DENILSON ATILIO GODRY FARIAS

# PARALELIZAÇÃO DA TÉCNICA *BRANCH AND BOUND* COM PVM

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática. Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Roberto A. Hexsel

CURITIBA
2000

UFPR
Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

# PARECER

Nós, abaixo assinados, membros da Comissão Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Denilson Atílio Godry Farias, avaliamos o trabalho intitulado **"Paralisação da Técnica Branch-and-Bound com PVM"**, cuja defesa foi realizada no dia 20 de março de 2000. Após a Avaliação, decidimos pela Aprovação do Candidato.

Curitiba, 20 de março de 2000.

Prof. Dr. Roberto André Hexsel
Presidente - DINF/UFPR

Prof. Dr. Walter Godoy Júnior
CEFET-PR

Prof. Dr. Elias Procópio Duarte Júnior
DINF/UFPR

# AGRADECIMENTOS

**Inspiração** - Para quando o caminho precisa ficar mais leve.

Às vidas de Vicente e Ana Godri Farias, meus pais.

Ao amor e compreensão de Munique, Elora e Nicole, minha família.

**Motivação** - Para quando forças precisam ser reunidas.

Ao amigo e Prof. Renato Carmo, pela precisão de suas palavras e sua preocupação, pela gentileza com que sempre doa seu tempo e seus melhores pensamentos.

Aos meus amigos sinceros, que cobram, incentivam e não permitem desistir.

**Transpiração** - Para quando o esforço pode ser dividido.

Ao Aldri Luiz dos Santos por sua ajuda e paciência impagáveis, que a luz do Sol Nascente sempre o ilumine.

Ao pessoal do laboratório Prof. Marcos Castilho e Andréa, ao pessoal que passou pelo PET neste período, que vieram em meu socorro na hora em que máquinas, sistemas operacionais e o homem não se entendem.

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ALGORITMOS

# RESUMO

Este trabalho aborda a implementação paralela da técnica *Branch-and-Bound* em problemas de otimização combinatória, especificamente busca em grafos. É utilizado na implementação o modelo de programação paralela por troca de mensagens com o uso da biblioteca *Parallel Virtual Machine* (PVM) sobre o sistema operacional Linux em uma arquitetura *multicomputador*.

É analisado o comportamento da técnica *Branch-and-Bound*, em particular a relação entre (a) três critérios de busca, (b) a utilização dos recursos de memória e (c) granularidade de, processamento e comunicação entre processos.

É proposto um esquema de implementação com processos mestre-escravos semi-distribuído, onde o processo mestre é responsável pela distribuição de tarefas e os processos escravos pela disseminação de resultados parciais no sistema. Resultados experimentais dessa implementação são exibidos e analisados, assim como algumas características relevantes ao desempenho global encontradas no uso da biblioteca PVM para esta arquitetura.

De um modo geral obtivemos em média para os problemas investigados uma eficiência da execução paralela da ordem de 98% em comparação à execução serial.

# ABSTRACT

This work presents a parallel implementation employing the Branch-and-Bound technique for solving combinatorial optimization problems, specifically search in graphs. The implementation is based on message passing, using the Parallel Virtual Machine (PVM) library on top of the Linux operational system on a network of PCs.

The behaviour of this implementations of Branch-and-Bound is analyzed,whith emphasis on the relationships between (a) three graph search strategies, (b) memory resources utilization, (c) granularity (computation versus communication).

The implementation consists of a master and slave processes. The master process is responsible for allocation of work amongst the slaves, and the slave processes for performing the graph search and communicating the parcial results to the master.

We present experimental data of several runs of the program on the network of PCs, and analyse the performance of the implementation.

We also discuss several issues related to the global performance attained, and on the use of the PVM library in this architecture.

Our implementation achieved up to 98% of eficiency in some of the experiments performed.

# INTRODUÇÃO

Este trabalho descreve o estudo e a implementação paralela da técnica para resolução de problemas de otimização combinatória denominada *Branch-and-Bound*, utilizando a biblioteca de programação paralela por troca de mensagens *Parallel Virtual Machine* (PVM).

A motivação principal para este trabalho foi a possibilidade de estudar e conhecer duas áreas de grande interesse para nós: a resolução de problemas de otimização combinatória usando *Branch-and-Bound* e a experiência de paralelização de um algoritmo serial e todos os desafios que estas duas áreas envolvem.

Este trabalho enfoca problemas cujas abstrações possam ser expressas por grafos. O trabalho descreve uma implementação paralela da técnica *Branch-and-Bound*, que permite a solução de diversos problemas de aplicação prática que possam ser modelados como problemas de busca em grafos.

É grande o interesse da comunidade acadêmica no estudo da técnica *Branch-and-Bound*, por que ela permite encontrar a solução ótima para um vasto conjunto de problemas de otimização combinatória. Entretanto, o número de possibilidades a serem analisadas para se chegar a uma solução ótima em muitos destes problemas de otimização por vezes tornam inviáveis o uso de *Branch-and-Bound* em implementações seriais. Com o uso de processamento paralelo é possível viabilizar a solução de problemas maiores, que venham a tornar seu uso em aplicações práticas uma realidade.

Até pouco tempo, ambientes de processamento paralelo eram de acesso restrito, de alto custo e que demandavam tempo e recursos humanos altamente especializados. Atualmente, a disseminação crescente do uso de estações de

trabalho de baixo custo interligadas por redes locais de capacidade computacional e velocidade cada vez maiores, permitem que seja possível o desenvolvimento de aplicações que utilizem recursos desses ambientes de processamento para o estudo e implementações através da programação paralela. A programação paralela por troca de mensagens é um dos modelos mais disseminados para processamento paralelo nestes ambientes, e a biblioteca *Parallel Virtual Machine* (PVM), uma das ferramentas mais utilizadas.

O trabalho está dividido no estudo das duas áreas de conhecimento: a compreensão da técnica *Branch-and-Bound* e sua paralelização utilizando o modelo de troca de mensagens com PVM. O texto aborda as características de comportamento e implementação tanto de *Branch-and-Bound* como paralelização por troca de mensagens com PVM, e está dividido em cinco capítulos.

O capítulo 1 apresenta algumas definições básicas sobre paralelismo, conceitua o que são arquiteturas paralelas, apresenta o paradigma de programação paralela por troca de mensagens e contém uma breve descrição do funcionamento da biblioteca PVM. Também são apresentadas as medidas de desempenho para determinação da eficiência de uma implementação paralela, e um método para facilitar as decisões necessárias no desenvolvimento de um projeto paralelo. No final do capítulo são ainda apresentadas algumas definições básicas sobre a Teoria de Grafos, com os termos que serão utilizados no decorrer do texto.

O capítulo 2, faz uma apresentação detalhada da técnica Branch-and-Bound, utilizando um algoritmo genérico para demonstrar seu comportamento e suas vantagens na resolução de problemas combinatoriais.

O capítulo 3, aborda os problemas encontrados durante o projeto da paralelização de *Branch-and-Bound*, relata algumas abordagens de implementação encontradas na literatura e destaca os pontos centrais críticos que geram múltiplas

possibilidades de escolha na implementação paralela da técnica.

O capítulo 4, apresenta a implementação paralela *de Branch-and-Bound* deste trabalho. O capítulo começa com a descrição do problema escolhido para exemplificar a técnica e sua implementação serial. A seguir são apresentadas as primeiras versões paralelas da técnica implementadas, feitas a partir da metodologia para definição de projetos paralelos, e de algumas evidências encontradas no Capítulo 3. Segue uma discussão sobre o fraco desempenho obtido com estas primeiras versões e sobre as possibilidades de evitar algumas abordagens equivocadas que impediram uma implementação paralela eficiente da técnica. O capítulo se encerra com a apresentação da versão eficiente empregada na paralelização da técnica, e algumas considerações importantes sobre esta implementação, que foram pontos chaves para a obtenção de um bom desempenho.

O capítulo 5, apresenta os testes realizados, seus resultados e análises sobre o comportamento da técnica *Branch-and-Bound* e a implementação paralela proposta.

Finalmente, o capítulo 6 apresenta as conclusões finais deste trabalho e propostas para trabalhos futuros.

# CAPÍTULO 1
# DEFINIÇÕES PRELIMINARES

Este capítulo apresenta os conceitos básicos sobre processamento paralelo e suas medidas de desempenho, além de uma breve definição sobre grafos, sendo estes conceitos necessários por serem amplamente utilizados durante o decorrer do texto.

## 1.1 Paralelismo

Um computador paralelo é um conjunto de processadores capazes de trabalhar cooperativamente para resolver um problema computacional. Trabalho cooperativo entre processadores é uma definição convenientemente abrangente por que pode incluir desde supercomputadores compostos por centenas ou milhares de processadores, estações com múltiplos processadores, até redes de estações de trabalho. [Fos95]

A computação paralela, que há poucos anos era vista como uma rara e exótica sub-área da computação, tem se transformado em uma área central de estudos no meio acadêmico impulsionada principalmente por três fatores relevantes:

1. A crescente necessidade por aplicações que oferecem uma demanda cada vez maior de processamento,
2. Os limites físicos impostos ao crescimento de arquiteturas seriais,
3. O enorme incremento na capacidade de tráfego em redes de computadores.

Estes são os principais fatores que determinam o interesse no processamento

paralelo na busca de maior poder de computação, baseado em princípios bastante intuitivos como a divisão de tarefas e o trabalho cooperativo.

Arquiteturas paralelas exigem, naturalmente, o estudo de novos algoritmos e programas capazes de tirar proveito deste paralelismo.

A união de arquiteturas paralelas e de novos algoritmos que tirem proveito destas arquiteturas, possibilitam um processamento de maior desempenho. Estas novas soluções trazem entretanto, novos desafios a serem enfrentados, tais como a sobrecarga oriunda da criação de processos concorrentes e principalmente, da comunicação necessária entre estes.

Para tornar a computação paralela uma prática corriqueira, é preciso avançar em alguns de seus fatores essenciais, que atualmente são o grande desafio nos estudos sobre paralelismo:

a) Portabilidade, a capacidade de usar os mesmos programas em diferentes arquiteturas paralelas,
b) Escalabilidade, a possibilidade de obter um padrão de desempenho linear do tempo de execução e ocupação de memória, em instâncias de tamanhos diferentes de um mesmo problema, e
c) Modularidade, que permita uma manutenção eficiente destes programas.

## 1.2 Classificação de Arquiteturas Paralelas

Diversas taxonomias para arquiteturas paralelas foram propostas [Lei92], sendo geralmente classificadas de acordo com seu modo de funcionamento.

Flynn [Fly72], propôs o primeiro método para classificar estas arquiteturas, distinguindo quatro tipos de máquinas de acordo com seus fluxos de instrução e dados:

1. SISD (Single Instruction Single Data), um único fluxo de instruções executa um único fluxo de dados. Este tipo corresponde a uma máquina monoprocessador.

2. SIMD (Single Instruction Multiple Data), os processadores executam a mesma instrução em fluxos de dados diferentes.

3. MISD (Multiple Instruction Single Data), instruções diferentes executam sobre um único fluxo de dados. Esse tipo de máquina não existe na realidade, por não fazer sentido múltiplos programas processando os mesmos dados num mesmo instante.

4. MIMD (Multiple Instruction Multiple Data), este tipo de arquitetura representa um sistema de processadores autônomos que executam programas diferentes sobre fluxos de dados diferentes.

Ducan [Duc90] estendeu esta classificação incluindo o modo de sincronização entre os processadores:

1. Máquinas síncronas, um relógio global sincroniza os diferentes processadores numa base de tempo comum.

2. Máquinas assíncronas, cada máquina funciona de maneira autônoma e possui um relógio local próprio.

Mais recentemente, Patterson [PH96] propõe que máquinas paralelas podem ser ainda divididas de acordo com a organização de suas memórias:

1. Máquinas de memória compartilhada centralizada, um número de processadores compartilha uma única memória central, a qual são interligados por um barramento. Estas máquinas são chamadas de *multiprocessadores* ou arquiteturas com *memória compartilhada distribuída.*

2. Máquinas de memória fisicamente distribuída, cada processador possui seu próprio sistema de memória. Denominadas de *multicomputadores.*

Este trabalho está dirigido ao uso da arquitetura MIMD assíncrona ou *multicomputador*, que é a definição de um conjunto de processadores independentes processando instruções e dados em uma memória local, que periodicamente trocam informações entre si [Alm94]. Entre possíveis implementações MIMD [Fos95], utiliza-se neste trabalho a implementação baseada numa rede local de estações de trabalho.

Emconseqüência dos avanços na velocidade e capacidade de tráfego em redes locais, ao baixo custo em relação a outras arquiteturas paralelas disponíveis no mercado, a grande disseminação desta arquitetura em ambientes comerciais e científicos e ao desempenho satisfatório na comunicação entre as estações de trabalho encontrados nesta implementação [San99], este é um dos modelos que tem possibilitado a disseminação e avanço de estudos em programação paralela.

## 1.3 Paralelismo por Troca de Mensagens

Assim como arquiteturas paralelas, são diversas as possibilidades propostas para linguagens de programação paralela [GR86] [GR88] [GR92] [Tur93] [Fos95] [Qui94], sendo dois os motivos fundamentais da existência de uma grande variedade de paradigmas disponíveis para programação paralela [Kit95].

O primeiro é a possibilidade de tornar paralelo um mesmo problema serial

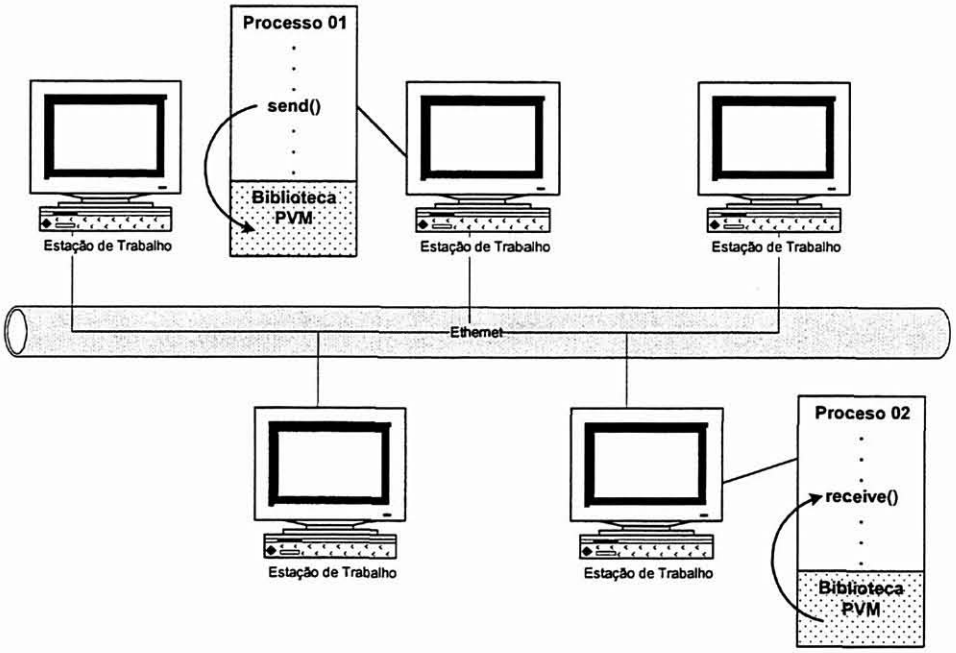
de diferentes maneiras, considerando diferentes abstrações. E o segundo motivo é a necessidade de se considerar a arquitetura alvo da aplicação que pela grande diversidade de taxonomias encontradas, favorece o aparecimento de diferentes propostas de linguagens que representem a computação paralela em cada uma das arquiteturas paralelas disponíveis.

Na arquitetura *multicomputador* baseada em rede local, adotada para este trabalho, uma possibilidade para a implementação de algoritmos paralelos é a adoção do paradigma de programação baseado em troca de mensagens.

Impulsionado pela disponibilidade e avanços desta arquitetura *multicomputador*, o paralelismo por troca de mensagens é, provavelmente, o mais difundido modelo de programação paralela em uso atualmente [Fos95].

Neste modelo, uma coleção de processos executa programas escritos em uma linguagem seqüencial padrão, tal como C ou Fortran, usando uma biblioteca de funções que possibilite a criação de múltiplos processos e implemente troca de mensagens entre estes diversos processos através de primitivas do tipo *send* e *receive*, como mostra a Figura 1.1 para um modelo *multicomputador*.

**Figura 1.1 – *Multicomputador* com PVM**




As primitivas de comunicação para envio e recebimento são implementadas normalmente em duas modalidades:

1. Comunicação síncrona: nesta modalidade o transmissor envia a mensagem para o receptor, e aguarda até que este último sinalize o recebimento da mesma. Se o receptor não estiver pronto para receber a mensagem, o transmissor é bloqueado temporariamente. Se o receptor tentar receber a mensagem antes que esta tenha sido enviada, ele também será bloqueado temporariamente.

2. Comunicação assíncrona: esta modalidade permite que o emissor envie sua mensagem e prossiga com seu fluxo normal de execução sem ficar bloqueado esperando uma resposta do receptor. Caso o receptor não esteja pronto para

receber a mensagem, esta deve permanecer armazenada temporariamente em alguma memória temporária (*buffer*). Como na comunicação síncrona, caso o receptor queira receber uma mensagem não enviada, ficará temporariamente bloqueado. Sistemas que implementam esta modalidade geralmente oferecem primitivas adicionais que apenas verificam se alguma mensagem chegou ao receptor ou não, sem a necessidade de bloqueio.

Visto que a simples operação de enviar um dado de um processador a outro envolve, no modelo de troca de mensagens em arquiteturas *multicomputador*, o uso de inúmeros serviços intermediários entre a aplicação paralela, a biblioteca de troca de mensagens, chamadas ao sistema operacional, protocolos de comunicação da camada de rede, além de todo o hardware de rede envolvido entre um processador e outro, a comunicação passar a ser, em todo o projeto paralelo baseado neste modelo, um dos pontos chaves para o sucesso de implementações paralelas por troca de mensagens.

## A Biblioteca PVM

A biblioteca de funções para troca de mensagens usada neste trabalho é a *Parallel Virtual Machine* (PVM) [Gei94] [GS92], uma das mais difundidas, populares e acessíveis, provavelmente por ser de domínio público e considerada atualmente (2.000) como o padrão de fato, embora não seja o padrão de direito, imputado à biblioteca *Message Passing Interface* (MPI) [DOW96],

Recentes trabalhos ressaltam os níveis de desempenho no uso de PVM em redes de alta velocidade [BWT98], em versões otimizadas desta biblioteca [CDH+95], e mesmo em arquiteturas de redes locais comumente encontradas [San99].

O conceito chave da biblioteca PVM, como seu nome evidencia, é o fato dela trabalhar com um conjunto de computadores heterogêneos ligados em rede, como se fosse uma única máquina paralela virtual.

O PVM é composto de duas partes: a) um processo servidor (*daemon*) chamado de Pvmd executado em cada máquina da arquitetura, responsável por atender as solicitações de tarefas pertencentes à maquina onde está sendo executado e aos Pvmds de outras máquinas, e b) uma biblioteca de funções que implementam a comunicação entre os Pvmds.

Suas primitivas para criação de processos e troca de mensagens são orientadas para operar em arquiteturas de redes de computadores heterogêneas, tornando transparente conversões de formato de dados, roteamento e gerenciamento de processos. Todas as primitivas necessárias para enviar e receber dados, verificar a chegada de mensagens, sincronizações e semáforos são encontradas na biblioteca de funções.

O usuário pode escolher o protocolo de rede a ser utilizado, TCP ou UDP, na transmissão das mensagens. Em ambos os casos as mensagens serão empacotadas antes do envio, e desempacotadas pelo processo receptor após seu recebimento

Estes procedimentos geram um custo extra na comunicação, preço pago para tornar transparente ao usuário todos os mecanismos inerentes à comunicação e heterogeneidade. Detalhes sobre as perdas geradas por este custo extra são analisados em mais detalhes em [San99].

Em PVM, as linguagens C/C++ e Fortran podem ser adotadas para a parte seqüencial da programação, sendo que especificamente neste trabalho, é adotada a linguagem C no padrão ANSI.

## 1. 4 Medidas de Desempenho

Medidas de desempenho permitem a aferição e avaliação do desempenho obtido com a paralelização de um determinado algoritmo serial, sua implementação e arquitetura onde é executada. A seguir são discutidas métricas de desempenho que serão avaliadas no trabalho.

### 1.4.1 Utilização

Utilização é a fração de tempo em que um processo do sistema utiliza o processador sem ficar parado.

Um processo pode ficar parado na troca de contexto entre processos que estão competindo por um mesmo processador, na espera da liberação de um canal de comunicação congestionado, ou ainda na espera de alguma mensagem não transmitida por outro processo.

Um valor baixo de utilização pode apontar problemas no gerenciamento da comunicação entre os processos ou ainda que o processo pode perder muito tempo esperando por algum evento externo, tal como resultados produzidos por outros processos, indicando problemas na distribuição de carga do algoritmo.

### 1.4.2 *Speedup*

*Speedup* (aceleração), é a razão entre o tempo de execução de um algoritmo serial eficiente ($t_s$), e o tempo ($t_p$) necessário para a execução de um algoritmo paralelo usando **p** processadores, para uma mesma instância do problema.

$$\mathbf{S = t_s/t_p}$$

No algoritmo paralelo ideal, esta razão deve manter-se aproximadamente constante mesmo quando o conjunto de dados do problema diminui ou aumenta, e deve também, ter um crescimento linearmente proporcional ao aumento do número (**p**) de processadores.

Esta medida permite avaliar o grau de paralelização alcançado no projeto paralelo em relação a sua versão serial. Se há um custo adicional em arquitetura de utilizar (**p**) processadores no lugar de 1, espera-se também que o programa paralelo seja ao menos em torno de (**p**) vezes mais rápido do que o serial.

**Anomalias em *Speedup***

Algumas vezes, uma implementação paralela pode ter um crescimento no *speedup* maior que o linear na medida que o número de processadores aumenta. Nestes casos, chamados de *speedup* superlinear, duas situações podem estar ocorrendo:

1. Efeitos de *cache*. Freqüentemente cada processador tem uma quantidade pequena de uma memória de alta velocidade intermediária entre processador e

memória principal (memória *cache*). Aumentando-se o número de processadores numa solução paralela, mais dados serão acomodados em memória *cache* com tempo de acesso pelo processador menor que os dados armazenados na memória principal. Como resultado, o tempo total da computação paralela tende a cair, porque os processadores locais tem acesso mais rápido ao seus dados.

2. Anomalias em buscas. Este fenômeno é encontrado em alguns algoritmos de busca em árvores. Se a árvore de busca contém soluções em vários níveis de profundidade, e a implementação da busca em paralelo explorar ao mesmo tempo ramos independentes da árvore, eventualmente a execução em paralelo poderá explorar muito menos nós do que a execução em serial.

Em ambos os casos a eficiência poderá ser maior que 1 e o *speedup* então, será superlinear.

### 1.4.3 Eficiência

Eficiência é a razão entre o *speedup* obtido na execução em **p** processadores e **p**. Esta medida mostra o quanto o paralelismo foi explorado no algoritmo, isto é, o quanto os **p** processadores foram utilizados na computação do algoritmo.

$$\mathbf{E = S/p}$$

A eficiência descreve a qualidade da implementação paralela melhor que o *speedup* porque permite analisar a paralelização como um todo, o quanto a implementação do projeto paralelo está conseguindo explorar os recursos disponíveis na arquitetura paralela.

### 1.4.4 Escalabilidade

Um algoritmo paralelo é escalável quando há um crescimento, ao menos linear, no nível de paralelismo, com o aumento do tamanho do problema e do número de processadores. Manter a escalabilidade é importante por que permite ao usuário resolver problemas de instâncias maiores com aproximadamente o mesmo *speedup* apenas aumentando o número de processadores.

Se isso não acontece, ou seja, se o *speedup* diminui em problemas maiores, mesmo quando se aumenta o número de processadores em uso, pode haver a indicação de que ou o número maior de processadores está causando um fluxo muito maior na comunicação total, ou a distribuição de tarefas não prevê ou é ineficiente em relação ao aumento de processadores na solução do problema.

Outros fatores podem ainda ser analisados para avaliar o comportamento de algoritmos paralelos tais como:

a) **Concorrência**, que é capacidade de realizar várias ações simultaneamente sobre subconjuntos de dados distintos.
b) **Granularidade**, é o grau de interdependência entre os processos que determina a freqüência com que os processos devem comunicar-se, denomina-se **granularidade grossa** quando o processo pode trabalhar a maior parte do tempo sem a necessidade de comunicação com outros processos e **granularidade fina** caso contrário, quando as interações são relativamente freqüentes.
c) **Localidade**, que é a disponibilidade dos dados aos processos em memória local ao invés de remota.

d) **Modularidade**, que é a capacidade de decomposição de entidades complexas em componentes simples, facilitando assim a manutenção e a reutilização do algoritmo paralelo.
e) **Portabilidade**, que é a capacidade do algoritmo poder ser utilizado em arquiteturas paralelas diferentes.

## 1.5 Metodologia de Desenvolvimento de Projetos Paralelos

Uma das questões que mais afligem o programador iniciante em projetos paralelos é a dificuldade em saber por onde começar. Somam-se a esta, todas as dificuldades em escolher uma entre as várias opções que se mostram disponíveis para divisão de tarefas, distribuição de dados e comunicação entre processos.

Ian Foster, em [Fos95], propõe uma metodologia para projetos de programas paralelos, visando diminuir os esforços necessários na implementação destes projetos, buscando reconhecer de antemão falhas que comprometam eficiência e escalabilidade, procurando garantir ainda a modularidade necessária para uma manutenção de baixo custo. Apresentaremos a seguir uma breve explanação desta metodologia, que foi amplamente usada durante todo este trabalho.

A metodologia proposta, consiste em dividir o estudo do projeto paralelo em 4 etapas distintas:

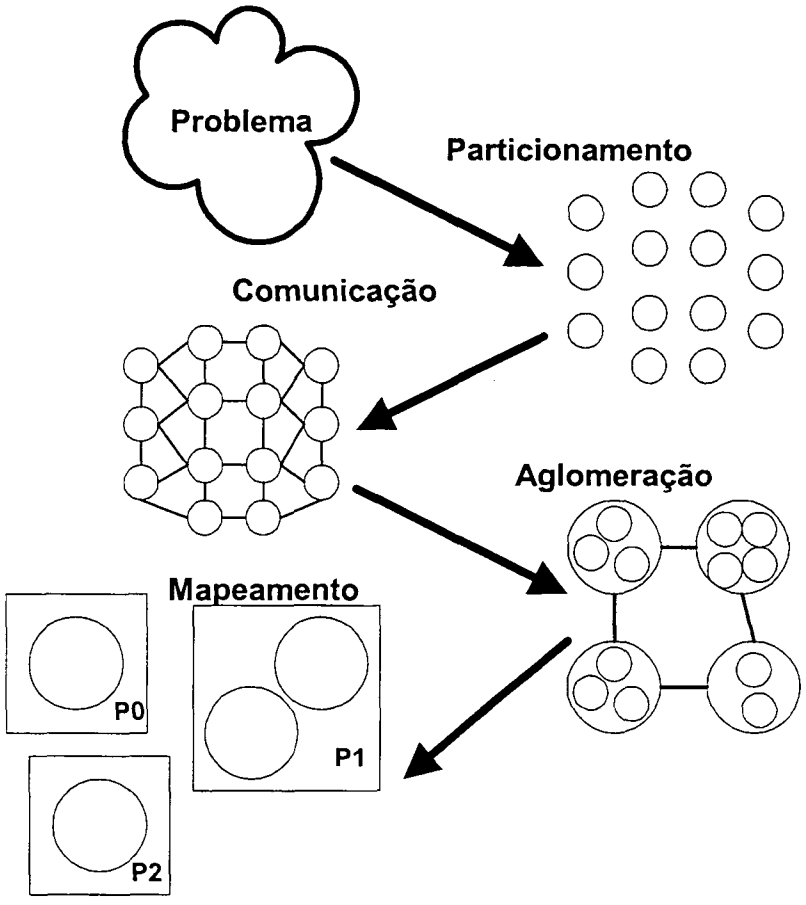
a) **Particionamento**, que é a decomposição do processamento em pequenas tarefas.
b) **Comunicação**, que é a coordenação da troca de dados entre as tarefas.
c) **Aglomeração**, que é a combinação, quando necessário, de um número de pequenas tarefas em uma tarefa maior, aumentando a granularidade, visando

melhores benefícios de desempenho e menores custos de comunicação.

d) **Mapeamento**, que é a atribuição do conjunto de tarefas aos processadores disponíveis de maneira a maximizar a eficiência ou utilização dos mesmos.

Na Figura 1.2, temos a representação destas 4 etapas no projeto de programas paralelos. A partir da especificação do problema, desenvolve-se o particionamento, determina-se os requisitos de comunicação, aglomera-se tarefas e finalmente realiza-se o mapeamento das tarefas, atribuindo-as aos processadores.

**Figura 1.2 – Etapas da Metodologia Ian Foster**



### 1.5.1 Particionamento

Na etapa de particionamento o foco das atividades está em explorar oportunidades de execução paralela, ignorando questões práticas como o número de processadores e a arquitetura a ser usada.

Um bom particionamento decompõe o problema em pequenas tarefas que possam ser executadas concorrentemente, associadas com os dados sobre os quais operam.

O estudo do particionamento é dividido em:

a) **Particionamento por Domínio**: quando o particionamento é realizado sobre a estrutura de dados do problema. Primeiro o programador explora a divisão dos dados em pequenas partes que possam ser processados independentemente, para depois associar as tarefas a cada parte destes dados.
b) **Particionamento Funcional**: aqui o particionamento é explorado na divisão de tarefas disjuntas concorrentes, para só depois associar a cada tarefa os dados que se farão necessários.

O particionamento é avaliado de acordo com as respostas das seguintes questões:

1. O particionamento gera mais tarefas ao menos uma ordem de magnitude a mais que o número de processadores? Senão haverá pouca flexibilidade nas etapas seguintes do projeto.
2. O particionamento evita computação redundante e requisições de

armazenamento de dados em memórias remotas? Senão, poderá não ser escalável para tratar instâncias maiores do problema.

3. As tarefas tem tamanhos semelhantes? Se não, pode ser difícil alocar a cada processador uma carga equânime de trabalho.
4. O número de tarefas é escalar com o tamanho do problema? O ideal é que se possa aumentar o número de tarefas com o aumento do tamanho do problema. Caso contrário, será difícil tratar problemas maiores quando houverem mais processadores disponíveis.

### 1.5.2 Comunicação

Tarefas geradas na etapa de particionamento em geral podem ser executadas concorrentemente mas raramente são executadas independentemente de tal forma a não necessitarem trocar dados umas com as outras. Dados e resultados parciais precisam ser transferidos entre tarefas, e para tal, um projeto para realizar eficientemente esta comunicação se faz necessário.

A etapa de comunicação é categorizada em quatro aspectos:

1. **Comunicação Local ou Global**: a) cada tarefa se comunica com um número reduzido de outras tarefas vizinhas, b) cada tarefa precisa se comunicar com muitas outras tarefas.
2. **Estruturada ou Não Estruturada**: c) a comunicação entre tarefas vizinhas pode ser representada por uma estrutura regular (como uma árvore ou uma matriz), d) a rede de comunicação só pode ser representada por formas arbitrárias.
3. **Estática ou Dinâmica**: e) os padrões na comunicação não se alteram durante a execução do programa, f) os padrões podem ser diferentes durante vários estágios do ciclo de vida do programa.

4. **Síncrona ou Assíncrona**: g) a comunicação entre tarefas é realizada em tempos determinísticos e há cooperação entre as tarefas para transferência de dados, h) cada tarefa pode solicitar ou enviar dados de acordo com sua demanda.

O projeto da comunicação, deve ser qualificado pelas seguintes questões:

1. Todas as tarefas realizam um mesmo número de operações de comunicação? Cargas não balanceadas de comunicação sugerem um projeto provavelmente não escalável.
2. Cada tarefa se comunica somente com um número reduzido de outras tarefas? Caso contrário pode haver muita competição pelo meio de comunicação, causando excesso de congestionamento e colisão de mensagens.
3. As operações entre tarefas são executadas concorrentemente? Se não, há grandes chances do programa ser ineficiente e não escalável.

### 1.5.3 Aglomeração

Das duas etapas anteriores, particionamento e comunicação, surge um modelo abstrato para a paralelização do projeto, que eventualmente pode não ser eficiente para a execução em uma arquitetura paralela particular disponível. Podemos, por exemplo, ter criado muito mais tarefas que processadores.

A partir desta etapa do projeto, passamos do modelo abstrato para o concreto, concentrando esforços para obter uma melhor eficiência no uso dos processadores e diminuir a comunicação entre as tarefas.

A avaliação de três pontos determinam as decisões a serem tomadas nesta etapa:

1. **Estudo da granularidade**: é possível reduzir o volume total de comunicação entre tarefas aumentando a granularidade, ou seja, aglomerando diversas tarefas em uma, reduzindo assim a necessidade de comunicação entre tarefas.
2. **Estudo da flexibilidade**: a capacidade de criar um número variável de tarefas é crítico para um programa se tornar portável e escalável.
3. **Redução de custos**: em grandes projetos pode ser determinante aproveitar o código e as estruturas de dados usados em um programa serial já implementado.

Determina-se a eficiência nesta etapa do projeto, na resposta das questões:

1. A aglomeração reduz custos de comunicação, aumentando a localidade? Se não, revise o projeto de particionamento e comunicação.
2. A aglomeração criou tarefas maiores, com custos de computação e comunicação similares? Caso contrário podem ocorrer graves problemas de balanceamento de carga.
3. O número de tarefas pode aumentar de acordo com o tamanho do problema?
4. Se foi feita a paralelização de um algoritmo serial, foi considerada a aglomeração que melhor permite a reutilização de código?

### 1.5.4 Mapeamento

Nesta etapa do projeto as tarefas são atribuídas aos processadores disponíveis na arquitetura alvo. Esse problema não existe em arquiteturas de um único processador ou em máquinas de memória compartilhada.

As duas estratégias que visam aumentar a utilização e reduzir a comunicação dos processadores são:

1. Alocar tarefas que executam concorrentemente em diferentes processadores.
2. Alocar tarefas que se comunicam freqüentemente nos mesmos processadores.

Essas estratégias são conhecidas como estratégias de compartilhamento de carga, cujo objetivo final é alocar as tarefas de tal forma a aproveitar ao máximo os recursos de processadores e memória disponíveis em arquiteturas paralelas homogêneas ou heterogêneas em uso.

O problema de alocar tarefas em processadores pode ser simples o suficiente para que a distribuição da carga possa ser definido antes da execução do programa (distribuição estática), ou complexo ao ponto de serem necessárias várias estratégias diferentes durante a execução do mesmo programa (distribuição dinâmica), em ambos os casos deve-se levar em consideração que os custos de comunicação destas transferências de tarefas podem ter um impacto importante na eficiência alcançada na implementação do projeto.

Alguns dos algoritmos de distribuição de carga mais conhecidos são:

a) **Bissecção Recursiva**: esta técnica é usada em problemas particionados por domínio. O número total de processadores disponíveis na arquitetura é repetidamente dividido em conjuntos menores até que seja igual ao número de tarefas a serem executadas.
b) **Balanceamento Local**: periodicamente cada processador compara sua carga de tarefas com seus vizinhos, se necessário os processadores trocam tarefas quando um está mais sobrecarregado que outro.
c) **Métodos Probabilísticos**: é umas das abordagens mais simples, pois resume-se a alocar tarefas aleatoriamente aos processadores. Se temporariamente durante a execução da implementação paralela as tarefas são trocadas de processadores, pode-se esperar que a distribuição seja similar.
d) **Mapeamento Cíclico**: os processadores são enumerados em uma lista circular e

cada tarefa é inicialmente alocada a um processador, em tempos determinados cada tarefa é alocada ao próximo processador da lista.

e) **Esquema Mestre e Escravo:** uma tarefa (mestre) é exclusivamente responsável pela alocação de trabalho para as demais tarefas (escravos). A tarefa mestre apenas gerencia o trabalho das tarefas escravos, não realizando o trabalho propriamente dito.
f) **Esquema Descentralizado**: neste esquema não há um controle centralizado. Cada tarefa alocada requisita mais trabalho de outras tarefas, podendo várias políticas de requisição serem definidas, tais como a comunicação apenas com vizinhos ou comunicação aleatória.

Os algoritmos de distribuição Mestre e Escravo e Descentralizado requerem ainda um mecanismo para determinar quando o processamento total foi finalizado, caso contrário correm o risco de deixar processos esperando por tarefas que não existem mais.

As questões que avaliam esta etapa do projeto são:

1. Se for um esquema centralizado, certificar-se de que o processo mestre não está demasiadamente sobrecarregado.
2. Considerar a distribuição probabilística e o mapeamento cíclico por que são simples.
3. Verificar se a capacidade de processamento em arquiteturas heterogêneas é equânime. Se não, considere mais tarefas para processadores mais potentes.

## 1.6 Grafos

Este trabalho implementa a técnica *Branch-and-Bound* (B&B) em problemas cujas estruturas de dados possam ser representadas por grafos. Esta seção formaliza uma terminologia e notação bastante introdutória sobre grafos, que são adotadas no decorrer do texto.

Um grafo é um par ordenado G = (V,A), onde V é um conjunto finito de **vértices** e A um conjunto finito de **arestas** onde $A \subseteq \{ \{u,v\}: u,v, \in V ; u <> v\}$.

O número de vértices de um grafo é representado por **n** = |V|, sendo que **m** = |A| representa o número de arestas.

Se G = (V,A) é um grafo e **a** = $[v_1,v_2] \in V$, então dizemos que $v_1$ é **adjacente** a $v_2$ (e vice-versa) e que a aresta ***a*** é **incidente** a $v_1$ e $v_2$.

O grafo é **não orientado** se as arestas não tem orientação e **orientado** caso contrário, onde para todo par (u,v) diz-se que a aresta vai de *u* (origem) para *v* (destino).

Um **caminho** de v1 até $v_i$ em um grafo G = (V,A) é uma seqüência de arestas (v1,v2), (v2,v3), (v3,v4), ..., ($v_i$-2,$v_i$-1), ($v_i$-1,$v_i$), tal que todo vértice $v \in V$ e toda aresta $a \in A$, e os vértices das seqüências das arestas são distintos. O tamanho do caminho é denotado por |P| e vale ***p***.

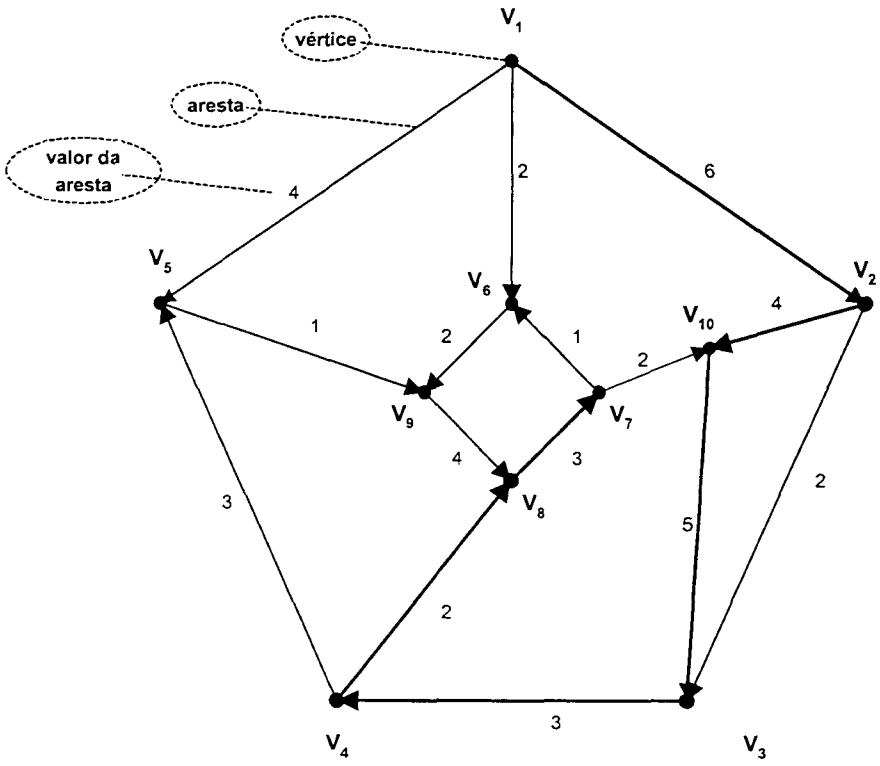
Um **circuito** é um caminho onde v1 (vértice origem) e $v_i$ (vértice destino) são iguais.

Um grafo é **valorado** G = (V,A,P={p(u,v)}) se suas arestas têm um valor (ou peso), representado pela função p(u,v), sendo que o valor de retorno desta função pode representar, por exemplo, o custo de ir do vértice *u* até o vértice adjacente *v*.

Um grafo é **completo** se para todo par de vértices (u,v) existe uma aresta $a \in A$, neste caso o número de arestas **m** será de ½ **n (n-1)**, para um grafo não orientado.

A Figura 1.3, mostra um grafo orientado valorado, e um caminho ($v_1$,$v_2$,$v_{10}$,$v_3$,$v_4$,$v_8$,$v_7$) de tamanho $p = 23$, do vértice $v_1$ até $v_7$.

**Figura 1.3 – Grafo**



# CAPÍTULO 2
# *BRANCH AND BOUND*

Otimização Combinatória é a área concentrada em encontrar a melhor solução dentre um conjunto de soluções possíveis, em determinados problemas matemáticos discretos [RND77]. Problemas estes, que tenham um número finito de soluções, onde a cada uma destas soluções possa ser atribuído um número que indique o valor da solução, para que então seja possível escolher a melhor entre todas as soluções enumeradas [Fou84]. A melhor solução pode ser a de menor ou maior valor, dependendo do objetivo e da formulação do problema.

A técnica *Branch-and-Bound* (B&B) é uma metodologia bem conhecida para resolver vários destes problemas matemáticos discretos, que são NP-difíceis, em otimização combinatória, tais como o Problema do Caixeiro Viajante, Problema da Mochila e Programação Inteira [LW66].

Uma das possíveis abordagens para entender o mecanismo de funcionamento de *Branch-and-Bound* é a partir do estudo do comportamento da busca exaustiva denominada *Backtracking*. *Backtracking* é uma forma comum de busca exaustiva em otimização combinatorial [Qui90], onde o problema inicial é decomposto em todos os subproblemas possíveis, dos quais é selecionada a melhor solução.

O algoritmo 2.1 apresenta uma forma recursiva de implementação de *Backtracking*:

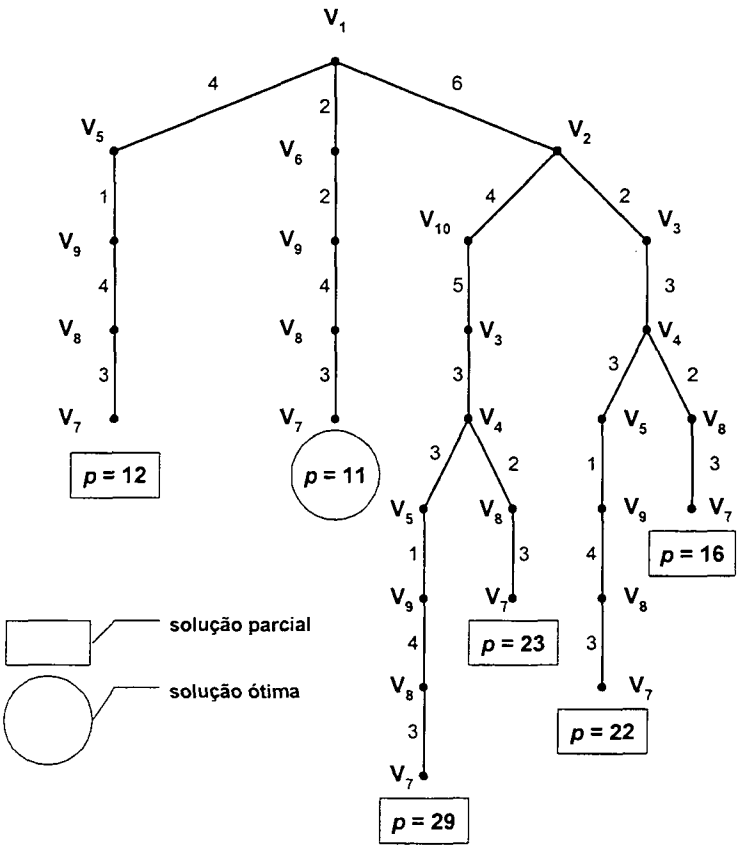
**Algoritmo 2.1 - *Backtracking***

**Backtracking**(P)
**Entrada**: Um problema P.
Se P contém uma solução do problema
solução ótima ← valor **p** de P
imprima P
Senão
Para cada subproblema P(i) de P
**Backtracking**(P(i))

Este processo de decomposição aplicado ao problema original pode ser representado por uma árvore, denominada de **árvore de espaço de estados** ou **árvore de busca** [Qui90], a qual passaremos a chamar simplesmente de árvore de estados.

Na Figura 2.1, temos a árvore de estados aplicando o Algoritmo 2.1 de *Backtracking* recursivo na solução de um problema de **caminho mínimo** entre o vértice inicial $v_1$ e o vértice final $v_7$ para o grafo apresentado na Figura 1.3. As folhas da árvore representam todas as possíveis soluções para o problema.

**Figura 2.1 – Árvore de Espaço de Estados com *Backtracking***




Neste exemplo de aplicação de Backtracking são explorados um total de **27 nós** na árvore de estados, para ter a certeza de que todas as possibilidades para a solução do problema foram avaliadas, e que a solução ótima é a de valor (**p**) = 11.

*Branch-and-Bound*, que pode ser traduzido como Ramificação e Corte, é a decomposição de um problema em outros menores e eliminação daqueles que pode-se verificar não levarão a uma solução desejada.

No algoritmo de *Branch-and-Bound* tira-se proveito de informações sobre as soluções parciais encontradas, desprezando as soluções sem chances de serem ótimas, evitando assim, ser necessário testar exaustivamente todas as ramificações da árvore de espaço de estados, para se chegar a solução ótima [Qui94]. O

algoritmo B&B decompõe um problema repetidamente usando certas restrições até que se obtenha a solução ótima ou que se prove sua inexistência.

O *branch* do B&B refere-se ao processo de particionamento sucessivo do problema inicial, enquanto o *bound* refere-se aos valores de soluções intermediárias, usados para limitar a decomposição de subproblemas inviávies, garantindo que a solução final é ótima sem a necessidade de uma busca exaustiva, ou então que não há solução para o problema [PS82].

O algoritmo 2.2 para a técnica *Branch-and-Bound*, é uma variação do algoritmo 2.1 recursivo de *Backtracking* onde a inclusão da expressão condicional (*Se solução* < *valor limite*) dentro da função recursiva, avalia se a solução parcial do subproblema em análise não é melhor que a melhor solução já encontrada, realizando o corte na árvore de estados quando o valor do subproblema não é mais promissor.

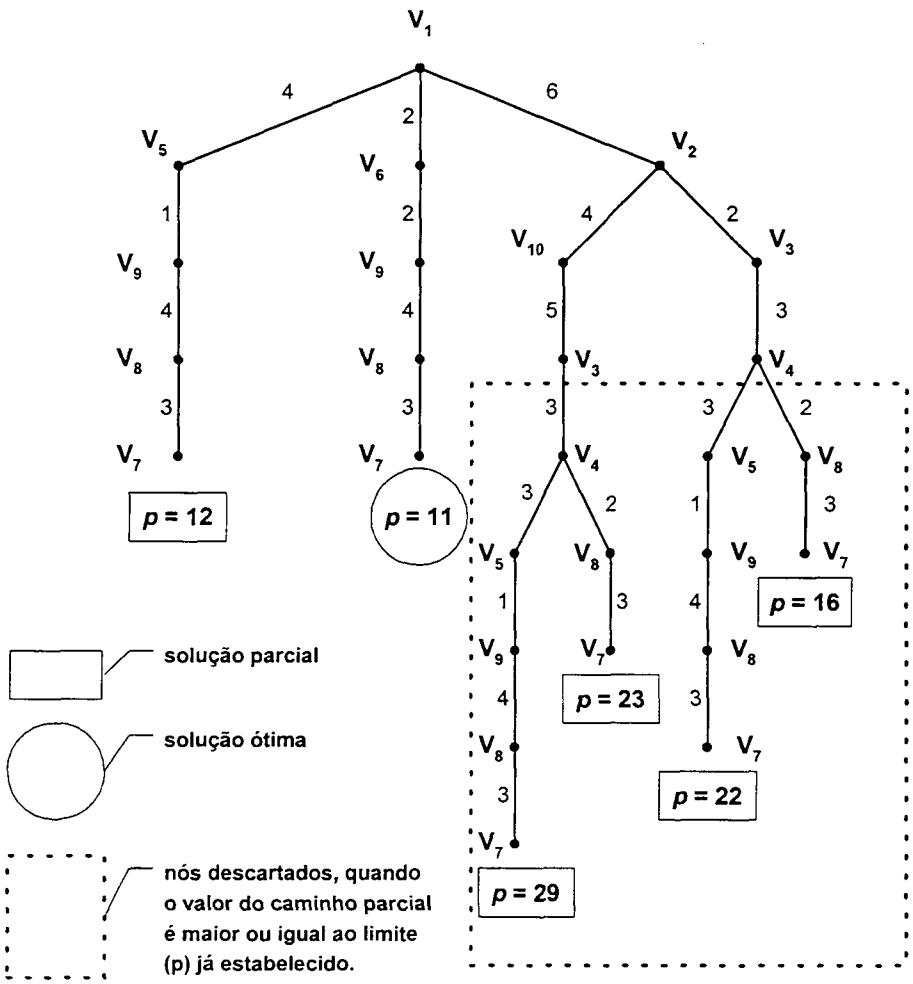
**Algoritmo 2.2 - *Branch-and-Bound* Recursivo**

```
valor limite ← ∞
Branch-and-Bound(P)
        Entrada: Um problema P.
        solução ← valor p de P
        Se solução < valor limite
                Se P é uma solução do problema
                        valor limite = solução ótima = solução
                        imprima P
                Senão
                        Para cada subproblema P(i) de P
                                Branch-and-Bound(P(i))
```

Na Figura 2.2, temos a árvore de estados para a mesma solução do problema

de **caminho mínimo** para o grafo apresentado na Figura 1.3 aplicando o algoritmo 2.2 de B&B recursivo.

**Figura 2.2 – Árvore de Espaço de Estados com B&B**




Ramos da árvore são descartados sempre que seus valores parciais forem maiores ou iguais ao valor limite mínimo de uma solução parcial já encontrada.

Neste exemplo, uma primeira solução parcial é encontrada pelo ramo da aresta (v1,v5) com p = 12, percorrendo-se a árvore de estados da esquerda para a direita, que se torna o primeiro valor limite (*bound*). Em seguida encontra-se uma solução parcial melhor pelo ramo da aresta (v1,v6) com p = 11 atualizando-se então o valor limite. O processamento do ramo originado pela aresta (v1,v2) é então interrompido quando os valores parciais dos **caminhos** [(v1,v2,v10) com p=15] e [(v1,v2,v3) com p=11] se tornam iguais ou maiores ao valor limite atual que é p = 11. Sem mais nós para analisar o algoritmo termina, sendo que sua última solução parcial encontrada (p=11) é então a **solução ótima**. Enquanto que aplicando *Backtracking* foram necessários a análise de 27 nós da árvore de estados, em *Branch-and-Bound* a possibilidade do corte limitou a exploração em apenas **14 nós**.

Um algoritmo B&B pode ser dividido em quatro partes [Yan94]: a) Regra de Particionamento, b) Regra de Seleção, c) Regra de Eliminação, d) Condição de Término, discutidas detalhadamente a seguir.

## 2.1 Regra de Particionamento

A regra de particionamento é responsável pela decomposição de um problema em subproblemas usando alguma estratégia heurística para esta decomposição. O tipo de problema a ser resolvido determina as possibilidades de implementação desta regra.

No algoritmo 2.2 do B&B recursivo, a regra de particionamento é genericamente representada pelas linhas:

Para cada subproblema P(i) de P
**Branch-and-Bound**(P(i))

Os nós intermediários da árvore de estados, que representam as possíveis soluções (**P(i)**) a serem analisadas, são normalmente mantidos em uma estrutura de dados tipo lista, onde a escolha do método para ordenação desta lista determina a estratégia a ser adotada na regra de seleção da técnica de B&B, como veremos a seguir.

## 2.2 Regra de Seleção

A regra de seleção examina os subproblemas ainda não analisados, e seleciona um deles para a próxima expansão.

No algoritmo 2.2 do B&B recursivo, a recursão é embutida na regra de seleção, o que determina que a escolha dos próximos subproblemas a serem examinados ocorra na ordem de uma busca em profundidade.

Se ao contrário de recursão, os nós intermediários forem armazenados numa estrutura de dados tipo lista, a estratégia adotada para manutenção desta lista determinará o modo como os subproblemas serão ordenados e selecionados para expansão.

Quatro diferentes abordagens têm sido propostas para a manutenção da regra de seleção. Um algoritmo B&B é chamado de:

a) **Busca em Largura:** se a lista de nós a serem examinados é mantida em uma lista do tipo FIFO.

b) **Busca em Profundidade:** se a lista de nós é mantida como uma lista do tipo LIFO.

c) **Melhores Primeiro:** se a lista é ordenada com as melhores soluções parciais primeiro.

d) **Aleatório:** se a seleção dos nós a serem examinados é arbitrária.

É sabido que o uso da regra de seleção com a lista ordenada pela pesquisa melhores primeiro, é o método mais atrativo em termos de complexidade de tempo, quando heurísticas sobre o problema são totalmente desconhecidas [Yan94].

A regra de seleção é um fator importante na eficiência da implementação de B&B paralelos [Yan94], visto que sua eficiência pode reduzir os custos com comunicação em algoritmos paralelos, o que normalmente é um fator crítico, dado os custos elevados das operações de comunicação.

## 2.3 Regra de Eliminação

A regra de eliminação especifica quais os subproblemas que não podem gerar uma solução ótima, melhor do que a solução atual encontrada durante as operações de particionamento e seleção, e os elimina da árvore de estados.

No algoritmo 2.2 do B&B recursivo, a regra de eliminação é genericamente representada pelas linhas:

solução ← valor **p** de P
Se solução < valor limite
**continua...**

Onde somente os subproblemas **P**, com valor **p** menor que o valor limite atual continuarão a ser analisados. Subproblemas com valores iguais ou maiores que o valor limite são descartados e o algoritmo continua com a seleção de um novo subproblema.

## 2.4 Condição de Término

A condição de término da técnica B&B ocorre quando não há mais subproblemas na árvore de estados para decomposição, encerrando assim o processo. Se os subproblemas são mantidos em uma lista, a condição de lista vazia indica o final do processo.

A condição de término da técnica B&B, de analisar todos os subproblemas até a exaustão, garante que todas as soluções possíveis são examinadas, e que a solução encontrada, se existir, é ótima.

As principais vantagens do uso de B&B na busca de soluções ótimas em problemas combinatoriais são [Hom96]:

a) Na média, não é necessário o processamento de todas as possíveis combinações.

b) A computação de cada nodo da árvore de estados é totalmente independente dos outros nós, permitindo que ramos diferentes da árvore de estados possam ser processados concorrentemente.

c) A possibilidade de processar cada subproblema gerado na fase de particionamento de forma independente, possibilita uma paralelização por domínio trivial no método B&B, pelo menos no nível conceitual.

# CAPÍTULO 3
# PARALELIZAÇÃO DE TÉCNICAS *BRANCH AND BOUND*

Este capítulo contém 4 seções. A primeira seção apresenta os resultados obtidos em alguns trabalhos realizados sobre o estudo da paralelização de B&B. A segunda seção analisa propostas de bibliotecas genéricas para solução de problemas de B&B paralelo, a terceira descreve as diversas abordagens possíveis e faz uma avaliação das maiores dificuldades encontradas na paralelização de B&B, e a quarta seção apresenta uma proposta para classificação de algoritmos B&B.

Vários estudos têm sido realizados sobre paralelização de B&B como, por exemplo, implementações de problemas específicos, avaliações de desempenho, análises de estratégias de manutenções de filas, estrutura de dados adequadas, e estratégias de seleção [Yan94] bem como implementações eficientes de algoritmos de B&B paralelos [LM89][OT89][CT89].

## 3.1 Abordagens Paralelas de B&B

Quinn, em [Qui90], apresenta as análises sobre a implementação de duas formas de paralelização de B&B. Na primeira abordagem, denominada de Algoritmo Síncrono com Fila Centralizada, um processo é o único responsável pela manutenção da fila de subproblemas. Em tempos determinados todos os outros processos devolvem seus subproblemas ao processo centralizador e esperam até que este processo reorganize sua fila de subproblemas e redistribua as tarefas.

Quinn aponta várias desvantagens na implementação deste algoritmo em arquiteturas *multicomputadores*, como o desproporcional tamanho da memória do processador centralizador, o aumento linear da comunicação com o aumento do número de processos envolvidos sacrificando a escalabilidade do algoritmo e o tempo de espera sem processamento de vários processos enquanto o processo centralizador reorganiza e redistribui subproblemas.

Na sua segunda abordagem, denominada de Algoritmo Assíncrono com Fila Distribuída, cada processo recebe um subproblema inicial, e depois a cada interação usa uma heurística para incluir parte dos subproblemas gerados em sua fila local e enviar a outra parte com seu valor limite local para seu processo vizinho. Quinn sugere que a grande dificuldade para a implementação desta abordagem é definir a heurística para distribuição das tarefas, mas que por outro lado evita as desvantagens encontradas no Algoritmo Síncrono Centralizado.

Laursen, em [Lau93], sugere que a maior parte do trabalho necessário para a implementação de B&B paralelo é gasto com a criação de algoritmos muito complicados para o gerenciamento da comunicação entre os processos.

Nesta proposta Laursen propõe duas estratégias mais simples para o gerenciamento da comunicação:

Na primeira, denominada de Distribuição Estática de Subproblemas, um particionamento inicial é realizado de modo a dividir os subproblemas em cotas de tamanho similar entre os processos. Depois que cada processo recebe sua cota, não há mais comunicação entre os processos até que cada um tenha terminado seus subproblemas, quando então um dos processos seleciona dentre várias, a solução ótima. Para esta estratégia o autor sugere que os pontos críticos são, primeiro encontrar uma boa heurística para definir a distribuição inicial dos subproblemas e segundo, o fato de os processos trabalharem somente com seus valores limites locais.

Em sua segunda estratégia, chamada de Distribuição Dinâmica de Subproblemas, a distribuição de tarefas é realizada de forma mais usual, onde um processador é responsável por enviar subproblemas aos demais, mas a comunicação é sincronizada. Os processos sabem previamente, quando, o que, e com quem devem se comunicar. As dificuldades expostas para esta abordagem são encontrar um protocolo de comunicação simples e livre de erros, decidir quem se comunica com quem e encontrar um intervalo de tempo para o qual a comunicação seja eficiente.

Diderich e Gengler, em [DG94] e [DG95], propõem em seu Algoritmo B&B Paralelo Sincronizado, que todos os processos mantenham sua própria fila de subproblemas e que cada um possa, independentemente, enviar um pedido de sincronização para os demais processos. Quanto todos os processos estiverem sincronizados a comunicação das informações locais sobre estado da fila de tarefas e o valor limite local são trocadas entre um processo e seu vizinho. Um processador pode solicitar uma sincronização quando encontrar um valor limite melhor que o atual ou estiver sem subproblemas em sua fila. Duas dificuldades são mencionadas na implementação desta abordagem, a primeiro é que quando um processo fica sem tarefas e seus vizinhos próximos não tiverem subproblemas de sobra, há uma espera muito longa e muitos pedidos de sincronização até que esta informação chegue aos vizinhos distantes; a segunda dificuldade destacada é que em média 15% do tempo total de processamento é gasto apenas com os pedidos de sincronização entre processos.

## 3.2 Bibliotecas para B&B Paralelo

Existem propostas e implementações de bibliotecas genéricas para implementação de algoritmos em B&B, implementadas para uso em arquiteturas MIMD heterogêneas, algumas destas baseadas em PVM, como por exemplo: BOB [CR95], D. Homeister [Hom96], Sabor [Mei96][ST96]. Outra biblioteca, a ZRAM [BMFN96] é implementada com base na Message Passing Interface (MIP).

Com o objetivo de tornar transparente a implementação de B&B Paralelo, estas bibliotecas possibilitam direcionar maiores esforços em implementações práticas de soluções de problemas combinatoriais, procurando isolar seus usuários dos problemas associados à implementação de B&B e sua paralelização.

Uma breve análise realizada na documentação destas bilbiotecas para paralelização de B&B disponíveis sugere que os recursos de memória ficam rapidamente escassos, devido ao fato de que para se tornarem suficientemente genéricas, estas bibliotecas não determinam (ou implementam), nenhuma estrutura de dados para manipulação do problema. Desta forma, como a estrutura de dados não é conhecida pela biblioteca, a estratégia de distribuição de tarefas é implementada de tal forma que toda vez que um problema é decomposto pela regra de particionamento, a estrutura de dados bem como todos os dados relativos ao escopo de cada subproblema gerado devem ser transmitidos para os processos que executam o algoritmo B&B. Além do grande desperdício de tempo necessário com a comunicação, ocorre uma replicação excessiva de dados esgotando rapidamente os recursos disponíveis de memória.

Por outro lado, implementações específicas que tendem a ser mais eficientes são por demais complexas quanto à distribuição de tarefas e ao gerenciamento de comunicação, impondo limites rigorosos a sua reusabilidade, tanto na sua solução algorítmica quanto na arquitetura paralela usada [Lau93].

Deste modo, apesar de a técnica B&B ser considerada como a melhor técnica para o estudo de problemas combinatoriais e da atenção dispensada pela comunidade envolvida com processamento paralelo à paralelização do método, o uso de B&B tem se mostrado eficiente apenas para resolver problemas cujas instâncias são consideradas pequenas, não sendo praticável no uso de problemas realísticos [Lau93].

Entre os estudos avaliados, o maior obstáculo para o uso de B&B está na necessidade de processar o número exponencial de subproblemas gerados em problemas realísticos, onde a principal dificuldade é identificada como os limites de memória local para processamento e gerenciamento da fila de tarefas [Fos95] [Hom96] [Mei96] [CR95] [Qui94] [Yan94] [Moh83].

## 3.3 Dificuldades na Paralelização de B&B Paralelo

As dificuldades na paralelização de B&B paralelo podem ser separadas em duas categorias: a) distribuição de tarefas e b) gerenciamento de comunicação [Lau93].

### 3.3.1 Distribuição de Tarefas

A escolha da estratégia de distribuição de tarefas deve maximizar o uso dos processadores, evitando que estes fiquem parados aguardando por trabalho, e se possível, procurar eliminar ramos na árvore de estados cujas soluções se tornaram inviáveis, reduzindo assim trabalho desnecessário.

Para a implementação da estratégia de distribuição de tarefas em paralelização de B&B deve-se avaliar e escolher dentre as possíveis opções:

a) **Distribuição centralizada ou distribuída**: um processo controla a distribuição

de tarefas, ou esta função é distribuída entre outros processos?

b) **Distribuição estática ou dinâmica**: há uma quantidade pré-estabelecida no número de tarefas que cada processo pode executar, ou isso vai depender da capacidade de processamento local dos processadores envolvidos?

c) **Gerenciamento de valores limites (*bound*)**: cada processo terá conhecimento de limites melhores gerados em outros processos (gerenciamento global), ou apenas usarão seus limites locais (gerenciamento local)?

d) **Gerenciamento das filas de tarefas**: as estratégias de manutenção da fila de tarefas são parametrizáveis ou fixas no algoritmo? Processos podem trocar informações sobre suas filas?

### 3.3.2 Gerenciamento da Comunicação

O gerenciamento da comunicação é uma tarefa complexa em programação paralela, com efeito direto na eficiência do algoritmo paralelo implementado [Lau93][Yan94]. Um projeto eficiente de comunicação deve evitar congestionamentos na rede de comunicação e colisões de mensagens entre os processos, efeitos estes que podem suspender, mesmo que temporariamente, o trabalho dos processadores.

Em implementações de B&B paralelo, as decisões referentes a abordagem utilizada na comunicação entre processos resume-se em encontrar soluções eficientes para as seguintes questões:

a) **Comunicação síncrona ou assíncrona**: será estabelecido um tempo específico para a comunicação entre os processos, ou os processos se comunicam na ocorrência de eventos durante seu processamento?

b) **Comunicação centralizada ou distribuída**: um processo comanda os eventos de comunicação ou cada processo pode se comunicar espontânea e

independentemente?

c) **Gerenciamento de colisões**: dois ou mais processos podem competir ao mesmo tempo pelo canal de comunicação?

d) **Gerenciamento de tráfego**: há um volume pré-determinado na quantidade de mensagens que um processo pode transmitir, ou cada processo pode ocupar o canal de comunicação com o volume de mensagens necessários para sua demanda?

Levando-se em consideração apenas a categorização de projetos de B&B paralelo proposta por [Lau93], que descreve quatro características para divisão de tarefas mais quatro características para a comunicação, cada qual com suas alternativas de composição existem 56 possibilidades diferentes de implementação paralela de B&B.

Desta forma, não existe ainda um padrão definido para selecionar a melhor estratégia de distribuição de tarefas e comunicação, que seja sempre eficiente. A melhor abordagem para a paralelização de técnicas B&B será aquela que levar em consideração um maior número de critérios, tais como, os recursos da arquitetura disponível, o comportamento do problema ou grupo de problemas a serem implementados, as necessidades de manutenção, escalabilidade e custos de implementação.

## 3.4 Uma Taxonomia para Classificação de Algoritmos B&B Paralelos

Trienekens e Bruin, em [TB92], baseados nas dificuldades e decisões necessárias para a implementação de algoritmos B&B paralelos, propõem uma classificação destes algoritmos de acordo com as decisões tomadas sobre três parâmetros:

a) **Compartilhamento do conhecimento:** como os resultados locais parciais dos

subproblemas (valores limites) são compartilhados entre os processos, e a estratégia de como os processos reagem quando do conhecimento de um valor limite melhor do que o seu atual.

b) **Divisão de trabalho:** definição do número de processos, distribuição da quantidade de tarefas para cada processo e estratégias de balanceamento de carga entre estes.

c) **Sincronicidade:** se a comunicação entre os processos é síncrona ou assíncrona

Onde a) e b) estão associados às decisões a serem enfrentadas na estratégia de distribuição de tarefas e c) da comunicação entre processos.

# CAPÍTULO 4
# IMPLEMENTAÇÃO PARALELA DE *BRANCH AND BOUND*

Este capítulo apresenta o projeto e a implementação paralela da técnica B&B em arquiteturas *multicomputador* baseadas no uso de PC's e do sistema operacional Linux, com o uso da biblioteca PVM como ferramenta de programação por troca de mensagens.

O foco da discussão está centrado nas questões que mais instigaram e que mais impacto tiveram durante todo o processo de implementação deste trabalho, afetando diretamente nos resultados do aprendizado e de desempenho obtidos. As questões fundamentais são: a) Uso de metodologia no desenvolvimento de projetos paralelos, b) distribuição de tarefas e c) gerenciamento da comunicação.

Sendo a primeira questão mais ampla e genérica e as duas últimas especificamente relevantes na paralelização de B&B por troca de mensagens.

O procedimento adotado para o estudo e implementação deste projeto está dividido em etapas como seguem, descritas em mais detalhes ao longo do capítulo nas seguintes seções:

a) Definição da classe de problemas a serem explorados.
b) Seleção e implementação de um algoritmo B&B serial.
c) Aplicação da Metodologia à Implementação Paralela.
d) Distribuição de Tarefas.
e) Gerenciamento da Comunicação.
f) Projeto e implementação do algoritmo B&B paralelo.

## 4.1 Definição da classe de problemas a serem explorados.

Entre as inúmeras possibilidades de escolha de um problema de otimização combinatória, para servir como aprendizado, avaliação e testes para a implementação paralela da técnica B&B, procurou-se nesse trabalho escolher um problema cuja estrutura de dados permita a abstração de uma gama maior de problemas que sejam também de grande aplicação prática.

Com nosso interesse voltado para uma implementação que possa atuar na resolução de vários problemas, é adotado para este trabalho uma estrutura de dados para implementação de problemas cujas abstrações possam ser expressas por **grafos**. Mais especificamente, limitamos o escopo deste trabalho em problemas de **busca em grafos**, que trata de problemas onde seja necessário caminhar entre os vértices e/ou arestas do grafo e que a solução seja dada pelos vértices e/ou arestas visitados e o valor do tamanho do caminho total percorrido.

Entre as possibilidades de problemas possíveis de implementação em busca em grafos, foi selecionado o Problema do Caixeiro Viajante (PCV), por seu alto interesse teórico e prático, interesse esse fartamente explorado na literatura.

Segue uma breve descrição deste problema. Referências detalhadas podem ser encontradas em [PS82][LM92][LLK86][Qui94].

### Problema do Caixeiro Viajante (PCV)

No Problema do Caixeiro Viajante, nós temos um conjunto de **n** cidades ligadas entre si por caminhos e uma distância associada a cada um desses caminhos. O objetivo é encontrar o menor caminho que começa e termina na mesma cidade, visitando cada uma das outras cidade apenas uma vez.

Formalmente, o mapa com estas **n** cidades e seus caminhos, pode ser representado por um **grafo completo** G = (V,A,P), onde V é um conjunto de

vértices (cidades), A é um conjunto de arestas {(u,v)} que são os caminhos conectando os vértices, e P é um conjunto de pesos não negativos {p(u,v)} (distância entre as cidades), que representam a distância entre os vértices *u* e *v*. Se o grafo é **não orientado** cada par de vértices (u,v) pode ser percorrido na direção de *u* até *v* e na direção de *v* até *u* e p(u,v) = p(v,u). Neste caso o grafo PCV é dito **simétrico**, e é este tipo de PCV o objeto de estudo deste trabalho.

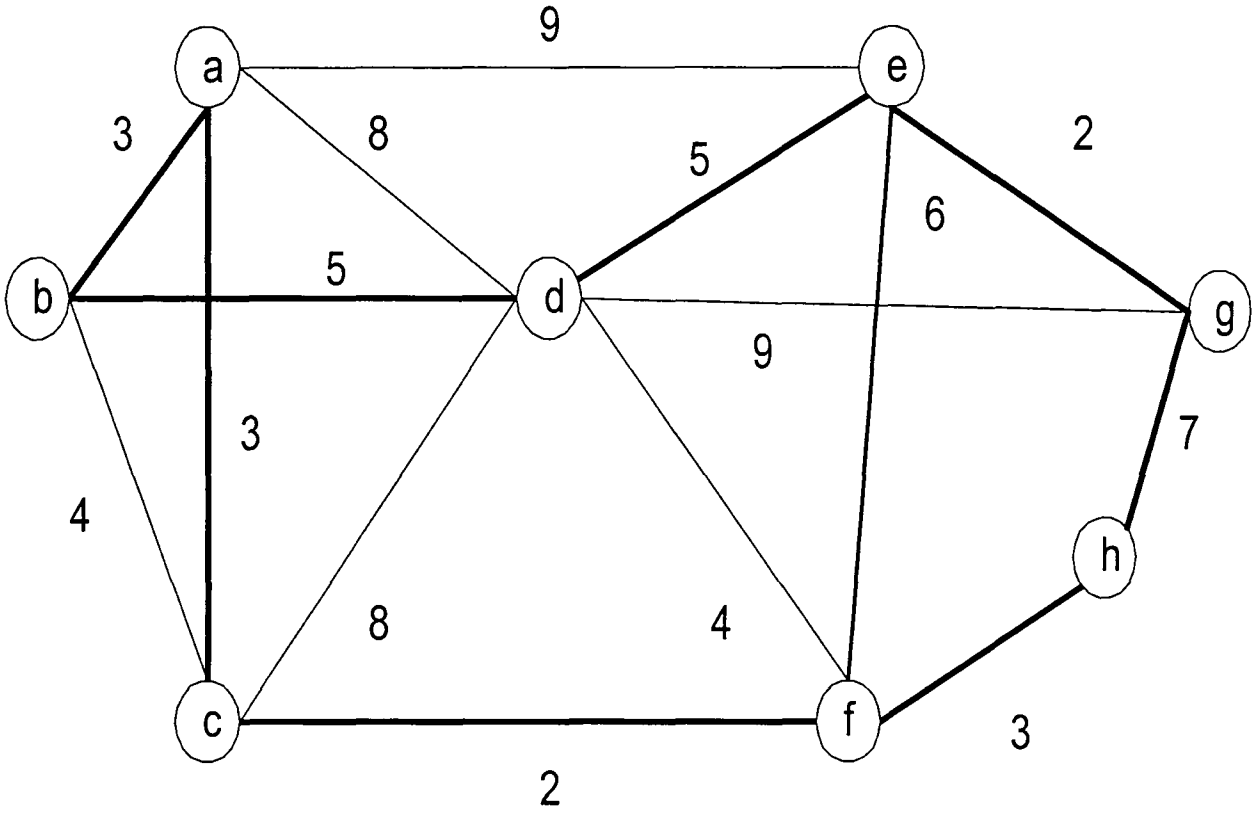
O número de caminhos possíveis para visitar estas **n** cidades é **(n-1)!/2**, e o exame sistemático de cada possibilidade para encontrar o menor caminho requer **O(n!)** passos.

A escolha de realizar os testes sobre a implementação de B&B em busca de grafos com o PCV simétrico, deve-se ao fato dele ser um problema NP-completo de crescimento exponencial, e de interesse prático na resolução de problemas de otimização combinatorial [PS82].

O fato de escolher para as avaliações grafos **completos** e não orientados torna as buscas por soluções em nossos exemplos de PCV mais exaustivas do que em grafos incompletos ou orientados (PCV assimétrico), pois qualquer caminho de tamanho igual a |V| é uma solução possível. Sendo ainda o grafo completo e não orientado, não há a preocupação de usarmos exemplos tendenciosos que possam induzir a interpretação dos dados nos testes, já que a escolha de grafos completos para a implementação do PCV simétrico gera os exemplos de variação mais pessimista em termos de números de soluções possíveis.

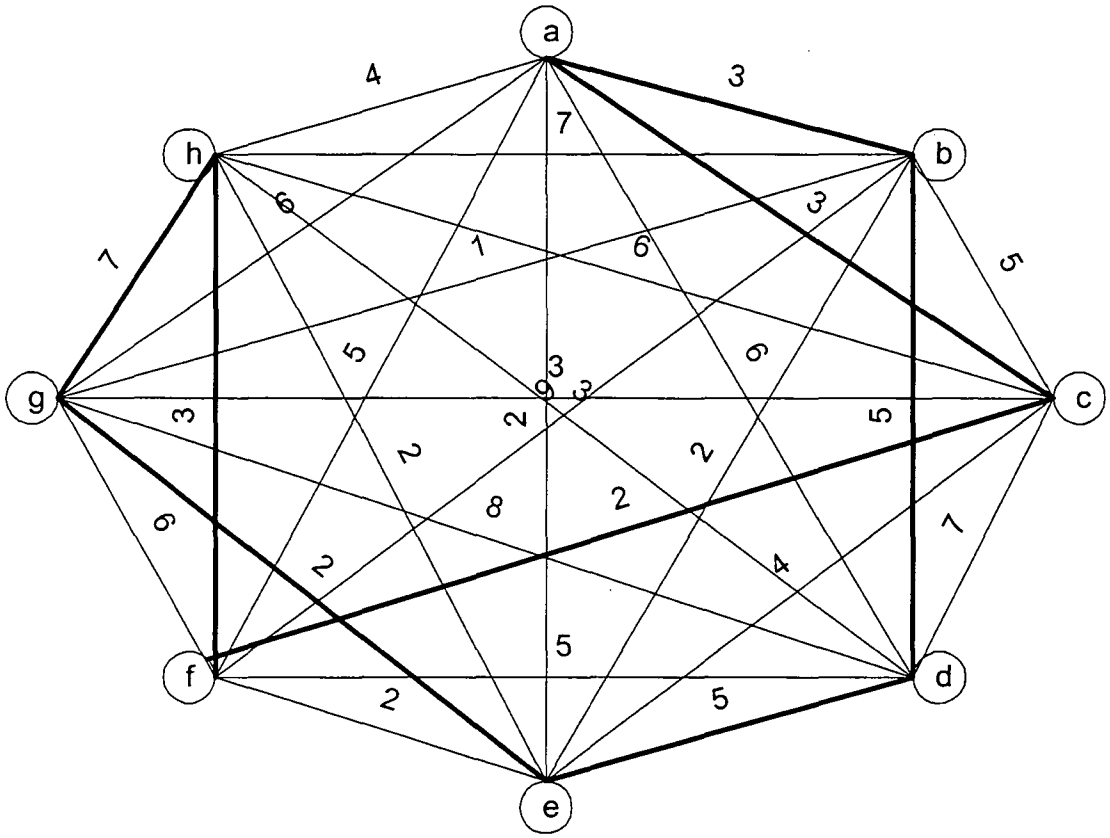
A Figura 4.1 mostra um grafo não completo, valorado e não orientado de |V| = 8 representando 8 cidades e |A| = 15 representando a distância entre as cidades e uma solução para o problema PCV simétrico em **linhas sólidas**.

**Figura 4.1 – Um problema PCV simétrico em grafo não completo**



A Figura 4.2 mostra a mesma solução em linhas sólidas para as 8 cidades representadas na Figura 4.1, mas agora com todas as possibilidades de caminhos, $|A| = \frac{1}{2}n(n-1) = 28$, de um grafo completo.

**Figura 4.2 – Um problema PCV simétrico em grafo completo**




É possível o uso de grafos completos para representar um PCV para casos realísticos onde nem todos os vértices possuem arestas entre si, bastando para isso, representar a não existência de ligação entre dois vértices com arestas de valor infinito.

Apesar desta escolha do problema PCV simétrico em grafos completos, outros testes com problemas de menor complexidade foram realizados, como Caminho Mínimo e *Floorplan Optimization* ambos são detalhados em [Fos95] e *Flowshop Scheduling Problem* detalhadamente explicado em [PS82].

Estes testes com outros problemas que não PCV, foram realizados com a intenção de validar as possibilidades de usar a mesma implementação em mais de um problema de busca em grafos. Seus desempenhos entretanto não foram tabulados por que os valores obtidos com a avaliação do PCV, que gera uma árvore de estados muito maior, bem como o número de possibilidades a serem analisadas, são suficientes para as análises necessárias de desempenho e comportamento da versão paralela em relação à serial implementadas neste trabalho.

## 4.2 Seleção e implementação de uma algoritmo B&B serial.

O algoritmo B&B serial, ponto de partida para início da paralelização, é derivado do algoritmo básico descrito em [PS82].

A Algoritmo 4.1 mostra o algoritmo já modificado, que é basicamente o mesmo algoritmo recursivo descrito no Algoritmo 2.2, onde a recursão é substituída por uma estrutura de dados que denominamos de ListaTarefas, utilizada para armazenar os subproblemas não explorados.

**Algoritmo 4.1 - *Branch-and-Bound* Serial Genérico**

**Branch-and-Bound Serial Genérico** (P)
**Entrada**: Um problema P.
valor limite ← ∞
ListaTarefas ← P
**Enquanto** ListaTarefas não estiver vazia
**a** ← (retira um subproblema de ListaTarefas)
solução ← valor **p** de **a**
**Se** solução < valor limite
**Se a** é uma solução do problema
valor limite = solução ótima = solução
imprima **a**
**Senão**
Para cada subproblema a(i) de **a**
ListaTarefas ← ListaTarefas + a(i)

A implementação da estratégia da regra de seleção neste algoritmo 4.1 (profundidade, largura, melhores primeiro ou aleatório), parte importante num algoritmo B&B, depende tão somente de como são implementadas as operações de inserção e remoção de elementos na ListaTarefas.

Removida a recursividade, o Algoritmo 4.1 é alterado para representar uma

solução de B&B que represente especificamente problemas de busca em grafos.

O Algoritmo 4.2 é uma evolução dos modelos genéricos apresentados até o momento para uma representação em pseudo-código de um algoritmo serial de B&B especificamente para busca em grafos, muito próxima da versão real implementada.

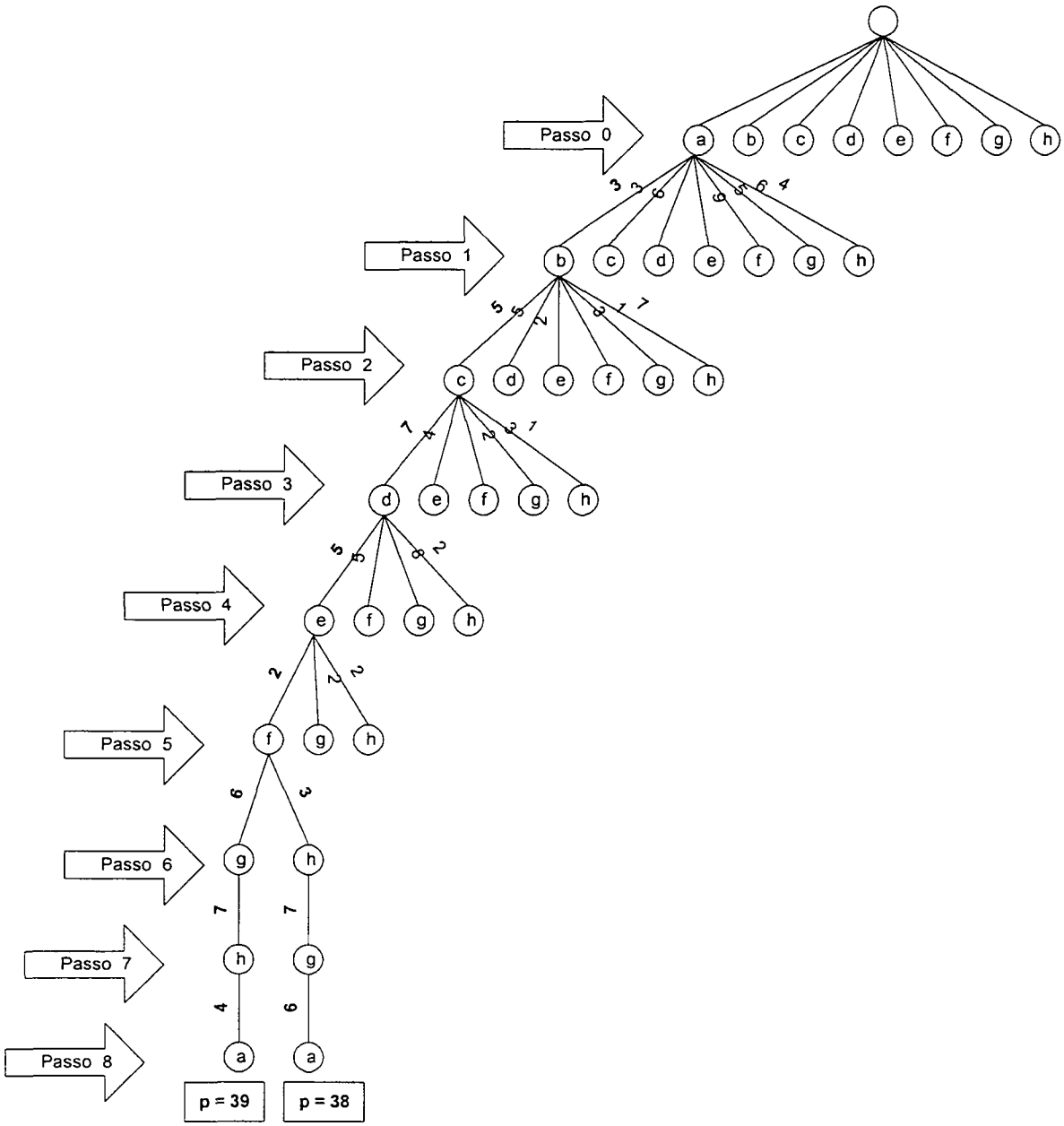
**Algoritmo 4.2 - *Branch-and-Bound* Serial para Busca em Grafos**

**Branch-and-Bound Serial para Busca em Grafos** (G,Vi)
**Entrada**: Um grafo G.
Um conjunto de vértices iniciais Vi, |Vi| > 0
**Estrutura Lista Tarefas**: {V, v, valorlimite} onde,
**V**, é o conjunto dos vértices já percorridos
**v**, é o vértice ainda não explorado
**valor limite**, é o valor (**p**) total dos pesos das arestas do caminho dos vértices, $v_1$ até $v_n$, contidos em V
valor limite ← ∞
**Para** cada vértice v(i) de Vi
ListaTarefas ← {vazio, v(i), 0}
**Enquanto** ListaTarefas não estiver vazia
**tarefa** ← (retira um elemento de ListaTarefas)
solução ← **tarefa**.valor limite
**Se** solução < valor limite
**Se tarefa**.V é uma condição de término
valor limite ← solução ótima ← solução
imprima **tarefa**.V e solução ótima
**Senão**
Para cada vértice *u* adjacente a **tarefa**.v
ListaTarefas ← ListaTarefas +
{ V + **tarefa**.v, *u*, **tarefa**.valor limite +
p(*u*,**tarefa**.v) }

A Figura 4.3, mostra os primeiros 9 passos na construção da árvore de

espaço de estados da técninca B&B pelo Algoritmo 4.2 aplicado no problema PCV da Figura 4.2, para uma ListaTarefas funcionando por seleção em profundidade. A partir de um vértice inicial qualquer, o algoritmo expande a árvore pelas arestas adjacentes aos vértice até que um caminho completo do PCV seja encontrado, neste ponto um valor limite passa a limitar a exploração de nós da árvore cujo caminho seja igual ou maior que o valor limite atual.

**Figura 4.3 – Árvore de estados para PCV simétrico em grafo completo**



A Tabela 4.1 mostra o comportamento da estrutura ListaTarefas em cada passo da execução do Algoritmo 4.2, similarmente a cada nível da árvore de estados ilustrada na Figura 4.3. As linhas sombreadas representam uma exclusão do elemento da estrutura ListaTarefas que no caso de uma busca em profundidade funciona como uma pilha do tipo FIFO.

**Tabela 4.1: Conteúdo de ListaTarefas**

| | Conjunto V | Vértice não explorado | Valor de (p) |
|---|---|---|---|
| **Passo 8** | {a, b, c, d, e, f, g, h} | a | 3+5+7+5+2+6+7+4 = **39** |
| **Passo 7** | {a, b, c, d, e, f, g} | h | 3+5+7+5+2+6+7 |
| **Passo 6** | {a, b, c, d, e, f} | g | 3+5+7+5+2+6 |
| | {a, b, c, d, e, f} | h | 3+5+7+5+2+3 |
| **Passo 5** | {a, b, c, d, e} | f | 3+5+7+5+2 |
| | {a, b, c, d, e} | g | 3+5+7+5+2 |
| | {a, b, c, d, e} | h | 3+5+7+5+2 |
| **Passo 4** | {a, b, c, d} | e | 3+5+7+5 |
| | {a, b, c, d} | f | 3+5+7+5 |
| | {a, b, c, d} | g | 3+5+7+8 |
| | {a, b, c, d} | h | 3+5+7+2 |
| **Passo3** | {a, b, c} | d | 3+5+7 |
| | {a, b, c} | e | 3+5+4 |
| | {a, b, c} | f | 3+5+2 |
| | {a, b, c} | g | 3+5+3 |
| | {a, b, c} | h | 3+5+1 |
| **Passo 2** | {a, b} | c | 3+5 |
| | {a, b} | d | 3+5 |
| | {a, b} | e | 3+2 |
| | {a, b} | f | 3+3 |
| | {a, b} | g | 3+1 |
| | {a, b} | h | 3+7 |
| **Passo 1** | {a} | b | 3 |
| | {a} | c | 3 |
| | {a} | d | 6 |
| | {a} | e | 9 |
| | {a} | f | 5 |
| | {a} | g | 6 |
| | {a} | h | 4 |
| **Passo 0** | {} | a | 0 |
| | {} | b | 0 |
| | {} | c | 0 |
| | {} | d | 0 |
| | {} | e | 0 |
| | {} | f | 0 |
| | {} | g | 0 |
| | {} | h | 0 |

Definida a limitação de uma classe de problemas específicos a serem utilizados (buscas em grafos), o problema exemplo a ser explorado PCV simétrico

em grafos completos, e a implementação do algoritmo serial, o próximo passo do trabalho é decidir sobre como transformar este algoritmo seqüencial em paralelo e determinar as características necessárias para que a implementação paralela seja eficiente.

## 4.3 Aplicação da Metodologia à Implementação Paralela.

Esta seção demonstra como a aplicação da metodologia para desenvolvimento de projetos paralelos descrita na Seção 1.4, norteou as decisões tomadas na implementação paralela da técnica B&B deste trabalho.

Além disso, são apresentadas as estratégias **erradas** de distribuição de tarefas e comunicação que foram inicialmente implementadas em versões anteriores à versão definitiva apresentada neste trabalho, resgatando assim um pouco do histórico e da evolução a que estão sujeitas o desenvolvimento de projetos paralelos.

### 4.3.1 Particionamento

Na etapa do particionamento é possível adotar uma de três estratégias para decompor o processamento em pequenas tarefas: a) por domínio quando os dados do problemas são divididos entre os processos, b) por decomposição funcional onde partes diferentes do algoritmo são divididas entre os processos, c) uma combinação entre decomposição por domínio e funcional.

O particionamento em B&B é por domínio, como o próprio autor da metodologia sugere e como pode ser amplamente visto em toda a bibliografia encontrada. A técnica B&B, que decompõe um problema maior repetidamente, gerando assim uma árvore da solução de ramos independentes, permite que a

exploração de ramos diferentes possa ser efetivamente executada concorrentemente por diferentes processos.

Uma vez que estava claro que o ganho na paralelização de B&B só seria possível com uma exploração paralela eficiente de ramos diferentes, as mais importantes questões sobre o particionamento em B&B, recaem principalmente sobre as decisões tomadas na distribuição de tarefas (ramos da árvore) entre os processos. As decisões de projeto no particionamento são:

a) Como fazer a atribuição de ramos diferentes aos diversos processos?
b) Como evitar que algum processo fique desocupado?
c) Uma vez feita a distribuição, haverá troca de informações entre os processos a respeito do andamento de suas explorações em seus respectivos ramos?

Neste ponto havia uma outra preocupação que não havia sido encontrada nas referências, mas que parecia, intuitivamente, ser promissora. Seria possível concentrar o esforço de exploração de todos os processos em algum ramo ou ramos da árvore, que se mostrassem mais próximos da solução ótima? Mesmo que para isso todos os processos selecionassem seus subproblemas, segundo o critério adotado para a seleção, em uma mesma fila global de subproblemas?

A única certeza até aquele momento era a necessidade de aproveitar o princípio da localidade, para evitar a transmissão constante de cópias de dados entre os processos. Esse é um alerta explícito na metodologia, e uma evidência constatada nas implementações de bibliotecas genéricas de B&B analisadas durante o levantamento das referências bibliográficas.

Com o escopo de problemas de busca em grafos definido, e por conseqüência a estrutura de dados sendo um grafo, a questão da localidade se tornou uma questão simples de resolver: basta que cada processo tenha sua própria cópia local do grafo a ser processado. Isso se tornou viável por que ocupa-se pouca

memória para armazenar os grafos, podendo-se difundi-los a todos os processos com custo de reduzido.

### 4.3.2 Comunicação

Na etapa da comunicação, procuram-se formas eficientes para coordenar a execução de troca de dados entre as tarefas.

A recomendação mais importante, quando se trata de comunicação, encontrada não só na metodologia, mas em toda a bibliografia consultada sobre paralelização mais genericamente ou de B&B especificamente, recai sobre um princípio básico: a comunicação deve ser a mínima possível.

As perguntas que vêem logo a seguir são: Quanto é esta comunicação mínima? Quais são os parâmetros de mínimo aceitável? O quanto comunicação mínima é representada por volume de dados transmitidos ou por freqüência das transmissões?

Sem estas respostas, mas através de algumas pistas extraídas da metodologia e das referências estudadas, pôde-se estabelecer alguns critérios a serem adotados em pelo menos duas das mais importantes questões sobre a estratégia de comunicação, que não tratam de minimizar a comunicação mas sim do gerenciamento da mesma:

a) A comunicação deveria ser assíncrona.
b) A comunicação deveria ser dinâmica em relação a quantidade de dados a serem transmitidos.

Com a comunicação sendo assíncrona, os processos não precisam ter períodos específicos para se comunicarem (períodos estes que sempre são um parâmetro difícil de estabelecer), além de diminuir as possibilidades de tempo de espera por

congestionamento na comunicação, já que os processos podem se comunicar em tempos distintos uns dos outros.

A comunicação sendo dinâmica, assim como a comunicação assíncrona, não precisa estabelecer um parâmetro difícil de medir (que é a quantidade de dados permitida para transmissão de cada processo), permite uma liberdade maior na implementação, e pode respeitar melhor a capacidade individual de processamento e comunicação de cada processador envolvido.

Estas duas escolhas também são estabelecidas com vistas a garantir maiores possibilidades na escabilidade da implementação, já que não contém critérios que podem se alterar com mudanças ou no tamanho do problema ou no acréscimo do número de processadores.

### 4.3.3 Aglomeração e Mapeamento

Na aglomeração deve-se combinar, se necessário, um número de pequenas tarefas em uma tarefa maior, visando melhores benefícios de desempenho e menores custos de comunicação. No mapeamento deve-se atribuir o conjunto de tarefas aos processadores disponíveis na arquitetura alvo.

Como o particionamento em paralelização de técnicas B&B é por domínio, onde não as tarefas, mas o conjunto de dados deve ser dividido entre os processadores, não há muita discussão sobre decisões a respeito de aglomeração e mapeamento.

Tanto a metodologia como as referências apontam para que cada processo tenha granularidade grossa, ou seja, executem o maior número possível de tarefas evitando comunicação, e que cada processador da arquitetura seja ocupado por apenas um processo.

No mapeamento ainda resta a questão de estabelecer qual processo deve ser executado em qual processador. Na biblioteca PVM, a função que cria os processos

(*pvm_spawn*), possui uma heurística própria denominada *Round-Robin*, que é uma implementação conhecida de mapeamento cíclico, onde os processadores são mantidos em uma lista circular e os processos são distribuídos obedecendo a ordem da lista.

Com estes critérios de particionamento, comunicação, aglomeração e mapeamento estabelecidos conceitualmente, foi implementada uma primeira versão, que deixa de lado em parte a preocupação com a capacidade dos recursos disponíveis na arquitetura, como custos de comunicação, para explorar conceitos do comportamento paralelo de B&B.

Para a primeira implementação, foi estabelecida a criação de dois tipos de processos distintos:

a) Um processo denominado **mestre** que detém o controle sobre todo o processamento, cria outros processos, mantém o melhor valor limite global atualizado e uma lista de tarefas com subproblemas não explorados.

b) Um número variável de processos denominados **escravos** que executam basicamente o mesmo algoritmo serial de *Branch-and-Bound* (Algoritmo 4.2), nos subproblemas recebidos, mantendo um valor limite local e sua própria fila de tarefas.

Com a adoção de um sistema mestre-escravo, e algumas modificações no Algoritmo Síncrono com Fila Centralizada [Qui90] já descrito, foi possível fazer uma implementação que respondesse as questões que ainda persistiam:

- *Como fazer a distribuição de ramos diferentes aos processos escravos?*

O processo mestre gera um número de subproblemas iniciais maior ou igual

ao número de processos escravos e envia um subproblema a cada escravo. Os demais subproblemas permanecem na fila do mestre, aguardando por solicitações dos escravos.

*- Como evitar que algum processo fique desocupado?*

O processo mestre pode manter uma fila de tarefas global e enviar um novo subproblema toda vez que algum escravo ficar desocupado. Se a fila do mestre estiver vazia ele pode solicitar algum subproblema de outros escravos, cujas filas locais estejam com mais de um tarefa.

*- Haverá troca de informações entre os processos, a respeito do andamento de suas explorações sobre seus respectivos ramos?*

O processo mestre pode atualizar sua fila de tarefas com os subproblemas do escravo que tiver encontrado um valor limite melhor do que o valor limite global mantido pelo mestre.

*- Seria possível concentrar o esforço de exploração de todos os processos em algum ramo ou ramos da árvore, que se mostrassem mais próximos da solução ótima?*

Sim, se ao atualizar seu valor limite global o mestre trocar as filas de tarefas dos escravos que estão com um valor limite local pior, por algum subproblema de sua fila global, já atualizada com subproblemas do escravo que encontrou este melhor valor limite atual.

*- Os processos podem selecionar seus subproblemas, segundo o critério adotado para a seleção, em uma mesma fila global de subproblemas?*

Sim, já que o mestre mantém esta fila global.

- *Como aproveitar o princípio da localidade?*

O mestre carrega e transmite aos escravos o grafo do problema. Com cada processo tendo sua própria cópia do grafo, a transmissão de subproblemas resume-se na comunicação dos campos encontrados na estrutura ListaTarefas do Algoritmo 4.2: rótulo do vértice não explorado, caminho percorrido até o vértice e o valor do caminho percorrido.

O autor do Algoritmo Síncrono com Fila Centralizada, alerta que um controle centralizado da fila de tarefas em implementações por passagem de mensagem em arquiteturas *multicomputador*, pode gerar problemas uso excessivo de memória no processo mestre, escravos podem ficar desocupados esperando o mestre reorganizar sua fila de tarefas, e prováveis congestionamentos podem acontecer ao aumentar o número de processos. Para evitar esses efeitos, alguns cuidados foram tomados na implementação.

O primeiro cuidado implementado foi que, diferentemente do Algoritmo Síncrono com Fila Centralizada, o mestre não recebe **toda** a lista de tarefas de **todos** os escravos, num determinado instante sincronizado. A fila global do mestre só é atualizada com subproblemas dos escravos que encontrarem um valor limite melhor que o valor limite global atual e isso não ocorre num tempo determinado, mas somente quando um escravo encontra um valor limite melhor.

Esta implementação funciona, mas o desempenho medido, ao menos nas condições da arquitetura disponível, é muito ruim. Entenda-se por desempenho ruim as medições encontradas de *speedup* menor ou igual a 1,0 (um), usando-se 3 processadores.

Algumas modificações foram implementadas causando melhoras significativas no desempenho deste modelo, tais como:

- O mestre não interromper os escravos para forçar uma troca de subproblemas quando do aparecimento de um valor limite melhor, esta troca só acontece quando cada escravo se comunicar com o mestre, ou por falta de subproblemas ou para avisar sobre o encontro de um valor limite melhor.
- Apenas o mestre poder solicitar ao escravo para iniciar uma comunicação, uma tentativa de evitar escravos competindo ao mesmo tempo pelo canal de comunicação com o mestre.

Com estas modificações, entretanto, ficou claro a possibilidade dos escravos ficarem trabalhando em ramos não promissores da árvore de estados, mesmo quando outro escravo tenha encontrado um valor limite melhor.

As melhoras de desempenho não foram ainda significativas, mas em alguns casos conhecer o comportamento destas abordagens de paralelização de B&B possibilitou identificar os pontos necessários para uma implementação eficiente de B&B paralelo.

Entre os comportamentos analisados destacam-se: o bom funcionamento do princípio da localidade com cada processo tendo sua cópia do grafo, o custo baixo da carga inicial de cada grafo nos processos em relação ao tempo total de processamento, a identificação do comportamento de exploração dos ramos da árvore de estados pelo algoritmo serial e paralelo e o crescimento das listas de tarefas, e principalmente como os elevados custos de comunicação na programação por troca de mensagens em arquiteturas *multicomputador* (ao menos nas utilizadas neste trabalho), afetaram o desempenho total das implementações e ainda o fato de que ter um processo mestre com muitas atividades, eliminava a possibilidade de ter um escravo atuando no mesmo processador.

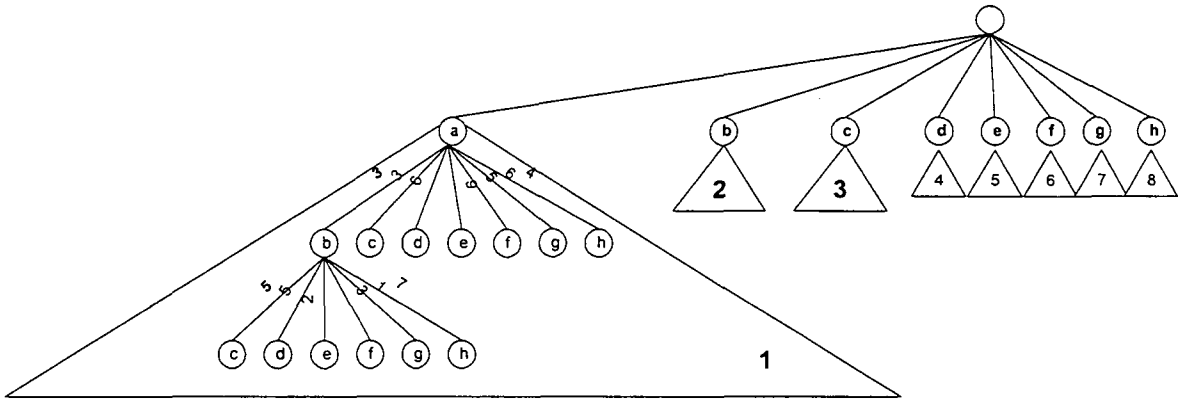
## 4. 4 Projeto e implementação do algoritmo B&B paralelo.

A idéia inicial para se chegar a uma implementação paralela eficiente de uma solução para problemas de buscas em grafos com a técnica B&B com PVM, parte do princípio de primeiro conhecer o comportamento na exploração da árvore de estados gerada pelo algoritmo B&B nesta classe de problemas, para em seguida usar estas características de comportamento para criar uma estratégia de paralelização que evite sobrecargas na distribuição de tarefas e na comunicação entre processos e que simplifiquem ao máximo as atribuições do processo mestre, permitindo assim, que um processo escravo concorra pelo mesmo processador.

A decisão foi adotar uma estratégia de paralelização que tornasse possível uma implementação eficiente do conceito de exploração de ramos diferentes da árvore de estados concorrentemente.

Na Figura 4.4, pode-se ver o conceito básico de explorar concorrência na técnica B&B que inspira esta implementação, onde os triângulos representam os ramos da árvores a serem explorados.

**Figura 4.4 – Comportamento do PCV em B&B por profundidade**




**Ordem de exploração dos ramos da árvores em seleção por profundidade: 1, 2, 3, 4, 5, 6, 7, 8.**

Numa versão serial da técnica B&B como a do Algoritmo 4.2, com a regra de seleção da fila de tarefas em profundidade, primeiro são explorados todos os nós do ramo 1 da árvore (que começa com o vértice **a**), para só então começar a exploração do ramo 2 (que começa com o vértice **b**).

Em uma implementação paralela, também por profundidade, **n** processos podem explorar **n** ramos ao mesmo tempo, se a cada processo for enviado como subproblema inicial o nó origem de cada ramo (no exemplo, os vértices **a**, **b**, **c**, até o vértice **h**).

Uma diferença importante entre a versão serial e a paralela, seria que na versão serial, quando chegar a vez de explorar o ramo 2 da árvore, o valor limite até este momento, será o melhor valor limite encontrado em todo o ramo 1. Enquanto na versão paralela, um processo pode ter explorado todo o ramo 2 sem descartar caminhos com valores que talvez já fossem piores dos que os encontrados no ramo 1. Pode-se imaginar uma situação particular em que, por exemplo, o

melhor valor limite ótimo do ramo 1, já seja melhor que todos os primeiros nós do ramo 2, nesta situação a versão serial teria, levando-se em consideração o tempo para criação de processos, inicialização de dados, entre outros, um desempenho provavelmente melhor que uma versão paralela com 2 processos.

Há portanto uma questão importante a resolver na paralelização da técnica B&B que é: a possibilidade dos processos que estão executando concorrentemente o algoritmo B&B ficarem informados sobre o melhor valor limite no menor tempo possível depois que ele foi encontrado, tem um impacto melhor na eficiência da paralelização do que ter dois ou mais processos explorando concorrentemente o mesmo ramo na árvore de estados, como verificado com os enganos cometidos nas versões paralelas anteriores a esta implementação. A importância dos processos saberem qual é o melhor valor limite global entre todos os processos concorrentes, pode significar grandes economias de tempo de processamento pela possibilidade de não explorar subproblemas desnecessariamente.

Duas possibilidades podem resolver o problema de divulgar os melhores valores limites globais, num esquema mestre-escravo, aos processos que estão executando o algoritmo B&B nos diversos ramos da árvore de estados: a) um esquema **centralizado** onde cada processo pode enviar seu valor limite ao processo mestre que se encarrega de divulgar o valor atualizado entre os outros, ou b) um esquema **distribuído** onde todos os processos escravos podem divulgar seus melhores valores limites locais entre si.

### 4.4.1 Implementação Paralela de B&B

Nesta seção são apresentados os algoritmos implementados para a solução paralela de B&B deste trabalho.

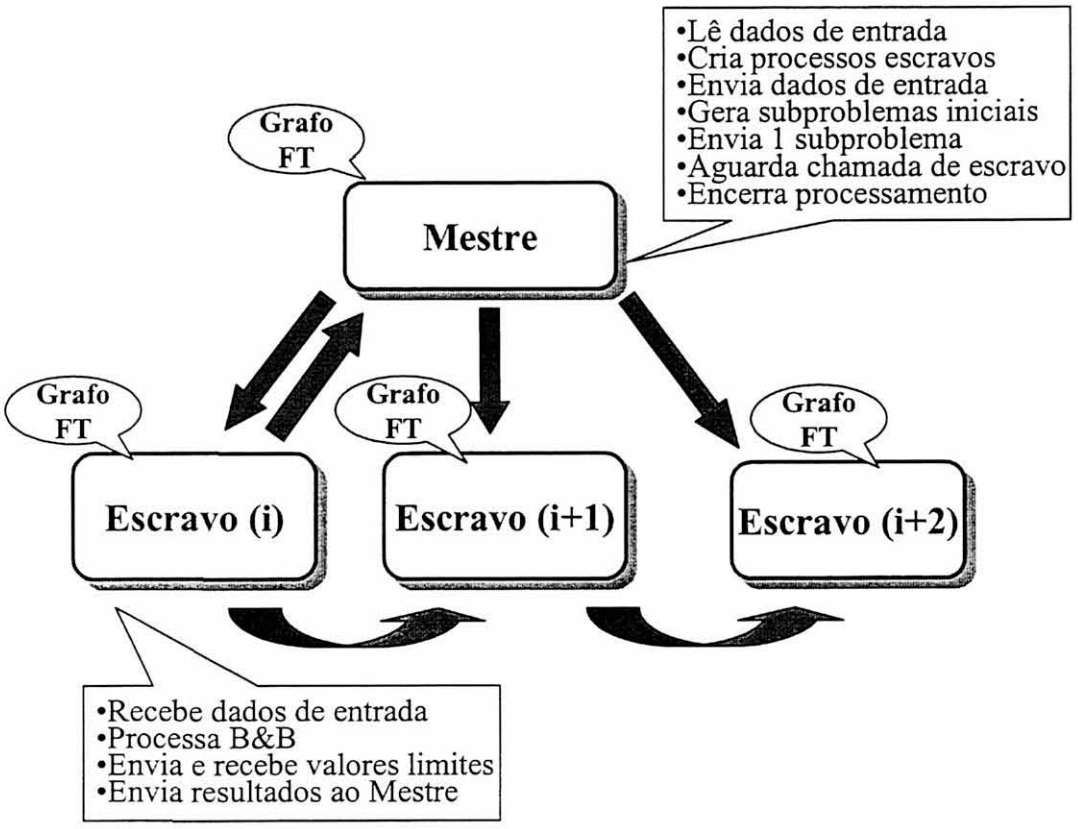
É mantido o esquema mestre-escravo, mas restringindo o trabalho do mestre à inicialização do sistema, distribuição de tarefas iniciais aos escravos, verificação de término e impressão de resultados finais.

Aos processos escravos é reservado o trabalho de realizar o algoritmo

*Branch-and-Bound* nos ramos da árvore de estado e compartilhar a atualização de um valor limite melhor, quando encontrado, com os demais processos escravos do sistema.

A Figura 4.5, mostra o comportamento do processo mestre interagindo com os processos escravos. Cada processo mantém um cópia do grafo (Grafo) assim como sua fila local de tarefas (FT), desta maneira os dados transmitidos na comunicação entre os processos se limita aos números inteiros que representam os rótulos dos vértices, os caminhos percorridos e do valor limite.

**Figura 4.5 – Implementação Paralela de B&B**



O esquema geral de funcionamento do processo mestre é implementado conforme as etapas descritas no algoritmo 4.3 descrito a seguir:

**Algoritmo 4.3 - Processo Mestre**(G,Vi)

**Entrada**: Um grafo G.
Um conjunto de vértices iniciais Vi, |Vi| > 0

**Cria N** processos Escravos
**Carrega** Grafo de entrada e **Envia** aos processos Escravos

valor limite global ← ∞
solução ótima ← {}
**Para** cada vértice v(i) de Vi
ListaTarefasGlobal ← {vazio, v(i), 0}

**Para** cada escravo e(i) de **N**
**tarefa** ← (retira um elemento de ListaTarefasGlobal)
**Envia tarefa.v** para e(i)

**Enquanto** ListaTarefasGlobal não estiver vazia **E** escravos tiverem tarefa
**Se Recebeu tarefa** de algum escravo
solução ← **tarefa**.valor limite
**Se** solução < valor limite global
valor limite global ← solução
solução ótima ← **tarefa**
**Se** ListaTarefasGlobal <> {vazio}
**tarefa** ← (retira um elemento de ListaTarefasGlobal)
**Envia tarefa** para e(i)
**Envia** ao processos Escravos solicitação de fim de processamento
**Imprima solução ótima**.V e valor limite global

O processo mestre segue a seguinte seqüência de procedimentos: ler o arquivo de entrada, criar processos escravos, enviar grafo de entrada para processos escravos, guardar em sua lista de tarefas os vértices iniciais, enviar a cada escravo um vértice inicial, aguarda a chamada de algum escravo, receber a solução encontrada pelo escravo que fez a chamada, atualizar sua solução global, enviar

outro vértice inicial de sua lista de tarefas para o escravo solicitante. O processo mestre envia uma indicação de fim de processamento a todos os escravos, imprime a solução ótima final e encerra seu processamento, quando todos os escravos estão sem subproblemas e a lista de tarefas do mestre está vazia.

A implementação do processo escravo é descrita no algoritmo 4.4 a seguir:

**Algoritmo 4.4 - Processo Escravo()**

**Recebe** Grafo do processo Mestre
valor limite local ← ∞
solução ótima ← {}
lista de processos escravos (**e**) ← {}
**Recebe tarefa.v** do processo Mestre
**Recebe** lista de processos escravos (**e**) do processo Mestre
ListaTarefasLocal ← {vazio, **tarefa**.v, 0}
fim-de-processamento ← falso
**Enquanto Não** for fim-de-processamento
    **Se** ListaTarefasLocal <> {vazio}
        **tarefa** ← (retira um elemento de ListaTarefasLocal)
        solução ← **tarefa**.valor limite
        **Se** solução < valor limite local
            **Se tarefa**.V é uma condição de término
                valor limite local ← solução
                solução ótima ← **tarefa**
                **Envia** valor limite local para Escravo e(i+1)
            **Senão**
                Para cada vértice *u* adjacente a **tarefa**.v
                    ListaTarefasLocal←ListaTarefasLocal +
                        { V+**tarefa**.v, *u*,**tarefa**.valorlimite +
                        p(*u*,**tarefa**.v) }
    **Senão**
        **Envia** solução ótima para Mestre

    **Se Recebeu** algo de algum processo
        **Se Recebeu** solução de Escravo e(i-1)
            Se solução < valor limite local
                valor limite local ← solução
                **Envia** valor limite local para Escravo e(i+1)
        **Se Recebeu tarefa** do processo Mestre
            ListaTarefasLocal ← {vazio, **tarefa**.v, 0}
        **Se Recebeu** fim-de-processamento do processo Mestre
            fim-de-processamento ← verdadeiro

A seqüência de procedimentos do processo escravo é: logo que é criado pelo processo mestre o processo escravo recebe o grafo e um vértice inicial que inclui em sua lista de tarefas local para começar sua exploração, inicia a execução de seu algoritmo *Branch-and-Bound* serial e a cada valor limite local melhorado envia este valor para o processo escravo vizinho, se sua lista de tarefas ficar vazia envia sua solução ótima local para o processo mestre indicando que está sem tarefas. A cada interação do algoritmo (que significa retirar um subproblema da lista de tarefas e processá-lo), o processo escravo verifica se há alguma mensagem a ser recebida de outro processo. Se for o envio de um valor limite por outro processo escravo, verifica se esse é melhor que o seu valor limite atual, se for, atualiza seu valor limite e envia essa informação para seu processo escravo vizinho. Se for um novo vértice enviado pelo processo mestre, atualiza sua fila de tarefas e continua a exploração a partir do vértice enviado. Se for um aviso de fim de processamento enviado pelo mestre, simplesmente encerra seu processo.

Muitas características importantes em relação à esta implementação, estão omitidas na descrição da forma geral dos algoritmos dos processos Mestre e Escravo para não poluir com muitas informações extras a compreensão do leitor sobre a estratégia geral de funcionamento desta solução proposta.

Todas essas informações relevantes sobre a implementação dos algoritmos descritos, são detalhadas na seção seguinte deste capítulo.

### 4.4.2 Considerações sobre a implementação

Esta seção aborda algumas considerações sobre a implementação do processo mestre descrito no Algoritmo 4.3 e do processo escravo descrito no Algoritmo 4.4. A apresentação inicia com descrições sobre detalhes da implementação que caracterizam o sistema paralelo proposto e encerra com a discussão das três questões mais importantes sobre o uso da biblioteca PVM neste trabalho, que tem uma influência decisiva sobre o desempenho da solução paralela implementada.

#### Opções do execução

O usuário pode, através da linha de comando, configurar alguns parâmetros que determinam características de comportamento, tanto da implementação serial como da paralela. Os parâmetros semelhantes as duas implementações são: **o nome do arquivo** de entrada (grafo do problema), o **método de seleção** a ser adotado nas operações da ListaTarefas (se profundidade = 0, largura = 1 ou melhores primeiro =2), e se o sistema deve **calcular** (=1) ou não (=0) um valor limite inicial.

Na implementação paralela, há ainda, dois parâmetros adicionais: o **número de processos escravos** a serem criados pelo processo mestre e um **número** que define o número de operações de retirada de elementos da ListaTarefas a serem realizados antes de verificar o recebimento de mensagens, a razão deste parâmetro é descrita na seção **A função *pvm_probe*()** deste capítulo.

Os parâmetros que definem o método de seleção e o número de interações, são enviados aos processos escravos, pelo processo mestre, junto com a transmissão do arquivo de entrada.

### A carga do grafo

A carga do arquivo que representa o grafo do problema e os rótulos dos vértices iniciais, é feita pela leitura do arquivo de entrada pelo processo mestre que o envia linha por linha à todos os processos escravos do sistema, através da função de biblioteca *pvm_mcast()*. A biblioteca PVM disponibiliza a função chamada de *pvm_mcast*(), que implementa a difusão (*multicasting*) de uma mensagem para um conjunto de processos. Com uma transmissão do tipo **1** (mestre) **para N** (escravos), o uso da função *pvm_mcast* não representou nenhum problema.

### Cálculo do valor limite inicial

Se o usuário não optar pelo cálculo de um valor limite inicial, este valor é inicializado com 30.000 (infinito, para efeitos práticos é um valor muitas vezes maior que os possíveis valores limites do problema). Se o usuário optar pelo cálculo de um valor limite inicial, este é realizado da seguinte forma: o processo mestre, durante a carga do grafo problema, soma os valores das arestas de um caminho qualquer de tamanho |V|+1, e transmite este valor juntamente com a descrição do grafo aos processos escravos.

### Estruturas de Dados Grafo e ListaTarefas

Um grafo G é implementado como um vetor G de |V| listas. Para cada $v \in \{1...|V|\}$, G[$v$] é uma lista contendo todas as arestas adjacentes ao vértice $v$. Esta implementação é usualmente conhecida como **lista de adjacência** de G. Todas as operações necessárias para operações elementares em grafos: inclusão e exclusão de vértices e arestas, pesquisa e outras, são também implementadas no programa.

A estrutura denominada ListaTarefas, que armazena os subproblemas não

explorados durante a execução do algoritmo *Branch-and-Bound*, é implementada como uma lista encadeada de estruturas do tipo **tarefa** composta por rótulo do vértice ($v$) não explorado, vetor de rótulos que representa o caminho até ($v$) e um valor inteiro que representa o tamanho do caminho percorrido. A estrutura ListaTarefas aceita inclusões do tipo FIFO, LIFO e de forma ordenada.

Estas implementações do grafo e de ListaTarefas são idênticas nas versões serial e paralela.

### Geração de tarefas iniciais

Eventualmente, pode haver menos vértices iniciais no problema do que o número de escravos ativos no sistema. O processo mestre nestes casos gera o número de particionamentos (*branch*), sobre os vértices iniciais, até que o tamanho de sua ListaTarefasGlobal seja igual ou maior que o número de processos escravos do sistema.

### A função *pvm_probe*()

Se a versão serial examina cada ramo da árvore a partir de cada vértice de origem do PCV, e na versão paralela pode-se atribuir ramos diferentes aos processadores envolvidos examinarem ao mesmo tempo, a expectativa para esta nova abordagem na implementação, que reduzia ao máximo as atividades processo mestre e o volume na comunicação, era de se obter um *speedup* próximo do ideal.

Contudo, ao se repetirem os testes da versão paralela anterior, constatou-se que o desempenho, apesar de uma ligeira melhora, continuava totalmente insatisfatório com um *speedup* em torno de 1 (um).

O que havia acontecido? Se o problema era comunicação, por que então a

melhora de desempenho não foi proporcional a sua redução? E, se ainda assim, o problema for comunicação como diminuir ainda mais seu volume?

Antes de sair em busca de outras otimizações, voltou-se a atenção ao comportamento do particionamento da árvore de estados nos problemas examinados. A abordagem paralela implementada pode ser vista como **n** implementações da versão serial sendo executadas em **n** processadores independentes, com os vértices iniciais de cada ramo da árvore sendo divididos entre estes **n** processos seriais. Poderia o custo de criar estes **n** processos e a transmissão de alguns números inteiros ter um impacto tão significativo no desempenho global?

Excetuando-se a comunicação dos vértices de início do mestre aos escravos (um número inteiro solicitado pelo escravo apenas quando está sem tarefas), e a transmissão de um processo escravo a outro de seu valor limite local melhorado (um inteiro) quando encontrado, a implementação do algoritmo *Branch_and-Bound* entre a versão serial e paralela são idênticas?

A menos das chamadas de comunicação, as versões serial e paralela são idênticas, com apenas outra sutil diferença: na versão paralela a cada interação do algoritmo, há uma verificação de recebimento de alguma mensagem enviada por outro processo, linha abaixo no algoritmo 4.4:

**Se Recebeu** algo de algum processo

Esta operação pode ser implementada pela função da biblioteca PVM denominada *pvm_probe*(). Após cuidadosa análise, constatou-se que esta verificação de recebimento de mensagens era responsável pelo fraco desempenho da versão paralela, pois há um alto custo de tempo na execução desta função.

A função:

int <identificador> ***pvm_probe***(int <processo>, int <tipo>),

Verifica se um determinado processo (um número <processo> para um processo específico ou –1 para qualquer processo), enviou um determinado tipo de mensagem (um número <tipo> para um tipo específico ou –1 para qualquer tipo de mensagem), se há uma mensagem para ser desempacotada e entregue, a função retorna um número <identificador> que identifica esta mensagem, senão retorna 0 (zero) quando não há nenhuma mensagem a ser recebida, ou –1 se ocorreu algum erro.

A documentação disponível consultada sobre o PVM, não menciona os detalhes de a implementação desta função ou custos envolvidos. A outra função denominada de *pvm_nrecv*(), que pode ser usada como substituição à função *pvm_probe*(), produziu os mesmos resultados.

A solução encontrada foi aumentar a granularidade do processamento do algoritmo *Branch-and-Bound* propriamente dito, verificando o recebimento de mensagens não mais a cada interação do algoritmo, mas apenas depois de um número **n** de interações. Denominamos este retardo na verificação de mensagens de **probe_delay**. Cada interação do algoritmo representa uma operação de retirada de um elemento da ListaTarefas a ser analisado, a variável **probe_delay**, passa a definir o número de operações que serão realizadas antes da verificação de mensagens recebidas com o uso da função *pvm_probe*().

Os detalhes da determinação de um valor ideal para o ***probe_delay***, serão apresentados no **Capítulo 6** do texto.

## A função *pvm_spawn*()

Durante os testes de validação da versão paralela, um efeito começou a se fazer presente: muitas vezes, ao executar a mesma compilação do programa, com o mesmo arquivo de entrada e os mesmos parâmetros de execução, havia uma variação de até 50% no tempo total de execução.

Como os testes estavam sendo realizados em 3 estações de trabalho distantes entre si (três salas separadas do laboratório), e de não estarem disponíveis com exclusividade para estes testes, iniciou-se um acompanhamento para identificar se a causa seria a possibilidade de processos de outros usuários estarem competindo pelos processadores.

O que se constatou curiosamente foi que, vezes sim, vezes não, a função da biblioteca PVM responsável pela criação de processos e mapeamento destes processos nos processadores disponíveis, a função *pvm_spawn()*, não operava como se esperava inicialmente.

Com a disponibilidade de 3 processadores para estes testes, e o processo mestre tendo suas tarefas reduzidas, criava-se 3 processos escravos, imaginado-se que cada um seria alocado em cada uma das estações de trabalho. Uma das estações teria o processo mestre e mais processo escravo, e as outras duas, um processo escravo cada. Freqüentemente porém, o que acontecia era termos uma estação com o processo mestre e 2 (dois) processos escravos, outra estação com o outro processo escravo e a outra estação sem processos. O mapeamento estava ineficiente e aleatório.

A consulta à documentação da biblioteca PVM, informa que a função *pvm_spawn*(), utiliza a heurística denominada Round-Robin de mapeamento cíclico começando a atribuição de processos com a próxima estação de sua tabela, o que atenderia nossas necessidades. Porém uma leitura mais atenciosa confirma que depois de algumas execuções em um mesmo *daemon* ***Pvmd*** (os aplicativos *runtime*

do PVM que controlam a comunicação entre os processadores envolvidos) o PVM irá usar métricas de desempenho retiradas das estações envolvidas para fazer o mapeamento dos processos. Não por coincidência, as 3 estações utilizadas são de capacidades diferentes, e a que recebia uma carga maior de processos, a mais potente delas.

Identificado o não determinismo no mapeamento, quando utilizado a heurística interna da biblioteca PVM, implementou-se um mapeamento cíclico no processo mestre que garante uma distribuição homogênea de processos escravos entre os processadores.

**Difusão dos valores limites**

Entre as opções de um esquema **centralizado** ou **distribuído**, de difundir os melhores valores limites encontrados durante a execução do sistema paralelo, está implementada na versão paralela deste trabalho, um esquema distribuído em que os processos escravos trocam seus valores limites locais entre si.

Com o uso da função *pvm_cast*(), seria possível que cada processo escravo, toda vez que melhorasse seu valor limite local, envia-se essa informação a todos os outros processos escravos ativos no sistema.

No Algoritmo 4.4, a linha:

**Envia** valor limite local para Escravo e(i+1)

Poderia ser trocada por uma linha tal como:

**Envia** valor limite local para lista de escravos (**e**)

Optou-se entretanto na implementação desta versão paralela, na criação de

uma lista circular de processos escravos, onde cada processo escravo ao melhorar seu valor limite local, transmite esse valor somente ao seu processo escravo vizinho da direita na lista. Esse por sua vez, ao receber algum valor limite que seja melhor que o seu, também repassa esse valor ao seu próximo vizinho da direita. Desta forma, garante-se que todos os processos escravos estarão com seus valores limites locais atualizados sempre pelo melhor valor durante a execução do programa.

Esta decisão de transmissão **1 para 1**, foi tomada para reduzir o risco de um possível aumento no volume de comunicações que poderia gerar o fato de vários processos escravos estarem transmitindo seus valores limites locais num esquema **1 para N** simultaneamente.

Durante a inicialização dos processos escravos, o processo mestre cria esta lista circular com um número inteiro que identifica um processo para o PVM, e transmite o número do processo vizinho a cada processo escravo.

# CAPÍTULO 5
# ANÁLISE E TESTES

Este capítulo apresenta os testes de comportamento da técnica B&B e o desempenho alcançado em sua paralelização. Inicialmente é apresentada a arquitetura utilizada na realização dos testes e os arquivos de grafos utilizados para resolução do PCV. A seguir é abordado o comportamento da técnica B&B em problemas de busca em grafos e finalmente é discutido o desempenho alcançado na paralelização da técnica.

## 5.1 Arquitetura disponível

Estes testes foram realizados numa arquitetura paralela do tipo *multicomputador*, composta de 3 estações do tipo PC, uma com processador de modelo Pentium II de 300 MHz e 128 Mbytes de RAM, outra com Pentium II 300 MHz com 64 Mbytes de RAM e a terceira com processador de modelo K6 330 e 64 Mbytes de RAM, interligadas por Ethernet em par trançado categoria 5 e placas de interface de rede de 100 Mbits de diferentes fabricantes. A versão do sistema operacional Linux em uso é a 2.0.33 e a versão da biblioteca PVM é a 3.3.11 em todas as máquinas.

Os programas foram compilados com gcc versão 2.95.2, com a opção de otimização -O2.

Todos os testes realizados na versão paralela, usaram sempre as 3 estações descritas, com **1** processo **mestre** e **3** processos **escravos**.

## 5.2 Arquivos de testes

Foram gerados 4 arquivos diferentes de grafos completos para realização dos testes, denominados: solução fácil (SF), solução aleatória (SA), um caso com valores reais (SR) e ainda um arquivo com um mapa de distâncias rodoviárias reais do estado do Rio Grande do Sul (RGS).

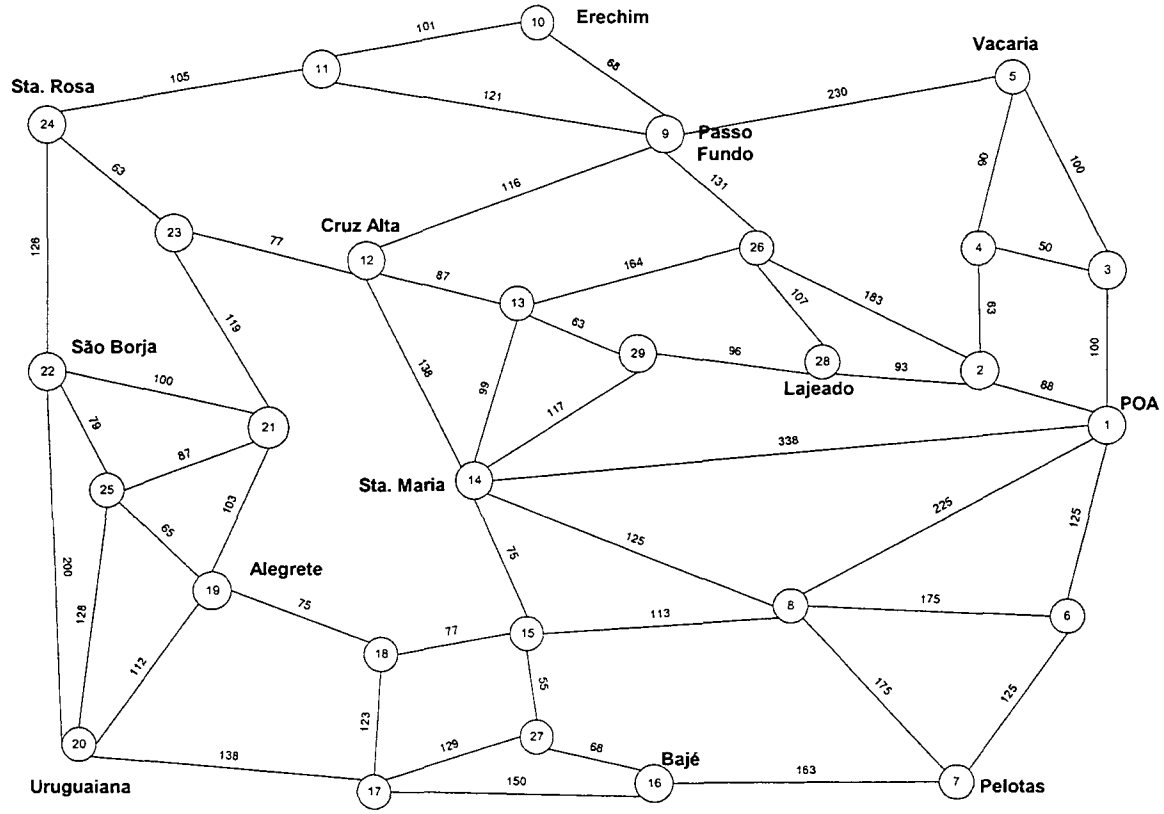
Os arquivos gerados para representar a solução fácil (SF) tem o peso de suas arestas variando entre os inteiros 1 e 9 mas somente as arestas ($v_0$-$v_1$, $v_1$-$v_2$, $v_2$-$v_3$, ..., $v_{n-2}$-$v_{n-1}$, $v_{n-1}$-$v_0$) que representam os vértices do grafo em ordem crescente, tem o valor igual a 1 (um). Desta forma a solução ótima para o PCV será sempre o caminho formado pelo ramo extremo esquerdo na árvore de estados, ou seja, é a primeira solução encontrada por uma busca em profundidade, vértices ($v_0$,$v_1$,$v_2$,$v_3$,...$v_{n-1}$,$v_0$).

Nos arquivos que representam a solução aleatória (SA), todas as arestas tem pesos que variam entre os inteiros 1 e 9. Nota-se que essa estreita faixa de variação gera caminhos de pesos diferentes mas numericamente muito próximos, o que conduzem à exploração dos nós da árvore de estado, pelo algoritmo B&B, até muito próximo das folhas.

No caso em que são usados valores reais (SR), o valor das arestas é a distância em quilômetros que separam capitais brasileiras. Nota-se que, como a análise é feita sobre grafos completos, mesmo para pares de cidades que não possuem rodovias diretas entre si (Curitiba-Fortaleza), têm em suas arestas a distância entre elas ao invés de uma distância infinita. A solução do PCV nestes casos poderia representar, por exemplo, uma solução para rotas aéreas de vôos diretos.

A Figura 5.1, mostra o grafo incompleto que representa uma mapa do estado brasileiro do Rio Grande do Sul com 29 (|V|=29) de suas principais cidades interligadas por suas rodovias (|A|=53). Este mapa foi escolhido por que a malha rodoviária cobre todo o estado de forma relativamente homogênea.

**Figura 5.1 – Grafo de 29 cidades do estado do Rio Grande do Sul**




**Distância em quilômetros rodoviários aproximados.**

## 5.3 Comportamento da estrutura ListaTarefas

A estrutura de dados ListaTarefas, armazena os subproblemas a serem explorados durante a execução do algoritmo *Branch-and-Bound.* Nesta implementação da técnica para busca em grafos, um subproblema é um vértice não explorado do grafo que representa um nó na árvore de estados. O tipo **tarefa** tem **92 bytes** de tamanho e armazena cada subproblema na lista encadeada ListaTarefas.

Para efeito de comparação é apresentado na Tabela 5.1, o comportamento da lista encadeada ListaTarefas em três formas possíveis da regra de seleção da técnica *Branch-and-Bound.* Nessa tabela, as duas colunas de cada estratégia mostram, (a) o tamanho máximo (TM) que a ListaTarefas atingiu na execução do programa, e (b) o espaço de memória (EM), em bytes, necessário para armazenar o tamanho máximo que a ListaTarefas atingiu. O valor de EM é obtido por TM* 92.

Os valores das Tabelas 5.1 e 5.2, foram obtidos com a execução da versão serial da implementação da técnica *Branch-and-Bound* e com o cálculo de um valor limite inicial.

**Tabela 5.1: Tamanho máximo da ListaTarefas em diferentes estratégias de seleção**

| Arquivo | | Profundidade | | Melhores Primeiro | | Largura | |
|---|---|---|---|---|---|---|---|
| \|V\| | Tipo | TM | EM | TM | EM | TM | EM |
| 5 | SA | 11 | 1.012 | 100 | 9.200 | 120 | 11.040 |
| 6 | SA | 16 | 1.472 | 521 | 47.932 | 720 | 66.240 |
| 7 | SA | 22 | 2.024 | 2.225 | 204.700 | 4.720 | 434.240 |
| 8 | SA | 29 | 2.668 | 8.136 | 748.512 | 37.968 | 3.493.056 |
| 9 | SA | 36 | 3.312 | 27.123 | 2.495.316 | 247.479 | 22.768.068 |
| 10 | SA | 44 | 4.048 | 79.568 | 7.320.256 | 1.449.737 | 133.375.804 |
| 11 | SA | 56 | 5.152 | 510.814 | 46.994.888 | **memória** | **insuficiente** |

Os resultados obtidos indicam que, para efeitos de utilização de memória, a *busca em profundidade* é a melhor heurística para a regra de seleção em algoritmos *Branch-and-Bound* na solução de problemas de buscas em grafos. O número

máximo de subproblemas na lista de tarefas será sempre menor ou igual a |A|+1, para um grafo completo. Já com as estratégias de busca *melhores primeiro* e *largura*, apesar de não terem um comportamento determinístico, seus valores apontam para um comportamento de crescimento polinomial e exponencial respectivamente. Em ambos os casos a memória é exaurida rapidamente. Note que a lista de tarefas pode sofrer grandes variações de tamanho dependendo da qualidade do valor limite inicial estabelecido e da faixa de valores das arestas do grafo.

A Tabela 5.2 mostra o número total de subproblemas (TS) inclusos na ListaTarefas durante a execução do programa com as 3 estratégias de busca. Este número é o total de nós explorados na árvore de estados pelo algoritmo *Branch-and-Bound*.

**Tabela 5.2: Número total de subproblemas explorados em diferentes estratégias de seleção**

| Arquivo | | Profundidade | Melhores Primeiro | Largura |
|---|---|---|---|---|
| \|V\| | Tipo | TS | TS | TS |
| 5 | SA | 185 | 185 | 319 |
| 6 | SA | 893 | 903 | 1.836 |
| 7 | SA | 3.889 | 3.424 | 11.001 |
| 8 | SA | 14.434 | 11.696 | 94.792 |
| 9 | SA | 47.595 | 37.289 | 629.793 |
| 10 | SA | 126.975 | 104.296 | 4.043.626 |
| 11 | SA | 580.106 | 654.873 | **memória insuficiente** |

Enquanto as buscas *profundidade* e *melhores primeiro* se aprofundam na pesquisa na árvore de estados, de modo a encontrar e melhorar rapidamente seus valores limites o que evita a exploração futura de ramos não promissores, a busca em *largura* expande completamente cada nível da árvore e determinará seus valores limites somente quando explorar o nível das folhas da árvore, quando então todos os valores limites serão encontrados um após o outro.

Apesar de haver uma tendência de crescimento no tamanho da ListaTarefas nas diferentes estratégias de busca, seus tamanhos podem sofrer uma grande variação dependendo das características do grafo explorado. Em casos extremos pode-se imaginar um grafo onde o primeiro caminho encontrado na busca em profundidade é o caminho ótimo, similar aos grafos do tipo Solução Fácil (SF) e por outro lado, grafos em que a única diferença no tamanho dos caminhos ocorre somente nas nós folhas da árvore, quando independentemente do método de busca, todos os nós da árvore serão explorados, e o comportamento da técnica B&B terá um desempenho semelhante ao comportamento da técnica de *Backtracking*.

O método de seleção deve ser, portanto, cuidadosamente escolhido, de acordo com as características do problema a ser implementado, levando-se em consideração: a) facilidade com que podem encontrar valores ótimos intermediários que evitem a exploração de nós desnecessariamente, b) os custos de manutenção dos subproblemas na fila de tarefas e c) os recursos de memória consumidos para armazenar esta fila de tarefas. É possível ainda considerar métodos diferentes de seleção que se alternam durante a execução do programa, como por exemplo começar a exploração por profundidade até que se encontre uma valor limite e continuar com melhores primeiro ou largura.

## 5.4 Cálculo de um valor limite inicial

A Tabela 5.3, mostra uma comparação entre o total de subproblemas (TS) explorados quando é realizado o cálculo (TS-com) de um valor limite inicial ou é empregado o valor de 30.000 (TS-sem).

**Tabela 5.3: Subproblemas explorados com e sem o cálculo de um valor limite inicial.**

| Arquivo | | Profundidade | | Melhores Primeiro | | Largura | |
|---|---|---|---|---|---|---|---|
| \|V\| | Tipo | TS-com | TS-sem | TS-com | TS-sem | TS-com | TS-sem |
| 5 | SA | 185 | 185 | 185 | 185 | 120 | 325 |
| 6 | SA | 893 | 896 | 903 | 903 | 720 | 1.956 |
| 7 | SA | 3.889 | 3.893 | 3.424 | 3.424 | 4.720 | 13.699 |
| 8 | SA | 14.434 | 14.434 | 11.696 | 11.696 | 37.968 | 109.600 |
| 9 | SA | 47.595 | 47.757 | 37.289 | 37.289 | 247.479 | 986.409 |
| 10 | SA | 126.975 | 126.993 | 104.296 | 104.296 | 1.449.737 | **mem. insufic.** |
| 11 | SA | 580.106 | 580.106 | 654.873 | 654.873 | **memória** | **insuficiente** |

Calcular um valor limite inicial mais aproximado à solução ótima do problema, ao contrário de iniciar com um valor suficientemente alto poderá ter vantagens nos casos em que: a) os valores das arestas do grafo estiverem numa faixa bastante ampla criando caminhos de tamanhos significativamente diferentes, ou b) quando se pretende usar busca em largura como estratégia de seleção na fila de tarefas, já que a busca em largura encontra todos os valores da solução no mesmo nível de exploração da árvore e ter um valor limite aproximado durante a execução do algoritmo B&B representa, na maioria dos casos, cortes significativos na árvore de busca.

## 5.5 Tempo de carga do sistema

A Tabela 5.4, mostra o tempo de carga do grafo e dos vértices inicias na versão serial, e na versão paralela este tempo de carga mais o tempo de criação e transmissão do grafo e dados iniciais aos processos escravos.

**Tabela 5.4: Tempo de Carga do Sistema em milésimos de segundos.**

| Arquivo | | Serial | | Paralelo | |
|---|---|---|---|---|---|
| \|V\| | Tipo | **sem** valor limite início | **com** valor limite início | **sem** valor limite início | **com** valor limite início |
| 5 | SA | 27 | 33 | 56 | 146 |
| 6 | SA | 28 | 30 | 29 | 146 |
| 7 | SA | 24 | 31 | 96 | 45 |
| 8 | SA | 16 | 19 | 50 | 71 |
| 9 | SA | 31 | 36 | 98 | 66 |
| 10 | SA | 23 | 26 | 56 | 154 |
| 11 | SA | 32 | 40 | 178 | 73 |
| 12 | SA | 19 | 24 | 115 | 300 |
| 13 | SA | 26 | 34 | 205 | 104 |
| 14 | SA | 25 | 32 | 201 | 196 |
| 15 | SA | 31 | 40 | 267 | 364 |
| 16 | SA | 26 | 33 | 230 | 196 |
| 17 | SA | 31 | 34 | 113 | 206 |
| 18 | SA | 29 | 45 | 566 | 307 |
| 19 | SA | 32 | 38 | 242 | 423 |
| 20 | SA | 26 | 33 | 280 | 198 |
| **Média** | | **27** | **33** | **174** | **187** |

Na versão paralela, a carga inicial do sistema mostrou-se de 5,5 a 6 vezes mais lenta que a versão serial com o uso de 1 processo mestre e 3 escravos. Descontando o valor de leitura do grafo pelo processo mestre, de 82 a 85% do tempo de inicialização da versão paralela é usado para criação e transmissão de dados aos escravos.

Foram efetuadas medidas realizando o cálculo do valor limite inicial e sem este cálculo, mas as diferenças encontradas não são significativas.

Excetuando-se os grafos muito pequenos, de 5 a 10 vértices, o tempo total consumido para iniciar o sistema mostra-se insignificante no cômputo do tempo

total de execução do sistema, tanto em sua versão serial como em sua versão paralela. Na versão paralela este baixo custo na carga inicial do sistema torna-se de grande importância para o desempenho eficiente da solução implementada, uma vez que transmitido o grafo aos processos escravos não há mais requisições de dados do problema durante a execução do programa, garantindo um aproveitamento máximo do princípio da localidade.

## 5.6 Desempenho da função *pvm_probe*()

O uso da função da biblioteca PVM, *pvm_probe*(), tem um impacto importante no total do tempo de execução do sistema paralelo com discutido em 4.4.2.

A Tabela 5.5 compara o tempo de execução da versão serial e a mesma versão serial com a inclusão de uma chamada da função *pvm_probe*() em cada interação do algoritmo *Branch-and-Bound*. Incluir uma chamada à função *pvm_probe*() no algoritmo serial permite isolar os custos associados ao tempo de execução desta função.

**Tabela 5.5: Tempo de execução da versão serial por profundidade sem e com a função *pvm_probe*()**

| Arquivo | | Serial s/ *pvm_probe*() | Serial c/ *pvm_probe*() |
|---|---|---|---|
| \|V\| | Tipo | tempo em ms | tempo em ms |
| 10 | SF | 61 | 262 |
| 11 | SF | 222 | 932 |
| 12 | SF | 1.576 | 6.868 |
| 13 | SF | 4.461 | 19.126 |
| 14 | SF | 10.294 | 44.172 |
| 15 | SF | 21.330 | 91.540 |
| 16 | SF | 45.474 | 192.590 |
| 10 | SA | 716 | 2.872 |
| 11 | SA | 3.244 | 13.126 |
| 12 | SA | 46.215 | 178.478 |
| 13 | SA | 583.673 | 2.145.898 |
| 14 | SA | 11.374.275 | 38.423.872 |
| 10 | SR | 3.010 | 11.384 |
| 11 | SR | 8.512 | 32.664 |
| 12 | SR | 29.772 | 114.844 |
| 13 | SR | 118.498 | 440.960 |
| 14 | SR | 292.004 | 1.106.008 |
| 15 | SR | 2.776.390 | 10.204.874 |
| **Tempo Total** | | **15.319.727** | **53.030.470** |

A inclusão de uma chamada à função *pvm_probe*() em cada interação do algoritmo serial, resultou em um tempo de processamento de 3,5 a 4 vezes maior, consumindo cerca de 71% do tempo total de processamento.

Na implementação paralela, os processos escravos são os responsáveis pela execução do algoritmo *Branch-and-Bound*, com código idêntico ao da versão serial, a menos das as inclusões necessárias para a comunicação com os outros processos. É possível diminuir o elevado custo de processamento da função *pvm_probe*(), aumentando a granularidade do algoritmo com a redução de chamadas a esta função. O uso da variável **probe_delay** determina quantas interações do algoritmo B&B são realizadas antes de usar a função *pvm_probe*().

A Tabela 5.6 mostra os valores utilizados na variável **probe_delay** para encontrar um valor adequado para a execução dos testes de desempenho da implementação paralela.

**Tabela 5.6: Tempo de execução da versão paralela com busca em profundidade, variando-se o probe_delay**

| Arquivo | | Valores de probe_delay | | | | | |
|---|---|---|---|---|---|---|---|
| \|V\| | Tipo | 10 | 20 | 100 | 1.000 | 10.000 | 20.000 |
| 13 | SF | 4.090 | 4.039 | 3.073 | 3.121 | 3.576 | 3.101 |
| 12 | SA | 16.550 | 13.348 | 12.759 | 12.049 | 12.790 | 12.669 |
| 14 | SA | 3.431.362 | 3.055.538 | 2.762.509 | 2.695.521 | 2.688.692 | 2.688.132 |
| 13 | SR | 36.115 | 32.063 | 29.052 | 28.291 | 28.254 | 28.196 |
| 14 | SR | 151.213 | 133.797 | 119.509 | 116.467 | 116.092 | 116.087 |
| **Total** | | **3.639.330** | **3.238.785** | **2.926.902** | **2.855.449** | **2.849.404** | **2.848.185** |

Há um ganho considerável no desempenho para valores menores ou iguais a 1.000, cerca de **27%** no tempo total com **probe_delay** = 10 até **probe_delay** = 1000. Acima deste valor o tempo de execução não se altera significativamente. É importante ressaltar que a granularidade controlada pela variável **probe_delay**, pode ter outros valores ideais para arquiteturas, problemas ou tamanhos de grafos diferentes dos abordados neste trabalho.

## 5.7 Desempenho da implementação paralela

A Tabela 5.7, apresenta os resultados obtidos com a implementação proposta da técnica *Branch-and-Bound* neste trabalho em relação à implementação serial da técnica. Todos os resultados foram obtidos com o cálculo do valor limite inicial e na versão paralela com a variável **probe_delay** igual a 1000. Estes testes foram realizados na arquitetura disponível já mencionada, sempre com 1 processo mestre e 3 processos escravos para a versão paralela. Os valores apresentados são a média obtida com os resultados de 3 execuções do programa para cada problema.

**Tabela 5.7: Tempo de execução da versão serial e paralela em milésimos de segundo.**

| Arquivo | | Profundidade | | Melhores Primeiro | | Largura | |
|---|---|---|---|---|---|---|---|
| \|V\| | Tipo | Serial | Paralela | Serial | Paralela | Serial | Paralela |
| 10 | SF | 40 | 743 | 9.173 | 1.055 | 97 | 15.979 |
| 11 | SF | 147 | 1.082 | 118.739 | 20.246 | | s/ memória |
| 12 | SF | 1.040 | 1.362 | | 354.345 | | |
| 13 | SF | 2.944 | 3.520 | | | | |
| 14 | SF | 6.794 | 51.580 | | | | |
| 15 | SF | 14.078 | 397.390 | | | | |
| 16 | SF | 30.013 | 427.709 | | | | |
| 17 | SF | 71.105 | 392.592 | | | | |
| 18 | SF | 255.488 | 2.155.389 | | | | |
| 10 | SA | 473 | 941 | 1.391.131 | 32.649 | 506.278 | 15.353 |
| 11 | SA | 2.141 | 1.582 | 38.143.851 | 2.405.918 | | |
| 12 | SA | 30.502 | 12.890 | | | | |
| 13 | SA | 385.224 | 132.639 | | | | |
| 14 | SA | 7.507.022 | 2.642.820 | | | | |
| 15 | SA | 158.328.643 | 50.232.001 | | | | |
| 10 | SR | 1.987 | 1.031 | 19.002.932 | 796.270 | s/ memória | 15.543 |
| 11 | SR | 5.618 | 3.211 | | 11.257.540 | | |
| 12 | SR | 19.650 | 9.090 | | | | |
| 13 | SR | 78.209 | 27.948 | | | | |
| 14 | SR | 192.723 | 114.751 | | | | |
| 15 | SR | 1.832.417 | 893.214 | | | | |
| 16 | SR | 9.250.674 | 3.754.423 | | | | |
| 17 | SR | 57.929.688 | 18.612.343 | | | | |
| **29** | **RGS** | **3.711.439** | **1.397.590** | | | | |
| **Total em Horas** | | **66,57** | **22,57** | **16,30** | **4,13** | **0,14** | **0,01** |

A interpretação do valores obtidos sobre o desempenho da implementação paralela em relação à versão serial, requer uma análise distinta entre as estratégias de seleção usadas.

### 5.7.1 Busca em Profundidade

A busca em profundidade se mostrou o método de seleção ideal para a exploração da árvore de estados em problemas de buscas em grafos, principalmente em problemas PCV. As razões para uma melhor eficiência da busca em

profundidade são: a) a necessidade de pouco espaço em memória para armazenar a lista de tarefas, b) a rapidez e freqüência em que encontra e atualiza seus valores limites e c) a facilidade de inserção e retirada de subproblemas na lista de tarefas.

O desempenho da versão paralela em profundidade, que na soma total do tempo consumido para executar todos os exemplos da Tabela 5.7, alcança um ***speedup*** de **2,95** que leva a uma **eficiência** da paralelização de **98%** nos 3 processadores usados. Esse desempenho corresponde às expectativas levantadas durante o projeto desta versão paralela de que dividir a exploração da árvores de estados entre **n** processos escravos em **n** processadores disponíveis, com um mínimo de comunicação entre mestre e escravos e um processo mestre com atribuições também mínimas, torna possível se alcançar um *speedup* próximo de $\mathbf{t_s/t_p = n}$, que leva a uma eficiência perto de **100%**.

Os melhores desempenhos da versão paralela acontecem nos 2 exemplos que demandam o maior tempo total de execução, arquivos **|15|-SA** com um *speedup* superlinear de **3,15** e o **arquivo |17|-SR** com um *speedup* de **3,11** também superlinear. O *speedup* superlinear numa implementação paralela em problemas de busca, sinaliza a possibilidade de se ter explorado menos ramos da árvore de busca do que na versão serial. De fato, descartar ramos não promissores é a grande vantagem encontrada na execução de problemas maiores e de solução não óbvia.

Fica constatado que nos exemplos denominados de solução fácil (SF), não houve ganhos significativos na versão paralela. Em grafos (SF) em que a vantagem de descartar ramos não promissores não pode ser muito explorada, já que a solução ótima é encontrada no primeiro ramo da árvore também pela versão serial, a solução paralela tende a um fraco desempenho.

No problema representado pelo arquivo **RGS**, que é a implementação sobre um caso real de caixeiro viajante, onde o grafo é incompleto e na implementação as arestas não existentes são representadas com valores infinitos, obteve-se um *speedup* de **2,66** e uma eficiência de **89%**. O fato de casos reais serem

normalmente representados por grafos incompletos, possibilita que problemas de busca com um número maior de vértices possam ser executados. O tempo de execução paralela do problema **RGS** que tem |V| = 29, é aproximadamente a **metade do tempo** necessário para a execução do problema **SA** com |V| = 14.

### 5.7.2 Busca por Melhores Primeiro

O número total de subproblemas explorados pelo método melhores primeiro é similar ao encontrado na estratégia de busca em profundidade, como mostram as tabelas 5.2 e 5.3. Há entretanto um crescimento significativo no tempo total de execução neste método. Isso ocorre por dois motivos: a) o rápido consumo de memória exigido pelo método melhores primeiro, já que sua fila de tarefas fica com um grande número de subproblemas não explorados (Tabela 5.1), e b) o custo computacional de manter a fila de tarefas ordenada. Nesta implementação, particularmente, é feita uma busca seqüencial a cada nova inserção de um novo subproblema na fila, para mantê-la ordenada.

O desempenho da versão paralela do método melhores primeiro apresenta um *speedup* superlinear, o que evidencia uma grande diferença entre o total de nós explorados entre a versão paralela e serial. A razão para que isto aconteça está na diferença do conteúdo inicial da fila de tarefas da versão serial e da fila de tarefas de cada escravo na versão paralela. Na versão serial, a lista de tarefas inicial contém todos os vértices de início do problema, enquanto que na versão paralela a lista de tarefas inicial tem somente o vértice enviado pelo processo mestre, restringindo a exploração dos nós a um ramo específico da árvore. Como conseqüência temos: a) um menor número total de subproblemas na lista de tarefas dos processos escravos, b) menos trabalho na manutenção da lista ordenada, c) um número menor de problemas a serem explorados e d) encontro de valores limites em menos tempo.

O custo elevado de manter a lista de tarefas ordenada na busca por melhores primeiro, pode ser observado comparando-se o tempo da execução serial para os arquivos SF e SA com |V| = 10 da Tabela 5.7 entre os método de busca melhores primeiro e largura. Apesar da versão com busca em largura explorar um número muito maior de nós, ver tabelas 5.1, 5.2 e 5.3, seu tempo total de execução menor é resultado da facilidade de inserção de novos subproblemas em sua lista de tarefas.

### 5.7.3 Busca em Largura

A execução em paralelo com a estratégia de seleção por busca em largura aponta para um efeito interessante: o tempo total de execução praticamente não se altera, independentemente dos valores das arestas para grafos com o número de vértices igual (SF-SA-SR). Isto ocorre por que a busca em largura não encontra valores limites que possam evitar a exploração de nós não promissores na árvore de busca, exigindo que a exploração de quase toda a árvore seja feita até suas folhas, independentemente dos valores das arestas do grafo.

O *speedup* superlinear observado entre a versão serial e a paralela da busca em largura acontece pela mesma razão de diferença entre o conteúdo inicial da lista de tarefas de cada versão, como ocorre com a busca por melhores primeiro.

# CAPÍTULO 6
# CONCLUSÃO

A necessidade de conhecer e atuar em duas áreas diferentes: programação paralela por troca de mensagens e *Branch-and-Bound* especificamente, duas áreas até então, completamente inexploradas em nossos estudos, foi o ponto marcante imposto pela proposta deste trabalho.

Depois de uma solução algorítmica serial alcançada, pensar em sua paralelização não é uma tarefa direta. Novos conceitos pertencentes ao paradigma da programação paralela, como *deadlock*, congestionamento, capacidade de transmissão, colisões, balanceamento de carga, além dos infinitos "travamentos" que agora são simultâneos e em várias máquinas, entram no conjunto de possibilidades a serem revistas e cautelosamente estudadas e por vezes atormentam e dificultam esta tarefa quando ela já parecia solucionada. Chegar a uma solução paralela eficiente e apresentável, é o resultado de inúmeras versões de programas, testes, e algoritmos diversos para simulações de partes da paralelização.

Já com a técnica *Branch-and-Bound*, se não há grandes dificuldades em assimilar seus conceitos teóricos, a dificuldade se encontra na dinâmica de modelar problemas para que se adaptem em sua estrutura de funcionamento.

Aqui obtivemos gratos resultados, além do domínio dos problemas desenvolvidos neste trabalho, incluindo alguns que foram usados como testes, o conhecimento desta técnica resultou na implementação, com resultados bastante positivos, de um problema que já há algum tempo perseguíamos: uma solução para resolver o problema de grade horária escolar para instituições de ensino de primeiro e segundo grau. Desta solução serial temos hoje um produto denominado de Mr. Class, já disponível no mercado, em seu terceiro ano de plena comercialização. Esta menção a um produto comercial desenvolvido a partir dos conhecimentos acadêmicos adquiridos com este trabalho, deve servir de estímulo para que mais

pesquisadores estejam atentos às possibilidades de tornarem seus conhecimentos em soluções, por que não dizer, lucrativas dentro da sociedade.

A impressão que fica em relação ao que encontramos sobre *Branch-and-Bound* e sua paralelização, é principalmente a importância e grande interesse que despertam na comunidade acadêmica, parte por sua atração teórica (quando se torna possível analisar melhor resultados e comportamentos de problemas NP-difíceis), parte por suas contribuições em aplicações práticas com grande interesse da sociedade, principalmente no segmento industrial.

Muitas são as possibilidades de melhorias e avanços práticos e teóricos na paralelização de B&B para busca em grafos, apenas considerando-se este texto há vários exemplos como: a avaliação das implementações serial e paralela em mais arquiteturas (diferentes redes e mais processadores principalmente), outras instâncias destes ou diferentes problemas, análises mais profundas de estratégias de seleção (como a combinação de usar profundidade até se atingir o primeiro valor limite e depois seguir com melhores primeiro, ou ainda a possibilidade de usar estratégias diferentes para cada processo escravo durante uma mesma execução), uma maior generalização do código para utilização em problemas em grafos, a definição de mais e melhores medidas de desempenho e uma avaliação exaustiva do comportamento da execução de *Branch-and-Bound* (como uma melhor análise do crescimento das filas, ocorrência da primeira solução, ocorrência de cortes).

É desejável também, a implementação de uma biblioteca genérica para resolver problemas específicos em grafos com B&B: problemas do caixeiro viajante, caminho mínimo, planaridade, árvores geradoras mínimas, *matching*, componentes conexas, fechamento transitivo, entre outros. Com a possibilidade de usar uma estrutura de dados pré-definida na implementação paralela da técnica B&B, pode-se aproveitar melhor o princípio da localidade e evitar problemas de consumo excessivo de memória observados em bibliotecas desenvolvidas para serem totalmente genéricas.

# Referências Bibliográficas


[Alm94] ALMASI, George S., *Highly Parallel Computing,* The Benjamin/ Cummmings Publishing Company Inc., Redwood City CA, 1994.

[BMFN96] BRÜNGGER, A.; MARZETTA, A.; FUKUDA, K.; NIEVERGELT, J., *The parallel search bench zram and its applications*, CroSCutS, 1996.

[BWT98] BLUM, Joachim M.; WARSCHKO, Thomas M.; TICHY, Walter F., *PSPVM: Implementing PVM on a high-speed Interconnect for Workstation Clusters*, University of Karlsruhe, Germany, available at http://www.ipd.ira.uka.de/parastation, 1998.

[CDH+95] CHANG S.; DU D.H.; HSIEH J.; et al, *Enhanced PVM Communications over a High-Speed LAN, IEEE Parallel & Distributed Tecnology*, vol. 3, no. 3, pp. 20-32, 1995.

[CR95] CUN, B. Le, ROUCAIROL C., *BOB: Une Plate-forme Unifiée de Développement pour les Algorithmes de type Branch-and-Bound*, Rapports de Recherche n. 95/20, Out., 1995.

[CT89] CLAUSEN J.; TRÄFF, J.L., *Implementation of Parallel Branch and Bound algorithms - Experiences with the graph partitioning problem*, DIKU report 89/16, Departament of Computer Science, University of Copenhagen, 1989.

[DG94] DIDERICH, Claude G.; GENGLER, Marc, *Parallélisation de lálgorithme de branch et bound.* EPFL Supercomputing Review, n. 6, Nov., 1994.

[DG95] DIDERICH, Claude G.; GENGLER, Marc, *Experiments with a Parallel Synchronized Branch and Bound algorithm.* Parallel Algorithms for Irregular Problems: State of the Art, 177-193, 1995.

[DOW96] DONGARRA, Jack J.; OTTO Steve W.; WALKER David, *A Message Pasing Standard for MPP and Workstations*, Comunnications of the ACM, vol. 37 no 7, pp. 84-90, Jul. 1996.

[Duc90] DUCAN, R. *A survey of parallel computer architectures.* IEEE Trans. On Computers, vol. 23 no 2, pp. 5-16, Ago. 1990.

[Fly72] FLYN, M.J. *Some Computer Organizations and their Effectiveness*, IEEE Trans. On Computers, C-21, pp. 948-960, Sep. 1972.

[Fos95] FOSTER, I.T. *Designing and Building Parallel Programs*, Addison-Wesley, EUA, 1995

[Fou84] FOULDS, L.R. *Combinatorial Optimization for Undergraduates*. New York, N.Y. Spring Verlag, 1984.

[Gei94] GEIST, A., Beguelin, A., at al, *PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, MA, 1994.

[GR86] GEHANI, N.H.; ROOME, W.D., *Concurrent C. Software Practice and Experience*, v.16, n.9, p.821-44. Set. 1986.

[GR88] GEHANI, N.H.; ROOME, W.D., *Rendezvous facilities: Concurrent C and ADA language*. IEEE Trans. on Software Engineering, v.14, n.11. Nov. 1998.

[GR92] GEHANI, N.H.; ROOME, W.D. *Implementing Concurrent C. Software Practice and Experience*, v.22, n.3, p.267-85. Mar. 1992.

[GS92] GRANT, B.K.; SKELLUM, A. *The PVM system: an in-depth analysis and documenting study-concise edition*. Numerical Mathematics Group, Lawrence Livermore National Laboratory, 1992.

[Hom96]HOMEISTER, Dieter., *Efficient Implementation of Parallel Branch & Cut*. Fifth SIAM Conference on Optimization, available at http://www.iwr.uni-heidelberg.de, May., 1996.

[Kit95] KITAJIMA, J.P. *Programação paralela utilizando mensagens*, XIV Jornada de Atualização em Informática, Canela, 1995.

[Lau93] LAURSEN, Per S., *Simple approaches to parallel Branch and Bound*, Parallel Computing no 19, Elsevier Science Publishers, pp. 143-152, Mai. 1993.

[Lei92] LEIGHTON, F.T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo CA, 1992.

[LM89] LÜLING, R.; MONIEN B., *Two strategies for solving the vertex cover problem on a transputer network.* Lecture Notes in Computing Science, no 392, pp. 160-170, 1989.

[LM92] LÜLING, R.; MONIEN B., *Load Balancing for Distributed Branch-and-Bound Algorithms.* In International Parallel Processing Symposium, pp. 543-549, 1992.

[LLK86] LAWLER, E.L.; LENSTRA J.K.; KAN, A.H.G., The Traveling Salesman Problem. A guided tour of Combinatorial Optimization. E.L Lawler, pp. 361-401, 1.986

[LW66] LAWLER, E.L.; WOOD D.E., *Branch-and-Bound methods, a survey.* Operations Research, vol. 14, pp. 699-719, 1966.

[Mei96] MEIRA, Wagner Jr., at al, *Parallel Branch-and-Bound: Design and Performance Understanding.* VIII Simp.de Arq. de Computadores e Proc. de Alto Desempenho, pp. 119-128, 1996.

[Moh83] MOHAN J., *Experience with two parallel programs solving the traveling salesman problem.* Proceedings of the 1983 International Conference on Parallel Processing, pp. 191-193, IEEE, New York, 1983.

[OT89] ORTEGA, M.; TROYA J., *Live nodes distribuition in Parallel Branch and Bound Algorithms*, Microprocessing & Microprogramming, no 25, pp. 301-306, 1989.

[PH96] PATTERSON, David A; HENNESSY, John L., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 2nd edition, 1996.

[PS82] PAPADIMITRION C. H.; STEIGLITZ K. *Combinatorial Opimization: Algorithms and complexity*, Prentice-Hall Inc., New Jersey, 1982.

[San99] SANTOS, Aldri L. dos, *Avaliação de desempenho da comunicação com PVM em ambiente Linux,* Dissertação de Mestrado, Depto. de Informática, Setor de Ciências Exatas, Universidade Federal do Paraná, Curitiba, 1999.

[Qui90] QUINN, M.J., *Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer.* IEEE Trans. on Computers, vol. 39-3, pp. 384-387, Mar. 1990.

[Qui94] QUINN, M.J. *Parallel Computing Theory and Pratice,* 2. ed., McGraw-Hill, Oregon State EUA, 1994.

[RND77] REINGOLD E.M.; NIEVERGELT J.; DEO N., *Combinatorial Algorithms: Theory and Practice. Prentice-Hall*, Englewood Cliffs, New Jersey EUA, 1977.

[ST96] SODERO, A.C.; TAVARES A.I., *SABOR - Sistema para Análise e Desenvolvimento de Algoritmos de Branch-and-Bound Orientado a Objetos em Redes de Estações de Trabalho - Guia do Usuário*, Universidade Federal de Minhas Gerais, Depto. Ciência da Computação, 1996.

[TB92] TRIENEKENS H.J.M.; BRUIN A. de, *Towards a Taxonomy of Parallel Branch and Bound Algorithms*, University Rotterdam, Departament of Computer Science, Report EUR-CS-92-01, 1992.

[Tur93] TURCOTTE, L.H. *A survey of software environment for exploiting networked computing resources*. Enginneering Research Center for Computacional Field Simulation, Mississipi State, EUA, 1993.

[Yan94] YANG, Myung K., *Evaluation of a Parallel branch-and-Bound Algotithm on a Class of Multiprocessors*. IEEE Trans. Parallel and Distributed Systems, vol. 5, pp. 74-86, Jan. 1994.