ISSN 0101-7438

Seção de *Software*

Virgílio José Martins Ferreira Filho
Departamento de Engenharia Industrial
Universidade Federal do Rio de Janeiro
Rio de Janeiro – RJ
virgilio@ufrj.br

MPI: UMA FERRAMENTA PARA IMPLEMENTAÇÃO PARALELA

Aníbal Alberto Vilcapona Ignácio
Programa de Engenharia de Produção/COPPE
Universidade Federal do Rio de Janeiro
Rio de Janeiro – RJ
E-mail: avilcap@pep.ufrj.br

Virgílio José Martins Ferreira Filho *
Programa de Engenharia de Produção/COPPE e
Departamento de Engenharia Industrial
Universidade Federal do Rio de Janeiro
Rio de Janeiro – RJ
E-mail: virgilio@ufrj.br ou virgilio@pep.ufrj.br

* *Corresponding author*/autor para quem as correspondências devem ser encaminhadas

1. Introdução

Muitos problemas interessantes de otimização não podem ser resolvidos de forma exata, utilizando a computação convencional (seqüencial) dentro de um tempo razoável, inviabilizando sua utilização em muitas aplicações reais na engenharia e na indústria. Embora os computadores estejam cada vez mais velozes, existem limites físicos e a velocidade dos circuitos não pode continuar melhorando indefinidamente. Por outro lado nos últimos anos tem-se observado uma crescente aceitação e uso de implementações paralelas nas aplicações de alto desempenho como também nas de propósito geral, motivados pelo surgimento de novas arquiteturas que integram dezenas de processadores rápidos e de baixo custo. Essas arquiteturas compõem ambientes com memória distribuída como, por exemplo, estações de trabalho ou PCs conectados em rede.

Segundo Tanenbaum (1999), os computadores paralelos podem ser divididos em duas categorias principais SIMD (*Simple Instruction Multiple Data*) e MIMD (*Multiple Instructions Multiple Data*). As máquinas SIMD executam uma instrução de cada vez, sobre

diversos conjuntos de dados; nessa categoria estão as máquinas vetoriais e as matriciais. As máquinas MIMD rodam programas diferentes, em processadores diferentes, e podem ser divididas em multiprocessadores que compartilham a memória principal e os multicomputadores que não compartilham nenhuma memória. Os multicomputadores podem ser divididos em máquinas MPPs (*Massive Parallel Processor*) e COWs (*Cluster of Workstations*). Os MPPs são os supercomputadores que utilizam processadores padrão como o IBM RS/6000, a família Dec Alpha ou a linha Sun UltraSPARC. Uma outra característica dos MPPs é o uso de redes de interconexão proprietárias, de alta performance, baixa latência e banda passante alta.

Um COW é composto de algumas centenas de PCs ou estações de trabalho, ligados por intermédio de uma rede comercial. Este ambiente paralelo já é uma realidade em empresas e universidades, tornando-se mais acessível que outros computadores paralelos de alto custo comercial. Deng & Korobka (2001) apresentaram um sistema COW chamado Galaxi, implementado sobre uma rede de alta velocidade (Fast e Gigabit Ethernet, com velocidades superiores a 100 MBPS). Nesse ano, a NTT Data Corp testará uma super-rede de computadores com a Intel, a Silicon Graphics (SGI), a Nippon Telegraph e a Telephone East Corp. (NTT East), envolvendo cerca de 1 milhão de PCs, com o intuito de criar um supercomputador virtual com capacidade de processamento cinco vezes maior que o mais rápido computador (IDGNow (2001)).

Além da rede de comunicação, é necessário uma camada de *software* que possa gerenciar o uso paralelo destas máquinas. Para tanto existem bibliotecas especializadas para tratamento da comunicação entre processos e a sincronização de processos concorrentes (Martins *et al.* (2002), Ignacio *et al.* (2000)). De uma forma geral, as bibliotecas são utilizadas sem maior dificuldade tanto nas máquinas MPP quanto nas máquinas COW, de maneira que as aplicações podem ser transferidas entre ambas plataformas. Os dois sistemas baseados na troca de mensagens mais usados em multicomputadores são o MPI (*Message-Passing Interface*) e PVM (*Parallel Virtual Machine*). O PVM é um sistema de mensagens de domínio público, projetado inicialmente para rodar em máquinas COW, tendo diversas modificações implementadas para rodar em máquinas MPP. A seção de software de Pesquisa Operacional (vide Souza *et al.* (1998)) já trouxe um artigo sobre o PVM (para maiores informações ver Geist *et al.* (1994)). Neste artigo é apresentado o MPI, que oferece mais recursos que o PVM, com mais opções e mais parâmetros por chamada, e que vem se tornando o padrão das implementações paralelas no MPP e no COW.

Um dos principais objetivos do desenvolvimento de aplicações paralelas é a redução do tempo computacional. Contudo não se deve buscar simplesmente otimizar uma simples medida de aceleração (razão entre o tempo do programa seqüencial e o tempo de execução da versão paralela). Devem ser também consideradas outras medidas como: eficiência e escalabilidade. A eficiência é usada para medir a qualidade de um algoritmo paralelo e é definido como a fração do tempo em que os processadores estão ativos (razão entre a aceleração e o número de processadores) caracterizando a utilização dos recursos computacionais, independentemente do tamanho do problema. A escalabilidade é uma medida de desempenho que indica a variação do tempo de execução e da aceleração, com o acréscimo do número de processadores e/ou tamanho do problema.

O presente trabalho apresenta as principais características de uma padronização de interface para troca de mensagens MPI, bem como as implementações existentes e suas aplicações como ferramenta de solução nos problemas da área de Pesquisa Operacional.

2. *Message Passing Interface* (MPI)

Segundo Gropp *et al.* (1994), Foster (1995) e Pacheco (1999), o MPI é um padrão de interface para a troca de mensagens em máquinas paralelas com memória distribuída e não se devendo confundi-lo com um compilador ou um produto específico.

Antes de se mostrar as características básicas do MPI, é apresentado um breve histórico do surgimento do mesmo.

2.1 Histórico do MPI

O MPI é o resultado do esforço de aproximadamente 60 pessoas, pertencentes a 40 instituições, principalmente dos Estados Unidos e Europa. A maioria dos fabricantes de computadores paralelos participou, de alguma forma, da elaboração do MPI, juntamente com pesquisadores de universidades, laboratórios e autoridades governamentais. O início do processo de padronização aconteceu no seminário sobre Padronização para Troca de Mensagens em ambiente de memória distribuída, realizado pelo *Center for Research on Parallel Computing*, em abril de 1992. Nesse seminário, as ferramentas básicas para uma padronização de troca de mensagens foram discutidas e foi estabelecido um grupo de trabalho para dar continuidade à padronização. O desenho preliminar, foi realizado por Dongarra, Hempel, Hey e Walker, em novembro 1992, sendo a versão revisada finalizada em fevereiro de 1993 (pode ser obtido em ftp://netlib2.cs.utk.edu/mpi/mpi1.ps).

Em novembro de 1992 foi decidido colocar o processo de padronização numa base mais formal, adotando-se o procedimento e a organização do HPF (the High Performance Fortran Forum). O projeto do MPI padrão foi apresentado na conferência *Supercomputing* 93, realizada em novembro de 1993, do qual se originou a versão oficial do MPI (5 de maio de 1994). Essa versão em *postscript* pode ser obtida na Netlib, enviando um e-mail a netlib@ornl.gov com a mensagem "*send mpi-report.ps from mpi*" (também pode ser obtida em ftp://netlib2.cs.utk.edu/mpi/mpi-report.ps).

Ao final do encontro do MPI-1 (1994) foi decidido que se deveria esperar mais experiências práticas com o MPI. A sessão do Forum-MPIF de *Supercomputing* 94 possibilitou a criação do MPI-2, que teve inicio em abril de 1995. No *SuperComputing'96* foi apresentada a versão preliminar do MPI-2. Em abril de 1997 o documento MPI-2 foi unanimemente votado e aceito. Este documento está disponível via HTML em http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html ou em *postscript* em http://www.mpi-forum.org/docs/mpi-20.ps.

2.2 MPI

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas. Por isso, o padrão MPI é algumas vezes referido como MPMD (*multiple program multiple data*).

Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. Dado o fato do número de processos no

MPI ser normalmente fixo, neste texto é enfocado o mecanismo usado para comunicação de dados entre processos. Os processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro). Um grupo de processos pode invocar operações coletivas (*collective*) de comunicação para executar operações globais. O MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*) que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

Os algoritmos que criam um processo para cada processador podem ser implementados, diretamente, utilizando-se comunicação ponto a ponto ou coletivas. Os algoritmos que implementam a criação de tarefas dinâmicas ou que garantem a execução concorrente de muitas tarefas, num único processador, precisam de um refinamento nas implementações com o MPI.

MPI - Básico

Embora o MPI seja um sistema complexo, um amplo conjunto de problemas pode ser resolvido usando-se apenas 6 funções, que servem basicamente para: iniciar, terminar, executar e identificar processos, enviando e recebendo mensagens.

`MPI_INIT`	Inicia uma execução MPI
`MPI_FINALIZE`	Finaliza a execução
`MPI_COMM_SIZE`	Determina o número de processos
`MPI_COMM_RANK`	Determina a identificação de processos
`MPI_SEND`	Envia a mensagens
`MPI_RECV`	Receber mensagens

Todos os procedimentos acima, exceto os dois primeiros, possuem um manipulador de comunicação como argumento. Esse manipulador identifica o grupo de processos e o contexto das operações que serão executadas. Os comunicadores proporcionam o mecanismo para identificar um subconjunto de processos, durante o desenvolvimento de programas modulares, assegurando que as mensagens, planejadas para diferentes propósitos, não sejam confundidas.

A seguir é apresentado um exemplo de um programa paralelo. Esse programa busca no espaço de soluções viáveis a melhor solução, segundo uma função objetivo. O ambiente paralelo tem 5 processos. O processo 0 é encarregado de controlar a busca. No inicio, ele faz uma amostragem do espaço de soluções viáveis para encontrar 4 diferentes soluções iniciais que são enviadas aos processos 1,2,3,4 utilizando-se o procedimento MPI_SEND. Esses processos, ao receber a mensagem, iniciam o processo de busca. Quando os processos de busca atingem seu critério de parada eles enviam o resultado ao processo 0, utilizando a função MPI_GATHER. O processo 0 então avalia a melhor solução encontrada para enviar essa nova solução a todos os outros processo utilizando, a função MPI_BCAST e se inicia um novo processo de busca, a partir de uma solução corrente melhor.

```
# Include<mpi.h>
main ()
        MPI_Status status;
        MPI_Init();
        MPI_Comm_rank(...,&my_rank);
        MPI_Comm_size(...,&p);
        If(my_rank=0)
                Amostragem_do_Espaço_de_Soluções
        for(source=1;source<p )
                docase(source)
                        case ´1´:  MPI_Send (solução_inicial_1,...)
                        case ´2´:  MPI_Send (solução_inicial_2,...)
                        case ´3´:  MPI_Send (solução_inicial_3,...)
                        case ´4´:  MPI_Send (solução_inicial_4,...)
else
        for(source=1;source<p)
                docase(source)
                        case ´1´:  MPI_Recv (solução_inicial_1,...);
                                   Busca_1(solução_inical_1);
                                   MPI_Gather(...,melhor_solução_1,...,0,...);
                        case ´2´:  MPI_Recv (solução_inicial_2,...);
                                   Busca_2(solução_inicial_2);
                                   MPI_Gather(...,melhor_solução_2,...,0,...);
                        case ´3´:  MPI_Recv (solução_inicial_3,...);
                                   Busca_3(solução_inicial_3);
                                   MPI_Gather(...,melhor_solução_3,...,0,...)
                        case ´4´:  MPI_Recv (solução_inicial_4,...);
                                   Busca_4 (solução_inicial_4);
                                   MPI_Gather(...,melhor_solução_4,...,0,...);

/* Inicia processo iterativo entre o processo 0 e os processo 1,2,3,4

While (Condição_de_Parada)
        If(my_rank=0)
                        Escolha_ Melhor_ Solução;
                        MPI_Bcast(...,melhor_solução_corrente,...,0,..);
        Else
                for(source=1;source<p)
                docase(source)
                        case'1':   Busca_1(melhor_solução_corrente);
                                   MPI_Gather(....,melhor_solução_1,...,0,...);
                        case ´2´:  Busca_2(melhor_solução_corrente);
                                   MPI_Gather(....,melhor_solução_2,...,0,...);
                        case ´3´:  Busca_3(melhor_solução_corrente);
                                   MPI_GATHER(...,melhor_solução_3,...,0,...);
                        case ´4´:  Busca_4 (melhor_solução_corrente);
                                   MPI_GATHER(...,melhor_solução_4,...,0,...);
MPI_Finalize();
```

As funções MPI_INIT e MPI_FINALIZE são usadas, respectivamente, para iniciar e finalizar uma execução MPI. A MPI_INIT deve ser chamada antes de qualquer função MPI e deve ser acionada por cada processador. Depois de ser acionada a MPI_FINALIZE, não se pode acessar outras funções da biblioteca.

A função MPI_COMM_SIZE determina o número de processos da execução corrente e a MPI_COMM_RANK os identifica, usando um número inteiro. Os processos, pertencentes a um grupo, são identificados com um número inteiro precedido de um 0. As funções MPI_SEND e MPI_RECV são usadas para enviar e receber mensagens.
Antes de proceder com aspectos mais sofisticados do MPI, é importante assinalar que a programação de troca de mensagens não é determinística ou seja, a chegada das mensagens enviadas por dois processos A e B ao processo C não é definida. Isto é, o MPI não garante que uma mensagem, enviada de um processo A e de um processo B, chegue na ordem que foi enviada. A responsabilidade disso depende do programador que deve assegurar a execução determinística, quando for solicitado.

A especificação da origem (*source*), no MPI_RECV, pode permitir o recebimento de mensagens vindas de um único processo específico, ou de qualquer processo. Esta segunda opção permite receber dados que tenha qualquer origem e isso algumas vezes é útil. Porém, a primeira opção é preferível porque as mensagens chegam na ordem do tempo em que foram enviadas.

Um mecanismo adicional para distinguir as mensagens é através do uso do parâmetro *tag*. O processo de envio deve associar uma *tag* (inteiro) a uma mensagem. O processo de recepção pode especificar que deseja receber mensagens com um *tag* especifico ou com qualquer *anytag*. O uso de um *tag* é preferível, devido à redução da possibilidade de erros.

Operações Globais

Os algoritmos paralelos, freqüentemente, precisam de operações coordenadas envolvendo múltiplos processos. Por exemplo, todos os processos podem precisar transpor uma matriz distribuída ou somar um conjunto de números distribuídos em cada um dos processos. Claramente, as operações podem ser implementadas por um programador, utilizando-se funções de recebimento e envio de funções. Por comodidade e para permitir implementações otimizadas, o MPI também fornece uma serie de funções especializadas de comunicação que executam operações comuns desse tipo.

As funções são:

- Barreira (*Barrier*): sincroniza todos os processos.
- Difusão (*Broadcast*): envia dado de um processo a todos os demais processos.
- Juntar (*Gather*): junta dados de todos os processos em um único processador.
- Espalhar (*Scatters*): distribui um conjunto de dados de um processo para todos os processos.
- Operação de redução (*Reduction operations*): soma, multiplicação, etc., de dados distribuídos.

A função MPI_BARRIER( ) é usada para sincronizar a execução de processos de grupo. Nenhum processo pode realizar qualquer instrução, até que todos os processos tenham

passado por essa barreira. Essa é uma maneira simples de separar as duas etapas da execução para assegurar que as mensagens geradas não se misturem. Por exemplo, a função MPI_BARRIER pode ser inserida, antes de uma segunda operação de envio, para garantir um determinismo na execução. É evidente, nesse exemplo e em outros, que a necessidade de barreiras explícitas pode ser evitada pelo uso de *tag*, de origem e/ou contextos específicos.

As funções MPI_BCAST( ), MPI_GATHER( ), e MPI_SCATTER( ) são rotinas coletivas de movimentação de dados, nas quais todos os processos interagem com origens distintas para espalhar, juntar e distribuir os dados, respectivamente. Em cada caso, os primeiros três argumentos especificam o local e o tipo de dados a serem comunicados e o número de elementos a serem enviados para cada destino. Outros argumentos especificam o local, o tipo de resultado e o número de elementos a serem recebidos de cada origem.

A função MPI_BCAST( ) implementa um processo de dispersão de dados do tipo um para todos, no qual um único processo origem envia um dado para todos os outros processos e cada processo recebe esse dado.

A função MPI_GATHER( ) implementa um processo de junção dos dados de todos os processos em um único processo. Todos os processos, incluindo a origem, enviam dados localizados na origem. Esse processo coloca os dados em locais contíguos e não opostos, com dados do processo "i", precedendo os dados do processo "i+l".

A função MPI_SCATTER( ) implementa uma operação de dispersão de um conjunto de dados para todos os processos; isto é, o reverso da MPI_GATHER. Note a sutil diferença entre essa função e a MPI_BCAST: enquanto na MPI_BCAST, todo processo recebe o *mesmo* valor do processo de origem, no MPI_SCATTER, todo processo recebe um valor *diferente.*

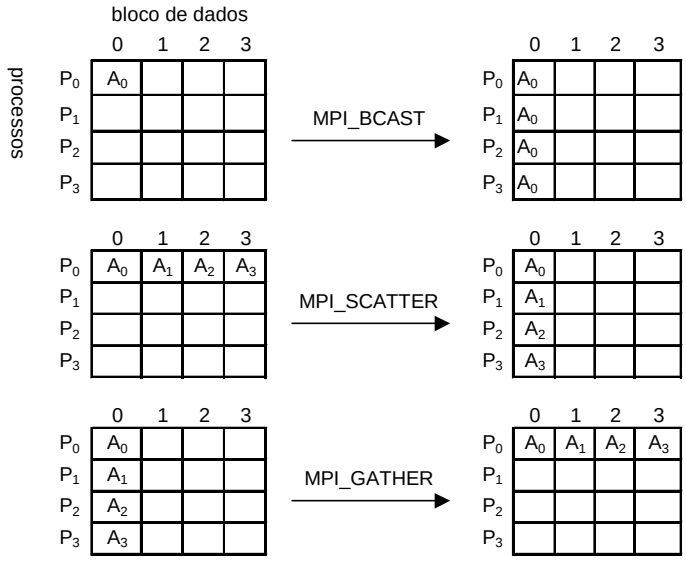

Figura 1 – Operações Globais

A MPI também fornece operações de redução. As funções MPI_REDUCE( ) e MPI_ALLREDUCE( ) implementam operações de redução. Eles combinam valores fornecidos no *bufer* de entrada de cada processo, usando operações especificas, e retorna o valor combinado, ou para o *bufer* de saída de um único processo da origem (MPI_REDUCE) ou para os *bufer* de saída de todos os processo (MPI_ALLREDUCE). Essas operações incluem o máximo, o mínimo (MPI_MAX, MPI_MIN), soma, produto (MPI_SUM, MPIPROD) e as operações lógicas.

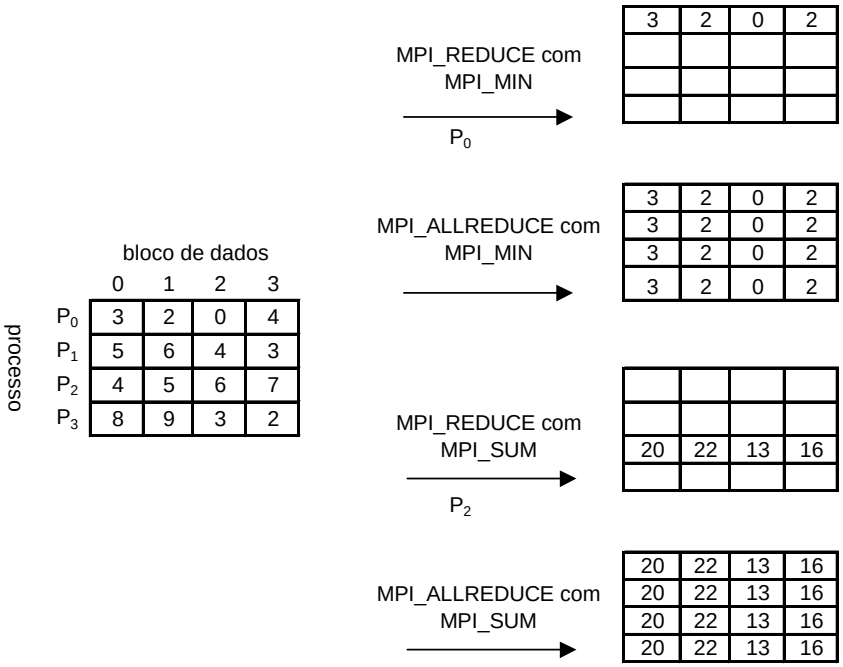

Figura 2 – Operações de Redução

Existem muitas outras funções MPI, que não foram descritas, neste texto que objetiva dar os princípios básicos de uma implementação. Algumas características não cobertas são: a possibilidade de comunicação assíncrona, quando um cálculo acessa elementos de uma estrutura de dados compartilhados em um modo não estruturado; a composição modular usada quando são implementados programas grandes de sistemas complexos onde módulos são implementados independentemente; o uso de ferramentas que ajudam a melhorar a performance dos algoritmos, investigando o ambiente dentro do qual estão sendo executados; o uso de diferentes tipos de dados, usados por exemplo para fazer a conversão entre diferentes representações de dados em ambientes heterogêneos.

Também é importante assinalar as diferenças do MPI para o MPI-2, dada pela incorporação de gerenciamento dinâmico de processos e na manipulação de *Threads*, o que sempre foi um problema para as bibliotecas de troca de mensagens. No MPI-2 tem-se a possibilidade de restringir o acesso à comunicação de algumas *Threads*, como por exemplo, apenas uma *Thread* (a principal) pode executar as funções do MPI.

Compilação e Execução

Uma vez escritos os programas usando as funções do MPI é necessário compilá-los e executá-los. Os métodos de compilação e execução dependem da implementação utilizada, mas, de forma bastante genérica, os principais passos para compilar e executar um programa em MPI são:

1) Utilizar compiladores que reconheçam automaticamente os procedimentos MPI, ou incluir manualmente as bibliotecas MPI no processo de compilação;
2) Verificar quais são os nós disponíveis no ambiente paralelo;
3) Fixar os parâmetros para o ambiente paralelo: por exemplo, escolher os nós a serem usados e o protocolo de comunicação;
4) Executar o programa (que deverá ser executado em cada um dos nós escolhidos).

3. Implementações de MPI

Muitas implementações de MPI foram desenvolvidas. Todas elas foram implementadas sobre uma máquina virtual formada por muitos computadores heterogêneos (*workstations*, PCs, Multiprocessadores), em cada um dos quais se executa um processo, usando-se um controle de mensagem entre essas máquinas. Entre essas implementações, o MPICH foi desenvolvido pela Argonne National Laboratory, utilizando-se MPI 1.1. padrão. Esta implementação está disponível e pode ser obtido no endereço http://www-unix.mcs.anl.gov/mpi/mpich. Esta ferramenta trabalha sobre as seguintes máquinas paralelas: IB SP-2, Intel Paragon, Gray T3D, Meiko CS-2, TC C-5, NCUBE NCUBE-2, Convex Exemplar, SGI Challenge, Sun Multiprocessors, CRAY YMP e CRAY C-90. Pode também ser executado sobre PCs (utilizando Linux, freeBSD ou Windows), Sun, HPO, SGI, IB RS/6000 e DEC workstations. MPICH/NT é uma outra implementação para Windows NT e está disponível pela Mississippi State University. Essa versão suporta comunicação com memória compartilhada entre estações e comunicação TCP/IP entre processos de múltiplas estações executando concorrentemente. Outra implementação disponível de MPI é o LAM que pode ser obtida no endereço http://www.mpi.nd.edu/lam, baseada no MPI 2.0 padrão e desenvolvida pelo laboratório de Computação Cientifica da Universidade de Notre Dame. Essa implementação fornece algumas funções de monitoramento, as quais são muito úteis para administrar processos. Esse software tem sido testado e verificado sobre o Solaris, IRIX, AIX, Linux e HP-UX. O MPI/Pro e um software comercializado por MPI Software Technology que pode ser executados sobre Windows NT, TCP/IP, SMP e VIA. O PaTENT MPI 4.0 é um software comercializado pela empresa Genias para cluster de PC's (intel x86 Win32).

Outras companhias têm desenvolvido implementações específicas para máquinas paralelas, tais como Alpha, Cray, IB, SGI, DEC, e HP. O Cornell Theory Center fornece muitos tutoriais para o MPI, no endereço http://www.tc.cornell.edu/UserDoc/SP/.

4. Uso do MPI em Problemas de Pesquisa Operacional

Existem muitas implementações paralelas de algoritmos exatos e metaheurísticas para se resolver problemas de otimização, utilizando-se o MPI. Algumas delas são apresentadas a seguir:

Ribeiro *et al.* (2002) desenvolvem uma heurística GRASP paralela para o problema de projeto de redes de dois caminhos (*2 - path network design problem*) que consiste em encontrar dois caminhos entre pares de pontos origem-destino. Aplicações desse tipo de problema podem ser encontradas em projetos de redes de computadores, nos quais os caminhos com poucos arcos são buscados a fim de forçar uma alta confiabilidade e maior rapidez. Os resultados computacionais mostram a eficiência do algoritmo paralelo e uma aceleração linear para até 32 processadores.

Sarma & Adeli (2001) apresentam uma heurística de algoritmos genéticos, na otimização de estruturas de aço da construção civil, sujeita às normas americanas de construção com aço. Utilizam dois esquemas inovadores na implementação paralela, com respeito à evolução dos indivíduos nos processadores e aos mecanismos de migração, cujos resultados computacionais apresentam acelerações eficientes.

Marco & Lanteri (2000) apresentam dois níveis de estratégia para a paralelização de um Algoritmo Genético, aplicados a problemas de desenho ótimo de formas que minimizem o choque aerodinâmico. Os algoritmos são executados no sistema SGI Origin 2000, IBM SP-2 e um cluster de PCs Pentium pro interconectados, através de uma rede FastEthernet de 100 Mbits/s.

Morales *et al.* (2000) desenvolvem três algoritmos paralelos para um problema simples de alocação, utilizando uma simples topologia de anel para os processadores, a fim de tornar o algoritmo portável. Os resultados computacionais mostram que o algoritmo fornece a solução ótima dos problemas.

Batoukov & Serevik (1999) apresentam um esquema geral para o algoritmo *branch-and-bound*, utilizando balanceamento de carga dinâmico. Na implementação utilizam a capacidade ociosa de uma rede de computadores para formar um supercomputador virtual. Os autores ressaltam a eficiência desse algoritmo pela baixa taxa de comunicação entre os processos e a estrutura de dados utilizada. Resultados computacionais são apresentados para diversas instâncias do problema da mochila.

Marzetta *et al.* (1999) desenvolvem uma biblioteca orientada a objetos baseada no C++ e no MPI, para resolver diversos problemas de otimização combinatória como planejamento de vôos nas empresas de linha aéreas, considerando alocação da frota com janela de tempo o algoritmo heurístico proposto para este problema encontra a solução com uma qualidade de 5.8% (limite superior – limite inferior/limite inferior * 100%).

Martins *et al.* (1998, 2000) desenvolvem uma implementação paralela de GRASP para resolver problemas de Steiner em grafos. O balanceamento de carga é usado para maximizar o uso das capacidades dos processadores, num entorno heterogêneo de clusters de estações, e os resultados computacionais mostram que, de 40 problemas testados, a heurística paralela encontra a solução ótima em 33 deles.

Homer (1997) desenvolve e implementa um algoritmo paralelo para o problema de corte de peso máximo em um grafo não direcionado. A implementação é iniciada com o algoritmo serial de Goemans e Williamson, a partir do qual diferentes versões são implementadas, variando-se a parte de pontos-interiores do algoritmo.

Eckstein *et al.* (1997) desenvolvem um conjunto de bibliotecas para a implementação paralela do algoritmo *branch-and-bound*.

Brüngger *et al.* (1997) mostram o uso da biblioteca para resolver algumas instâncias do problema quadrático de alocação.

Fachat & Hoffmann (1997) descrevem a implementação de conjunto de algoritmos de *Simulated Annealing* em paralelo e que utilizam balanceamento de carga dinâmico com o intuito de fazer uso da máxima potência de processamento.

5. Conclusões

Nos dias de hoje o conceito de máquinas paralelas formadas por cluster de PCs ou estações de trabalho (COW) já é uma realidade e podem ser encontradas em qualquer instituição por menor que seja. As melhoras significativas no desempenho, na confiabilidade e na alta velocidade de transmissão as tornam uma alternativa barata para aplicações paralelas junto ao desenvolvimento de bibliotecas para trocas de mensagem como o MPI. O MPI tem se tornado um padrão de comunicação para troca de mensagens, permitindo que cada fabricante implemente, da melhor maneira possível, a exploração dos recursos específicos de sua arquitetura. Isso possibilita o desenvolvimento de ferramentas eficientes e portáveis para diversos sistemas comerciais e de domínio público como: Windows, NT, AIX, Linux freeBSD etc. Os sistemas COW e MPP, juntamente com o MPI, estão sendo usados para resolver, com êxito, problemas desafiadoras na área da Pesquisa Operacional que anteriormente não podiam ser tratados, mas se tornaram viáveis quando olhados do ponto de vista da programação paralela.

Independentemente da qualidade do algoritmo paralelo usado, alguns aspectos da implementação podem conduzir a resultados pobres, como por exemplo gargalos na plataforma paralela ou a escolha inadequada do ambiente paralelo COW. Também deve-se ter em conta que o ganho de uma implementação paralela em geral é mais notório em grandes instâncias de problemas difíceis, podendo até mesmo ser desfavorável em problemas "fáceis".

Referências Bibliográficas

(1) Batoukov, R. & Serevik, T. (1999). A generic parallel branch and bound environment on a network of workstations. *Proceedings of High Performance Computing on Hewlett-Packard Systems*, 474-483.

(2) Brüngger, A.; Marzetta, A.; Clausen, J. & Perregaard, M. (1997). Joining forces in solving large-scale quadratic assignment problems in parallel. *Proceedings of the 11th International Parallel Processing Symposium*, 418-427.

(3) Deng, Y.F. & Korobka, A. (2001). The performance of a supercomputer built with commodity components. *Parallel Computing*, **27**, 91-108.

(4) Eckstein, J.; Hart, W.E. & Philips, C.A. (1997). An adaptable parallel toolbox for branching algorithms. *XVI International Symposium on Mathematical Programming*, Lausanne, p.82.

(5) Fachat, A. & Hoffman, K.H. (1997). Implementation of ensemble-based simulated annealing with dynamic load balancing under MPI. *Computer Physics Communications* **107**, 49-53.

(6) Foster, I. (1995). *Designing and building parallel programs: Concepts and tools for parallel software engineering.* Addison-Wesley.

(7) Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R. & Sunderman, V. (1994). *PVM: Parallel Virtua Machine - A user's guide and tutorial for networked parallel computing*. MIT Press.

(8) Gropp, W.; Lusk, E. & Skjellum, A. (1994). *Using MPI: Portable Parallel Programming with the Message Passing-Interface.* MIT Press.

(9) Homer, S. (1997). Design and performance of parallel and distributed approximation algorithms for maxcut. *Journal of Parallel and Distributed Computing*, **41**, 48-61.

(10) Ignacio, A.A.V.; Ferreira Filho, V.J.M. & Galvão, R.D. (2000). Métodos Heurísticos Num Entorno Paralelo. *Anais do XXXII Simpósio Brasileiro de Pesquisa Operacional*, 769-788.

(11) IDGNow. (2001). Intel, NTT e Silicon Graphics farão super-rede com 1 mi de PCs, 30/12/2001. <http://idgnow.terra.com.br/idgnow/pcnews/2001/11/0086>.

(12) Marco, N. & Lanteri, S. (2000). A two-level parallelization strategy for Genetic Algorithms applied to optimum shape design. *Paralell Computing*, **26**, 377-397.

(13) Martins, S.L.; Ribeiro, C.C. & Rodriguez, N.R. (2002). Parallel Computing Environments. *Handbook of Applied Optimization* (P. Pardalos & M.G.C. Resende), 1029-1043, Oxford.

(14) Martins, S.L.; Pardalos, P.; Resende, M.G.C. & Ribeiro, C.C. (2000). A parallel GRASP for the Steiner tree problem in graphs using a hybrid local search strategy. *Journal of Global Optimization*, **17**, 267-283.

(15) Martins, S.L.; Ribeiro, C.C. & Souza, M.C. (1998). A parallel GRASP for the Steiner problem in graphs. *Lecture Notes in Computer Science*, **1457**, 285-297.

(16) Marzetta, A.; Brüngger, A.; Fukuda, K. & Nievergelt, J. (1999). The parallel search bench ZRAM and its applications. *Annals of Operations Research*, **90**, 45-63.

(17) Morales, D.; Almeida, F.; Garcia, F.; Roda, J.L.; Rodriguez, C. (2000). Design of parallel algorithms for the single resource allocation problem. *European Journal of Operational Research*, **126**, 166-174.

(18) Pacheco, S.P. (1999). A User's Guide to MPI". Disponível no <http://nexus.cs.usfca.edu/mpi/>.

(19) Ribeiro, C.C. & Rosseti, I. (2002). A parallel GRASP heuristic for the 2-path network design problem. Artigo submetido.

(20) Sarma, K.C. & Adeli, H. (2001). Bilevel parallel genetic algorithms for optimization of large steel structures. *Computer-Aided Civil and Infrastructure Engineering*, **16**, 295-304.

(21) Souza, H.G.; Schiozer, D.J. & Monticelli, A.J. (1998). Uma Introdução ao PVM – Como a paralelização pode ajudar a resolver problemas complexos de otimização. *Pesquisa Operacional*, **18**, 63-73.

(22) Tanenbaum, A.S. (1999). *Structured Computer Organization.* Prentice Hall.