+1 para Y. Os cálculos podem ser arranjados em uma matriz S de dimensões (m+1) x (n+1), onde cada

posição S[r, s] representa a similaridade entre X[1..r] e Y[1..s]. O escore de similaridade S pode ser calculado assim: S[r, s] = max(S[r,s-1] – k, S[r-1,s-1] + p(r, s) , S[r-1,s] – k).

```
se rank == 0 então
     leia x e y;
     MPI_Broadcast(m,n); //envia comprimento de X e Y
     MPI_Broadcast(X); //envia a seqüência X
     pn = n / p;
     o = pn;
     para i = 1 até p-1 faça
     MPI_Send(Y[o], pn, i); //envia partes de Y
     o = o + pn;
senão
     MPI_Broadcast(m,n); //recebe comprimento de X e Y
     pn = n / p;
     MPI_Broadcast(X); //recebe X
     MPI_Recv(Y, pn, 0); //recebe a parte de Y

S = CreateMatrix(m, pn);
pm = m / p;
para k = 0 até m paço pm faça
     se rank != 0 então
          MPI_Recv(S[0], m, rank-1);
     para r = k + 1 até k+pm faça
          para s = 1 até pn faça
               S[s][r] = max(S[s-1][r]-d, S[s][r-1]-d,
                             S[s-1][r-1]+p(x[r], y[r]));
     se rank != size - 1 então
          MPI_Send(S[pn], m, rank+1);
```

QUADRO I: Algoritmo simplificado do Cálculo de Similaridade em Paralelo.

A aplicação genoma recebe como entrada a seqüência genética de dois indivíduos. O processo mestre (*rank 0*), além do cálculo, fica responsável por transmitir as seqüências genéticas para os demais processos. A primeira seqüência é replicada, na integra, para todos os processos. A segunda seqüência é dividida, em partes iguais, de acordo com o número de processos e cada segmento é transmitido para um processo. No fim, uma operação de redução do tipo MAX é executada e o valor (*score*) do maior seguimento encontrado nos dois indivíduos é retornado.

5.3.2 Métrica de Poder Computacional.

Para aproveitar a heterogeneidade do cluster na distribuição de carga, é preciso classificar os nodos de acordo com seus recursos, tais como quantidade de memória disponível, taxa de transferência do dispositivo de rede e poder de processamento da CPU, entre outros. Neste trabalho, o poder computacional tem sido usado pelo fato de ser a característica diferencial no cluster utilizado, uma vez que os demais recursos são semelhantes entre os nodos.

Para este trabalho adotou-se o poder computacional como forma de classificar os diferentes nodos do cluster. A unidade da métrica escolhida, para quantificar o poder computacional de cada nodo, foi o BogoMIPS (LINUX ONLINE, 2006) e esta é utilizada nas próximas experimentações. Trata-se de uma métrica criada por Linus Torvalds, calculada pelo *kernel* do Linux antes da carga do sistema operacional e utilizada como medida de tempo por diversos dispositivos de hardware.

O nome da métrica tem uma origem curiosa. MIPS é a sigla para Milhões de Instruções Por Segundo (*Millions of Instructions Per Second*). Bogo advém de *bogus*, gíria em inglês para algo imaginário. Assim, alguns autores interpretam o BogoMIPS como sendo "o número de milhões de vezes por segundo que o processador faz absolutamente nada". De fato, o BogoMIPS é calculado com um laço de repetição em espera ocupada (*busy wait*).

A motivação para a escolha do BogoMIPS é que a métrica é calculada no momento da carga do sistema operacional, o que previne qualquer sobrecarga da abordagem de balanceamento de carga para quantificar o poder computacional do nodo. Uma vez que o valor do BogoMIPS é disponibilizado pelo sistema operacional. O que não aconteceria se fosse adotado o próprio MIPS ou o MFLOPS, conforme Legrand (LEGRAND et al, 2004), pois esses valores são obtidos por *benchmarks* que não fazem parte do sistema operacional.

Além disso, existe certa variação nos valores em MFLOPS retornados por diferentes *benchmarks*, o que não acontece com o BogoMIPS. Duas máquinas com uma mesma configuração irão obter o mesmo valor ou um valor muito próximo, qualquer que seja a distribuição do Linux. Destaca-se também o fato desta métrica apresentar um valor constante, obtendo sempre o mesmo valor toda vez que o SO for carregado. Outro ponto, é que utilizar o MFLOPS pode não trazer uma considerável melhora nos resultados, já que este é considerado adequado quando se conhece o número de operações de ponto-flutuante do problema (HOCKNEY & BARRY, 1994).

Para avaliar o benefício do BogoMIPS como métrica do poder computacional, um experimento foi realizado para comparar a versão original da LAM/MPI e outra que foi estendida com o emprego do BogoMIPS, BAL2. Nos gráficos gerados, o eixo X representa o número de nodos sobrecarregados artificialmente no momento da distribuição da carga. A aplicação genoma foi utilizada para avaliar o desempenho da métrica e a FIGURA 15 mostra os resultados em termos de desvio padrão.

O alto desvio padrão no tempo de execução é conseqüência das alternativas existentes para se distribuir a carga. O modo como a carga artificial foi injetada no cluster permitiu realçar tal conseqüência, já que as cargas artificiais ocupam inicialmente as estações lentas até que comece a ocupar as estações rápidas. Nesse experimento, devido a existência da carga artificial, as alternativas para a distribuição dos processos da aplicação genoma são previsíveis.

Por exemplo, na FIGURA 15, quando 3 nodos são sobrecarregados artificialmente, restam cinco nodos sem carga e oito processos da aplicação genoma para distribuir. Dentre estes cinco nodos, um é uma estação lenta. Assim, se o balanceamento de carga não respeita o poder computacional dos nodos, existe a probabilidade de que, em 27%

dos casos, a estação lenta receba dois processos, o que eleva o tempo de execução da aplicação gerando o alto desvio padrão.

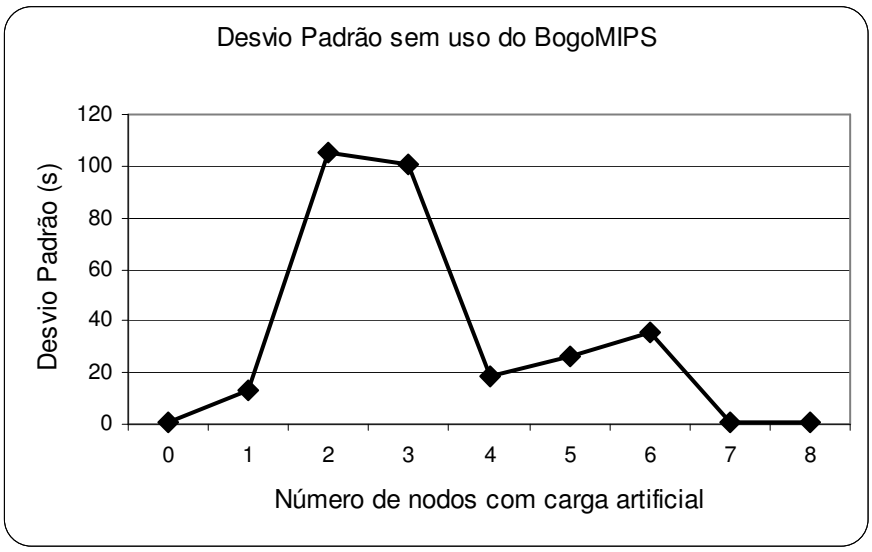

FIGURA 15: Desvio padrão do tempo de execução sem respeitar o poder computacional.

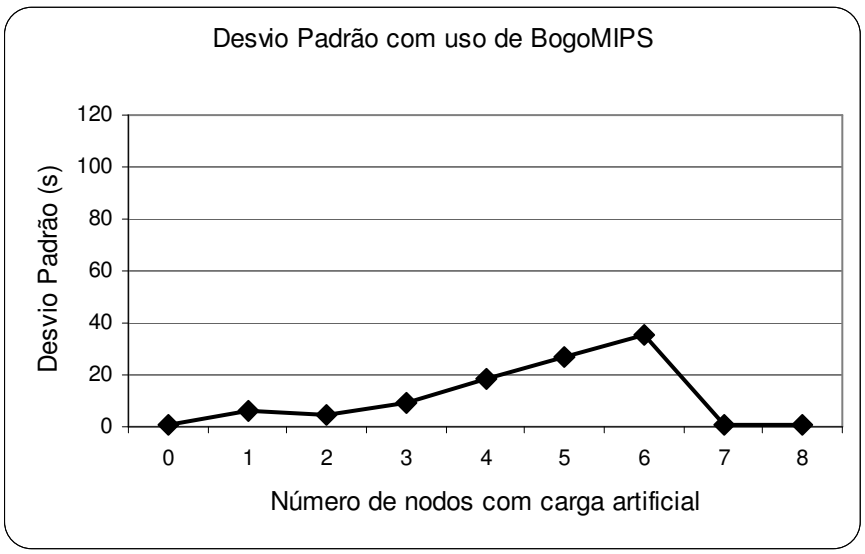

FIGURA 16: Desvio padrão do tempo de execução respeitando o poder computacional.

A FIGURA 16 mostra um experimento similar, mas neste caso, a técnica de balanceamento de carga está respeitando o poder computacional antes de verificar o índice de carga de CPU dos nodos para distribuir a carga. Observe que nos pontos 2 e 3, o valor do desvio padrão é 1/10 do valor para estes mesmos pontos na FIGURA 15. Isso ocorre porque com cinco nodos e oito processos para distribuir, sendo que apenas um nodo é uma estação

lenta, não existe a probabilidade de que esta receba dois processos, mantendo os resultados estáveis.

Nos pontos 4, 5 e 6, tanto da FIGURA 15 como da FIGURA 16, o desvio padrão volta a subir por conseqüência da comunicação entre os processos MPI da aplicação genoma, que troca dados de forma circular. Como o número de nodos é inferior à metade do número de processos (8), as alternativas de distribuição e comunicação entre estes podem gerar gargalos.

5.4 Testando um Balanceamento mais Eficiente: Segunda Versão.

Uma vez definido o parâmetro de carga como sendo a média de uso de CPU, no intervalo de um minuto, é preciso então determinar como balancear com base nesta informação. Uma opção é interpretar a informação de forma discretizada, adotando um limiar para que todos os nodos que apresentem média de uso de CPU (carga) superior a este limiar não recebam mais carga, conforme ilustra a FIGURA 17.

FIGURA 17: Comparativo entre evolução da média de carga e nível de carga instantâneo.

Note que o limiar não impede que a carga do nodo ultrapasse o seu valor. Como o parâmetro de carga representa a média de uso de CPU em um intervalo de tempo (média de carga), a seguinte situação pode acontecer: um nodo com carga de 0,2 recebe 10 novos processos em menos de 30 segundos, tendo cada processo um tempo de execução previsto de dez minutos. Os 30 segundos não é tempo suficiente para que a média de um (1) minuto represente toda a nova carga do nodo, e nesse caso, a carga do nodo irá ultrapassar o valor de

10 (valor obtido empiricamente) após três minutos de execução, contados a partir do instante em que o primeiro processo entra em execução. Se for utilizado um intervalo superior a um minuto (cinco ou quinze minutos) para a tomada de decisão, o resultado será pior, lembrando que a média de um intervalo maior reage ainda mais lentamente.

Portanto, nota-se a necessidade de se adequar corretamente o valor do limiar. Para isso, várias simulações foram executadas utilizando diferentes valores da média de uso de CPU para o limiar (0,60; 0,70; 0,80 e 0,90). Cada limiar foi simulado com aplicações sendo disparadas a cada 10, 20 e 40 segundos, somando um total de sete aplicações por simulação.

A escolha de sete aplicações é para garantir que disparando uma nova aplicação a cada 10 segundos, tenham-se novas aplicações em todo o intervalo da média de carga. Com novas aplicações a cada 20 e 40 segundos é preciso manter as sete aplicações para que o nível final de carga seja equivalente em todas as simulações. Para este experimento não foi necessário carga artificial e aplicação genoma foi utilizada para a tomada de tempo. Desta forma, o experimento é formado pela execução de sete aplicações genoma.

Outra característica das simulações é que a aplicação genoma é disparada solicitando cinco processos, ou seja, utilizando cinco dos oito nodos do cluster. Isto possibilita que a técnica de balanceamento de carga decida quais nodos irão receber os processos da nova aplicação e permite verificar se houve redução no tempo de execução.

Teoricamente, quanto maior o tempo entre cada nova aplicação, o limiar pode ser maior. Enquanto que se aplicações podem ser disparadas com um intervalo de tempo inferior a 30 segundos, recomenda-se um limiar baixo (0,60) para evitar que o nodo seja sobrecarregado antes que o índice de carga tenha tempo de representar toda a nova carga.

Como a idéia deste algoritmo de balanceamento de carga para a LAM/MPI não é restringir o acesso ao cluster, quando todos os nodos estiverem com a carga superior ao

limiar, o escalonamento dos processos irá respeitar, em ordem decrescente, o poder computacional dos nodos. Caso haja interesse em restringir que o cluster trabalhe em sobrecarga, o comportamento aqui descrito pode ser desativado, impedindo que novas aplicações sejam disparadas no cluster.

Com os resultados das simulações, iniciou-se o trabalho de análise. Imediatamente os resultados confirmaram a hipótese citada anteriormente. Com novas aplicações disparadas em intervalos de 40 segundos (FIGURA 18), a melhor média de tempo de execução foi obtida quando utilizado o limiar de 0,90.

Outro comportamento que se manteve, indiferente do limiar adotado, foi que quanto maior o tempo entre cada nova aplicação melhor é o resultado. Além do motivo óbvio, de que cada aplicação provocou menor interferência uma à outra, um segundo motivo é que a média de carga teve mais tempo para assimilar a nova carga antes do surgimento de mais uma aplicação, permitindo uma melhor distribuição dos novos processos.

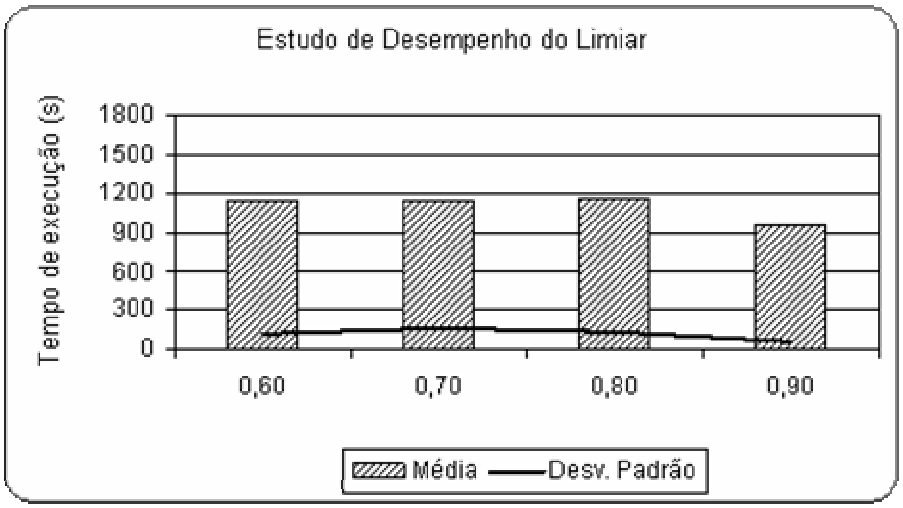

FIGURA 18: Gráfico comparativo do tempo de execução de cada limiar, simulado com novas aplicações a cada 40 segundos.

Os resultados do experimento apresentados na FIGURA 19 mostram um comparativo entre a segunda versão e a versão original. Apesar de prover melhorias na maioria das situações, pode-se observar uma deficiência no algoritmo implementado. Trata-se do caso de detectar quando o cluster é considerado sobrecarregado, pois enquanto o algoritmo

encontrar um nodo com índice de carga inferior ao limiar (ocioso), este nodo irá receber toda a nova carga, é o que acontece no ponto 7, quando apenas um nodo está ocioso. Quando não existem mais nodos ociosos, a técnica BAL2 distribui os novos processos de forma semelhante ao comportamento ORIG – um processo para cada nodo – respeitando apenas o poder computacional. É possível adequar o algoritmo proposto, para que decida pelo comportamento ORIG em algum ponto anterior. A FIGURA 20 mostra uma visão geral do ganho promovido por este algoritmo, o qual atinge mais de 63% quando apenas um nodo é sobrecarregado. A situação, já comentada, quando 7 nodos são sobrecarregados trata-se de uma situação exclusiva (“ponto fora da curva”).

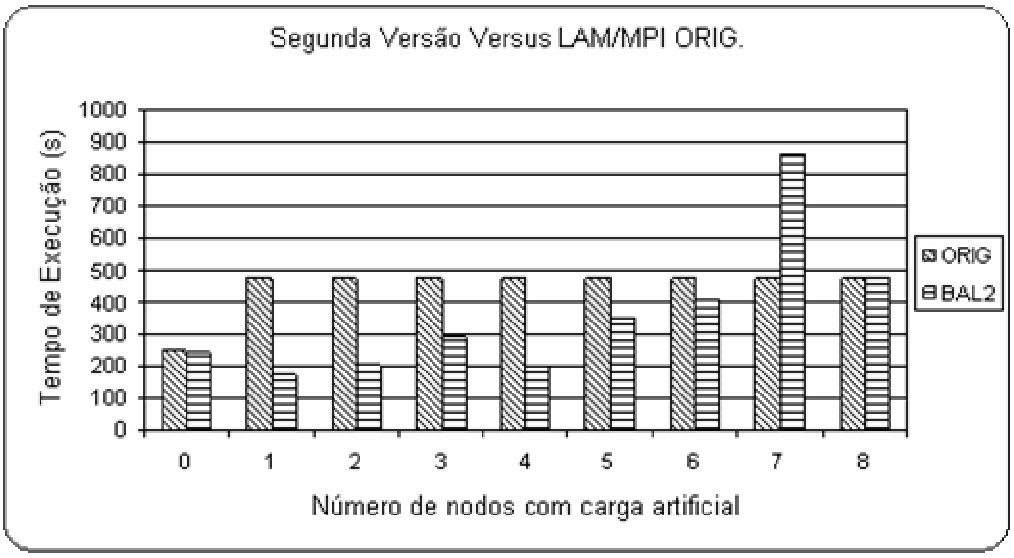

FIGURA 19: Comparativo do tempo de desempenho entre a segunda versão (BAL2) e o comportamento padrão da biblioteca LAM/MPI (ORIG).

Para o ambiente de pesquisa deste trabalho, um bom ponto pode ser quando o número de nodos ociosos for inferior a 25%. Mas se 25% for aplicado a um cluster de 100 estações, pode levar a perda de desempenho. Outra solução seria verificar o número de nodos livres (L) e o número de processos solicitados (p), e considerar até que ponto é viável que processos da mesma aplicação compartilhem recursos. Para um experimento simples, adotou-se que cada processo pode compartilhar o processador com outro processo da mesma aplicação ($p \leq L * 2$).

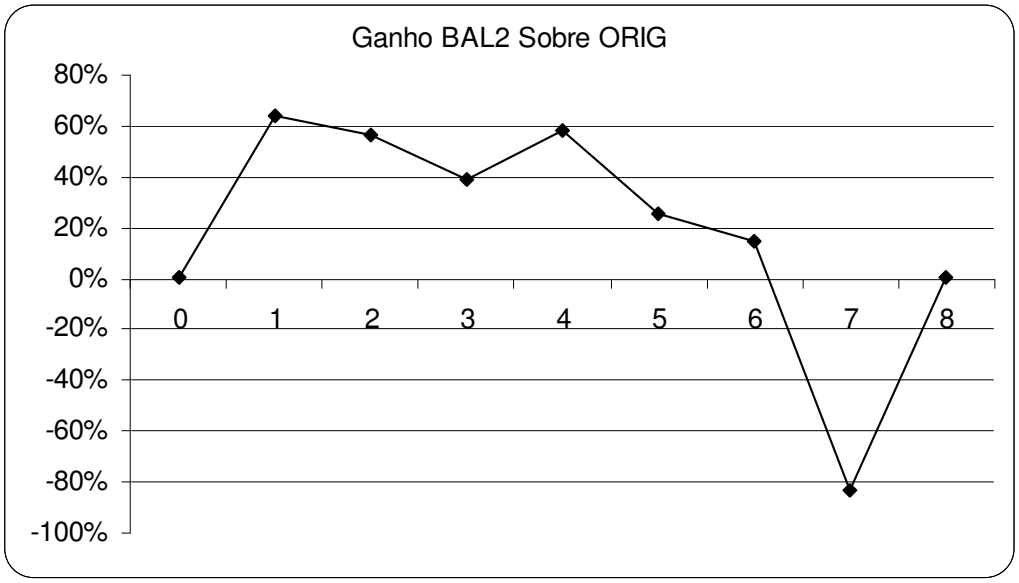

FIGURA 20: Ganho de desempenho da segunda versão sobre a original

```
L = número de nodos livres
p = número de processos solicitados (novos)

se p <= L * 2 então
     distribuir processos utilizando BAL2
senão
     distribuir  processos  utilizando  ORIG
```

QUADRO II: Pseudo-algoritmo para detecção de cluster sobrecarregado.

O pseudo-algoritmo do QUADRO II foi implementado e o resultado pode ser consultado na FIGURA 21. O principal objetivo foi alcançado, fazer com que o balanceamento de carga não apresente resultados piores que o comportamento padrão da LAM/MPI. É possível observar também que no ponto 6 houve aumento de desempenho. Isso ocorre porque os oito processos não são mais distribuídos para os únicos dois nodos ociosos, adequando melhor a carga do cluster. Mas nenhum estudo mais profundo foi realizado para avaliar se a lógica adotada é a mais adequada, mesmo porque, ainda restam questões mais sérias a respeito desta proposta de balanceamento de carga.

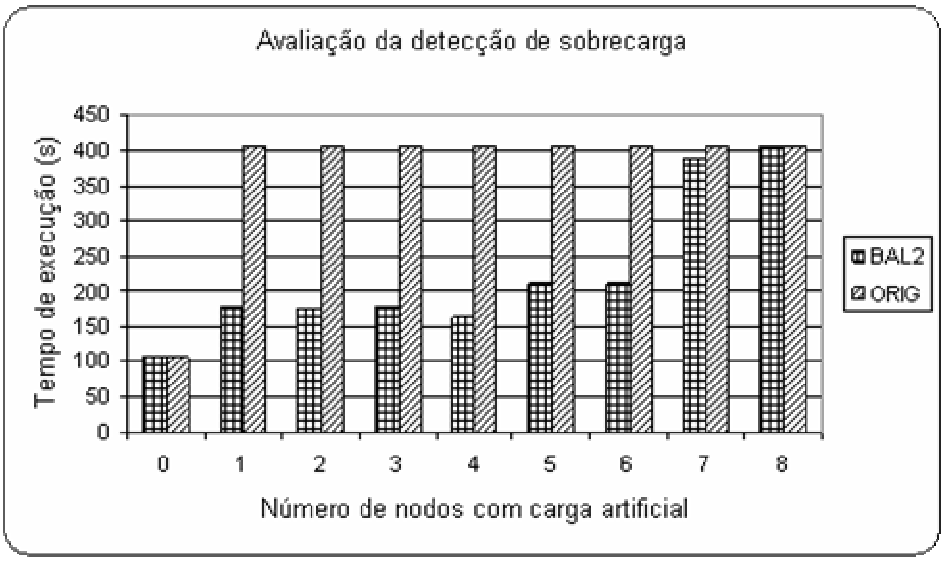

FIGURA 21: Avaliação do pseudo-algoritmo de detecção de sobrecarga.

Dando prosseguimento na avaliação do método de tomada de decisão, foi possível observar que o ideal seria um limiar auto-ajustável. Por mais que o limiar fosse adotado como um parâmetro passado pelo administrador do cluster, que conhece a utilização do mesmo, o limiar se torna inútil quando o estado de carga do cluster é de sobrecarregado.

Outro ponto detectado, é que basear a tomada de decisão no limiar (média de carga), despreza a diferença de poder computacional dos nodos, podendo gerar no longo prazo, estações lentas sobrecarregadas e estações rápidas ociosas. Tal situação pode ocorrer quando todos os nodos apresentarem a média de carga de CPU superior ao limiar, então a distribuição de carga vai respeitar a ordem decrescente do poder computacional, mas se as aplicações disparadas solicitarem um número de processos igual ao número de nodos, todos os nodos receberão novas cargas em igual quantidade.

Como a pesquisa deseja utilizar o balanceamento de carga para tirar proveito de clusters heterogêneos, e a abordagem apresentada nesta seção não apresentou um resultado satisfatório, partiu-se para um novo método de tomada de decisão.

5.5 Testando um Balanceamento mais Eficiente: Terceira Versão.

Na busca por uma técnica de balanceamento de carga que tire proveito de clusters heterogêneos, um novo método de tomada de decisão foi elaborado. A idéia é considerar a informação sobre o número de processos em execução em cada nodo.

A média de carga que até então era interpretada como limiar de carga, pode ser interpretada como o número de processos em execução no nodo. Por exemplo, um nodo com uma média de carga de 1,98 possui, no mínimo, dois processos em execução. Pode possuir mais do que dois processos, se os processos em execução ficarem ociosos por períodos relativamente longos, como na realização de E/S com elevada freqüência ou na espera por resultados de outros processos.

Nos testes realizados, notou-se que quando eram disparados muitos processos, por exemplo, 300, que passavam mais tempo no estado de espera do que efetivamente ocupando a CPU, a média de carga não ultrapassava o valor de 0,10. Esse é um exemplo de que a média de carga pode ser interpretada como o número de processos que efetivamente estão ocupando a CPU.

Além desta nova interpretação, adicionou-se um princípio que consiste na idéia de que dividir o tempo de CPU com um processo da mesma aplicação paralela, resulta em um tempo de execução total menor do que se o tempo de CPU for divido com um processo de outra aplicação, quando se consideram os casos em que mais de uma aplicação paralela encontra-se em execução no cluster. Resumindo, o cluster pode apresentar um *throughput* maior, se existir mais de uma aplicação paralela disputando o cluster e os processos de uma mesma aplicação paralela competirem preferencialmente pela mesma CPU.

Imagine o seguinte, um sistema formado por dois processadores em que o processador “A” é 1/3 mais rápido que o processador “B”, que o sistema executa sempre a mesma aplicação e a carga da aplicação é dividida em duas partes iguais. No início, em que os

dois processadores encontram-se ociosos, cada processador recebe uma parte da carga da aplicação x_1, e o tempo de execução é regido pelo processador "B" (situação inicial).

Considere que no instante T_1, antes do término da aplicação x_1, uma nova aplicação x_2 seja submetida, então o sistema pode escolher entre entregar uma parte da carga para cada processador (situação 1) ou as duas partes para o processador mais rápido "A" (situação 2) em virtude de evitar que duas aplicações distintas disputem o CPU mais lento, como representado pela FIGURA 22. Nesse caso, qualquer das situações irá apresentar o resultado da aplicação x_2 no mesmo instante T_6. Agora observe que se no instante T_4 uma nova aplicação x_3 for submetida ao sistema, a situação 2.1 apresenta um desempenho superior, finalizando a aplicação x_3 em T_8 enquanto a situação 1.1 finaliza a mesma aplicação em T_9.

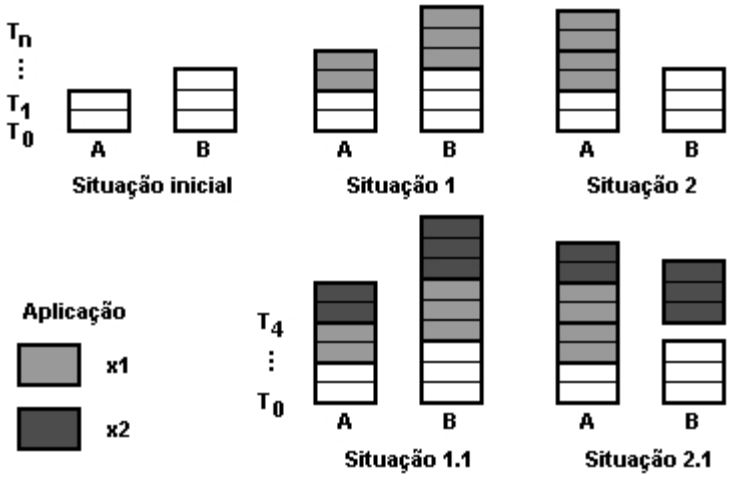

FIGURA 22: Distribuição de carga respeitando o poder computacional e carga atual.

Com base nessa interpretação, uma terceira técnica que realiza a distribuição da carga entre os nodos, respeitando inclusive o poder computacional destes, foi elaborada e pode ser vista no QUADRO III. A técnica consulta quantos nodos forem necessários até que todos os processos solicitados pelo usuário sejam atribuídos a um processador.

A decisão para a escolha de qual processador irá receber os processos MPI é inicialmente baseada na média de carga existente no processador, seguida do poder computacional do mesmo. Todo processador ocioso pode receber novos processos. Se este não é o caso, o algoritmo de distribuição de carga considera o número de nodos disponíveis

(n), o número total de processos (MPI ou não MPI) no cluster (p) e um limiar de carga (t) o qual pode ser ajustado.

Quando o cluster está trabalhando com até *n*t* processos, os nodos são classificados em dois grupos, o das estações rápidas e o das estações lentas. A distribuição da carga depende então da origem do processo MPI e do grupo considerado. No grupo das estações rápidas, é permitido que um processo MPI compartilhe um processador com outro processo de qualquer aplicação. Entretanto, no grupo das estações lentas, um processo MPI só pode compartilhar um processador com outro processo da mesma aplicação MPI. Se o número de processos, disputando recursos do cluster, superar o valor de *n*t*, o algoritmo busca o nodo mais adequado para receber cada novo processo, deixando de considerar que o cluster está dividido em dois grupos.

```
t = limiar definido pelo administrador (t=2)
p = número de processos em execução
n = número de nodos do cluster
np = número de novos processos
enquanto np > 0 faça
     enquanto verdadeiro faça
          se p > n * t então
               Busca o nodo(i) com a melhor relação
               poder computacional por processo;
               fim;

          senão
               se ocioso(i) então
                    fim;
               se nodo_rápido(i) então
                    permite competir com até 1
                    processo de outra aplicação
               senão
                    permite competir apenas com 1
                    processo da mesma aplicação
          i = i + 1;
     fim
     atribui_processo(i);
     np = np - 1;
fim
```

QUADRO III: Pseudo-algoritmo responsável pela distribuição de carga BAL3.

A busca pelo melhor nodo consiste em dividir o poder computacional do nodo pela estimativa da ocupação da CPU. O nodo que apresentar a melhor relação poder computacional por processo (BogoMIPS/p), é o nodo que deve receber o novo processo. Os processos recém distribuídos também são computados quando da distribuição de um segundo processo, da mesma aplicação, para qualquer outro nodo.

Por exemplo, um novo processo deve ser distribuído, o nodo 1 possui poder computacional de 5000 BogoMIPS e a sua média de carga é de 3,95, enquanto o nodo 2 possui poder computacional de 3000 BogoMIPS e média de carga de 1,98. Calculando[4] a relação poder computacional por processo, o nodo 1 resulta em 1250 BM/p (leia-se BogoMIPS por processo) e o nodo 2, 1500 BM/p. Logo, o nodo 2, mesmo sendo de poder computacional inferior, irá receber o novo processo. Para os experimentos realizados neste trabalho, o valor adotado para o limiar *t* foi 2 (t = 2), mais experimentos com outros valores para *t* não devem ser ignorados.

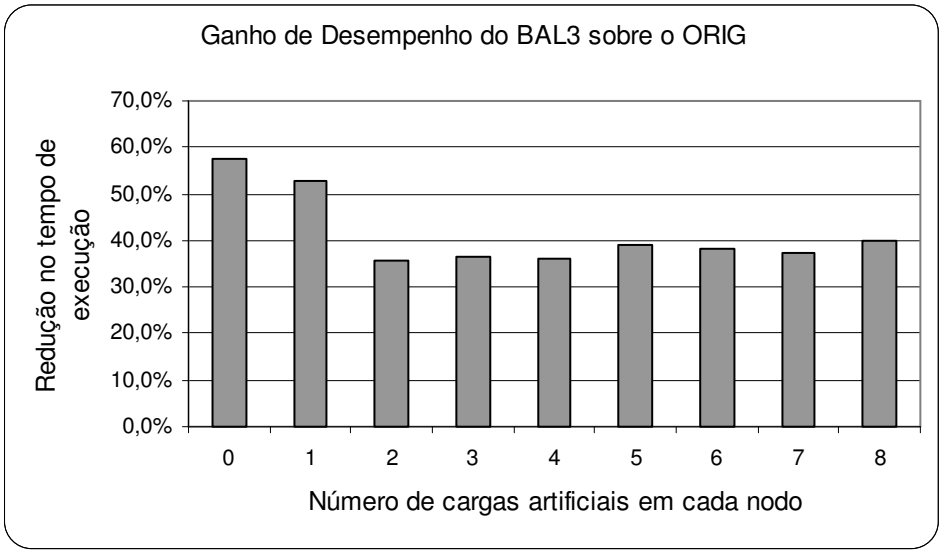

FIGURA 23: Ganho médio da versão balanceada (BAL3) sobre a versão original da LAM/MPI.

Com esta técnica, os resultados mostraram um desempenho satisfatório, conforme vistos na FIGURA 23. O problema de se trabalhar com o cluster sobrecarregado foi

[4] Para os cálculos, deve ser utilizado o menor inteiro maior que o valor recuperado pelo histórico de carga. O motivo deste arredondamento foi explanado no inicio desta subseção.

minimizado, pois a distribuição dos novos processos respeita a relação poder computacional/processo. Além disso, a técnica mostrou ser capaz de aproveitar a diferença de poder computacional existente entre os nodos do cluster.

Um resultado curioso foi o ganho de desempenho alcançado pela técnica de balanceamento com o cluster ocioso (ponto 0). O ganho de desempenho superior a 57%, mostra na verdade como é possível perder desempenho apenas porque o cluster está mal configurado. Recorde que na aplicação genoma, o processo mestre (*rank 0*) é responsável por transmitir as seqüências genéticas para os demais processos. Acontece que na LAM/MPI ORIG a ordem de distribuição dos processos respeita a ordem do arquivo de definição de nodos (*boot schema*), que neste caso inicia com as estações lentas e termina com as estações rápidas. Permitir que o processo mestre fosse alocado a uma estação lenta provocou a redução do desempenho, o que não acontece com a técnica de balanceamento de carga, pois esta opera sobre um vetor ordenado segundo o poder computacional das estações, alocando o processo mestre a uma estação rápida. É importante mencionar que essa diferença no desempenho é específica para a aplicação genoma.

Nos demais casos, o ganho de desempenho se manteve entre 30 % e 40 %, sendo que os processos foram distribuídos utilizando a regra de poder computacional/processo. Como todos os nodos se encontravam com a mesma carga em cada simulação, as estações lentas só receberam os processos da aplicação genoma enquanto $p \leq n * t$.

Após todos os experimentos controlados realizados por este trabalho, que podem até mesmo ter ocultado comportamentos indesejados, uma nova bateria de experimentos foi realizada utilizando um gerador de carga aleatório. A intenção deste último experimento é avaliar o desempenho do balanceamento de carga, ou seja, da decisão de qual nodo irá receber a nova aplicação MPI, e verificar a existência de comportamentos indesejados. Este

experimento não avalia a habilidade das técnicas quando operando em um cluster sobrecarregado.

O gerador de carga tem como funcionalidade básica definir aleatoriamente quais nodos estarão com carga durante a execução da aplicação genoma. Uma vez definido os nodos que irão receber a carga artificial, três simulações são disparadas sequencialmente utilizando a mesma definição de carga. Das três simulações, duas executam técnicas de balanceamento de carga distintas (BAL2 e BAL3). Na terceira simulação é verificado o comportamento padrão da biblioteca LAM/MPI.

Para garantir, no caso das técnicas de balanceamento de carga, que nenhuma simulação gere interferência na subseqüente, existe um intervalo de espera suficientemente grande entre o fim de uma simulação e o início da próxima. Um intervalo de 90 segundos também acontece depois que as cargas artificiais são atribuídas, cujo objetivo é permitir que a média de uso de CPU tenha tempo de se ajustar corretamente à nova carga.

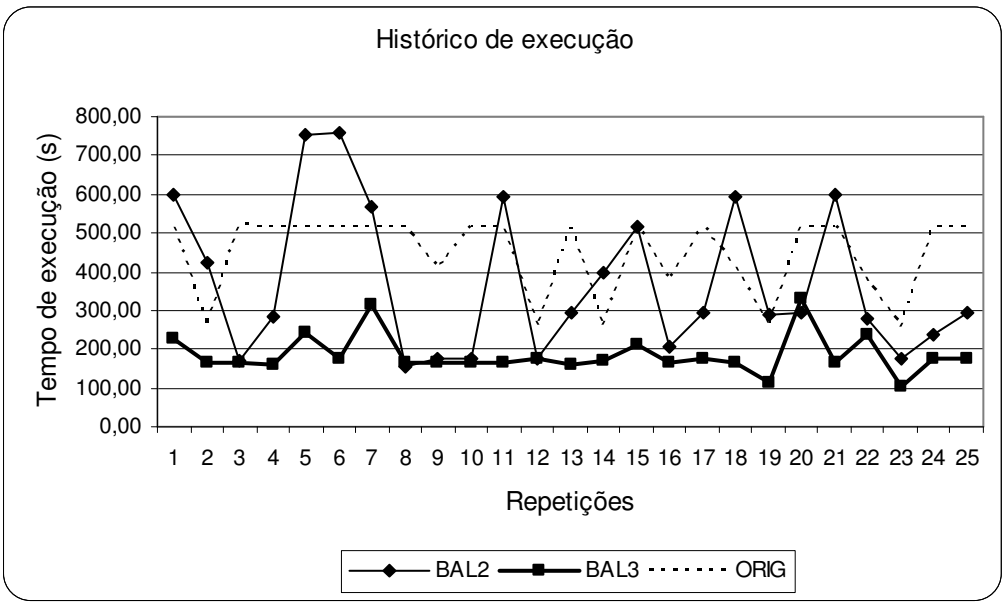

FIGURA 24: Gráfico do histórico de execução do experimento utilizando gerador de carga aleatória.

O gráfico comparativo representando o histórico de execução das duas principais técnicas propostas neste trabalho e aquele que é padrão da biblioteca LAM/MPI aparece na

FIGURA 24. Pode-se observar que a técnica de balanceamento de carga com tomada de decisão baseada no número de processos que efetivamente ocupam a CPU (BAL3) apresenta o melhor resultado na maioria dos casos.

```
para i = 7 até 0 faça
     nodo[i] = r / 2^i;
     r = r % 2^i;
```

QUADRO IV: Pseudo-algoritmo para determinar nodos que receberão carga artificial.

Para determinar o número de nodos que receberão a carga artificial, um valor (s) de 0 a 255000 é sorteado. O valor *s* é então divido por 1000 e o resto da divisão é atribuído a *r*. Logo $\{r \in \mathrm{N} \mid 0 \leq r < 255\}$. Para se conhecer quais nodos receberão carga artificial, o valor de *r* é então dividido por uma potência de 2 (2^i) como apresentado no QUADRO IV. Então um vetor, com tamanho igual ao número de nodos do cluster, armazena em cada posição um valor 0 ou 1. As posições marcadas com 1 indicam os nodos que receberão a carga artificial nesta repetição da simulação. Na FIGURA 25 é possível verificar o número de nodos com carga artificial para cada repetição da simulação.

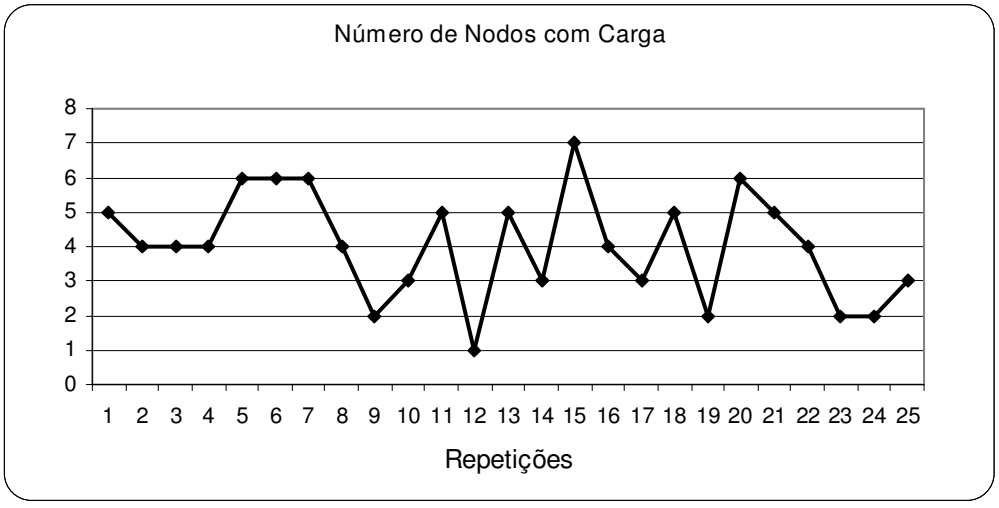

FIGURA 25: Valores obtidos pelo gerador de carga aleatória.

Também vale salientar que a técnica BAL3 em nenhum momento apresenta desempenho inferior ao comportamento padrão da biblioteca LAM/MPI. O mesmo não pode ser afirmado a respeito da técnica BAL2, isso acontece porque caso os nodos ocupados com a

carga artificial sejam as estações rápidas, toda a nova aplicação MPI será alocada para as estações lentas. No caso da técnica BAL3, as estações rápidas serão verificadas e se possível, estas também receberão os processos da nova aplicação. Essa propriedade permite que a técnica BAL3 atinja um desempenho médio 50,2 % superior, se comparada com a técnica BAL2.

Outro comportamento que deve ser destacado, e que reforça o ganho de desempenho da técnica BAL3, é a pequena variação no tempo de execução, que é melhor visualizado pelo pequeno desvio padrão mostrado na FIGURA 26. Isso significa que o ganho de desempenho não acontece apenas para alguns casos, e que a utilização do cluster heterogêneo tende a ser a melhor possível a maior parte do tempo. O desempenho da técnica BAL3 chega a ser 57% superior ao modo de distribuição padrão da biblioteca LAM/MPI. Inclusive, o desvio padrão do tempo de execução da LAM/MPI não é tão elevado, este corresponde praticamente a metade do desvio padrão da técnica BAL2, que apresenta um ganho de desempenho médio de apenas 16,7 % em relação ao ORIG.

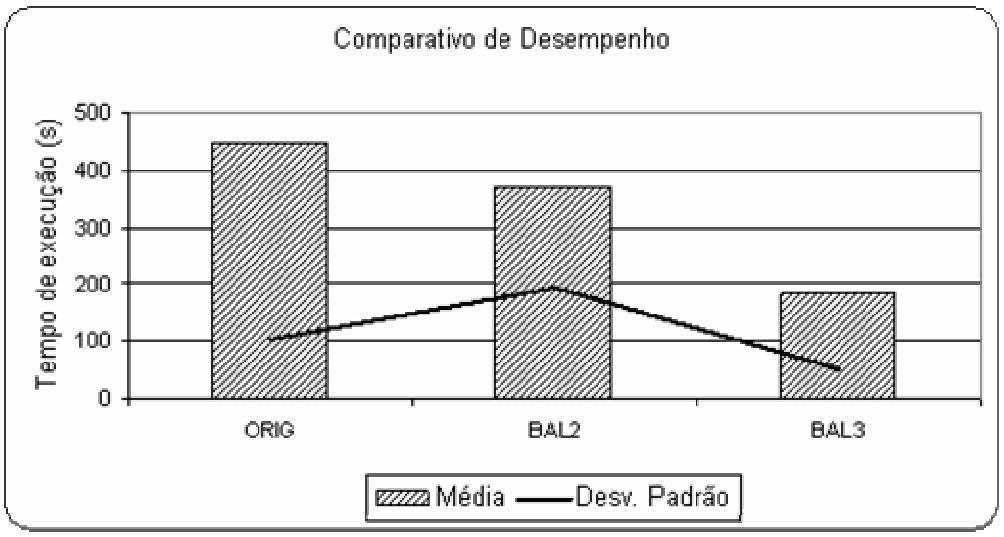

FIGURA 26: Média de desempenho e Desvio Padrão das técnicas de balanceamento de carga e distribuição padrão da LAM/MPI, no experimento de carga aleatória.

Mas ainda restam pontos que podem ser aperfeiçoados na técnica de balanceamento de carga BAL3. É possível citar um desses pontos, que é o problema de novas

aplicações surgirem em um curto espaço de tempo, mas eliminar este problema irá gerar uma sobrecarga (*overhead*) indesejada no sistema, pois seria preciso reduzir o intervalo da média de carga para 30 segundos, por exemplo, o que atualmente não é realizado pelo sistema operacional. Entretanto, é impossível mensurar quanto o desempenho irá melhorar com o aperfeiçoamento deste e outros pontos.

5.6 Propondo um Balanceamento mais Eficiente: Quarta Versão.

Em todos os experimentos realizados neste trabalho, observou-se que a carga de trabalho em um cluster pode sofrer oscilações muito variadas durante o dia, fazendo com que os valores sobre as cargas dos nodos, obtidos do sistema operacional, possam muitas vezes não representar a real situação das cargas durante a efetiva distribuição dos processos da aplicação distribuída. Este fato acontece porque a informação fornecida pelo sistema operacional se refere a uma situação já ocorrida, apesar de ser em momento muito próximo.

Entretanto, dependendo da freqüência e das rotinas dos usuários do cluster, estas oscilações podem gerar comportamentos previsíveis. Na área de arquitetura de computadores, a previsão de instruções de desvios em dois níveis (YEH & PATT, 1991, 1992) é muito usual no projeto de processadores atuais e os resultados são muito satisfatórios. Neste sentido, o presente trabalho propõe investigar o uso desta técnica na previsão da oscilação da carga de trabalho em ambientes de clusters de forma a realizar o balanceamento não somente em função da informação de carga obtida do sistema operacional, mas também em função de uma previsão sobre a oscilação da mesma.

A idéia é dividir o dia em intervalos de tempo subseqüentes e coletar continuamente as informações de carga fornecidas pelo sistema operacional nestes intervalos. O grau de discretização destes intervalos será objeto de investigação. Os valores das cargas obtidas serão classificados em função de um padrão a ser também determinado, como por exemplo, alto, médio e baixo. Os valores obtidos sobre as cargas serão mantidos em 2 tabelas

estruturadas em 2 níveis, indexadas no primeiro nível pelo histórico global corrente das variações de carga e no segundo nível pelo padrão deste histórico obtido na primeira tabela. A organização desta estrutura e a sua avaliação é tema de pesquisa futura.

6 CONCLUSÕES E TRABALHOS FUTUROS

Com a realização dos experimentos, apresentados e discutidos nos capítulos anteriores, foi possível avaliar diferentes variações de balanceamento de carga para a LAM/MPI. Com relação a uma possível sobrecarga (*overhead*) provocada pelo código adicional de obtenção e distribuição de carga, foi constatado que o mesmo é desprezível para as aplicações aqui experimentadas (menos de 0,25%), não causando nenhum impacto externo que possa prejudicar os resultados aqui relatados. Para aplicações com tempo de execução superior, o impacto é ainda menor.

Durante os experimentos também foi possível constatar o bom comportamento do nosso melhor algoritmo quando trabalhando com clusters heterogêneos, atingindo um ganho de 58,5 % de desempenho no melhor caso. Isso é possível porque a carga é distribuída de acordo com o poder computacional dos nodos. Cada versão de balanceamento de carga avaliada mostrou-se mais eficiente, especialmente por se tornar cada vez mais estável, apresentando bom desempenho em diversas situações.

O uso da extensão da biblioteca LAM/MPI em um cluster homogêneo também proverá aumento no desempenho se o cluster trabalhar em regime de sobrecarga, pois a extensão irá realizar a distribuição dos novos processos para os nodos com a menor quantidade de carga, resultando em um *throughput* maior. O balanceamento de carga oferece a possibilidade de explorar os recursos de um cluster Beowulf de forma transparente para o usuário da aplicação paralela.

Mas ainda restam pontos que podem ser aperfeiçoados. Avaliar a possibilidade de obter a média de carga de CPU em intervalos de tempo menores é um deles. Prevenir a sobrecarga do cluster quando novas aplicações são disparadas em um curto intervalo de tempo é outro. Avaliar o uso do algoritmo com uma métrica de poder computacional mais específica, como o MFLOPS, para que as aplicações com uso intenso de operações em ponto flutuante

possam alcançar maior desempenho. Acredita-se ainda, que o próprio algoritmo proposto possa ser otimizado, para os casos de realizar o balanceamento de carga com o cluster sobrecarregado, reduzindo a sua complexidade.

Outra questão que já está sendo desenvolvida é a capacidade do balanceamento de carga prever a variação de carga do sistema em função do histórico de carga anterior, tal como esboçado na seção 5.6, decidir onde alocar os processos considerando que a carga poderá se alterar. O objetivo é adequar a carga do cluster de acordo com a periodicidade e as tendências do seu uso. O algoritmo irá operar de forma semelhante à previsão de desvios em dois níveis usada nos processadores atuais.

REFERÊNCIAS BIBLIOGRÁFICAS

AGARWALA, S; POELLABAUER, C; KONG, J; et al. System-Level Resource Monitoring in High-Performance Computing Environments, **Journal of Grid Computing.** Volume 1, número 3, 2003.

ANDRESEN, D.; SCHOPF, N.; BOWKER, E.; at al. Distop: A low-overhead cluster monitoring system. **In Proceedings of the PDPTA**. Las Vegas, pág. 1832 – 1836, junho de 2003.

ATTIYA, G.; HAMAM, Y. Two phase algorithm for load balancing in heterogeneous distributed systems. **Proceedings of 12th Euromicro Conference Parallel, Distributed and Network-Based Processing.** on 11-13, pág. 434 – 439, Fevereiro de 2004.

AVERSA, L.; BESTAVROS, A. Load balancing a cluster of web servers using distributed packet rewriting. **IEEE Int'l Performance, Computing and Communication Conf**. pág. 24 – 29, 2000.

AVRESKY, D.; NATCHEV, N. Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures. **IEEE Transactions Computers**. Volume 54, número 5, pág. 603 – 615, maio de 2005.

BAKER, M. **Cluster Computing White Paper**, Final Release, Version 2.0, University of Portsmouth, UK, 2000.

BAUMGARTNER, K. M.; WAH, B. W. A Global Load Balancing Strategy for a Distributed Computer System. **Proceedings of Int. Conf. on Future Trends in Distributed Computing Systems**. Hong Kong, pág. 93 – 102, 14-16 de setembro de 1998.

BELL, G.; GRAY, J. What's next in high-performance computing? **Communications of the ACM**. Volume 45, número 2, pág. 91 – 95, fevereiro de 2002.

BEVILACQUA, A. Dynamic load balancing method on a heterogeneous cluster of workstations. **Informatica**. Volume 23, número 1, pág. 49 – 56, março de 1999.

BLAZEWICZ, J.; DELLl'OLMO, P.; DROSDOWSKI, M. Scheduling multiprocessor tasks on two parallel processors. **RAIRO Operations Research**, 36 pág. 37 – 51, 2002.

BOHN C. A.; LAMONT G. B. Asymmetric load balancing on a heterogeneous cluster of pcs. **In Proceedings of the PDPTA**. Las Vegas, volume 5, pág. 2515 – 2522, 1999.

BURNS, G.; DAOUD, R. Robust Message Delivery with Guaranteed Resources. **Proceedings, MPIDC'95**. Maio de 1995.

CARNS, P. H.; LIGON III, W. B.; MCMILLAN, S. P.; ROSS, R. B. An Evaluation of Message Passing Implementations on Beowulf Workstations. **Proceedings of the 1999 IEEE Aerospace Conference**. Março de 1999.

CASAVANT T, L; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Software Eng., 14(2):141–154, 1988.

CHAU, S. C.; FU, A. W. Load balancing between computing clusters. **Proceedings of the Fourth International Conference.** Parallel and Distributed Computing, Applications and Technologies, PDCAT'2003. pág. 548 – 551, 27-29 de Agosto de 2003.

CHIOLA, G. Some Research Projects on Clusters of Personal Computers. **24th. Proceedings of Euromicro Conference.** Volume 2, pág. XLVII – XLLIV, de 25 a 27 de Agosto de 1998.

CHOI, M.; YU, J.; KIM, H.; et al. Improving performance of a dynamic load balancing system by using number of effective tasks on Cluster Computing. **Proceedings of IEEE International Conference**. pág. 436 – 441, 2003.

DECKER, T. Virtual Data Space---A universal load balancing scheme. **In Proceedings of the 4-th International Symposium on Solving Irregularly Structured Problems in Parallel.** Volume 1253 of Lecture Notes in Computer Science, pág. 159 – 166, 1997.

DROZDOWSKI, M.; WOLNIEWICZ, P. Out-of-core divisible load processing. **IEEE Transactions on Parallel and Distributed Systems**, 14, 10, pág. 1048 – 1056, 2003.

FEITELSON, D, G. **Job scheduling in multiprogrammed parallel systems**. IBM Research Report RC 19790 (87657). Agosto de 1997.

FEITELSON, D. G. Metrics for Parallel Job Scheduling and Their Convergence. **In Revised Papers From the 7th international Workshop on Job Scheduling Strategies For Parallel Processing**. Londres, pág.188-206, 2001.

FOSTER, I. **Designing and Building Parallel Programs**. Addison-Wesley, Reading, MA, 1995.

FOX, G.; WILLIAMS, R.; MESSINA, P. **Parallel Computing Works!** San Francisco, CA: Morgan Kaufmann Publishers, 1994.

GEORGE, W. Dynamic load-balancing for data-parallel MPI programs. **In Message Passing Interface Developer's and User's Conference** (MPIDC'99). pág. 95 – 100, 1999.

HAASE, J.; ESCHMANN, F.; WALDSCHMIDT, K. The SDVM - An Approach for Future Adaptive Computer Clusters. **Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium** (IPDPS'05). Washington, volume 17, pág. 278a – 278a, de 04 a 08 de abril de 2005.

HAWICK, K. A., GROVE, D. A.; VAUGHAN, F. A. Beowulf - A New Hope for Parallel Computing? **Proc. 6th IDEA Workshop.** Rutherglen, Victoria, janeiro de 1999.

HOCKNEY, R.; BERRY, M. Public International benchmarks for parallel computers report. **PARKBENCH Committee**, Report-1, 7 de fevereiro de1994.

HOGANSON, K. E. Workload execution strategies and parallel speedup on clustered computers. **IEEE Transactions Computers**. Volume 48, número 11, pág. 1173 – 1182, novembro de 1999.

JALOTE, P. **Fault Tolerance in Distributed Systems.** New Jersey: Prentice Hall, 1994.

IBRAHIM, M. A. M.; XINDA, L. Performance of dynamic load balancing algorithm on cluster of workstations and PCs. **Proceedings of Algorithms and Architectures for Parallel Processing.** Fifth International Conference on 23-25, pág. 44 – 47, Outubro de 2002.

KACER, M.; TVRDÍK, P. Load balancing by remote execution of short processes on linux clusters. **IEEE/ACM International Symposium on Cluster Computing and the Grid**, 2002.

KOLEN, J. F.; HUTCHESON, T. A low-cost, high-density mounting system for computer clusters. **Cluster Computing Proceedings IEEE Computer Society**. Los Alamitos, volume 0, pág. 157 – 162, de 8 a 11 de outubro de 2001.

LAM/MPI Implementation – Linux man page. Disponível em <http://www.die.net/doc/linux/man/man7/libmpi.7.html>. Acesso em 09 de outubro de 2005.

LEGRAND, A.; RENARD, H.; YVES, R.; et al. Mapping and load-balancing iterative computations. **IEEE Transactions on Parallel and Distributed Systems**, 15, 6, 546 – 558, 2004.

LI, K.; CHANG, H.; YANG, C.; et al. On Construction of a Visualization Toolkit for MPI Parallel Programs in Cluster Environments. **19th International Conference on Advanced Information Networking and Applications** (AINA). Volume 2, pág. 211 – 214, de 25 a 30 de março de 2005.

Linux Online – Documentation. Disponível em <http://www.linux.org/docs/>. Acesso em 17 de agosto de 2006.

LLING, R.; MONIEN, B.; RAMME, F. A Study of Dynamic Load Balancing Algorithms. **In Proceedings of the 3rd IEEE SPDP**, pages 686–689, 1991.

LUSK, E. Programming with MPI on clusters. **Proceedings of the IEEE International Conference on Cluster Computing**. pág. 360 – 362, de 8 a 11 de outubro de 2001.

MARTINS JR, A. S.; GONÇALVES, R. A. L. ***Checkpoint* Automático de Aplicações Distribuídas em *Clusters* MPI**. Dissertação de Mestrado. Programa de Pós-Graduação em Ciência da Computação. UEM. 02/09/2005.

MEREDITH, M.; CARRIGAM, T.; BROLMAN, J.; et al. Exploring beowulf clusters. **Jornal of Computing in Small Colleges**. Volume 18, número 4, pág. 268–284, 2003.

NGUYEN, V. A. K.; PIERRE, S. Scalability of computer clusters. **Electrical and Computer Engineering**. Canadian Conference on Volume 1, pág. 405 – 409, de 13 a 16 de maio de 2001.

NGUYEN, K. N.; LE, T. T. Evaluation and comparison performance of various MPI implementations on an OSCAR Linux cluster. **Information Technology: Coding and Computing [Computers and Communications], Proceedings**. ITCC 2003. International Conference. San jose, pág. 310 – 314, 28 a 30 de Abril de 2003.

NUPAIROJ, N.; NI, L. M. Performance evaluation of some MPI implementations on workstation clusters. **Proceedings of Scalable Parallel Libraries Conference**. pág. 98 – 105, de 12 a 14 de outubro de 1994.

ONG, H; FARREL, P. A.. Performance comparison of lam/mpi, mpich, and mvich on a linux cluster connected by a gigabit ethernet network. Proceedings of the 4th Annual Linux Showcase & Conference, October 2000.

Open Source Initiative OSI. Disponível em <http://www.opensource.org/docs/definition.php>. Acesso em 07 de junho de 2005.

PFISTER, G. F. Clusters of computers for commercial processing: the invisible architecture. **IEEE Parallel & Distributed Technology**. Fall, volume 4, número 3, pág.12 – 14, 1996.

SAVVAS, I. K.; KECHADI, M. T. Dynamic task scheduling in computing cluster environments. **Parallel and Distributed Computing**. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop, pág. 372 – 379, 5 a 7 de julho de 2004.

SHEN, K.; YANG, T.; CHU, L. Cluster load balancing for fine-grain network services. **Proceedings of International Parallel & Distributed Processing Symposium**, 2002.

SILBERSCHATZ, A.; GALVIN, P.; GANE, G. **Sistemas Operacionais**. Rio de Janeiro: Campus, 2001, 1ª edição, pág. 4, 95, 122.

STERLING, T. An Introduction to PC Clusters for High Performance Computing. Conforme consta em **Cluster Computing White Paper**, Final Release, Version 2.0, Editor Mark Baker. University of Portsmouth, UK. 2000.

UNDERWOOD, K. D.; SASS, R. R.; LIGON, W. B. Cost effectiveness of an adaptable computing cluster. **Proceedings of the 2001 ACM/IEEE conference on SuperComputing**. Denver, novembro de 2001.

WILLIAMS, R. D. Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. **Jornal Concurrency**. Volume 3, pág. 457 – 481, 1991.

WONG, P.; JIN, H.; BECKER, J. Load balancing multi-zone applications on a heterogeneous cluster with multi-level parallelism. **Parallel and Distributed Computing.** Third International Symposium on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Third International Workshop, pág. 388 – 393, 5-7 de Julho de 2004.

YEH, T.; PATT, Y. N. **Two-Level Adaptive Training Branch Prediction**. The 24th ACM/IEEE International Symposium and Workshop on Microarchitecture, p. 51-61, Nov. 1991.

YEH, T.; PATT, Y. N. **Alternative Implementation of Two-Level Adaptive Branch Prediction**. The 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, p. 124-134, May. 1992.

ANEXO I – MÓDULO DE OBTENÇÃO DE CARGA

```
/*
 * mpi_libt.c usa a biblioteca libgtop (GNOME) para obter informacoes sobre
 * os recursos(numero de processos, uso de CPU, uso de memoria) da máquina.
 * Com os recursos obtidos, o programa armazena em um arquivo binario o
nodo
 * e o seu respectivo valor de recurso. Este arquivo sera utilizado pelo
 * mpirun para realizar o balanceamento de carga estatico.
 *
 * Parametros de entrada:
 * - c : medir recursos de CPU (padrao);
 * - p : medir recursos de processos;
 * - m : medir recursos de memoria;
 *
 * mpi_libt c
 *
*/

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

#include <mpi.h>

#include <glibtop/loadavg.h>
#include <glibtop/proclist.h>
#include <glibtop/mem.h>

#define FILENAME "/home/fabio/testes/load"

struct resource{
      short id;
      long value_i;
      double value_d;
      double bogo;
};


int main(int argc, char** argv){
  int     i, j;    //variaveis auxiliares
  int     pos_final;  //posicao para troca do elemento
  int     my_rank;
  int     size;    //numero de processos
  short   type;    //tipo de recurso selecionado

  short    out;    //local do arquivo de saida
  char *  log;    //nome do arquivo de log

  char line[50];    //recebe as linhas do arquivo
  char value[10];    //armazena valor recuperado bogomips
  char * find;    //flag indicando substring encontrada

  int     value_i;  //receber valor dos nodos
  double  value_d;  //receber valor dos nodos
  double  bogo;    //receber valor dos nodos
  struct resource   *values;
  struct resource   tmp;
```

```
FILE *    arq;
MPI_Status   status;
glibtop_loadavg  loadavg;  //informacoes de media de carga
glibtop_proclist   process;  //informacoes dos processos
glibtop_mem   memory;  //informacoes de memoria

char * temp;
int * t_len;

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

if (argc == 2){
  switch (argv[1][0]){
    case 'c': type=1;
    break;
    case 'p': type=2;
    break;
    case 'm': type=3;
    break;
    default:
    printf("O parametro informado nao e valido!\n");
    exit(0);
  }
}
values=(struct resource *) calloc(size, sizeof(struct resource));

if (my_rank == 0){

  printf("Iniciando processo de avaliacao de carga...\n");

  switch (type){
    case 1:
      for(i=1; i < size; i++){
        MPI_Recv(&value_d, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
        values[i].id= i;
        values[i].value_d= value_d;
      }
      values[0].id=0;
      glibtop_get_loadavg(&loadavg);
      values[0].value_d= loadavg.loadavg[0];
    break;
    case 2:
      for(i=1; i < size; i++){
        MPI_Recv(&value_i, 1, MPI_LONG, i, 0, MPI_COMM_WORLD, &status);
        values[i].id=i;
        values[i].value_i=value_i;
      }
      values[0].id=0;
      glibtop_get_proclist(&process, GLIBTOP_KERN_PROC_ALL, 0);
      values[0].value_i=process.number;
    break;
    case 3:
      for(i=1; i < size; i++){
        MPI_Recv(&value_i, 1, MPI_LONG, i, 0, MPI_COMM_WORLD, &status);
        values[i].id=i;
        values[i].value_i=value_i;
      }
      values[0].id=0;
      glibtop_get_mem(&memory);
```

```
        values[0].value_i=memory.user;
      break;
    }

    // Recebendo dados do bogomips
    for(i=1; i < size; i++){
      MPI_Recv(&value_d, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, &status);
      values[i].bogo= value_d;
    }

    arq= fopen("/proc/cpuinfo", "r");

    if (arq == NULL)
      perror("Erro abrindo o arquivo");
    else{

      while( !feof(arq) ){
        fgets(line, 50, arq);
        find= strstr(line, "bogomips");
        if (find != NULL){
          find= strchr(line, ':');
          find= find + sizeof(char);   //pula o :
          j= 0;
          for (i = find-line+1; line[i] != ' '; i++){
            value[j]= line[i];
            j++;
          }
          values[0].bogo= atof(value);
          break;
        }
      }
    }
    fclose(arq);

/*
 * Ordena os nodos em ordem decrescente de bogomips
 * e faixas com igual bogomips em ordem crescente de carga
 */
    if (type == 1){
      for (i=size-1; i > 0; i--){
        pos_final= i;
        for (j=0; j < i; j++){
          if (values[j].bogo < values[pos_final].bogo){
            pos_final= j;
          }
          else if (values[j].bogo == values[pos_final].bogo){
            if (values[j].value_d > values[pos_final].value_d){
              pos_final= j;
            }
          }
        }
        if (pos_final != i){
          tmp= values[pos_final];
          values[pos_final]= values[i];
          values[i]= tmp;
        }
      }
    }


/*
```

```
 * Gravar para arquivo, arquivo utilizado pelo mpirun
 */
    arq=fopen(FILENAME,"wb");
    for (i=0; i < size; i++){
      fwrite(&values[i], sizeof(struct resource), 1, arq);
    }
    fflush(arq);
    fclose(arq);
    free(values);
    printf("Finalizado.\n");

  } /* if (my_rank */
  else{
    switch (type){
      case 1:
        glibtop_get_loadavg(&loadavg);
        MPI_Send(&loadavg.loadavg[0], 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
      break;
      case 2:
        glibtop_get_proclist(&process, GLIBTOP_KERN_PROC_ALL, 0);
        MPI_Send(&process.number, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
      break;
      case 3:
        glibtop_get_mem(&memory);
        MPI_Send(&memory.user, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
      break;
    }

    arq= fopen("/proc/cpuinfo", "r");

    if (arq == NULL)
      perror("Erro abrindo o arquivo");
    else{

      while( !feof(arq) ){
        fgets(line, 50, arq);
        find= strstr(line, "bogomips");
        if (find != NULL){
          find= strchr(line, ':');
          find= find + sizeof(char);   //pula o :
          j= 0;
          for (i = find-line+1; line[i] != ' '; i++){
            value[j]= line[i];
            j++;
          }
          value_d= atof(value);
          break;
        }
      }
    }
    fclose(arq);
    MPI_Send(&value_d, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

ANEXO II – MODIFICAÇÕES NO PROGRAMA MPIRUN

Novos parâmetros para a execução do Balanceamento de Carga.

```
#define SIGBAL "/home/fabio/testes/sig"

  ao_setopt(ad, "cpu", 0, 0, 0);
  ao_setopt(ad, "ps", 0, 0, 0);
  ao_setopt(ad, "mem", 0, 0, 0);

  lam_ssi_base_open(ad);
  lam_ssi_base_ao_setup(ad);

  /* If we're going to run with totalview support, we need to
     initialize the totalview interface */

  lam_tv_init(argc, argv, ad);

  if (asc_compat(&argc, &argv, ad)) {
    errno_save = errno;
    sfh_argv_free(main_argv);
    ao_free(ad);
    errno = errno_save;
    perror("mpirun");
    lam_ssi_base_close();
    exit(errno_save);
  }

  if (ao_parse(ad, &argc, argv)) {
    errno_save = errno;
    show_help("mpirun", "usage", NULL);
    sfh_argv_free(argv);
    sfh_argv_free(main_argv);
    ao_free(ad);
    lam_ssi_base_close();
    exit(errno_save);
  }

  /* Check if mpirun was started with -tv option. exec
     "totalview mpirun -a ..." if this convenience argv option
     -tv was given */

  lam_tv_check(ad);

  if (ao_taken(ad, "cpu")){
      FILE *arq;
      arq=fopen(SIGBAL,"wb");
      close(arq);
  }
```

ANEXO III – MODIFICAÇÕES NA ROTINA ASC_SCHEDULE()

```
/*
 *
 *    MODIFICADA por Fabio Gorino (2005)
 *    Para realizar Balanceamento de carga.
 *
 * Copyright (c) 2001-2002 The Trustees of Indiana University.
 *                         All rights reserved.
 * Copyright (c) 1998-2001 University of Notre Dame.
 *                         All rights reserved.
 * Copyright (c) 1994-1998 The Ohio State University.
 *                         All rights reserved.
 *
 * This file is part of the LAM/MPI software package.  For license
 * information, see the LICENSE file in the top level directory of the
 * LAM/MPI source distribution.
 *
 * $HEADER$
 *    Ohio Trollius
 *    Copyright 1996 The Ohio State University
 *    GDB
 *
 *    $Id: asc_schedule.c,v 6.6 2002/10/09 20:56:58 brbarret Exp $
 *
 *    Function:   - schedules an application schema
 *                - LAM specific
 *                - generates a new application list with one
 *                  process, one node entries
 *                - four scheduling cases:
 *                  1) "foo -c # <nodes>"  # procs on this list of nodes
 *                  2) "foo -c #"          # procs using all nodes
 *                  3) "foo <nodes>"       1 proc on each of these nodes
 *                  4) "foo"               1 proc on every node
 *
 *    Accepts:    - parsed application schema
 *    Returns:    - expanded, precise application schema or NULL
 */

#include <all_list.h>
#include <app_schema.h>
#include <ndi.h>
#include <net.h>

#define FILENAME "/home/fabio/testes/load" //arquivo gerado pelo MPI_LIBT
#define SIGBAL "/home/fabio/testes/sig" //arquivo gerado pelo MPIRUN
#define MODO3

static void str_append(char **src, int *len, char *suffix);

struct resource{
      short id;
      long value_i;
      double value_d;
      double bogo;
};
```

```
LIST *            asc_schedule(applist)
LIST *            applist;

{
  LIST *          newapplist;       /* scheduled app schema */
  LIST *          newnodelist;      /* explicit node ids */
  LIST *          parsenodelist;    /* parsed nodes or default */
  struct aschema  newproc;          /* precise app process */
  struct aschema  *p;               /* current app process */
  struct ndi *    node;             /* current node ID */
  char *          nodev[3];         /* default node spec */
  int             n;

  int             i, flag;          //auxiliares
  int             n_nodes;          //Numero de nodos do cluster
  int             hiload;           //Numero de nodos com alta carga
  struct resource *values;          //Valores de carga ordenados
  short *         num_proc;         //Numero de processos atribuido a cada
nodo
  short           min_proc;         //Menor numero de processos atribuido a
um nodo

  short *         actual_load;      //Carga antes da nova aplicacao
  short           max_proc;
  float           max_val;          //Relacao poder computacional/carga

  FILE *          arq;

/*
 * Expand each entry in the parsed schema.
 */
  p = (struct aschema *) al_top(applist);
  newapplist = al_init(sizeof(struct aschema), 0);

  while (p) {
/*
 * Absent node information is replaced by all nodes.
 */
    if (al_count(p->asc_nodelist) == 0) {
      nodev[0] = "cmd";
      nodev[1] = "C";
      nodev[2] = 0;

      parsenodelist = ndi_parse(2, nodev, 0);

      if (parsenodelist == 0) {
        al_free(newapplist);
        return(0);
      }
    } else {
      parsenodelist = p->asc_nodelist;
    }
/*
 * Expand the node list into plain node identifiers.
 */
    newnodelist = ndi_resolve(parsenodelist);

    if (al_count(p->asc_nodelist) == 0) {
      al_free(parsenodelist);
    }
```

```
    if (newnodelist == 0) {
      al_free(newapplist);
      return(0);
    }
/*
 * An absent process count means one process on each node.
 */
    n = (p->asc_proc_cnt < 1) ? al_count(newnodelist) :
        p->asc_proc_cnt;

/*
 * Generate a one process, one node application schema.
 */
    node = (struct ndi *) al_top(newnodelist);
    newproc.asc_errno = 0;
    newproc.asc_proc_cnt = 1;
    newproc.asc_args = p->asc_args;
    newproc.asc_env = p->asc_env;
    newproc.asc_nodelist = 0;

/**/
/*
 * Recupera o arquivo gerado pelo mpi_libt com os valores de carga
 * dos nodos envolvidos.
 */
    /* Verifica se executa MODO BAL */
    n_nodes= 0;
    arq= fopen(SIGBAL, "r");
    if (arq != NULL){
      char *cmd =(char *) malloc(128);
      int cmd_len=128;
      char libt_num[10];

      n_nodes= getnall();
      printf("Usando indice de carga %.2f\n", INDICE);
      fclose(arq);
      remove(SIGBAL);

/*
 * Trata o argv para retirar informacoes importantes
 * para obter carga dos nodos envolvidos atualmente
 */
      cmd[0]= 0;
      str_append(&cmd, &cmd_len, "mpirun -ssi rpi tcp -np");
      sprintf(libt_num, " %d", n_nodes);

      str_append(&cmd, &cmd_len, libt_num);
      str_append(&cmd, &cmd_len, " mpi_libt c");
      system(cmd);
      free(cmd);
    }

    if (n_nodes > 0){
      arq=fopen(FILENAME,"rb");
      values=(struct resource *) calloc(n_nodes ,sizeof(struct resource));
      fseek(arq,0,SEEK_SET);
      n_nodes = 0;
      while (1){
        fread(&values[n_nodes], sizeof(struct resource), 1, arq);
        if (feof(arq)) {
          break;
```

```
        }
        n_nodes++;
      }
      num_proc=(short *) calloc(n_nodes, sizeof(short));
      fclose(arq);
      remove(FILENAME);

      actual_load = (short *) calloc(n_nodes, sizeof(short));
      for(i = 0; i < n_nodes; i++) {
        if (values[i].value_d > 0.5) {
          actual_load[i] = (short) values[i].value_d;
          actual_load[i]++; /* Arredonda para cima */
        }
        else {
          actual_load[i] = 0;
        }
      }



/**/
/*
 * Escalona os processos de arcodo com os recursos
 */
      min_proc = 0;
      max_proc = 1;
      hiload = 0;
      flag = 1;

/*********************************************************
 * Distribui processos de acordo com a carga de CPU por nodo
 * permite aplicacoes distintas diputem o mesmo nodo
 *********************************************************
*/

      while (n > 0) {
        while (1) {

         if (hiload >= n_nodes) {
           max_val = 0;
           for (i = 0; i < n_nodes; i++) {
             values[i].value_d = values[i].bogo /
               (actual_load[i] + num_proc[i]);
             if (values[i].value_d > max_val) {
               max_val = values[i].value_d;
               min_proc = i;
             }
           }
           i = min_proc;
           break;
         }
         else { /* (1) */
           /* Primeiro os nodos ociosos */
           if ((num_proc[i] == 0) && (actual_load[i] == 0)) {
              hiload = 0;
              break;
           }
           else { /* (2) */
             if (values[i].bogo > 4000) {
               /* disputa 50% com carga */
               if ((num_proc[i] == 0) && (actual_load[i] == 1)) {
```

```
                  hiload = 0;
                  break;
                }
                else if ((num_proc[i] < max_proc)&&(actual_load[i] == 0)) {
                  hiload = 0;
                  break;
                }
                else {
                  hiload++;
                }
              }
              else {
                if ((num_proc[i] < max_proc) && (actual_load[i] == 0)) {
                  hiload = 0;
                  break;
                }
                else {
                  hiload++;
                }
              }
            } /* else (2) */
          } /* else (1) */

          i++;
          if (i >= n_nodes) {
            i = 0;
            max_proc++;
          }
        }
        num_proc[i]++; /* processo atribuido */

        node= (struct ndi *) al_top(newnodelist);
        while (node->ndi_node != values[i].id){
          node= (struct ndi *) al_next(newnodelist, node);
        }

        newproc.asc_node = node->ndi_node;
        newproc.asc_srcnode = (p->asc_srcnode == -1) ?
                              node->ndi_node : p->asc_srcnode;
        newproc.asc_args->apa_refcount += 1;
        newproc.asc_env->ape_refcount += 1;

        if (al_append(newapplist, &newproc) == 0) {
           al_free(newapplist);
           al_free(newnodelist);
           return(0);
        }
        n--;
      }
      free(values);
      free(num_proc);
      free(actual_load);
    } /* if (n_nodes > 0 */
/*
 * Distribui ORIG
 */
    else{
      while (n > 0) {

        newproc.asc_node = node->ndi_node;
        newproc.asc_srcnode = (p->asc_srcnode == -1) ?
```

```
          node->ndi_node : p->asc_srcnode;
        newproc.asc_args->apa_refcount += 1;
        newproc.asc_env->ape_refcount += 1;

        if (al_append(newapplist, &newproc) == 0) {
          al_free(newapplist);
          al_free(newnodelist);
          return(0);
        }

        node= (struct ndi *) al_next(newnodelist, node);

        if (node == 0) {
              node = (struct ndi *) al_top(newnodelist);
        }

        n--;
      }
    }

    al_free(newnodelist);
    p = al_next(applist, p);
  }
  p= (struct aschema *) al_top(newapplist);
  while (p){
    p= (struct aschema *) al_next(newapplist, p);
  }
  return(newapplist);
}

/*
 * Copia do mpirun.c
 */
static void
str_append(char **src, int *len, char *suffix)
{
  int slen1;
  int slen2;

  slen1 = strlen(*src);
  slen2 = strlen(suffix);

  if (slen1 + slen2 > *len) {
    while (slen1 + slen2 > *len)
      *len *= 2;
    *src = realloc(*src, *len);
  }

  strcat(*src, suffix);
}
```