Nuno Manuel Reis Amado

Algoritmos Paralelos de Indução de Árvores de Decisão

Faculdade de Engenharia da Universidade do Porto
2001

Nuno Manuel Reis Amado

Algoritmos Paralelos de Indução de Árvores de Decisão

Dissertação submetida para satisfação parcial dos requesitos do grau de mestre em Inteligência Artificial e Computação

Dissertação realizada sob a supervisão de
Professor Fernando Silva,
da Faculdade de Ciências da Universidade do Porto
e de
Professor João Gama,
da Faculdade de Economia da Universidade do Porto

Faculdade de Engenharia da Universidade do Porto
2001

Agradecimentos

O presente trabalho foi realizado sob a orientação do Professor Fernando Silva e do Professor João Gama aos quais desejo profundamente agradecer pelo apoio, disponibilidade e enorme paciência que sempre demonstraram.

Desejo também exprimir os meus agradecimentos ao LIACC - Laboratório de Inteligência Artificial e Ciências de Computadores por permitir as condições necessárias para o desenvolvimento deste trabalho.

Resumo

Actualmente, nos campos da Extracção do Conhecimento e da Aprendizagem Automática a quantidade de dados disponível para a construção de procedimentos de classificação está a crescer a um ritmo bastante rápido. Para a construção de um procedimento de classificação, uma maior quantidade de dados aumenta a possibilidade de se obterem procedimentos com maior precisão de classificação. Contudo, também implica um custo maior no tempo e na complexidade de obtenção de tais classificadores. Desta forma, existe uma grande necessidade de algoritmos capazes de construir classificadores a partir de conjuntos de dados bastante grandes e, simultaneamente, serem computacionalmente eficientes e escaláveis. Uma possível solução é a utilização de algoritmos paralelos de forma a reduzir a quantidade de tempo utilizada na construção de procedimentos de classificação a partir de conjuntos de dados bastante grandes e, desta forma, manter a precisão de classificação.

Neste trabalho, são apenas considerados os procedimentos de classificação baseados em árvores de decisão dado que estas têm sido extensamente utilizadas em diversas áreas e têm sido consideradas como os modelos mais adequados para a extracção do conhecimento e para a Aprendizagem Automática. São descritas algumas das estratégias utilizadas para a implementação de algoritmos de indução de árvores de decisão em paralelo baseados em técnicas como o paralelismo de controlo, paralelismo nos dados e paralelismo híbrido.

Algumas das ideias e estratégias apresentadas, assim como o algoritmo de indução de árvores de decisão do sistema C4.5, serviram como suporte para o desenvolvimento de um novo algoritmo paralelo de construção de árvores de decisão, à sua implementação usando MPI e análise de desempenho num *cluster* de memória distribuída.

Abstract

In the field of Data Mining and Machine Learning the amount of data available for building classification procedures is growing very fast. For the construction of classification procedures, the use of larger amounts of data increases the probability of achieving procedures with higher classification accuracy. However, it also implies higher costs in time and complexity for building those procedures. Therefore, there is a great need for algorithms capable of building classifiers from larger datasets and, simultaneously, being computationally efficient and scalable. One possible solution for reducing the amount of time used in the construction of classification procedures from very large datasets, and, by this, keeping their classification accuracy, is the use of parallelism.

In this work we only considered those classification procedures based upon decision trees since they have been extensively used in many fields and considered as the most suitable models for data mining and machine learning. We present a few strategies used in the implementation of parallel decision tree induction algorithms based on techniques such as task parallelism, data parallelism and hybrid parallelism.

Based upon the C4.5 decision tree induction algorithm and on some of the introduced strategies and techniques, we developed a new parallel decision tree induction algorithm, implemented it using MPI and analysed its performance on a distributed memory cluster.

Résumé

Dans le domaine de l'Extraction de la Connaissance (Data Mining) et de l'Apprentissage Automatique (Machine Learning) un grand volume de données est disponible aujourd'hui, en croissance rapide. Une grande quantité de donnés permet d'obtenir des procédures de classification plus précises mais leur construction est plus lente et complexe. C'est pour ça qu'il faut rencontrer des algorithmes pour construire les procédures qui soient plus efficientes et "echellables ".

Dans ce travail, on considère les procédures de classification basées sur des Arbres de Décision parce qu'elles ont été considérées comme les meilleurs modèles dans le domaine de l'extraction de la connaissance et de l'apprentissage automatique. Et pour maintenir la précision et en même temps réduire le temps dépensé avec l'obtention automatique des procédures, on utilise des "algorithmes parallèles ". On fait aussi dans ce travail, une description des stratégies utilisées dans l'implémentation des algorithmes d'induction des arbres de décision en parallèle basées sur des techniques de parallélisme de contrôle, parallélisme dans les données et parallélisme hybride.

Quelques idées et stratégies présentées, aussi bien que l'algorithme du système C4.5, ont constitué un support précieux pour le développement d'un nouvel algorithme de construction des arbres de décision en parallèle, qui a été le but principal de ce travail.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A Inteligência Artificial é um campo da ciência da computação que pretende compreender os princípios que tornam possível o desenho de sistemas inteligentes, cujas características são associadas às da inteligência no comportamento humano [Wis77, BF82, Cat91]. A capacidade de aprendizagem e de aplicação do conhecimento adquirido é claramente uma das características fundamentais da inteligência e uma das áreas investigação na Inteligência Artificial.

Muitas das actividades humanas onde é exigido a aplicação de conhecimento aprendido podem ser consideradas como tarefas de classificação. No sentido mais geral do termo, a *Classificação* ocorre sempre que é necessário tomar uma decisão ou realizar uma previsão com base na informação disponível.

1.1 Classificação

No âmbito deste trabalho, a classificação tem como objectivo a construção de um procedimento capaz de prever a classe de um exemplo com base num conjunto de características ou atributos que a descrevem. O procedimento de classificação é obtido a partir de um programa de aprendizagem ao qual é fornecido um conjunto de observações para as quais a verdadeira classe é conhecida.

Ao longo dos anos têm surgido vários métodos e modelos para a classificação. Estes podem ser agrupados em três áreas diferentes, a *Classificação Estatística*, a *Aprendizagem Automática* e as *Redes Neuronais*. Embora enquadrados em áreas diferentes, todos os métodos tentam encontrar procedimentos capazes de tomar decisões e ao mesmo tempo, serem suficientemente genéricos para poderem ser aplicados com sucesso, numa grande variedade de problemas práticos [MST94].

Dos diversos modelos que tem surgido nas diferentes áreas relacionadas com a classificação, as árvores de decisão têm sido consideradas como os modelos mais adequados para a extracção do conhecimento e para a Aprendizagem Automática, pois são suficientemente simples para serem facilmente compreendidos pelos humanos e facilmente expressos numa subrotina em qualquer linguagem de programação [MST94, Qui92].

Entre os vários algoritmos de classificação e, em especial, entre os algoritmos de indução de árvores de decisão, destaca-se o sistema C4.5 [Qui92]. Este tem sido tomado como ponto de referência para o desenvolvimento e análise de novas propostas. As árvores de decisão por ele construídas apresentam uma boa precisão na classificação e tem sido considerado como um dos algoritmos mais rápidos utilizados na extracção do conhecimento e na aprendizagem automática.

Para a construção de um procedimento de classificação é normalmente fornecido um conjunto de observações para as quais a verdadeira classe já é conhecida. De uma forma geral, a utilização de um conjunto maior de observações para a construção de um procedimento de classificação e, em particular, das árvores de decisão, proporciona uma melhor precisão de classificação do modelo obtido [Cat91]. Mas para que essa utilização seja eficaz é importante dispor de algoritmos de classificação computacionalmente eficientes e escalonáveis.

1.2 Motivação e Objectivos

A construção de árvores de decisão a partir de grandes conjuntos de dados poderia passar pela utilização de uma pequena amostra do conjunto total de observações. Contudo, as árvores de decisão obtidas desta forma não apresentam o mesmo nível de precisão de classificação que as árvores obtidas a partir da totalidade dos dados [Cat91]. O recurso ao paralelismo é uma forma de reduzir a complexidade computacional na construção de árvores de decisão a partir de grandes conjuntos de dados [FL98, SHKS99].

O recurso ao paralelismo na construção de árvores de decisão consiste na utilização de um conjunto de processadores que cooperam entre si para cada um deles realizando em simultâneo apenas uma parte do processamento total necessário. Mas, a implementação de algoritmos de construção de árvores de decisão em paralelo é uma tarefa bastante complexa e continua a ser, hoje em dia, uma área de importante investigação e desenvolvimento. Os problemas principais na implementação destes algoritmos encontram-se precisamente relacionados com a distribuição do processamento e dos dados pelos vários processadores [SHKS99].

Neste trabalho descrevem-se algumas das principais estratégias utilizadas na implementação de algoritmos de construção de árvores de decisão em paralelo, nomeadamente, a utilização de paralelismo de controlo, paralelismo nos dados (com distribuição vertical e horizontal) e paralelismo híbrido. Para além de se identificarem os principais problemas associados a cada uma dessas estratégias, apresentam-se também alguns soluções para os ultrapassar.

As ideias e metodologias referidas bem como o algoritmo do sistema C4.5, serviram de suporte ao desenvolvimento de um novo algoritmo de construção de árvores de decisão em paralelo.

1.3 Organização da tese

Os capítulos seguintes encontram-se organizados da seguinte forma:

- No capítulo 2 são introduzidos alguns conceitos e definições fundamentais na área da Classificação. Descrevem-se ainda alguns dos métodos utilizados nesta área. É ainda realizada uma descrição detalhada do processo de construção das árvores de decisão e do sistema C4.5 utilizado como base para o algoritmo desenvolvido neste trabalho.

- O capítulo 3 apresenta uma introdução à área da Computação Paralela. Define-se o paralelismo e descrevem-se as principais arquitecturas e sistemas utilizados. Neste capítulo abordam-se também sobre o desenho e implementação de algoritmos paralelos e a forma se prevê e avalia o desempenho de um sistema paralelo.

- No capítulo 4 apresentam-se algumas das principais estratégias e algoritmos utilizados na construção de árvores de decisão em paralelo. Estas estratégias encontram-se divididas em paralelismo de controlo, nos dados e híbrido.

- No capítulo 5 descreve-se um novo algoritmo de indução de árvores de decisão em paralelo com distribuição horizontal dos dados que tem por base o sistema C4.5 e que utiliza como camada de comunicação o sistema MPI.

- No capítulo 6 apresentam-se os resultados obtidos com a implementação do algoritmo e realiza-se uma análise crítica dos mesmos salientando os principais

- Por último, no capítulo 7, apresentam-se as conclusões do trabalho realizado e algumas propostas para desenvolvimentos futuros, pois um trabalho como este deixa lacunas que merecem ser colmatadas e abre novas perspectivas para outras, e por ventura melhores, abordagens.

Os capítulos 2 e 3 têm por objectivo introduzir os temas da Classificação e do Paralelismo, o que torna a sua leitura não essencial para a compreensão deste trabalho caso o leitor já se encontre familiarizado com estas áreas. Se o desejar poderá iniciar a leitura no capítulo 4.

Capítulo 2

Classificação

A *Classificação* pode ser encontrada numa grande variedade de actividades humanas. Esta dá-se sempre que é necessário tomar uma decisão baseada na informação disponível para a decisão. Por exemplo, tarefas como a atribuição ou recusa de um crédito a um cliente de um banco ou compra ou venda de acções podem ser consideradas como tarefas de classificação.

Para a classificação, é considerado o problema de construir um procedimento capaz de prever a classe de uma observação baseada num conjunto de características ou atributos dessa mesma observação. Este procedimento é normalmente obtido através de um programa de aprendizagem que o constrói a partir de um conjunto de observações para as quais a verdadeira classe já é conhecida. Este procedimento pode ser utilizado posteriormente para identificar a classe de novas observações.

A capacidade de aprendizagem e de aplicação do conhecimento é claramente uma das características principais da inteligência humana e uma das áreas de importante investigação dentro do campo da Inteligência Artificial. Logo, a classificação tem sido uma das áreas principais de investigação e desenvolvimento dentro da Inteligência Artificial.

Ao longo de vários anos têm sido propostos diversos algoritmos para a classificação que podem ser agrupados em três áreas de especial importância a nível histórico: a *Classificação Estatística*, a *Aprendizagem Automática* e as *Redes Neuronais*. Apesar de serem utilizadas metodologias diferentes pelos algoritmos em cada uma destas áreas, todos tentam igualar, ou mesmo superar, as decisões realizadas pelos humanos e, ao mesmo tempo, serem suficientemente genéricos para poderem ser aplicados numa grande variedade de problemas práticos.

Dentro dos diversos métodos utilizados na classificação salientam-se as árvores de decisão. Estas têm sido consideradas como os modelos mais adequados para a extracção do conhecimento e para a aprendizagem automática. Uma árvore de decisão é uma estrutura em árvore capaz de prever a classe de uma observação baseada em decisões realizadas sobre os atributos que descrevem essa mesma observação. Os procedimentos de classificação baseados em árvores de decisão são suficientemente simples para serem compreendidos pelos humanos e são de execução eficiente podendo ser facilmente expressos como uma subrotina em qualquer linguagem de programação.

Entre os vários algoritmos de indução de árvores de decisão, o sistema C4.5 proposto por Quinlan [Qui92], merece uma atenção especial por variadas razões. Em conjunto com alguns outros sistemas, este representa o resultado da investigação nas áreas da aprendizagem automática e da extracção do conhecimento que data desde o sistema ID3. Como tal, tem sido tomado como ponto de referência para o desenvolvimento e análise de novos métodos e algoritmos de classificação e indução de árvores de decisão [BFOS84]. Também, o sistema C4.5 apresenta uma boa precisão na classificação e tem sido considerado como um dos sistemas de *main-memory* mais rápidos entre os algoritmos utilizados na extracção do conhecimento.

Neste capítulo são apresentados, de uma forma resumida, alguns dos conceitos fundamentais relativos à classificação. São também apresentados, muito resumidamente, alguns dos métodos mais utilizados na classificação nas três principais áreas a nível histórico: a *Classificação Estatística*, a *Aprendizagem Automática* e as *Redes Neuronais*. É também realizada uma descrição, com maior detalhe, das árvores de decisão e da forma como estas são construídas e utilizadas na classificação. Por fim, é apresentado o algoritmo de construção de árvores de decisão do sistema C4.5 que foi utilizado como base para o algoritmo desenvolvido neste trabalho.

2.1 Conceitos

A *Classificação* aparece numa grande variedade de actividades humanas. No seu sentido mais lato a classificação ocorre sempre que é necessário tomar uma decisão ou realizar uma previsão baseada na informação disponível. Uma tarefa de classificação pode ser considerada como a aplicação formal de um método sempre que são tomadas decisões face a novas situações [MST94].

O conceito de classificação considerado neste trabalho é um pouco mais restrito que o anterior. É considerado o problema de construir um procedimento capaz de atribuir uma classe, de um conjunto pré definido de classes, a cada *observação* de uma sequência continua de *observações*. A atribuição é realizada com base num conjunto de *atributos* ou *características* que descrevem cada uma das observações.

Um procedimento de classificação é normalmente obtido através de um programa de aprendizagem que o constrói a partir de um conjunto de observações para as quais a verdadeira classe é conhecida. Este método de construção de um procedimento de classificação é muitas vezes referido como *reconhecimento de padrões*, *discriminação*, ou *aprendizagem supervisionada* (de forma a distinguir de *aprendizagem não supervisionada* ou *clustering* em que as classes são inferidas a partir das observações) [MST94].

Muitos dos termos utilizados para a classificação têm vários sinónimos. As *observações* são muitas vezes denominadas por exemplos, casos, objectos ou items. O conjunto de observações ou exemplos utilizados para a construção de um procedimento de classificação é normalmente referido como *conjunto de treino* ou, por vezes, de *dados de entrada*. O termo *conjunto de dados* aplica-se normalmente ao conjunto de todos os dados de um determinado *domínio* que se encontram disponíveis. O conjunto de treino é normalmente formado por um subconjunto do conjunto de dados. O termo *domínio* normalmente refere-se ao contexto dos dados utilizados para a tarefa de classificação, como por exemplo o domínio dos pedidos de crédito. O resultado de um programa de aprendizagem (o procedimento de classificação) pode também ser denominado de uma forma mais geral de *conceito* ou, talvez mais vagamente, de *produto* do processo de aprendizagem.

A construção do procedimento de classificação e a previsão da classe de um conjunto de observações são realizadas com base num conjunto de atributos ou características descritivas dessas mesmas observações. É comum nas declarações de dados fornecidas aos programas de aprendizagem distinguir-se entre *atributos contínuos*, cujos valores são números reais, e *discretos*, cujos valores encontram-se restritos a um número finito (normalmente pequeno) de valores possíveis. Os atributos discretos também são conhecidos como *atributos nominais*. Alguns sistemas permitem ainda definir *atributos estruturados* para os quais são definidas hierarquias, como por exemplo "Contínuo, Discreto" ao nível mais elevado da hierarquia e para o nível "Discreto" definidos dois valores "Discreto Ordenado, Discreto Não Ordenado".

O conjunto de valores possíveis para cada atributo e para as classes são declarados e fixados antes de se dar início ao processo de aprendizagem. Esta informação é denominada de *contexto de indução*. Alguns algoritmos permitem ainda que um valor de um atributo ou de uma classe possa ser especificado como *desconhecido*, isto é, que o seu verdadeiro valor não é conhecido, para os quais existe uma variedade de métodos para os tratar [BFOS84, Qui92].

2.1.1 Definição

A *classificação* tem dois significados distintos. Pode ter como objectivo encontrar um conjunto de classes ou grupos num conjunto de observações. Ou, conhecendo à partida a existência de um determinado número de classes, o objectivo é encontrar uma regra capaz de classificar uma nova observação numa das classes definidas [MST94]. A primeira definição é normalmente conhecida como aprendizagem não supervisionada (ou *clustering*) e a segunda, utilizada neste trabalho, como aprendizagem supervisionada. Em termos estatísticos a aprendizagem supervisionada é normalmente referida como discriminação, o objectivo de encontrar uma regra de classificação a partir de um conjunto de observações já correctamente classificadas.

No sentido mais geral do termo indução ou raciocínio indutivo, este pode referir-se a qualquer progressão de factos científicos para regras gerais. A indução tem origem no raciocínio humano, e, se os procedimentos estiverem, de alguma forma, restritos e formalizados, a indução pode ser realizada de acordo com um procedimento bem definido normalmente denominado por algoritmo de indução [Cat91].

A definição do termo indução assemelha-se bastante à definição de um programa de aprendizagem que utiliza um conjunto de observações (factos) para encontrar um procedimento de classificação (regra). Isto deu origem a que os algoritmos de aprendizagem supervisionada sejam muitas vezes referidos como *algoritmos de indução* ou *sistemas indutivos*.

2.1.2 Motivação

A Inteligência Artificial é um campo das ciências da computação que pode ser diversamente definido como o estudo dos computadores, o estudo da informação, o estudo da programação, ou o estudo dos algoritmos. Em [Wis77] os objectivos da Inteligência Artificial são apresentados de uma forma bastante abrangente:

"Os objectivos principais da Inteligência Artificial são tornar os computadores mais úteis e compreender os princípios que tornam a inteligência possível."

No *Handbook of AI* [BF82] a Inteligência Artificial é descrita como "preocupada em desenhar sistemas computacionais inteligentes, isto é, sistemas que apresentam características que estão associadas à inteligência no comportamento humano - compreender a linguagem, aprendizagem, raciocínio, resolução de problemas, etc..."

A capacidade de aprendizagem e da aplicação do conhecimento aprendido é claramente uma das características principais da inteligência e uma das áreas de importante investigação da Inteligência Artificial.

Muitas das actividades humanas podem ser vistas como tarefas de classificação, escolhendo as suas decisões de um pequeno número de possibilidades definidas como atribuir ou recusar um crédito, ou comprar, vender ou manter acções. Outras actividades, normalmente consideradas mais difíceis, como planeamento, escalonamento e desenho, que não aparentam ser tarefas de classificação, podem ser formuladas por componentes que são em si procedimentos de classificação. Logo, tarefas de classificação são objectos naturais de investigação no campo da Inteligência Artificial [MST94].

Os procedimentos de classificação, capazes de realizar tarefas de classificação ou decisão de forma automática, representam conceitos ou regras induzidas do conhecimento de um domínio específico onde serão aplicados. Como já foi referido, os sistemas indutivos são sistemas (ou algoritmos) capazes de induzir conceitos ou regras de forma automática a partir de observações de um determinado domínio. Ou seja, são sistemas capazes de criar procedimentos de classificação de forma automática a partir da informação disponível de um determinado domínio.

Em resumo, a criação de sistemas indutivos e de procedimentos de classificação, é fortemente motivada pela vontade de automatizar a aprendizagem de regras ou conceitos a partir da informação ou do conhecimento disponível de um determinado domínio, e criar procedimentos capazes de oferecer concelhos e realizar decisões de forma automática perante novos exemplos ou observações.

2.1.3 Aplicações

Alguns dos problemas mais urgentes que estão a surgir em diversas áreas como a ciência ou a industria podem ser vistos como problemas de classificação ou decisão apoiados, normalmente, em grandes quantidades de informação bastante complexa. Em muitas aplicações existem já bases de dados de conjuntos de exemplos previamente classificados que se encontram disponíveis para serem utilizadas na criação de procedimentos de classificação.

A aprendizagem supervisionada, realizada por um sistema indutivo, pressupõe a existência de algo que previamente classificou correctamente o conjunto de observações utilizado na aprendizagem. Então porquê tentar encontrar um regra de classificação que não é exacta se já existe algo capaz de classificar correctamente as observações? São várias as razões para criar um procedimento de classificação. A seguir são referidos alguns exemplos da utilização de tais procedimentos [MST94]:

- Máquinas de leitura automática de códigos postais, com procedimentos de classificação bastante rápidos, capazes de ordenar rapidamente grande parte da correspondência de uma central de correios.
- A decisão automática de atribuição de um crédito baseado apenas na informação fornecida por um questionário preenchido pelos clientes.
- Na área da medicina, procedimentos capaz de produzir um diagnóstico de confiança baseado apenas em sintomas externos dos pacientes.
- Aplicações em áreas como a meteorologia em que a verdadeira classe de uma observação só será conhecida no futuro como, por exemplo, a previsão do estado do tempo para o dia seguinte baseada em certas medidas meteorológicas do próprio dia. Neste caso a tarefa de classificação consiste em tentar encontrar um procedimento capaz de realizar previsões.

2.2 Algoritmos

Ao longo dos anos têm sido adoptados vários métodos para a classificação. Os vários métodos podem ser agrupados em três áreas com mais importância a nível histórico como a *Classificação Estatística*, a *Aprendizagem Automática* e as *Redes Neuronais*. Estes áreas têm envolvido profissionais e grupos académicos diferentes assim como têm sido dirigidas para questões distintas.

Contudo, os vários métodos de classificação apresentam alguns objectivos em comum [MST94]. Todos eles tentam encontrar procedimentos capazes de:

- Igualar, ou até superar, as decisões realizadas por um humano, com a vantagem de apresentar consistência e, até um certo grau, clareza;
- Serem capazes de tratar uma grande variedade de problemas e, tendo disponível informação suficiente, serem bastante genéricos;
- Serem utilizados em aplicações práticas com sucesso.

2.2.1 Classificação estatística

No campo da estatística a atenção é dirigida principalmente para as observações empíricas, não apenas como estas podem ser sumariadas, mas também com o processo de realizar inferências e decisões a partir dos dados. A estatística engloba uma grande variedade de áreas, algumas delas relevantes para o objectivo de produzir eficientemente classificadores precisos e compreensíveis a partir de um conjunto de observações [DH73].

A Análise Discriminante pode ser caracterizada genericamente por apresentar um modelo probabilístico que, ao contrário da simples identificação de uma classe, fornece a probabilidade de uma dada observação pertencer a uma determinada classe. Os métodos mais utilizados na classificação estatística normalmente assumem que irão ser utilizados por pessoas de alguma forma ligadas à estatística o que pressupõe a existência de alguma intervenção ao nível da selecção e tratamento das variáveis assim como a estruturação geral do problema.

Alguns dos métodos estatísticos clássicos assumem fortes simplificações de forma a reduzir a complexidade dos cálculos a realizar, mas, nestes casos, a precisão dos procedimentos de classificação obtidos pode contudo diminuir. Por outro lado, algumas das técnicas estatísticas são demasiado complexas que se tornam impraticáveis, como a regressão múltipla que envolve a inversão de matrizes cuja dimensão é determinada pelo número de atributos.

Muitos dos campos da estatística, alguns dos quais referidos a seguir, são normalmente utilizados como técnicas adjacentes aos vários métodos de classificação podendo ser bastante úteis para a construção de um procedimento de classificação.

A amostragem é um campo da estatística que estuda a teoria e a prática da aquisição de um conjunto de dados. Durante a construção de um procedimento de classificação muitas vezes é bastante mais importante, para uma melhoria na precisão do procedimento, a utilização de melhores métodos de aquisição de dados ou de reavaliação dos dados adquiridos do que apenas aumentar o número de exemplos utilizados.

Por vezes a aquisição de dados encontra-se separada da construção do procedimento de classificação impossibilitando qualquer acção sobre a forma como estes são adquiridos.

Contudo, existem outras opções como, por exemplo, o tratamento dos dados. Existem métodos automáticos de transformação dos dados denominados de métodos de indução construtiva, embora normalmente, métodos tradicionais de escolha de atributos e da sua transformação possam ser igualmente úteis. O campo relevante da estatística é denominado de análise de dados, que pode ser definida como a manipulação, sumarização e apresentação de dados para os tornar mais compreensíveis, logo revelando uma possível estrutura escondida por de trás dos dados. Uma melhor compreensão dos dados pode dar origem a melhores métodos para a sua transformação e utilização melhorando desta forma a precisão do procedimento de classificação.

Outros campos da estatística como a Análise Sequencial abordam questões como a quantidade de dados que deve ser sensivelmente recolhida e considerada na classificação. Por vezes pode ser preferível construir um classificador a partir de um conjunto de dados menor sacrificando, desta forma, a sua precisão quando é considerado o custo da obtenção de novas observações. Ou, por exemplo, pode ser utilizada uma regra de decisão linear como aproximação a uma regra de decisão bastante mais complexa de forma a aliviar o custo computacional.

A teoria da decisão é um outro campo da estatística que estuda qual a melhor acção a escolher a partir da avaliação de uma situação. Por exemplo, a atribuição de um crédito a um cliente que apresenta a probabilidade de 97% para o pagamento do empréstimo. Para a escolha da melhor acção, a teoria da decisão tem em conta, entre outros, os custos associados a decisões erradas, os custos das observações e os custos da própria decisão.

2.2.2 Aprendizagem Automática

O objectivo principal da *Aprendizagem Automática* é encontrar procedimentos computacionais baseados em operações lógicas ou binárias capazes de aprender uma tarefa a partir de um conjunto de exemplos. Os procedimentos de classificação aprendidos têm de ser suficientemente simples para que possam ser facilmente compreendidos. Estes devem-se assemelhar ao raciocínio humano de forma a possibilitar uma visão do processo de classificação [MST94]. Assim como na classificação estatística, um conhecimento do problema pode e deve ser explorado durante o desenvolvimento do procedimento de classificação, mas é assumido que a classificação de novas observações é realizada de forma automática e sem a intervenção humana.

Para a representação do conceito aprendido pelos algoritmos englobados na aprendizagem automática a atenção tem sido dirigida para as árvores de decisão. Nestes algoritmos a classificação de uma observação é o resultado de uma sequência de passos lógicos, ou decisões, sobre os atributos que a descrevem. As árvores de decisão são capazes de representar conceitos bastante complexos desde que sejam fornecidas observações suficientes para a aprendizagem (o que pode significar um grande conjunto de observações). Contudo, dentro da aprendizagem automática têm vindo a surgir outras abordagens, tanto para a representação dos conceitos aprendidos como para os programas de aprendizagem, como por exemplo, os algoritmos genéticos ou a *Inductive Logic Programming (ILP)*. Estes, em princípio, irão permitir trabalhar com dados mais genéricos como observações com número variável de atributos, hierarquias de classes, entre outros.

Muito resumidamente, os algoritmos de indução de árvores de decisão constroiem as árvores dividindo sucessivamente o conjunto de dados até todos os exemplos pertencerem à mesma classe. A cada divisão do conjunto de dados está associada uma decisão sobre um atributo. Devido às sucessivas divisões do conjunto de dados estes algoritmos são muitas vezes conhecidos por *Top Down Induction of Decision Trees - TDIDT*. Um dos programas mais conhecidos de indução de árvores de decisão é o sistema C4.5 que tem sido considerado como ponto de referência para o desenvolvimento e análise de novas propostas dentro dos algoritmos da família TDIDT [Qui92, BFOS84].

Alguns algoritmos de indução de árvores de decisão introduziram conceitos com origem na classificação estatística para a construção das suas árvores. Estes algoritmos utilizam alguns princípios da teoria de decisão Bayesiana para a escolha dos atributos associados às divisões do conjunto de dados. Outros algoritmos estendem os métodos de construção das árvores de decisão utilizadando uma representação do conhecimento em trama para ultrapassar observações com uma descrição dispersa numa estrutura de dados complexa (onde grande parte dos valores são quase sempre desconhecidos). De outra forma, as observações iriam requerer um número demasiado elevado de atributos ao ser utilizada a representação normal dos dados para algoritmos de indução de árvores. Outros sistemas ainda, enquadram-se sobre o tópico de métodos múltiplos. Estes sistemas constroem diversas árvores de decisão a partir dos mesmos dados e, no final, combinam, de alguma forma, as classificações obtidas numa só.

O termo *indução estruturada* foi dado por analogia à programação estruturada: em vez de realizar um classificador, por exemplo uma árvore de decisão, como uma unidade homogénea, este é decomposto em estruturas recursivas de classificadores. A classificação obtida por um componente é fornecida como um atributo no próximo componente mais elevado na estrutura. O classificador no topo da hierarquia produz resultado final. Assim como na programação estruturada, o objectivo é melhorar a compreensão do conceito obtido, oferecendo ainda a possibilidade da utilização de atributos estruturados [Sha87].

Uma outra abordagem, ao contrário de assumir que todo o conjunto de dados está disponível quando se inicia a aprendizagem, os algoritmos de aprendizagem incremental actualizam e refinam os seus conceitos à medida que são apresentadas novas observações. Muitos destes sistemas utilizam árvores de decisão para a representação do conceito aprendido. A principal motivação para estes métodos é a possibilidade de actualizar um conceito rapidamente assim que novos dados ficam disponíveis. Nestes métodos o custo de actualização das árvores é tipicamente muito mais pequeno quando comparado com o custo de as reconstruir do início através do método tradicional. Contudo, a aprendizagem incremental é mais lenta do que os métodos não incrementais e apresenta ainda algumas limitações como não permitir a utilização de atributos contínuos [UBC96].

Estudos recentes sobre conjuntos de dados utilizados na aprendizagem automática mostram que os sistemas indutivos baseados na teoria estatística Bayesiana apresentam uma melhor precisão de classificação que os métodos de indução de árvores de decisão. Contudo, alguns sistemas desta natureza mais simples, muitas vezes denominados de *naive bayes*, realizam algumas suposições um pouco arriscadas, como assumirem que os atributos são condicionalmente independentes em relação à classe [DP97].

Supondo que, num conjunto de dados sobre a atribuição de crédito, os valores das classes obedecem à seguinte regra: aceitar todas as pessoas que estão desempregadas ou que não têm conta bancária, mas nunca às que não têm conta bancária e estão desempregadas. É impossível encontrar uma função baseada na teoria bayesiana que traduza correctamente esta regra. Mas uma árvore de decisão de apenas nove nós pode representar este conceito correctamente. Dependências nos atributos, embora menos extremas, são muito comuns em aplicações reais.

A dependência condicional pode ser realizada quando estão disponíveis dados suficientes que justifiquem a construção de um modelo mais complexo baseado nas probabilidades condicionais conjuntas (condicionadas nos valores de dois ou mais atributos). As redes de inferência Bayesiana são sistemas de classificação estatística baseados nestes métodos. Estas redes foram utilizadas num dos primeiros sistemas periciais, mas foram construídas manualmente, e alguns dos seus mecanismos não apresentam uma justificação formal. Contudo, uma representação da rede, que mostra a relação entre os efeitos dos vários atributos sobre as probabilidades das classes, oferece uma maior compreensão do modelo do que uma grande tabela de números, o resultado tipicamente oferecido pelos métodos estatísticos clássicos.

Os algoritmos de indução de árvores de decisão apresentam diversas vantagens sobre os métodos de classificação estatísticos clássicos [Qui92, BFOS84] como, por exemplo:

1. A quantidade de cálculo realizada pelos algoritmos de indução de árvores de decisão é bastante mais pequena;
2. De uma forma geral, os classificadores produzidos (árvores de decisão) normalmente são mais fáceis de compreender.
3. A filtragem dos atributos irrelevantes para a tarefa de classificação é automática
4. Os procedimentos de classificação baseados em árvores de decisão apresentam melhor precisão na classificação em aplicações práticas do que os procedimentos baseados em métodos estatísticos.

Uma abordagem diferente dos algoritmos de indução de árvores de decisão que tem vindo a ser utilizada na aprendizagem automática são os algoritmos genéticos. Estes algoritmos são caracterizados por construírem sistemas tipicamente compostos de cadeias de vários classificadores onde os dados não estão disponíveis para avaliar e apurar todas as componentes individuais. Estes sistemas podem também ser descritos como desenhados para absorver nova informação continuamente a partir de ambientes com ruído e em tempo real, deduzindo conjuntos de hipóteses completas, expressas como regras e sem perturbar significativamente as capacidades já adquiridas. Os seus métodos são bastante diferentes dos métodos de indução de árvores de decisão onde as hipóteses são refinadas progressivamente em vez de serem construídas de uma vez e sem retrocesso.

Uma comparação realizada em domínios propostos para os algoritmos genéticos e que foram utilizados em algoritmos de indução de árvores de decisão mostrou que os algoritmos de indução de árvores necessitam de conjuntos de treino mais pequenos e obtêm uma precisão na classificação melhor que os algoritmos genéticos [MST94]. Os resultados sugerem também que o tempo de aprendizagem é menor nos algoritmos de indução de árvores de decisão, e que ambos os métodos exibem propriedades de escalonamento semelhantes.

2.2.3 Redes neuronais

As Redes Neuronais surgiram de diversas áreas, desde o fascínio de compreender e simular o cérebro humano, ao de copiar certas capacidades humanas como a fala e a utilização da linguagem, e até de disciplinas científicas e comerciais mais práticas como o reconhecimento de padrões, a modelização e a previsão [MST94].

Resumidamente, as redes neuronais consistem numa estrutura de nós interligados, em que cada nó tem como resultado uma função não linear da sua entrada. A entrada de um nó pode ser fornecida por outro nó ou directamente dos dados de entrada. A saída de alguns nós são consideradas como a saída da própria rede. A rede completa representa assim um conjunto bastante complexo de interdependências que pode incluir qualquer grau de não linearidade permitindo modelar funções muito gerais.

Em grande parte das redes neuronais a aprendizagem automática é realizada sobre uma estrutura fixa da rede, desenhada manualmente. A estrutura de uma rede neuronal mais comum, um *Percepton de Multi Camadas*, é formada por várias camadas de nós: uma camada de entrada, uma camada de saída e uma ou mais camadas intermédias "escondidas". Contudo não existe nenhuma regra formal para a escolha do número de nós em cada camada intermédia ou da topologia das ligações entre as camadas.

Nas redes neuronais mais simples, o resultado de um nó é fornecido a outro nó de forma semelhante a propagar uma mensagem por várias camadas de nós interligados. Contudo, comportamentos mais complexos podem ser modelados por redes em que o resultado final é fornecido aos nós de camadas anteriores criando, desta forma, um sistema que apresenta características de sistemas não lineares de grau bastante elevado e com realimentação. Tem sido argumentado que as redes neuronais simulam, até certo ponto, o comportamento das redes de neurónios no cérebro humano.

Assim que a estrutura de uma rede neuronal e fixada, existem diversos algoritmos desenhados para "treinar" a rede a partir de um conjunto de observações. Um dos algoritmos mais populares é o algoritmo de propagação do erro (*error backpropagation*). O treino da rede por este algoritmo pode ser bastante mais lento quando comparado com os algoritmos de indução de árvores de decisão. Contudo, é injusto comparar dois métodos apenas pelo tempo de aprendizagem quando estes apresentam diferentes níveis de precisão de classificação e quando, em alguns domínios, já foram apresentados melhores resultados por parte das redes neuronais, embora esta superioridade não seja constante.

Na classificação, as redes neuronais combinam a complexidade de algumas técnicas estatísticas com o objectivo da aprendizagem automática de simular o raciocínio humano. Contudo, a classificação é realizada a um nível bastante mais "inconsciente" não permitindo que os conceitos aprendidos sejam facilmente compreendidos pelos humanos, ou que a estrutura da rede seja facilmente alterada.

Comparativamente aos algoritmos de indução de árvores de decisão as redes neuronais apresentam, em alguns domínios, uma melhor precisão na classificação. Mas estes métodos apresentam também algumas desvantagens como tempos de aprendizagem maiores e uma maior dificuldade de compreensão dos conceitos mesmo em conjuntos de treino de tamanho moderado.

Como resultado de análises fornecidas por diversos estudos comparativos das varias vantagens dos diferentes algoritmos utilizados na classificação, muitos investigadores têm proposto sistemas híbridos dirigidos a retirar o melhor de dois ou mais métodos. Um exemplo destes sistemas são as *Perceptron Trees* que combinam os métodos das redes neuronais com as árvores de decisão. Tais sistemas híbridos podem usufruir de uma melhor precisão proporcionada pela flexibilidade da utilização de vários modelos para a representação dos conceitos, mas estes sistemas parecem também herdar, pelo menos, alguns dos problemas computacionais dos algoritmos em que se baseiam [Utg88].

2.2.4 Medidas de Desempenho

Para o desenvolvimento de um procedimento de classificação existem diversas questões relevantes no que diz respeito ao seu desempenho, quer durante a sua construção como na sua utilização. A seguir são apresentados alguns dos parâmetros considerados como mais importantes [MST94]:

Precisão: A precisão representa o grau de confiança de um procedimento de classificação, o que a torna no parâmetro mais importante a ter em conta em qualquer tarefa de indução. Sem a precisão tudo o resto é irrelevante.

A precisão é normalmente calculada como a razão entre o número de exemplos correctamente classificados sobre o número total de exemplos. Contudo, em alguns domínios, determinados erros podem ser considerados mais graves que outros e pode ser importante controlar o erro em relação a uma classe em particular [PFK98].

Experiências iniciais, em que foram utilizados conjuntos de dados com informação "perfeita", deram origem a diversas situações onde pequenos conjuntos de exemplos foram suficientes para conseguir obter procedimentos de classificação capazes de classificar observações em domínios consideravelmente grandes com elevada confiança. Infelizmente o mesmo não acontece para os domínios mais difíceis e com conjuntos de dados imperfeitos que têm vindo a ser abordados nos últimos anos.

Em certos domínios pode não ser possível atingir uma precisão perfeita, mas a utilização de grandes conjuntos de dados podem ajudar a melhorar a precisão do procedimento de classificação. Deve ser entendido que a utilização de conjuntos de dados maiores não são sempre a única nem a melhor forma de melhorar a precisão de um procedimento de classificação. Existem outras opções como a pesquisa de melhores atributos, ou o tratamento dos atributos disponíveis que podem dar origem a classificadores com maior grau de confiança a partir de conjuntos de dados mais pequenos. Estas opções são normalmente complementares e não excluem a utilização de um grande conjunto de dados. Os grandes conjuntos de dados existem, estão a crescer, e representam uma fonte potencial de precisão adicional que deve ser explorada.

Compreensão: Este parâmetro representa a facilidade de compreensão por parte dos humanos do procedimento de classificação. Se é pretendido que o procedimento de classificação seja executado por humanos é importante que este seja fácil de compreender caso contrário irão verificar-se erros na sua aplicação. É também importante que a pessoa que irá aplicar o procedimento, para além de o compreender, também acredite no processo de classificação. A compreensão do conceito obtido é bastante subjectiva e depende bastante do ponto de vista do observador. Um exemplo várias vezes mencionado é o caso da central nuclear Three-Mile Island onde os dispositivos automáticos recomendaram que a central fosse desligada mas esta recomendação não foi executada pelos operadores que não acreditaram no seu fundamento. Uma história semelhante é o caso do desastre de Chernobyl.

A compreensão do conceito não é apenas importante para o produto final. Também, durante o desenvolvimento de um procedimento de classificação, a sua compreensão permite reformular e melhorar os dados utilizados e consequentemente melhorar a sua precisão. Pode também diminuir o tamanho do procedimento e do conjunto de dados necessários para atingir um determinado nível de precisão.

Diversas qualidades estão relacionadas com o tamanho de um procedimento de classificação sendo a mais importante a compreensão do conceito obtido. O objectivo de minimizar o tamanho do procedimento de classificação, quer este seja representado por regras, árvores de decisão, ou outra estrutura, é bastante adoptado na construção dos algoritmos de aprendizagem com vista a aumentar a sua compreensão. Pela falta de uma medida geral para a compreensão, é normalmente utilizado o tamanho do conceito ou procedimento obtido.

Tempo de Aprendizagem: Este parâmetro representa o tempo necessário para um algoritmo de aprendizagem construir o procedimento de classificação a partir do conjunto de dados disponível. Quando são utilizados conjuntos de treino relativamente pequenos é muito pouco provável que o tempo de aprendizagem seja um factor condicionante. Mas, quando se utilizam grandes conjuntos de dados o tempo de aprendizagem tem de ser controlado pois existe o risco de se tornar insuportavelmente grande, visto que este cresce à medida que o número de exemplos utilizados cresce. Por exemplo, em ambientes que mudam muito rapidamente pode ser necessário aprender o procedimento de classificação, ou adaptar o procedimento já existente, muito rapidamente ou em tempo real. Pode também implicar que o procedimento de classificação seja obtido a partir de um número pequeno de observações.

Velocidade: A velocidade representa o tempo utilizado por um procedimento de classificação para classificar uma nova observação. Em algumas circunstâncias a velocidade de um classificador é um dos aspectos fundamentais. Por vezes pode ser preferível um classificador com uma precisão de 90% a um de 95% se este for 100 vezes mais rápido. Considerações desta natureza podem ser levantadas para casos como a ordenação de correspondência ou detecção de defeitos numa linha de montagem. O tempo de classificação não é normalmente considerado como um factor muito importante, mas pode-o ser em aplicações de tempo real, como o controlo de sistemas dinâmicos.

2.3 Árvores de decisão

Diversos modelos de classificação têm sido propostos ao longo dos anos, como por exemplo, redes neuronais, modelos estatísticos, árvores de decisão ou algoritmos genéticos. Dentro destes modelos as árvores de decisão têm sido considerados como os mais adequados para extracção do conhecimento e para a aprendizagem automática. Os algoritmos de indução de árvores de decisão apresentam tempos de aprendizagem menores que os outros métodos para além dos seus modelos (as árvores de decisão) serem mais simples de entender. Os procedimentos de classificação baseados em árvores de decisão são também de execução eficiente. As árvores de decisão podem ser facilmente expressas como uma subrotina em qualquer linguagem de programação, integradas numa aplicação, e executadas. (A subrotina pode ser uma expressão if-then-else encadeada ou um simples interpretador de uma estrutura codificada) [Cat91, MST94, Rug00, Qui92].

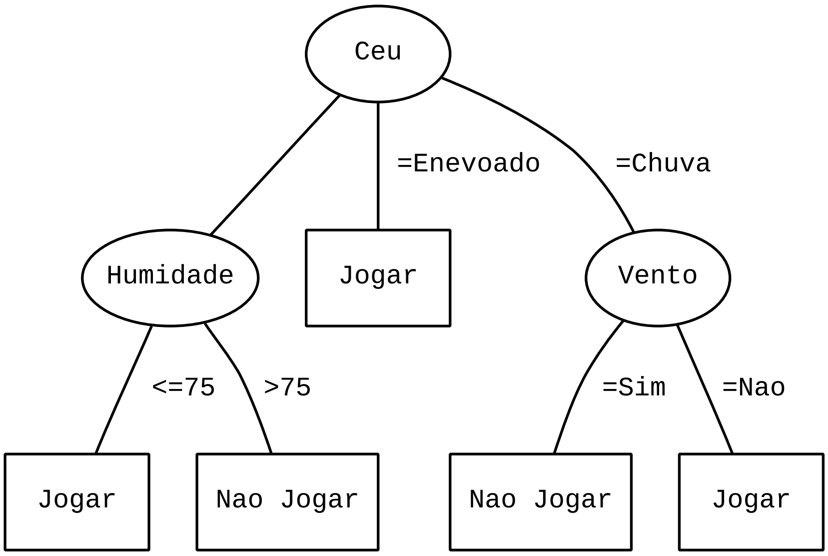


Figura 2.1: Exemplo de uma árvore de decisão.

2.3.1 Definição

Uma *árvore de decisão* que, como o próprio nome indica, é uma estrutura de dados em árvore que tanto pode ser:

- uma *folha*, que representa uma classe, ou
- um *nó de decisão* que representa um teste a ser realizado sobre o valor de um único atributo, com um ramo e uma sub-árvore para cada resultado possível desse teste.

Uma árvore de decisão pode ser utilizada para prever a classe de uma observação, ou exemplo, iniciando o processo de classificação na raiz da árvore e ir percorrendo a estrutura até ser encontrada uma folha. Em cada nó de decisão, o teste representado pelo nó é aplicado ao valor do atributo da observação e a classificação é continuada na sub-árvore correspondente ao resultado obtido. Quando, através deste processo, é finalmente (e inevitavelmente) encontrada uma folha, a classe prevista para o exemplo é a classe representada por essa folha.

A figura 2.1 mostra um exemplo de uma árvore de decisão utilizada para classificar o conjunto de exemplos representados na tabela 2.1. Cada um destes exemplos indica se, dependendo do estado do tempo descrito por quatro atributos (estado do céu, temperatura, humidade e vento), é um bom dia ou não para ir jogar futebol para o parque.

Tomando como exemplo a primeira observação:

Céu	Temperatura	Humidade	Vento	Classe
Limpo	75	70	Sim	Jogar
Limpo	69	70	Não	Jogar
Limpo	85	85	Não	Não Jogar
Limpo	80	90	Sim	Não Jogar
Limpo	72	95	Não	Não Jogar
Enevoado	72	90	Sim	Jogar
Enevoado	83	78	Não	Jogar
Enevoado	64	65	Sim	Jogar
Enevoado	81	75	Não	Jogar
Chuva	71	80	Sim	Não Jogar
Chuva	65	70	Sim	Não Jogar
Chuva	75	80	Não	Jogar
Chuva	68	80	Não	Jogar
Chuva	70	96	Não	Jogar

Tabela 2.1: Conjunto de observações utilizadas como exemplo.

Céu = Limpo; Temperatura = 75; Humidade = 70 e Vento = Sim.

A árvore de decisão ilustrada pode ser utilizada para classificar esta observação da seguinte forma. Na raiz da árvore é representado um teste sobre o atributo *Céu* que, para a observação considerada, é igual a *Limpo*. Neste ponto é considerado o ramo da esquerda e a sub-árvore associada. A raiz da sub-árvore esquerda representa um teste sobre o atributo *Humidade* que é comparado com o valor 75. Para a observação considerada este atributo é igual a 70. Este valor é menor do que 75 e, como indicado na árvore, é considerado mais uma vez o ramo e a sub-árvore da esquerda. Neste ponto é encontrada uma folha que representa a classe prevista para a observação. O resultado da classificação do exemplo considerado é *Jogar*. Isto é, um dia com *Céu Limpo* e *Humidade* igual a 70 é um bom dia para ir *Jogar* futebol para o parque.

2.3.2 Construção da árvore de decisão

Para a construção das árvores de decisão é normalmente utilizada uma técnica de "dividir para conquistar". Muito resumidamente, esta técnica consiste em dividir recursivamente o conjunto de treino até todos os subconjuntos conterem observações, ou exemplos, que pertencem a apenas uma classe. As divisões do conjunto de treino são realizadas com base nos valores possíveis de um dos atributos que descrevem os exemplos. A cada divisão é associado um nó de decisão na árvore que representa o teste realizado aos valores do atributo que deu origem à divisão. Quando todos os exemplos de um conjunto pertencerem à mesma classe, este é associado a uma folha na árvore que representa a classe dos exemplos.

Este processo de construção de uma árvore de decisão pode ser resumido pelos passos

Céu	Temperatura	Humidade	Vento	Classe	
Limpo	75	70	Sim	Jogar	
Limpo	69	70	Não	Jogar	
Limpo	85	85	Não	Não Jogar	(b)
Limpo	80	90	Sim	Não Jogar	
Limpo	72	95	Não	Não Jogar	
Enevoado	72	90	Sim	Jogar	(a)
Enevoado	83	78	Não	Jogar	
Enevoado	64	65	Sim	Jogar	
Enevoado	81	75	Não	Jogar	
Chuva	71	80	Sim	Não Jogar	(a)
Chuva	65	70	Sim	Não Jogar	
Chuva	75	80	Não	Jogar	(c)
Chuva	68	80	Não	Jogar	
Chuva	70	96	Não	Jogar	

Figura 2.2: Divisões do conjunto de treino da tabela 2.1 na construção da árvore da figura 2.3.

seguintes, onde T representa o conjunto de exemplos utilizados:

```
Constrói Árvore(T):
   Se todos os exemplos em T pertencem à classe Ci:
      Retorna uma folha que representa a classe Ci.
   Se não:
      Divide o conjunto de exemplos T baseado num único atributo A.
      Constrói o nó de decisão N que representa o teste sobre o atributo A.
      Para cada subconjunto Tj resultante da divisão:
         Filho j de N = Constrói Árvore(Tj).
      Retorna N.
```

Nas figuras 2.2 e 2.3 é ilustrado o processo de construção da árvore de decisão e das sucessivas divisões do conjunto de treino representado na tabela 2.1. A raiz da árvore é criada pela primeira divisão do conjunto de treino (a) que o divide em três subconjuntos. O primeiro subconjunto é utilizado para a construção da sub-árvore esquerda (valores do atributo *Céu* iguais a *Limpo*), o segundo utilizado para a sub-árvore do meio (*Céu* = *Enevoado*) e o terceiro utilizado para a sub-árvore direita (*Céu* = *Chuva*). A raiz da sub-árvore esquerda é criada pela divisão do primeiro subconjunto (b) baseada no atributo *Humidade*. Como resultado desta divisão são criados dois subconjuntos cada um formado por exemplos que pertencem a apenas uma classe. Ao primeiro subconjunto é associada uma folha na árvore que representa a classe *Jogar* e ao segundo uma folha que representa a classe *Não Jogar*. O mesmo processo é aplicado aos restantes subconjuntos que dão origem aos restantes nós da árvore.

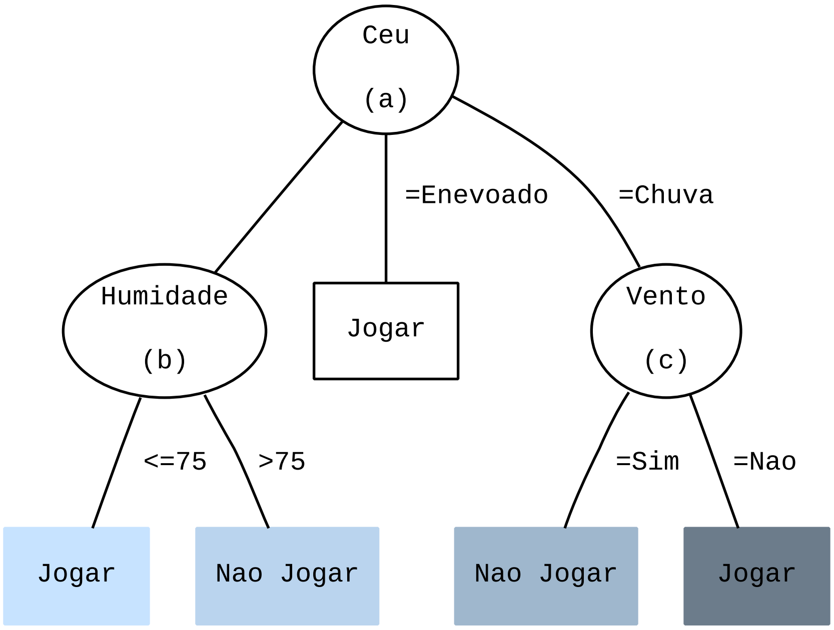


Figura 2.3: Ilustração do processo de construção da árvore de decisão a partir do conjunto de treino da tabela 2.1.

O tamanho e a precisão das árvores de decisão construídas por este processo dependente bastante da escolha dos atributos utilizados para a divisão do conjunto de treino. Os vários programas de indução de árvores de decisão utilizam, mais ou menos, a mesma técnica de "dividir para conquistar" na construção das suas árvores. A principal diferença entre os vários algoritmos reside na forma como são escolhidos os atributos para a divisão dos exemplos num nó.

A escolha de um atributo para a divisão de um conjunto de exemplos associado a um nó é normalmente realizada com base num *critério de preferência*. Os critérios de preferência mais populares são baseados na teoria da informação. Cada subconjunto, formado pela divisão do conjunto de dados, é avaliado pela informação esperada do seu conteúdo. Subconjuntos dominados por uma classe apresentam uma menor informação esperada. O critério de ganho, um critério deste tipo bastante simples e comum, escolhe o atributo que dá origem à divisão que apresenta a menor informação média esperada, onde a média é pesada de acordo com o tamanho relativo dos subconjuntos.

Os algoritmos de indução de árvores de decisão são, na sua generalidade, algoritmos "gulosos" e sem "previsão": a escolha de um atributo é fixada sem considerar o seu impacto nas divisões futuras dos subconjuntos resultantes da divisão actual. Uma má escolha de um atributo pode fragmentar o conjunto de treino e reduzir a precisão, logo um bom critério de escolha é fundamental.

Uma outra questão importante na construção das árvores de decisão é a sua simplificação. O processo de construção da árvore descrito tem como resultado árvores de decisão perfeitamente ajustadas ao conjunto de exemplos utilizados na sua construção. Como a construção da árvore de decisão só termina quando todos os subconjuntos forem formados por exemplos que pertencem a uma só classe, a árvore não irá apresentar quaisquer erros ao classificar o conjunto de treino utilizado para a sua construção. Contudo, é impossível concluir que, uma árvore de decisão que apresente uma precisão de 100% ao classificar o conjunto de treino utilizado para sua construção, irá classificar um novo conjunto de exemplos com a mesma precisão de 100%. Em tarefas reais este problema é normalmente acrescido de imperfeições no conjunto de treino. Os atributos disponíveis podem simplesmente não conter informação suficiente para prever a classe com exactidão, independentemente do número de exemplos utilizados no treino. Pior ainda, alguns valores dos atributos ou da classe podem ser incorrectos, possivelmente devido à não exactidão da fonte ou por erro humano na introdução dos dados. Tais erros são denominados de ruído.

Para a simplificação das árvores de decisão são normalmente utilizados dois métodos:

- O primeiro método consiste em utilizar um determinado critério para verificar se, antes de realizar a divisão dos exemplos num nó, a divisão é relevante para a classificação final. Caso não o seja, esta não é realizada e a exploração desse ramo da árvore é terminada.
- O segundo método, muitas vezes denominado de *poda*, é aplicado após a construção da árvore. Cada nó de decisão da árvore é visitado e os ramos considerados como não contribuírem significativamente para a classificação são simplificados.

Ambos os métodos podem dar origem a árvores que não são completamente consistentes com os exemplos de treino, mas a precisão da árvore ao classificar um novo conjunto de observações é normalmente superior [EMS93].

2.4 O sistema C4.5

Os algoritmos de classificação têm atraído um considerável interesse nas áreas de investigação da Inteligência Artificial, em particular nas áreas da extracção do conhecimento e da aprendizagem automática. Entre os algoritmos de classificação e, em especial, entre os algoritmos de indução de árvores de decisão, o sistema C4.5 proposto por J. R. Quinlan [Qui92] merece uma atenção especial por diversas razões [Rug00]. Por um lado e de forma semelhante a outros sistemas como, por exemplo, o sistema CART [BFOS84], o C4.5 representa o resultado de vários anos de investigação na aprendizagem automática e na extracção do conhecimento. Como tal, tem sido tomado como ponto de referência para o desenvolvimento e análise de novas propostas. Por outro lado os resultados de diversos testes realizados mostram que o algoritmo de indução de árvores de decisão do C4.5 apresenta uma boa precisão na classificação e é um dos mais rápidos entre os algoritmos de *main-memory* comparados para a extracção do conhecimento e aprendizagem automática [MST94, Rug00].

2.4.1 Construção da árvore de decisão

O algoritmo de construção de uma árvore de decisão do sistema C4.5 é muito semelhante ao algoritmo de construção de uma árvore apresentado. A grande diferença entre o algoritmo do sistema C4.5 e os outros algoritmos de construção de árvores de decisão reside, essencialmente, no critério de avaliação utilizado para a escolha do atributo em que se baseiam as divisões do conjuntos de treino.

O processo de construção das árvore de decisão no C4.5 pode ser dividido em duas fases. Na primeira fase a árvore é construída obedecendo à regra em que a construção só termina quando todos os exemplos associados a um nó pertencem à mesma classe (ou o seu número for menor que um valor pré determinado). Contudo, uma árvore de decisão construída de acordo com esta regra encontra-se, tipicamente, sobre especializada ao conjunto de treino utilizado. Na segunda fase do processo é realizada uma simplificação da árvore de decisão. Cada sub-árvore é avaliada e são simplificadas aquelas que forem consideradas como não contribuírem com um aumento significativo para precisão.

2.4.1.1 Crescimento

O algoritmo de construção de árvores de decisão do C4.5 associa a cada nó um conjunto de exemplos. No início do algoritmo existe apenas a raiz da árvore à qual está associado todo o conjunto de treino. Para cada nó da árvore são aplicados os seguintes passos ao seu conjunto de exemplos T:

```
Constrói Árvore(T):
1. Determina a distribuição dos exemplos pelas classes em T.
2. Se todos os exemplos em T pertencem à mesma classe Ci:
   2.1. Retorna uma folha que representa a classe Ci.
3. Se os exemplos em T pertencem a mais do que uma classe:
   3.1. Encontra a melhor divisão para o conjunto de exemplos T.
   3.2. Divide o conjunto de exemplos T e constrói um nó de decisão N.
   3.3. Para cada subconjunto Tj criado pela divisão:
        3.3.1. Filho j de N = Constrói Árvore(Tj).
   3.4. Retorna o nó de decisão N.
```

No início da construção de um nó da árvore de decisão o algoritmo começa por encontrar a distribuição dos exemplos pelas classes para o conjunto de exemplos T associados ao nó (passo 1). Para determinar as distribuições, o algoritmo percorre o conjunto de exemplos associados ao nó para determinar o número de exemplos que pertence a cada uma das classes. Se todos os exemplos pertencem à mesma classe, ou se o seu número for inferior a um determinado valor, então o nó é uma folha que representa a classe Ci ou, respectivamente, a classe mais frequente (passo 2.1).

Contudo, se o conjunto de exemplos T for formado por exemplos que pertencem a mais do que uma classe então T é dividido baseado nos valores de um único atributo (passo 3.2). O nó é então um nó de decisão que representa o teste realizado ao atributo escolhido. Após a divisão do conjunto de exemplos os mesmos passos são aplicados de forma recursiva a cada um dos subconjuntos resultantes da divisão (passo 3.3.1).

Como o algoritmo de construção da árvore do C4.5 pára a exploração de um nó quando o número de exemplos associados a esse nó for menor que um determinado valor pode dar-se o caso de uma folha cobrir exemplos que não pertencem à classe representada. Isto dá origem a erros de classificação. Quando uma folha é criada o erro de classificação é calculado como o número de exemplos que não pertencem à classe representada pela folha (passo 2.1).

Para um nó de decisão, o erro de classificação é calculado como a soma dos erros de classificação dos seus filhos. No caso do erro de classificação calculado para o nó de decisão for superior ao número de exemplos que não pertencem à classe mais frequente no nó, este é substituído por uma folha. Esta representa a classe mais frequente nos exemplos e o seu erro de classificação é igual ao número de exemplos cobertos que não pertencem a essa classe.

O passo mais importante do algoritmo e que o diferencia dos vários algoritmos de construção de árvores de decisão é o passo onde é escolhido o atributo que dá origem à divisão do conjunto de exemplos (passo 3.1). O tamanho e a precisão das árvores de decisão depende fortemente da escolha dos atributos para a divisão dos exemplos. Uma má escolha de um atributo pode dar origem a uma má divisão do conjunto de treino e reduzir a precisão, logo um bom critério de escolha é muito importante.

A divisão do conjunto de exemplos associados a um nó é realizada pela aplicação de um teste aos valores de um único atributo. O C4.5 admite dois tipos de atributos: atributos *contínuos*, cujos valores são números reais, e *discretos*, cujos valores encontram-se restritos a um número finito (normalmente pequeno) de valores possíveis. Baseado nestes dois tipos de atributos o C4.5 considera dois testes possíveis para a divisão do conjunto de exemplos:

- Testes sobre atributos discretos que têm como resultado a divisão do conjunto de exemplos em n subconjuntos, um para cada valor possível do atributo;
- Testes sobre atributos contínuos que têm como resultado a divisão do conjunto de exemplos em dois subconjuntos, um com os valores do atributo menores ou iguais a um determinado valor e o outro com os valores do atributo maiores.

Para a escolha do melhor atributo, o algoritmo de construção de árvores de decisão avalia cada uma das possíveis divisões obtidas pela aplicação do respectivo teste a cada um dos atributos. A avaliação de uma divisão é realizada segundo o critério *razão de ganho*. Para a avaliação de uma divisão este critério baseia-se na informação das distribuições dos exemplos pelas classes antes e depois da divisão.

Supondo um conjunto de exemplos T e k classes possíveis, então a quantidade média de informação necessária para identificar a classe de um exemplo em T pode ser calculada da seguinte forma:

$$Info(T) = -\sum_{j=1}^{k} \frac{freq(C_j,T)}{|T|} * log_2(\frac{freq(C_j,T)}{|T|})\ bits \tag{2.1}$$

em que $freq(C_j, T)$ é o número de exemplos em T que pertencem à classe C_j e $|T|$ é o número total de exemplos em T (cardinalidade de T). A quantidade $Info(T)$ é também conhecida como *entropia*.

O ganho de informação obtido com uma divisão do conjunto de exemplos pode ser calculado pela diferença entre a quantidade de informação necessária para a identificação da classe de um exemplo antes da divisão e a média pesada da quantidade de informação necessária nos vários subconjuntos criados com a divisão. Supondo que o conjunto de exemplos T é dividido em m subconjuntos, então o ganho de informação é definido como:

$$Ganho = Info(T)\ -\ \sum_{i=1}^{m} \frac{|T_i|}{|T|} * info(T_i)\ bits \tag{2.2}$$

O ganho de informação pode ser utilizado como critério de avaliação de uma divisão do conjunto de exemplos, mas este apresenta uma grave tendência para valorizar divisões que tenham como resultado um número elevado de subconjuntos. Por exemplo, considerando um conjunto de exemplos em que um dos atributos é um identificador sequencial dos exemplos. Então a divisão do conjunto baseada neste atributo tem como resultado tantos subconjuntos como o número de exemplos. Como cada subconjunto apenas contém um exemplo que, logicamente, pertence a uma única classe, a entropia de cada subconjunto iria ser 0 resultando num ganho de informação máximo. Mas, para a construção de uma árvore de decisão, uma divisão como esta é completamente inútil.

De forma análoga à definição de entropia, pode-se definir o potencial de informação gerado por uma divisão de um conjunto de exemplos T em m subconjuntos como:

$$Split\ Info = -\sum_{i=1}^{m} \frac{|T_i|}{|T|} * log_2(\frac{|T_i|}{|T|})\ bits \tag{2.3}$$

A razão:

$$Razao\ de\ Ganho = -\frac{Ganho}{Split\ Info}\ bits \tag{2.4}$$

Representa a proporção de informação gerada pela divisão que é útil à classificação. Se a divisão for trivial, como no exemplo anterior, o $Split\ Info$ vai ser maior e a razão

	Céu	Temperatura	Humidade	Vento	Classe
	Limpo	75	70	Sim	Jogar
	Limpo	69	70	Não	Jogar
T1	Limpo	85	85	Não	Não Jogar
	Limpo	80	90	Sim	Não Jogar
	Limpo	72	95	Não	Não Jogar
	Enevoado	72	90	Sim	Jogar
	Enevoado	83	78	Não	Jogar
T2	Enevoado	64	65	Sim	Jogar
	Enevoado	81	75	Não	Jogar
	Chuva	71	80	Sim	Não Jogar
	Chuva	65	70	Sim	Não Jogar
T3	Chuva	75	80	Não	Jogar
	Chuva	68	80	Não	Jogar
	Chuva	70	96	Não	Jogar

Figura 2.4: Divisão do conjunto de exemplos baseada no atributo *Céu*.

de ganho torna-se instável, penalizando as divisões que tenham como resultado um número elevado de subconjuntos.

O algoritmo de construção de árvores de decisão do C4.5 utiliza a razão de ganho como critério de avaliação das divisões do conjunto de exemplos. Para evitar a instabilidade deste critério quando são criados muitos subconjuntos, a divisão escolhida é aquela que, obviamente, maximiza a razão de ganho e que o respectivo ganho de informação é suficientemente grande, pelo menos tão grande como a média do ganho de todas as divisões avaliadas.

A avaliação de uma divisão de um conjunto de dados baseada num atributo discreto é bastante simples de realizar. Considerando o conjunto de exemplos que tem vindo a ser utilizado, novamente representado na figura 2.4. A divisão deste conjunto baseada no atributo *Céu* tem como resultado tantos subconjuntos como o número de valores possíveis do atributo. Neste caso são criados 3 subconjuntos cada um contendo os exemplos com os valores do atributo correspondentes.

A entropia do conjunto de exemplos e de cada subconjunto é:

$$\begin{array}{lllll}
Info(T) & = & -9/14 * log_2(9/14) - 5/14 * log_2(5/14) & = & 0.410 + 0.530 & = & 0.940 \\
Info(T_1) & = & -2/5 * log_2(2/5) - 3/5 * log_2(3/5) & = & 0.529 + 0.442 & = & 0.971 \\
Info(T_2) & = & -4/4 * log_2(4/4) - 0/4 * log_2(0/4) & = & 0 & & \\
Info(T_3) & = & -3/5 * log_2(3/5) - 2/5 * log_2(2/5) & = & 0.442 + 0.529 & = & 0.971
\end{array}$$

Logo, o ganho de informação correspondente à divisão do conjunto de exemplos pelo atributo é:

$$Ganho(Ceu) \quad = \quad 0.940 * 5/14 - 0.971 * 4/14 - 0 - 5/14 * 0.971 \quad = \quad 0.246$$

O potencial de informação e a razão de ganho correspondentes à divisão são calculados como:

$$\begin{aligned} Split\,Info(Ceu) &= -5/14 * log_2(5/14) - 4/14 * log_2(4/14) - 5/14 * log_2(5/14) \\ &= 0.530 + 0.516 + 0.530 \\ &= 1.577 \end{aligned}$$

$$Razao\,de\,Ganho(Ceu) = 0.246/1.577 = 0.156$$

A partir das equações 2.1 e 2.2 é fácil de verificar que, no caso de um atributo discreto ter sido utilizado para a divisão do conjunto de exemplos num nó antecessor da árvore, o seu ganho no nó actual é nulo. Logo, o algoritmo de construção de árvores de decisão do C4.5 nem sequer considera estes atributos para a divisão.

Para os atributos contínuos, o C4.5 considera a divisão do conjunto de exemplos associados a um nó em dois subconjuntos. O primeiro com os valores do atributo menores ou iguais que um determinado valor (também denominado de ponto de divisão) e o segundo com os valores maiores. Apesar do atributo ser contínuo, este apenas contém um número finito de valores. Supondo os valores ordenados de um atributo contínuo $v_1, v_2, ..., v_n$. Então, qualquer ponto entre v_i e v_{i+1} divide o conjunto de exemplos em dois subconjuntos $v_1, v_2, ..., v_i$ e $v_{i+1}, v_{i+2}, ..., v_n$ e pode ser considerado como um potencial ponto de divisão. Desta forma, apenas existem $n - 1$ divisões possíveis do conjunto de exemplos.

A avaliação de uma divisão do conjunto de exemplos baseada num atributo contínuo analisa as $n - 1$ possíveis divisões segundo o critério de razão de ganho que, se o conjunto de exemplos estiver ordenado pelos valores do atributo, é conseguido com apenas uma passagem pelos exemplos. O valor final da razão de ganho considerado para a avaliação corresponde ao maior valor da razão de ganho das $n - 1$ divisões avaliadas.

Após a avaliação e caso o atributo seja escolhido para a divisão dos exemplos, o ponto de divisão é o valor de todos os valores do atributo no conjunto de treino que não exceda o ponto médio dos limites da divisão $((v_i + v_{i+1})/2)$. Desta forma o C4.5 garante que o ponto de divisão que aparece na árvore de decisão é um valor que também aparece nos dados.

Na figura 2.5 é representado um subconjunto do mesmo conjunto de exemplos que tem vindo a ser utilizado nos exemplos anteriores. O subconjunto representado, ordenado pelos valores do atributo *Humidade*, é o primeiro subconjunto obtido pela divisão baseada nos valores do atributo *Céu*. A lado dos exemplos são exibidos os valores do ganho de informação e da razão de ganho para cada uma das 4 divisões possíveis baseadas no atributo *Humidade*.

Céu	Temperatura	Humidade	Vento	Classe
Limpo	75	70	Sim	Jogar
Limpo	69	70	Não	Jogar
Limpo	85	85	Não	Não Jogar
Limpo	80	90	Sim	Não Jogar
Limpo	72	95	Não	Não Jogar

Ganho	Razã de Ganho
0,322	0,446
0,971	1
0,737	0,759
0,171	0,237

Figura 2.5: Avaliação da divisão do conjunto de exemplos baseada no atributo *Humidade*.

2.4.1.2 Simplificação

Existem basicamente duas formas de realizar a simplificação de uma árvore de decisão. A primeira consiste em decidir não continuar a exploração de um nó com base em algum critério de verificação da relevância do atributo escolhido para a divisão dos exemplos num nó. Na segunda forma, a simplificação é realizada após a construção da árvore em que as sub-árvores avaliadas como não estarem a contribuir significativamente para a precisão são simplificadas [EMS93].

A primeira abordagem apresenta a vantagem de não ser gasto tempo na construção de partes da árvore de decisão que depois irão ser removidas. Mas um bom critério de avaliação da relevância de um atributo não é fácil de definir. Um critério demasiado restritivo pode parar a construção da árvore antes de os benefícios das sucessivas divisões se tornarem evidentes para a classificação. Por outro lado, um critério pouco selectivo pode resultar numa simplificação bastante pequena.

A segunda abordagem, apesar de só ser efectuada após a construção da árvore de decisão, é compensada por realizar uma melhor exploração das divisões do conjunto de treino. A completa exploração e posterior simplificação das árvores de decisão é um processo um pouco mais lento mas que normalmente tem como resultado árvores com melhor precisão na classificação.

O C4.5 utiliza uma variação de um método de simplificação posterior da árvore denominado de simplificação pessimista. A simplificação pessimista apresenta a vantagem em relação aos outros métodos de simplificação de não requerer que o conjunto de exemplos utilizados na simplificação seja distinto do conjunto utilizado na construção da árvore de decisão. Mas, em aplicações onde seja possível a utilização de conjuntos distintos pode dar origem a uma precisão significativamente maior.

O algoritmo de simplificação de árvores de decisão do C4.5 utiliza um ajuste estatístico para estimar o erro de classificação esperado para cada sub-árvore de decisão em novas observações.

A estimativa do erro de classificação de uma folha da árvore é calculado da seguinte forma. Se uma folha cobre N exemplos em que E desses exemplos não pertencem à classe representada, o erro de classificação da folha é dado por E/N. O erro de classificação pode ser considerado como E "acontecimentos" em N "tentativas". Se o conjunto de N exemplos cobertos pela folha for considerado como uma amostra pode-se averiguar o significado do erro de classificação em relação à probabilidade do erro para toda a população de onde foi retirada a amostra. Embora não se consiga determinar com exactidão a probabilidade do erro este apresenta uma distribuição de probabilidade representada por um intervalo de confiança. Para um determinado nível de confiança CF o limite superior da probabilidade pode ser facilmente encontrado a partir dos limites de confiança para a distribuição binomial aqui representada como $U_{CF}(E, N)$.

A estimativa do erro de classificação para uma folha que cubra N exemplos com um limite superior do erro previsto de $U_{CF}(E, N)$ é dado por $N * U_{CF}(E, N)$ exemplos mal classificados. O erro previsto para um nó de decisão é calculado como a soma dos erros previstos para cada um dos seus filhos.

A simplificação da árvore de decisão é realizada iniciando o processo pelos nós mais baixos da árvore. Um nó de decisão A é substituído por uma folha, que representa a classe predominante desse nó, se o erro de classificação estimado para o nó A for superior ao erro estimado para a folha quando esta cobre os mesmos exemplos que o nó.

De forma semelhante o algoritmo de simplificação do C4.5 também examina o filho do nó que cobre mais exemplos. Se o erro de classificação estimado para o nó filho, quando este é utilizado para classificar os mesmos exemplos cobertos pelo pai, for menor do que o erro estimado para o pai então o nó é substituído pelo seu filho e os outros filhos são removidos da árvore.

Pode-se perguntar o porquê do C4.5 apenas considerar o filho que cobre mais exemplos. A intuição sugere que este é o filho mais provável de contribuir para uma melhoria na precisão da árvore e que a análise de todos os outros ramos levaria bastante mais tempo. Experiências realizadas em alguns domínios mostraram que, apesar de alguns filhos diferentes do que cobria mais exemplos ocasionalmente deram origem a estimativas de erro mais baixas. A avaliação de todos os ramos raramente deu origem a um erro de classificação menor para a árvore simplificada. O tempo de simplificação foi maior mas não por um factor muito grande [Cat91].

Para exemplificar o algoritmo de simplificação é utilizada a árvore de decisão ilustrada na figura 2.6. Esta árvore foi obtida a partir de um conjunto de dados sobre as votação do Congresso Americano [Qui92]. Os número (N/E) indicados em cada folha representam que a folha cobre N exemplos com E mal classificados.

Para a simplificação da árvore o C4.5 utiliza um nível de confiança por defeito de 25% para o calculo do limite superior da probabilidade de erro. A sub-árvore com raiz no nó de decisão (c) não classifica erradamente nenhum exemplo do conjunto de treino. Para a primeira folha com $N = 6, E = 0$ e um limite superior do erro igual a $U_{25\%}(0, 6) = 0.206$ o número estimado de exemplos mal classificados é igual a $6 * 0.206 = 1.236$ quando esta classifica 6 exemplos novos. Para as restantes folhas, $U_{25\%}(0, 9) = 0.143$ e $U_{25\%}(0, 1) = 0.750$, fazem com que o erro previsto para a sub-

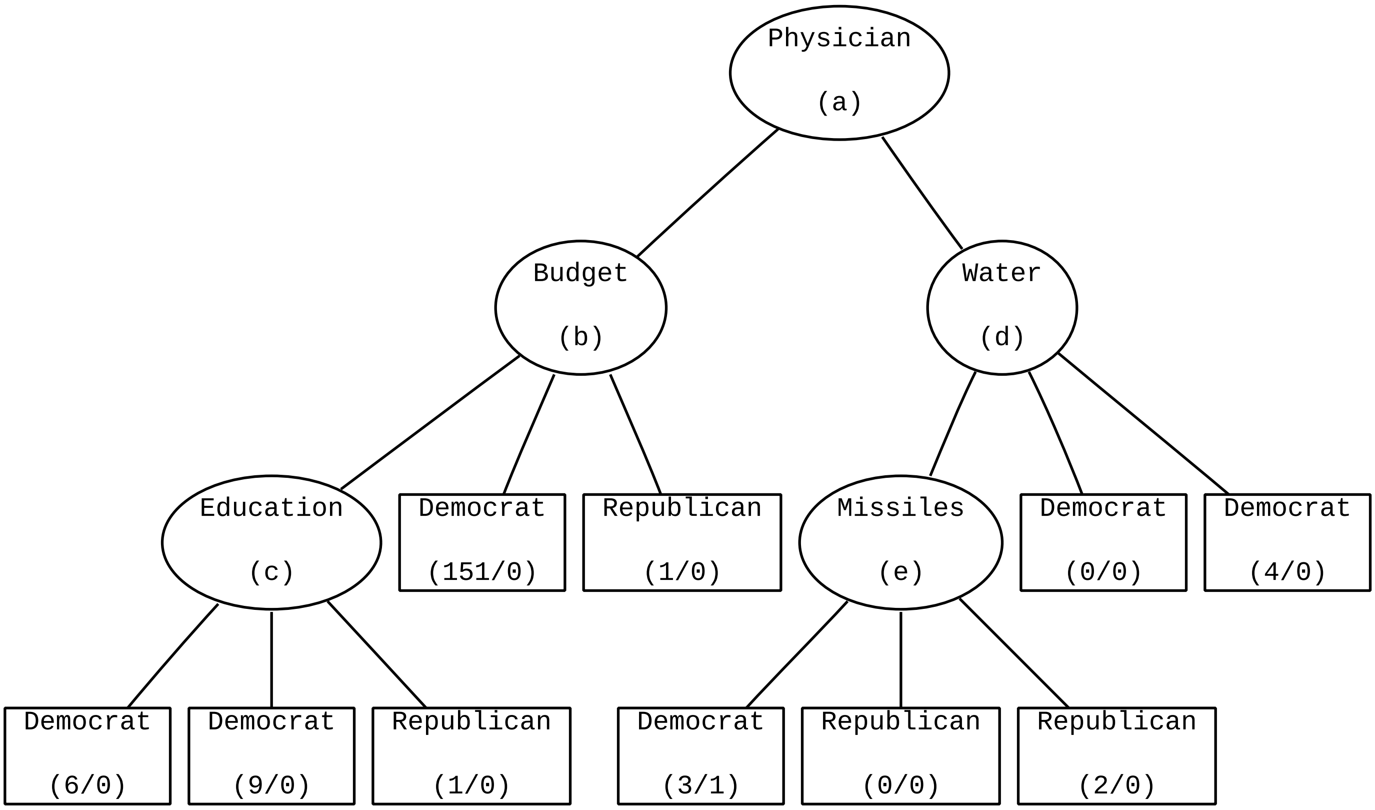


Figura 2.6: Árvore de decisão obtida a partir dos dados de votação do Congresso Americano.

árvore seja igual a $6 * 0.206 + 9 * 0.143 + 1 * 0.750 = 3.273$..

Se esta sub-árvore for substituída por uma folha que represente a classe *democrat* (a classe predominante), esta irá cobrir os mesmos 16 exemplos com apenas um exemplo mal classificado. O número estimado de exemplos mal classificados para esta folha é dada por:

$16 * U_{25\%}(1, 16) = 16 * 0.157 = 2.512$

Como a estimativa do erro de classificação para a sub-árvore (c) é superior à da folha, este nó é simplificado substituindo-o por uma folha que representa a classe *democrat*.

A sub-árvore imediatamente a cima (com raiz no nó (b)) tem agora o aspecto indicado na figura 2.7.

O erro de classificação estimado para esta sub-árvore (figura 2.7) é dado por:

$16 * U_{25\%}(1, 16) + 151 * U_{25\%}(0, 151) + 1 * U_{25\%}(0, 1) = 4.642$

Se esta sub-árvore for substituída por uma folha que represente, mais uma vez, a classe *democrat*, o erro previsto é igual a $168 * U_{25\%}(1, 168) = 2.610$. O erro de classificação

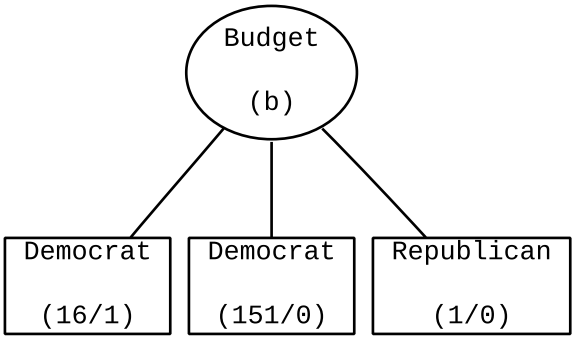


Figura 2.7: Sub-árvore obtida pela simplificação de um dos ramos da árvore da figura 2.6.

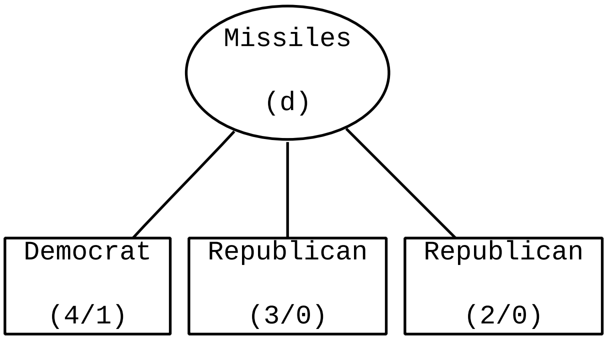


Figura 2.8: Simplificação da sub-árvore (d) substituída pela sub-árvore (e).

previsto para a folha é menor que o previsto para a sub-árvore logo esta é substituída pela folha.

Por último a sub-árvore com raiz no nó (d) apresenta uma estimativa do erro dada por:

$$\begin{array}{lcl} 0 + 4 * U_{25\%}(0,4) + (0 + 3 * U_{25\%}(1,3) + 2 * U_{25\%}(0,2)) & = & \\ 0 + 4 * 0.293 + (0 + 3 * 0.686 + 2 * 0.500) & = & 5.230 \end{array}$$

Se esta sub-árvore for substituída por uma folha que representa a classe *democrat*, a estimativa do erro para a folha é dada por:

$$9 * U_{25\%}(3,9) = 9 * 0.504 = 4.536$$

Mas se a sub-árvore for substituída pela sub-árvore filha com raiz no nó (e), isto é, se o nó de decisão for substituído pelo seu filho que cobre mais exemplos, este iria classificar os exemplos cobertos pelo pai como ilustrado na figura 2.8.

A estimativa do erro para a sub-árvore é dada por:

$$3 * U_{25\%}(0,3) + 4 * U_{25\%}(1,4) + 2 * U_{25\%}(0,2) = 3 * 0.367 + 4 * 0.547 + 2 * 0.500 = 4.289$$

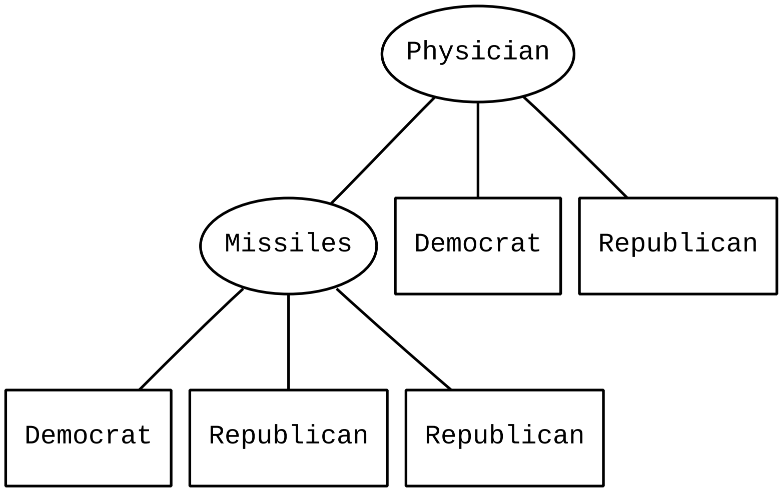


Figura 2.9: Resultado obtido pela simplificação da árvore da figura 2.6.

O erro previsto para a substituição é menor que a estimativa do erro para a sub-árvore original. Logo a sub-árvore com raiz no nó (d) é simplificada substituindo-a pela sub-árvore com raiz no nó (e).

Neste ponto não é possível realizar mais nenhuma simplificação que provoque uma redução do erro previsto. A simplificação da árvore de decisão da figura 2.6 tem como resultado a árvore apresentada na figura 2.9.

2.4.2 Valores desconhecidos

Até agora, para a descrição do algoritmo de construção de árvores de decisão do C4.5 foi sempre assumido que todos os valores eram conhecidos. Infelizmente os dados muitas vezes contêm valores de atributos desconhecidos. A existência de valores desconhecidos nos dados pode dever-se, por exemplo, a que um dado valor não seja importante para um determinado exemplo, ou a que não tenha sido registado quando os dados foram reunidos, ou ainda a ter sido mal introduzido pela pessoa encarregue da introdução dos dados.

Os valores desconhecidos nos dados levantam três questões fundamentais:

- Como avaliar diferentes divisões do conjunto de exemplos com número diferente de valores desconhecidos nos subconjuntos criados pelas divisões;
- Ao realizar a divisão de um conjunto de exemplos, como tratar os exemplos com valores desconhecidos no atributo escolhido para a divisão;
- Como classificar um novo exemplo quando este tem um valor desconhecido num atributo considerado num dos testes.

O critério de razão de ganho utilizado pelo C4.5 para a avaliação das possíveis divisões de um conjunto de exemplos é baseado na informação necessária para determinar a classe de um determinado exemplo. É óbvio que um teste não pode determinar a classe de um exemplo se o valor do atributo ao qual se refere não for conhecido. O ganho de informação obtido pela divisão de um conjunto de exemplos (equação 2.2) pode ser facilmente modificado para considerar apenas os exemplos onde o valor do atributo é conhecido:

$$\begin{aligned} Ganho(A) &= probabilidade\ de\ A\ ser\ conhecido\ *\ (info(T) - \textstyle\sum_{i=1}^{m} \frac{|T_i|}{|T|} * info(T_i)) \\ &+ probabilidade\ de\ A\ ser\ desconhecido\ *\ 0 \end{aligned}$$

$$Ganho(A) = F * ((info(T) - \sum_{i=1}^{m} \frac{|T_i|}{|T|} * info(T_i))\ bits \tag{2.5}$$

onde F é a fracção de exemplos com valores conhecidos para o atributo sobre o número total de exemplos. Para os cálculos de $info(T)$ (equação 2.1) apenas são considerados os exemplos com valores conhecidos no atributo.

De forma semelhante, a definição do potencial de informação gerado pela divisão (equação 2.3) pode ser alterada para considerar os exemplos com valores desconhecidos como se pertencessem a um novo subconjunto de exemplos. Se o teste considerado dividir o conjunto de exemplos em m subconjuntos o $Split\ Info$ é calculado como se a aplicação do teste desse origem a $n+1$ subconjuntos.

$$Split\ Info(A) = -\sum_{i=1}^{m} \frac{|T_i|}{|T|} * log_2(\frac{|T_i|}{|T|}) - \frac{|T_u|}{|T|} * log_2(\frac{|T_u|}{|T|}) \tag{2.6}$$

Onde T_u é o subconjunto dos exemplos com valores desconhecidos para o atributo A.

Supondo que, depois de escolhida a melhor divisão, o conjunto de exemplos associado a um nó é dividido em $T_1, T_2, ..., T_n$ subconjuntos. Então, ao ser atribuído um determinado exemplo a um subconjunto T_i pode dizer-se que esse exemplo pertence ao subconjunto T_i com probabilidade 1 e aos restantes subconjuntos com probabilidade 0.

Para tratar os exemplos com valores desconhecidos o C4.5 associa a cada exemplo um peso que representa a probabilidade de esse exemplo pertencer ao conjunto onde se encontra. No início do processo de construção da árvore de decisão é atribuído a todos os exemplos o peso 1. Se, durante a divisão do conjunto de exemplos associados a um nó, um determinado exemplo apresenta, para o atributo utilizado na divisão, um valor conhecido, o seu peso no subconjunto correspondente mantém-se inalterado. Caso contrário, o exemplo é associado a cada um dos subconjuntos criados com a divisão e o seu peso em cada um deles é multiplicado pela probabilidade de pertencer a esse subconjunto.

Ao serem atribuídos pesos aos exemplos os cálculos efectuados para a avaliação das divisões têm de considerar a soma dos pesos e não o número de exemplos nos conjuntos.

Também, os pesos associados aos exemplos de um determinado conjunto podem já ser o resultado de uma divisão anterior. O peso de um exemplo com um valor desconhecido no atributo considerado para a divisão, quando atribuído a cada um dos subconjuntos resultantes dessa mesma divisão, deverá ser calculado da seguinte forma:

$w' = w * prob(T_i)$

A probabilidade $prob(T_i)$ é estimada como a soma dos pesos dos exemplos com valores conhecidos que pertencem a T_i dividida pela soma dos pesos de todos os exemplos com valores conhecidos em T.

Uma abordagem semelhante é seguida quando a árvore de decisão é utilizada para classificar um novo exemplo. Se, ao ser percorrida a árvore, é encontrado um nó de decisão para o qual o valor do exemplo é desconhecido no atributo referido pelo teste, o C4.5 explora todos os resultados possíveis e combina a classificação obtida aritmeticamente. Como podem existir vários caminhos desde a raiz da árvore ou de uma sub-árvore até às folhas, a "classificação" tem como resultado uma distribuição de classes e não uma única classe. Quando a distribuição de classes para o novo exemplo é completamente determinada, isto é, depois de explorados todos os caminhos possíveis, a classe com a maior probabilidade é escolhida como a classe prevista para o exemplo.

Para ilustração seguinte da forma como são tratados pelo C4.5 os valores desconhecidos nos atributos é utilizado o mesmo conjunto de exemplos considerado para a ilustração das avaliações das divisões baseadas em atributos discretos e contínuos e representado novamente na tabela 2.2. Como representado na tabela 2.2 o valor do atributo *Céu* para o exemplo:

Céu = Enevoado, Temperatura = 72, Humidade = 90, Vento = Sim

é desconhecido (representado por ?).

Considerando os 13 exemplos restantes com valores conhecidos, são obtidas as seguintes distribuições dos exemplos para cada valor possível do atributo *Céu*:

Céu	Jogar	Não Jogar	Total
Limpo	2	3	5
Enevoado	3	0	3
Chuva	3	2	5
Total	8	5	13

Utilizando as frequências relativas anteriores para o calcular a avaliação da divisão pelo atributo obtém-se:

$$\begin{array}{lclclcl}
Info(T) & = & -8/13 * log_2(8/13) - 5/13 * log_2(5/13) & = & 0.961\ bits & & \\
Info(T_1) & = & -2/5 * log_2(2/5) - 3/5 * log_2(3/5) & = & 0.529 + 0.442 & = & 0.971\ bits \\
Info(T_2) & = & -3/3 * log_2(3/3) - 0/3 * log_2(0/3) & = & 0\ bits & & \\
Info(T_3) & = & -3/5 * log_2(3/5) - 2/5 * log_2(2/5) & = & 0.442 + 0.529 & = & 0.971\ bits
\end{array}$$

$Ganho(Ceu) = 13/14 * (0.961 - 5/13 * 0.971 - 4/13 * 0 - 5/13 * 0.971) = 0.199$

Céu	Temperatura	Humidade	Vento	Classe
Limpo	75	70	Sim	Jogar
Limpo	69	70	Não	Jogar
Limpo	85	85	Não	Não Jogar
Limpo	80	90	Sim	Não Jogar
Limpo	72	95	Não	Não Jogar
?	**72**	**90**	**Sim**	**Jogar**
Enevoado	83	78	Não	Jogar
Enevoado	64	65	Sim	Jogar
Enevoado	81	75	Não	Jogar
Chuva	71	80	Sim	Não Jogar
Chuva	65	70	Sim	Não Jogar
Chuva	75	80	Não	Jogar
Chuva	68	80	Não	Jogar
Chuva	70	96	Não	Jogar

Tabela 2.2: Conjunto de exemplos com valores desconhecidos.

O ganho de informação obtido é ligeiramente inferior a 0.245 bits obtido anteriormente. Contudo, o potencial de informação esperado pela divisão ainda é obtido considerando a totalidade dos exemplos do conjunto e é superior visto que é considerado um novo subconjunto para os valores desconhecidos:

$$\begin{aligned} Split\,Info(Ceu) &= -5/14 * log_2(5/14) \\ &\quad -3/14 * log_2(3/14) \\ &\quad -5/14 * log_2(5/14) \\ &\quad -1/14 * log_2(1/14) \\ &= 0.530 + 0.516 + 0.530 + 0.272 \quad = \quad 1.849 \end{aligned}$$

$Razao\ de\ Ganho(Ceu) = 0.199/1.849 = 0.108$

O *Split Info* obtido é igual a 1.849 bits, superior a 1.577 bits obtido anteriormente. Logo o valor da razão de ganho para a divisão desce de 0.156 para 0.108.

Se o conjunto de exemplos for dividido pelo atributo *Céu*, os 13 exemplos para os quais o valor do atributo é conhecido não apresentam qualquer dificuldade. O exemplo restante com o valor do atributo desconhecido é atribuído a todos os 3 subconjuntos resultantes da divisão com o peso igual a 5/13, 3/13 e 5/13 respectivamente.

A árvore de decisão obtida a partir do conjunto de exemplos utilizado neste exemplo (tabela 2.2), apesar de conter um exemplo com um valor desconhecido, mantém o mesmo aspecto da árvore já apresentada para este conjunto e novamente representada na figura 2.10.

Os números em cada folha na forma (N) ou (N/E) representam o número de exemplos N que são cobertos pela folha e o número de exemplos E que não pertencem à classe representada.

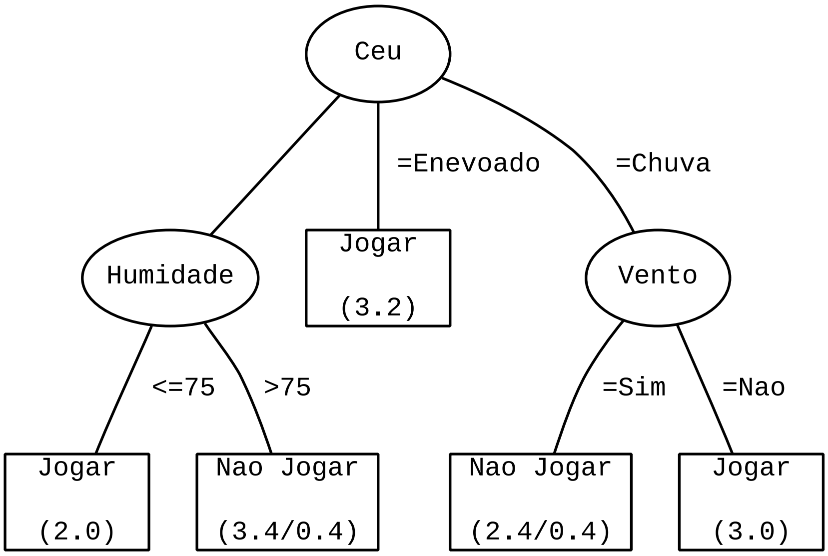


Figura 2.10: Árvore obtida a partir do conjunto de exemplos com valores desconhecidos da tabela 2.2.

Considere-se agora que esta árvore é utilizada para classificar um novo exemplo descrito por:

Céu = Limpo, Temperatura = 70, Humidade = ? e Vento = Não.

O valor *limpo* para o atributo *Céu* garante que este exemplo é classificado pela sub-árvore esquerda. Mas, neste ponto não é possível determinar qual dos ramos deverá ser escolhido para classificar o exemplo pois não é conhecido o valor do atributo *Humidade*.

Quando a árvore foi construída, cada uma das sub-árvores cobria 2.0 e 3.4 exemplos respectivamente. O novo exemplo é classificado considerando os pesos relativos, 2.0/5.4 e 3.4/5.4 para as possíveis classificações efectuadas por cada um dos ramos. A distribuição de classes final para o novo exemplo é dada por:

$$\begin{array}{lll} Jogar: & 2.0/5.4 * 1 + 3.4/5.4 * 0.4/3.4 & = & 0.44 \\ Nao\ Jogar: & 3.4/5.4 * 3/3.4 & = & 0.56 \end{array}$$

2.4.3 Análise da complexidade do algoritmo

A fase de crescimento da árvore de decisão é, sem qualquer dúvida, a fase mais "pesada" computacionalmente do sistema C4.5. De facto, a simplificação consiste simplesmente em atravessar a árvore de decisão verificando para cada nó uma condição de simplificação para os exemplos cobertos por esse nó. À excepção dos conjuntos de treino muito pequenos, o tempo gasto na simplificação é uma pequena fracção do tempo total de execução [Qui92, Rug00]. Logo a fase onde se deve centrar a atenção é a fase de crescimento da árvore de decisão. A seguir é apresentada uma análise do custo de construção de um único nó. Para tal é necessário introduzir algumas notações utilizadas:

- T o conjunto de exemplos associados ao nó;
- $|T|$ a cardinalidade de T, e analogamente para qualquer conjunto;
- n_c o número de atributos contínuos;
- n_d o número de atributos discretos ainda não seleccionados num nó antecessor;
- s o número de conjuntos formados pela divisão de T no nó;
- p a probabilidade de o atributo seleccionado ser contínuo (na raiz p é $n_c/(n_c+n_d)$ mas, como os atributos discretos não são seleccionados mais do que uma vez p cresce até 1 à medida que a árvore cresce, a não ser que não existam atributos contínuos).

A seguir é apresentado, de forma resumida, os passos realizados pelo algoritmo de construção de árvores de decisão na construção de um nó com um conjunto de exemplos T associados:

```
Constrói Árvore(T):
1. Determina a distribuição dos exemplos pelas classes em T.
2. Se todos os exemplos em T pertencem à mesma classe Ci:
   2.1. Retorna uma folha que representa a classe Ci.
3. Se os exemplos em T pertencem a mais do que uma classe:
   3.1. Para cada atributo Aj:
        3.1.1 Calcula a razão de ganho obtido pela divisão de T
              baseada no atributo Ak.
   3.2. Divide o conjunto de exemplos T baseado no melhor atributo
        Ak e constrói um nó de decisão N.
   3.3. Para cada subconjunto Tj criado pela divisão:
        3.3.1. Filho j de N = Constrói Árvore(Tj).
   3.4. Retorna o nó de decisão N.
```

Baseado no algoritmo anterior, a construção de um único nó da árvore de decisão requer em média os seguintes passos:

- $|T|$ para o cálculo das frequências (passo 1).
- $n_d.|T|$ para o cálculo do ganho de informação dos atributos discretos (passo 3.1.1).

- $n_c.|T|.(log|T|+1)$ para a ordenação dos exemplos em T e cálculo dos valores do ganho de informação e razão de ganho para cada divisão possível de um atributo contínuo (o sistema C4.5 utiliza o algoritmo de ordenação QuickSort) (passo 3.1.1).
- $p*|TS|/2$ para encontrar o ponto de divisão em todo o conjunto de treino quando é seleccionado um atributo contínuo (3.2).
- $(s+1)*|T|$ para a divisão dos exemplos em subconjuntos associados aos nós filhos (3.3.1).

O que dá origem ao número seguinte de passos por exemplo:

$$n_c * (log_2|T|+1) + p * |TS|/(2*|T|) + (s+2+n_d) \tag{2.7}$$

Um dos principais factores que prejudicam o desempenho do algoritmo de construção da árvore de decisão do C4.5 é evidenciado pela equação 2.7, onde custo de execução do segundo termo, nomeadamente $p*|TS|/(2*|T|)$, pode ser bastante elevado, principalmente para os nós que cobrem um número reduzido de exemplos. Este termo está associado ao custo de execução de uma pesquisa linear para encontrar o ponto de divisão em todo o conjunto de treino quando um atributo contínuo é seleccionado para a divisão (passo 3.2). É realizada uma pesquisa linear, pois o conjunto de exemplos pode não estar ordenado pelos valores do atributo. Um outro factor que prejudica o desempenho do algoritmo são as diversas ordenações realizadas sempre que um atributo contínuo é avaliado (primeiro termo na equação 2.7).

Em [Rug00] é apresentado um algoritmo de construção de árvores de decisão, denominado *Efficient C4.5 (EC4.5)*, baseado no algoritmo do C4.5 onde estes problemas são ultrapassados. No EC4.5 a pesquisa linear, realizada no C4.5 para determinar o ponto de divisão, é substituída por uma pesquisa binária. Também, neste algoritmo, o algoritmo de ordenação utilizado no C4.5 (QuickSort) é substituído por um outro algoritmo de ordenação mais adequado para a natureza "dividir para conquistar" do algoritmo de construção de árvores de decisão, reduzindo, desta forma, os custos associados ao primeiro termo na equação 2.7.

Também, em [SAM96], são apresentados dois algoritmos de construção de árvores de decisão, o SLIQ e o SPRINT, que evitam as sucessivas ordenações dos exemplos durante a avaliação dos atributos contínuos. Contudo, a abordagem seguida por estes algoritmos é bastante diferente do algoritmo anterior. Nestes algoritmos os atributos são guardados em listas separadas de valores que, para os atributos contínuos, são ordenadas apenas uma vez no início e a ordem dos valores é mantida durante a construção da árvore.

No algoritmo desenvolvido neste trabalho as diversas ordenações dos exemplos também são evitadas em que é seguida uma abordagem semelhante à apresentada em [SAM96]. Também são utilizadas listas separadas de valores e os valores dos atributos contínuos são ordenados apenas uma única vez antes do início da construção da árvore de decisão. Também, para determinar o ponto de divisão quando um atributo contínuo é escolhido para a divisão dos exemplos num nó, é realizada uma pesquisa binária aos valores do atributo dado que estes se encontram ordenados.

Capítulo 3

Computação paralela

A procura de mais poder tem sido um dos factores principais para o desenvolvimento dos computadores. Antes do nascer da era computacional, os cientistas e engenheiros pensavam que a capacidade de realizar algumas centenas de cálculos aritméticos por segundo seria quase impossível. Contudo, logo que estes obtiveram este poder começaram a exigir mais. Queriam agora conseguir realizar milhares de operações por segundo. A seguir aos milhares, pediram milhões (10^6), a seguir um milhar de milhões (10^9) até que o objectivo actual é de um bilião (10^{12}) operações por segundo (apesar de muitos já se encontrarem a discutir 10^{15} operações por segundo). Mas porquê esta necessidade crescente de mais poder computacional? A ciência, a engenharia, a economia, e até o entretenimento, todos eles exigem mais. Durante séculos, a Ciência baseou-se no paradigma básico de primeiro observar, depois teorizar e por fim testar a teoria através da experimentação. De forma semelhante, na Engenharia tradicionalmente desenhava-se (tipicamente em papel), depois construíam-se e testavam-se protótipos e por fim construía-se o produto final. Contudo, está a tornar-se menos dispendioso realizar simulações detalhadas em computador em vez de realizar numerosas experiências ou construir uma série de protótipos. Logo, os paradigmas de observar e experimentar nas Ciências, e desenhar e construir protótipos nas Engenharias estão cada vez mais a ser substituídos pela computação. Ainda, em alguns casos, é possível simular por computador fenómenos que eram impossíveis de ser estudados através da experimentação, como por exemplo a evolução do Universo. E à medida que se vão resolvendo os problemas outros mais complexos vão emergindo e exigindo um poder computacional de várias ordens de grandeza superior ao necessário para resolver os problema iniciais. Quanto maior e o conhecimento, maior e também a sede de se conhecer mais e mais complexas se vão tornando as questões para resolver.

Mas 10^{12} operações por segundo? Não será excessivo? Não é difícil encontrar problemas que requerem bastante mais poder computacional que o disponível actualmente. Por exemplo, simulações bio-moleculares ao nível atómico ou simulações que permitiriam acelerar o desenho e fabrico de circuitos integrados. De facto, o Governo Americano respondeu a este desafio e investiu milhões de dollars no financiamento da investigação e desenvolvimento de algoritmos e equipamento computacional que possa ser utilizado para resolver problemas nas Ciências e Engenharias esperando com isso proporcionar contribuições fundamentais para os conhecimentos teóricos e práticos actuais [Pac97].

Apesar da discussão ter sido dirigida para o problema da velocidade de cálculo, um outro problema fundamental no desenvolvimento de maior poder computacional esta relacionada com o desenvolvimento de uma maior capacidade de armazenamento (memória). Mesmo que seja possível construir um computador capaz de realizar 10^{12} operações por segundo, se este apenas dispuser de alguns milhões de palavras de memória, provavelmente irá ser de pouca utilidade. Logo, se o desenvolvimento de mais poder computacional supõe o desenvolvimento de simultâneo mais velocidade e mais capacidade de armazenamento [Pac97].

Suponhamos que se pretendia construir um computador capaz de realizar 10^{12} operações por segundo. Por exemplo, que era desejado um computador capaz de executar num segundo o código seguinte:

```
0.    x, y e z são vectores de números reais,
      cada um com 10^12 elementos
1.    Para i de 1 a 10^12
1.1.    z[i] = x[i] + y[i]
```

Possivelmente, a abordagem mais óbvia é a de procurar estender as tecnologias já conhecidas. Isto é, tentar construir mais ou menos um computador convencional baseado no modelo de von Neumann com um processador extremamente rápido e uma quantidade de memória suficientemente grande.

Mas, num computador convencional, iriam ser carregados a partir da memória os valores de $x[i]$ e $y[i]$, adicionados e o resultado guardado novamente na memória em z[i]. Logo, para executar este código seria necessário realizar pelo menos $3*10^{12}$ cópias por segundo entre a memória e os registos do processador. Se os dados viajarem da memória para o processador à velocidade da Luz ($3*10^8$ metros por segundo) e se for r a distância média entre um memória e a CPU, então r tem de satisfazer a seguinte igualdade:

$3*10^{12}*r$ metros $= 3*10^{18}$ metros por segundo $*1$ segundo,

Ou seja, r terá que ser igual a 10^{-14} metros. Ora o computador tem de conseguir conter pelo menos $3*10^{12}$ palavras de memória para conseguir guardar os valores de x, y e z. Se fosse utilizada para memória, como é típico num computador convencional, uma grelha quadrada com o comprimento de lado s e se a CPU estiver ligada ao centro do quadrado, então a distância média da memória à CPU seria de aproximadamente $s/2$. Logo $s/2 = r = 10^{-14}$ metros, ou $s = 2*10^{-14}$ metros. Como as palavras de memória formariam uma grelha de lado quadrado, uma linha típica de palavras de memória iria conter $\sqrt{3*10^{12}} = \sqrt{3}*10^6$ palavras. Logo, seria necessário conseguir encaixar uma palavra de memória num quadrado com comprimento de lado igual a $2*10^{-14}/\sqrt{3}*10^{6} = 10^{-10}$ metros, valor este que é aproximadamente igual ao diâmetro de um átomo relativamente pequeno. Por outras palavras, a não ser que se consiga descobrir uma forma de representar uma palavra de 32 bit (ou, mais provavelmente, de 64 bit) através de um único átomo, é impossível construir o computador pretendido do modo convencional [Pac97]. Então como será possível construir um computador capaz de realizar 10^{12} operações por segundo?

Recorramos a uma alegoria. Petrus é um empreiteiro romano especializado na escavação. O seu único trabalhador, Paulus, é capaz de escavar 1000 metros cúbicos num dia. Contudo, surgiu uma enorme quantidade de pedidos quando foi noticiado que Attila, o Huno, iria visitar Roma na semana seguinte. Petrus rapidamente se apercebeu que, para conseguir satisfazer a procura, necessitava de escavar aproximadamente 100.000 metros cúbicos por dia no espaço de uma semana. Ele, que não é nenhum tolo, resolveu o seu problema simplesmente contratando mais 99 homens, aumentando, desta forma, a sua força de trabalho de 1 para 100^4.

A analogia é clara: o único trabalhador representa o único processador e a memória de um computador actual, e o pedido de 100.000 metros cúbicos por dia representa o desafio colocado. A solução também é clara. Devem ser *contratados* mais trabalhadores, isto é, mais processadores e módulos de memória para a resolução do problema.

Um computador paralelo é simplesmente um computador (ou um conjunto de computadores) equipado com vários processadores capazes de trabalhar em conjunto para a resolução de um determinado problema [Pac97]. Entramos então no mundo da computação paralela.

3.1 Sistemas paralelos

A classificação original de computadores paralelos é normalmente conhecida como a taxonomia de Flyn. Em 1966 Michael Flyn classificou os computadores paralelos de acordo com o número de fluxos de instruções e com o número de fluxos de dados. A máquina clássica de von Neumann (um computador clássico com um único processador) é caracterizada por um único fluxo de instruções e um único fluxo de dados, logo é identificada, de acordo com a taxonomia de Flyn, como uma máquina *Single Instruction Single Data* (SISD). No extremo oposto encontram-se os sistemas *Multiple Instruction Multiple Data* (MIMD), onde um conjunto de processadores autónomos operam nos seus próprios fluxos de dados. De acordo com a taxonomia de Flyn esta é a arquitectura mais geral. Entre os sistemas SISD e MIMD encontram-se os sistemas *Single Instruction Multiple Data* (SIMD) e os sistemas *Multiple Instructions Single Data* (MISD). A seguir são discutidas resumidamente cada uma destas arquitecturas [Pac97, LER92].

3.1.1 A máquina clássica de von Neumann

A máquina clássica de von Neumann é formada por uma Unidade Central de Processamento (CPU - *Central Processing Unit*) e uma memória principal. A CPU é ainda formada por uma Unidade de Controlo e por uma Unidade aritmética e Lógica (ALU - *Arithmetic-Logic Unit*). Na máquina de von Neumann a memória guarda ambas as instruções e os dados, a unidade de controlo gere a execução dos programas, e a ALU realiza os cálculos pedidos pelo programa. Quando estão a ser utilizados pelo programa, as instruções e os dados são guardados em localizações de memória muito rápidas conhecidas como registos. É claro que esta memória rápida é bastante mais cara, logo existem relativamente bastante menos registos. Tanto os dados como as instruções dos programas são deslocados entre a memória e os registos na CPU. O caminho por onde se deslocam é denominado por *barramento* (*BUS*). O barramento consiste basicamente num conjunto de ligações paralelas juntamente com algum hardware de controlo de acesso. Para se tornar útil, a máquina clássica de von Neumann necessita ainda de alguns dispositivos adicionais, como dispositivos de entrada e saída, e dispositivos de armazenamento.

A velocidade de execução dos programas na máquina clássica de von Neumann encontra-se limitada pela velocidade à qual os dados e as instruções, inerentemente sequenciais, são transferidos entre a memória principal e a CPU Esta limitação é denominada por *Bottleneck de von Neumann*. Actualmente muito poucos computadores podem ser identificados como máquinas de clássicas von Neumann. A grande maioria dispõe de uma memória hierárquica, onde para além da memória principal e dos registos, existe uma memória intermédia mais rápida que a principal mas mais lenta que os registos, denominada de *Cache*. A cache surgiu da observação dos programas tenderem a aceder à memória de uma forma sequencial. Logo, se for armazenado um pequeno bloco de dados e um pequeno bloco de instruções na cache, grande parte dos acessos vai ser feito à cache e portanto serão mais rápidos [Pac97, LER92].

3.1.2 Arquitecturas pipeline e vectoriais

A primeira extensão ao modelo clássico de von Neumann foi o *pipelinning*. Se os vários circuitos que formam a CPU forem divididos em áreas funcionais, e se estas forem dispostas numa sequência, isto é, num pipeline, então o pipeline pode, em teoria, produzir um resultado a cada ciclo de instrução.

Para ilustrar o princípio de funcionamento de um pipeline, considere-se o programa seguinte em que é calculada a soma de dois vectores x e y e o resultado guardado em z.

```
0.   x, y e z são vectores de números reais
     cada um com 100 elementos
1.   Para i de 1 e 100
```

```
1.1.   z[i] = x[i] + y[i]
```

Supondo ainda que a única adição do programa consiste na seguinte sequência de operações:

1. Ler os operandos da memória;
2. Comparar os expoentes;
3. Realiza o Shift de um dos operandos;
4. Adiciona os operandos;
5. Normaliza o resultado;
6. Guarda o resultado na memória;

Se estiverem disponíveis unidades funcionais suficientes para realizar cada uma destas operações básicas e se estas se encontrarem dispostas num pipeline, isto é, a saída de uma unidade funcional é a entrada da unidade seguinte, então, enquanto $x[0]$ e $y[0]$ estão a ser adicionados (passo 4), a unidade anterior pode estar a realizar o Shift de $x[1]$ ou de $y[1]$. Também os expoentes de $x[2]$ e de $y[2]$ podem estar a ser comparados e $x[3]$ e $y[3]$ podem estar a ser carregados da memória. Logo, assim que o pipeline fica totalmente preenchido, é possível produzir seis vezes mais resultados do que com um único processador.

Uma melhoria adicional aos processadores pipeline pode ser obtida pela adição de instruções vectoriais ao conjunto de instruções básicas da máquina. No exemplo anterior em que são adicionados 100 pares de números reais, se não existirem instruções vectoriais, uma instrução correspondente a cada uma das operações básicas tem de ser carregada e interpretada 100 vezes. Com a utilização de instruções vectoriais, cada uma das instruções básicas apenas necessita de ser invocada uma única vez. A diferença é de alguma forma análoga à diferença entre os dois programas seguintes:

```
Para i de 1 a 100
  z[i] = x[i] + y[i]

z[1:100] = x[1:100] + y[1:100]
```

Outra forma possível de melhorar as máquinas vectoriais passa pela utilização de múltiplos bancos de memória. As operações de acesso à memória principal são várias vezes mais lentas que as operações que apenas envolvem a CPU (por exemplo, adicionar). A utilização de bancos de memória independentes podem, até certo ponto, ultrapassar este problema.

Por exemplo, supondo que é possível executar uma operação de CPU uma vez em cada ciclo de relógio, mas só é possível um acesso à memória em cada quatro ciclos. Então, se existirem quatro bancos de memória, e os dados se encontrarem adequadamente distribuídos ao longo dos bancos, é possível realizar um acesso à memória em cada ciclo.

A grande virtude dos processadores vectoriais é de serem bem percebidos e para os quais existem disponíveis compiladores bastante eficientes. Logo, é relativamente fácil desenhar programas com desempenhos bastante elevados. Desta forma, estes sistemas continuam a ser bastante populares para a computação científica de alto desempenho [Pac97, LER92].

Contudo, existem diversas contrapartidas. Os princípios do pipelinning e das instruções vectoriais não são adequados para programas que utilizam estruturas de dados irregulares ou que são compostos por muitos ramos. A chave do desempenho destas máquinas está em preencher totalmente o pipeline e mante-lo cheio. Se os operandos não se encontrarem convenientemente dispostos na memória tal não é possível. Além disso, se um programa contém bastantes ramos condicionais, irão existir poucas oportunidades para a utilização de instruções vectoriais. Mas o pior é de estes sistemas parecerem escalonar mal. Isto é, não é clara a forma de os modificar para que possam resolver problemas de maior dimensão. Mesmo que sejam adicionadas várias unidades funcionais ao pipeline e que seja possível mante-las cheias, o limite superior da sua velocidade irá ser um pequeno múltiplo da velocidade da CPU [Pac97, LER92].

Alguns autores classificam os processadores vectoriais como máquinas MISD, outros afirmam que não existem máquinas MISD, e que estas são apenas uma variante das máquinas SIMD. Ainda outros afirmam que não existem de todo máquinas realmente paralelas.

3.1.3 Sistemas SIMD

Um sistema SIMD puro (em oposição a um processador vectorial) é formado por uma CPU dedicada exclusivamente ao controlo, e um grande conjunto de ALUs subordinadas, cada uma com a sua memória. Durante cada ciclo de instrução o processador de controlo envia uma instrução a todos os processadores subordinados. Cada processador subordinado ou executa a instrução ou permanece em estado de espera.

Supondo três vectores x, y e z com 100 elementos cada e distribuídos de forma a que cada memória de cada processador subordinado contenha um elemento de cada vector. Pretende-se, então, executar a seguinte sequência de instruções (em série):

```
Para i de 1 a 100
  Se y[i] != 0
    z[i] = x[i] / y[i]
```

```
Se y[i] == 0
  z[i] = x[i]
```

Para o conjunto de instruções anteriores, cada processador subordinado vai executar algo semelhante à sequência seguinte de operações:

1. Testar y != 0.0
2. (a) Se $y! = 0$:
   $z[i] = x[i]/y[i]$
   (b) Se $y == 0$:
   não faz nada
3. (a) Se $y! = 0$:
   não faz nada
   (b) Se $y == 0$:
   $z[i] = x[i]$

É importante realçar que isto implica uma execução completamente síncrona de expressões. Por outras palavras, em qualquer instante de tempo, um determinado processador subordinado ou está "activo" e a executar exactamente as mesmas operações que todos os outros processadores ou está em estado de espera.

O exemplo torna evidentes as desvantagens dos sistemas SIMD. Num programa com muitos ramos condicionais ou com longas expressões dependentes de condições, é possível que vários processadores permaneçam em estado de espera por longos períodos de tempo. Contudo, o exemplo não torna claro que as máquinas SIMD são relativamente simples de programar se o problema for definido por uma estrutura regular. Por fim, estes sistemas escalonam bastante bem [Pac97, LER92].

3.1.4 Sistemas MIMD

A diferença chave entre os sistemas MIMD e os sistemas SIMD consiste na autonomia dos processadores. Num sistema MIMD cada processador consiste numa CPU completa com uma unidade de controlo e uma ALU o que permite a cada processador executar o seu próprio programa ao seu próprio ritmo. Em particular, e ao contrário dos sistemas SIMD, os sistemas MIMD são assíncronos. Normalmente não existe um relógio global, e, a não ser que os processadores estejam especificamente programados para sincronizar, pode não existir uma correspondência entre o que está a ser executado nos diferentes processadores, mesmo que estes se encontrem a executar o mesmo programa.

O universo dos sistemas MIMD encontra-se dividido entre os sistemas de memória partilhada e os sistemas de memória distribuída. Alguns autores distinguem entre as

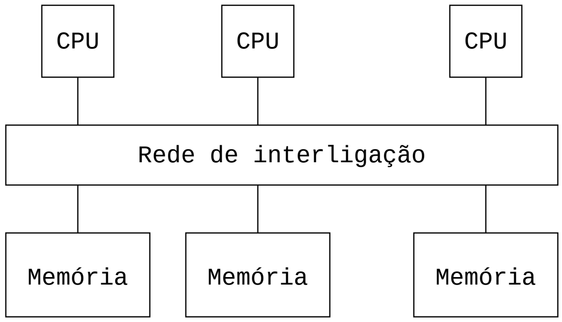


Figura 3.1: Arquitectura genérica de um sistema de memória partilhada.

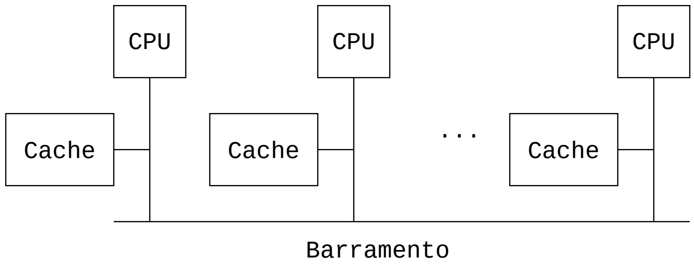


Figura 3.2: Sistema de memória partilhada baseado num barramento.

duas arquitecturas denominando os sistemas de memória partilhada por *multiprocessadores* e os sistemas de memória distribuída por *multicomputadores*. Contudo esta terminologia não é aceite por todos, e é bastante frequente o termo *multiprocessador* ser utilizado como um sinónimo de *processador paralelo*.

3.1.4.1 Sistemas de memória partilhada

Como o próprio nome indica, o sistema MIMD genérico de memória partilhada é formado por um conjunto de processadores e módulos de memória interligados por uma rede (figura 3.1) [Pac97, LER92].

Sistemas baseados em barramentos: A rede de ligação mais simples é baseada num barramento. Contudo, se vários processadores tentam simultaneamente aceder à memória, rapidamente o barramento fica saturado, e podem existir grandes atrasos entre o início da operação de acesso à memória até à cópia real dos dados. Logo cada processador normalmente dispõe de uma *cache* razoavelmente grande (figura 3.2). Devido à largura de banda limitada do barramento, estas arquitecturas não são desenhadas para serem utilizadas com um grande número de processadores.

Sistemas baseados em interruptores: Na maioria das arquitecturas de memória partilhada, onde não se utilizam barramentos, a rede de interligação é implementada através de uma rede de interruptores. Um exemplo de uma rede de interruptores é

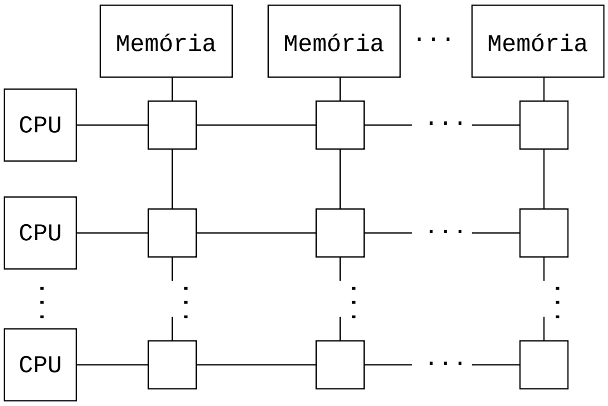


Figura 3.3: Cronbar Switch.

uma *Cronbar Switch*. Uma Cronbar pode ser ilustrada por uma trama rectangular de ligações com interruptores nos pontos de intersecção, e terminais no topo e num dos seus lados. Os processadores e os módulos de memória encontram-se ligados nos terminais da trama (figura 3.3). Os interruptores permitem a passagem simultânea de sinais nas direcções vertical e horizontal, ou podem redireccionar um sinal da direcção vertical para a horizontal, ou vice versa. Logo, se os processadores estiverem colocados no lado esquerdo da Cronbar e as memórias no topo, então qualquer processador pode aceder a qualquer módulo de memória. Ainda, qualquer outro processador pode aceder simultaneamente a qualquer outro módulo de memória. Isto é, a comunicação entre duas unidades não interfere com a comunicação entre qualquer outras duas unidades. Logo, as Cronbar não sofrem do problema de saturação encontrado nos barramentos. Infelizmente, são caras. Por exemplo, na Convex SPP 1200, para se dispor de mais do que 8 processadores numa única máquina, duas ou mais Cronbars são ligadas em anel. Este tipo de ligação implica que, quando um processador acede a uma memória ligada a uma outra Cronbar, o tempo de acesso é maior. Naturalmente esta situação é indesejada, mas é um compromisso que foi adoptado por quase todos os construtores de máquinas de memória partilhada. Tempos de acesso não uniformes são uma regra, não uma excepção. Tais sistemas são denominados de *nonuniform memory access* ou sistemas NUMA.

3.1.4.2 Sistemas de memória distribuída

Nos sistemas de memória distribuída, cada processador dispõe da sua própria memória privada. Um sistema genérico de memória distribuída pode ser representado como na figura 3.4. Se um sistema de memória distribuída for representado como um grafo, onde as “arestas” representam ligações, então existem dois tipos genéricos de grafos. Aqueles em que cada vértice corresponde a um par processador / memória, ou *nó* (figura 3.5a), e aqueles em que alguns vértices correspondem a nós e outros correspondem a interruptores (figura 3.5b). As redes do primeiro tipo são denominadas de *redes estáticas* e as redes do segundo de *redes dinâmicas*.

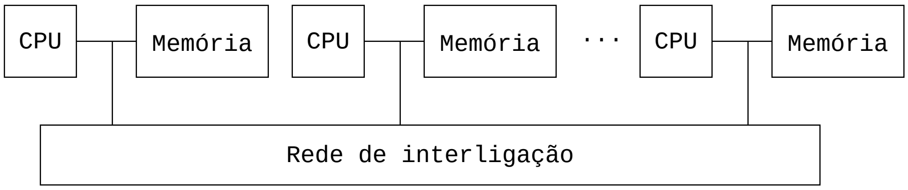


Figura 3.4: Arquitectura genérica de um sistema de memória distribuída.

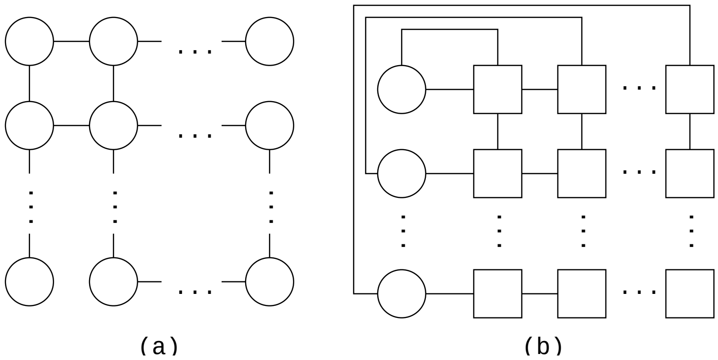


Figura 3.5: Diferentes tipos de redes de interligação: (a) rede estática; (b) rede dinâmica.

Do ponto de vista do desempenho e da facilidade de programação, a rede de interligação ideal é uma rede completamente ligada, em que cada nó se encontra directamente ligado a todos os outros nós (figura 3.6). Numa rede completamente ligada cada nó pode comunicar directamente com todos os outros nós. Ainda, as comunicações podem dar-se em simultâneo. Infelizmente o custo de uma rede deste tipo torna impraticável a construção de uma máquina com muitos nós [Pac97, LER92].

Redes de interligação dinâmicas: Talvez a forma prática mais próxima de se obter uma rede completamente ligada consiste numa *Crossbar switch* onde cada nó encontra-se ligado a um dos terminais da rede do lado esquerdo e a um outro no topo (figura 3.5(b)). Essencialmente o único atraso verificado nas comunicações deve-se à activação de um único interruptor, e, se o nó i está a comunicar com o nó j, então qualquer outro par de nós distinto pode comunicar entre si simultaneamente. Contudo, estas redes também apresentam custos elevados sendo invulgar encontrar Crossbars

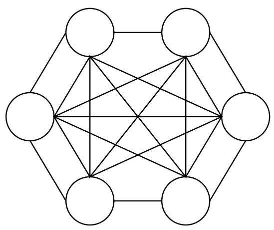


Figura 3.6: Rede de interligação completamente ligada.

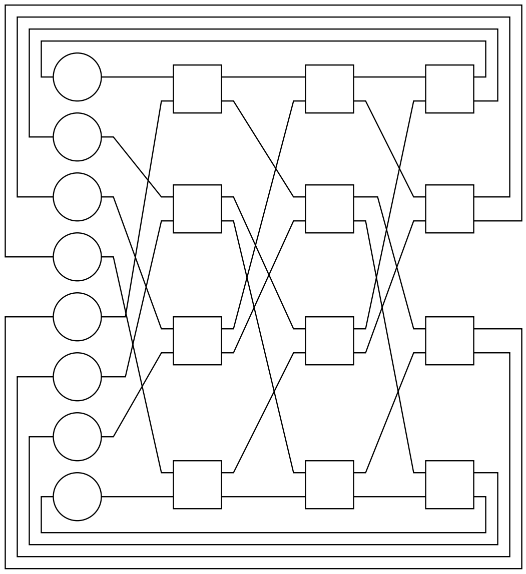


Figura 3.7: Configuração de uma rede Omega.

com mais de 16 processadores. Uma solução menos dispendiosa é a utilização de uma rede de interruptores de multicamadas (multistage switch network). Existem diversos tipos de redes de multicamadas diferentes. Um exemplo de uma rede, é a *Omega Network*, ilustrada na figura 3.7. Se existirem p nós, então numa Omega Network são utilizados $p * log_2(p)/2$ interruptores, o que torna seu custo bastante mais baixo que o das Crossbars onde são utilizados p^2 interruptores. Numa Omega Network, qualquer nó pode comunicar com outro nó. Contudo, existe uma probabilidade relativamente alta de a comunicação entre os nós interferir com a comunicação de outros dois. Também, o atraso na transmissão das mensagens entre os nós é superior, visto que $log_2(p)$ interruptores têm de ser activados.

Redes de interligação estáticas: No extremo oposto das redes completamente ligadas encontram-se os *vectores*, uma rede estática em que todos menos dois nós se encontram ligados a dois nós vizinhos adjacentes (figura 3.8(a)). Uma rede um pouco mais eficiente é um *anel* (figura 3.8(b)). Esta pode ser descrita como um vector onde os nós extremos encontram-se ligados entre si. A grande vantagem destas duas redes é serem pouco dispendiosas. Para além do custo dos nós, existe apenas o custo adicional de $p-1$ ou p ligações. Outra vantagem é a de escalonarem bastante bem. É muito fácil e pouco dispendioso aumentar o tamanho da rede para incluir um número arbitrário de nós. A principal desvantagem é de o número de ligações disponíveis ser extremamente limitado. Se dois nós estão a comunicar entre si, é muito provável que outros nós que pretendam comunicar não o consigam fazer. Ainda, num vector, dois nós que estejam a comunicar podem necessitar de fazer passar a mensagem entre um número de $p-1$ ligações, e, no caso de um anel, pode ser necessário fazer passar a mensagem até um número de $p/2$ ligações.

A rede de ligação estática prática que melhor se aproxima de uma rede completamente

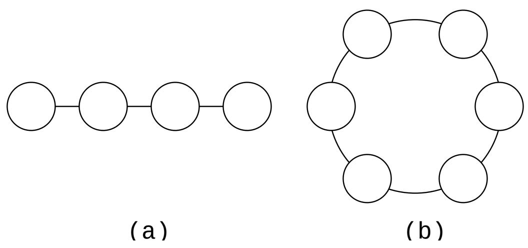


Figura 3.8: Redes de interligação estáticas em (a) vector e (b) anel.

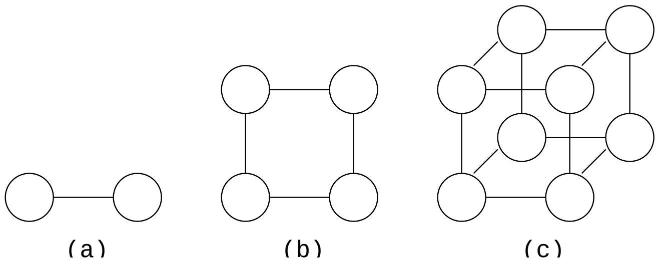


Figura 3.9: Hipercubos de (a) dimensão 1, (b) dimensão 2 e (c) dimensão3.

ligada é o *Hipercubo*. Os hipercubos são definidos de forma recursiva. Um hipercubo de dimensão 0 é formado por um único nó. Um hipercubo de dimensão $d > 0$ é formado a partir de dois hipercubos de dimensão $d-1$ em que os nós correspondentes são ligados entre si. Na figura 3.9 são ilustrados três hipercubos de dimensões 1, 2 e 3. Como, sempre que é aumentada a dimensão de um hipercubo o número de nós é duplicado, um hipercubo de dimensão d é formado por $p = 2^d$ nós. Também, ao aumentar a sua dimensão, é adicionada uma ligação a cada nó. Desta forma, um nó num hipercubo de dimensão d encontra-se directamente ligado a d nós. Comparativamente aos vectores ou às redes Omega, numa rede com esta configuração é mais simples encontrar um caminho para a comunicação entre dois nós sem interferências noutras comunicações. A principal desvantagem dos hispercubos é a sua relativa falta de escalonabilidade. Cada vez que é pretendido aumentar o tamanho da máquina é necessário duplicar o número de nós e adicionar uma nova ligação para cada nó.
No ponto intermédio entre os hipercubos e os vectores encontram-se as *meshes* e os *torus* que são respectivamente vectores e anéis de dimensão superior. Na figura 3.10 são ilustrados uma mesh de duas dimensões, uma mesh de três dimensões e um torus de três dimensões. Um torus de n dimensões pode ser obtido a partir de uma mesh de n dimensões pela ligação dos nós nos extremos. À medida que a dimensão aumenta torna-se menos provável que dois pares de nós a comunicar interfiram entre eles. Os torus e as meshes escalonam melhor que os hipercubos, embora não tão bem como os vectores ou os anéis.

Redes baseadas em barramentos: A última e provavelmente a mais simples rede de ligações consiste num barramento. Um conjunto de computadores ligados por uma rede local é um exemplo bastante comum. É claro que os barramentos

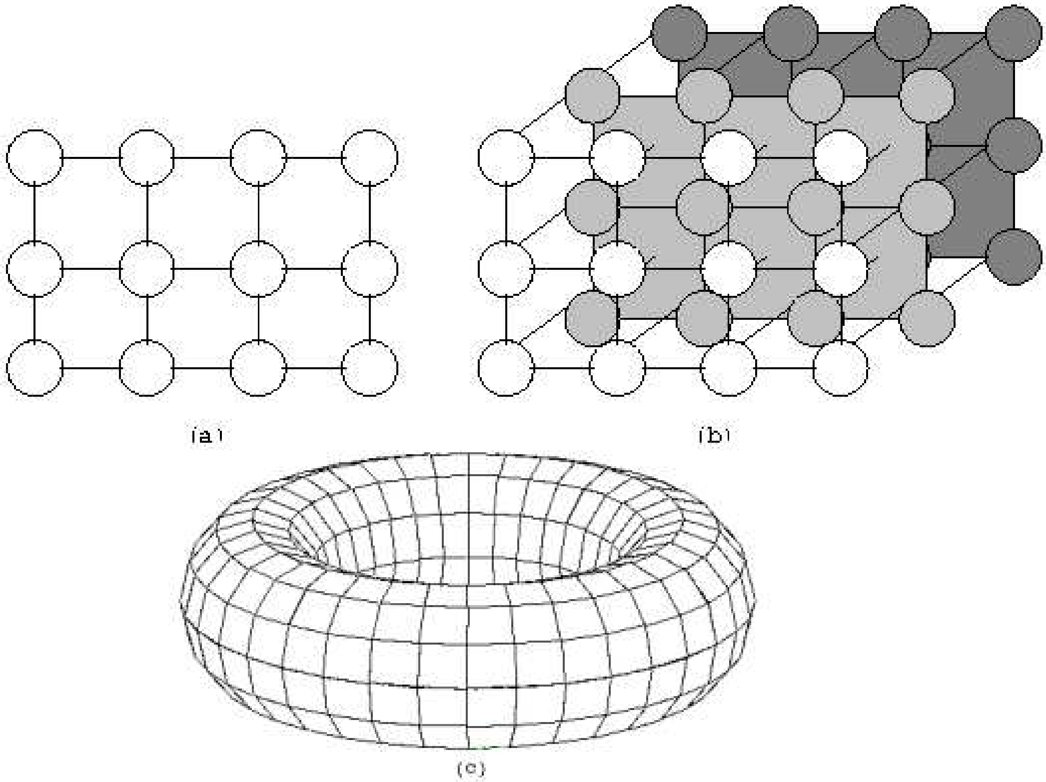


Figura 3.10: (a) Uma mesh de dimensão 2; (b) Uma mesh de dimensão 3; (c) Um torus de dimensão 2.

tendem a ser relativamente lentos. Pior ainda, os barramentos, especialmente as redes locais, ficam saturadas muito rapidamente no caso de "mais do que alguns nós" se encontrarem a comunicar. Embora as redes sejam bastante úteis para o desenvolvimento de aplicações, actualmente os sistemas baseados em barramentos não se apresentem muito prometedores para aplicações de larga escala.

3.2 Programação Paralela

Um processo é um bloco de construção fundamental na maioria dos paradigmas de programação paralela. Intuitivamente, um *processo* é uma instância de um programa ou parte de um programa que é executada de forma mais ou menos autónoma num processador (físico). Um programa é um programa paralelo se, em qualquer ponto da sua execução, contém mais do que um processo associado. Para que seja possível criar programas paralelos úteis devem existir formas de especificar, criar, destruir processos e de coordenar a sua interacção. Nesta secção, é feita uma breve descrição de como estes diferentes tópicos são abordados pelos diferentes paradigmas de programação paralela.

3.2.1 Memória partilhada

Apesar de convencionalmente se pensar num sistema de memória partilhada como um sistema em que os processos têm, mais ou menos, acesso igual a todas as posições de memória, é perfeitamente viável simular memória partilhada com hardware de memória distribuída, desde que existam mecanismos para criar um espaço de endereços global. Logo, é perfeitamente possível programar um sistema de memória distribuída utilizando primitivas de memória partilhada [Pac97, LER92].

Os sistemas de memória partilhada providenciam mecanismos de criação, tanto estática como dinâmica, de processos. Isto é, os processos podem ser criados no início da execução do programa por uma directiva ao sistema operativo, ou podem ser criados durante a própria execução do programa. A função mais conhecida de criação dinâmica de processos é a função $fork()$. Uma implementação típica permite a um processo iniciar outro, um processo filho, por uma chamada à função $fork()$. O processo inicial, o processo pai, pode esperar pelo fim dos processos filho por uma chamada a uma outra função bastante conhecida, a função $join()$.

A coordenação entre processos nos programas de memória partilhada é gerida tipicamente por três primitivas. A primeira especifica as variáveis que podem ser acedidas por todos os processos. A segunda evita que os processos acedam de forma imprópria aos recursos partilhados. Por fim, a terceira primitiva providencia meios de sincronização entre os processos [Pac97, LER92]. Para ilustrar estas três ideias é apresentado a seguir um exemplo bastante simples.

Neste exemplo cada processo calcula uma variável privada p_x. Uma variável privada é uma variável em que o seu conteúdo só é acessível a um único processo. Logo, cada processo define uma variável distinta p_x que não pode ser acedida pelos outros. O objectivo do programa é calcular a soma destas variáveis privadas, e de um único processo imprimir o resultado.

A primeira primitiva pode ser utilizada para declarar uma variável partilhada *soma*. Uma abordagem possível pode ser a definição de um prefixo *shared* incluído na declaração da variável. Desta forma a declaração de variáveis do programa pode ser realizada da seguinte forma:

```
int p_x;
shared int soma = 0;
```

O programa torna-se um pouco mais complexo para o cálculo da soma. Como é evidente, não se pode simplesmente executar em cada processo o cálculo `soma=soma+p_x;`

Um sistema típico executa algo semelhante à sequência seguinte de instruções máquina para o cálculo da soma:

- Carregar *soma* no registo A

Tempo	Processo 0	Processo 1
0	carregar $soma = 0$	Terminar o cálculo de p_x
1	carregar $p\_x = 2$	Carregar $soma = 0$
2	Adicionar $2 + 0$	Carregar $p\_x = 3$
3	Guardar $soma = 2$	Adicionar $3 + 0$
4		Guardar $soma = 3$

Tabela 3.1: Sequência de eventos.

- Carregar p_x no registo B
- Adicionar o conteúdo do registo B ao registo A
- Guardar o conteúdo do registo A em $soma$

Supondo que foram criados dois processos, 0 e 1, e o conteúdo de p_x em 0 é igual a 2, e em 1 é igual a 3. Então dependendo do instante em que cada processo realiza a soma de p_x com $soma$, o resultado pode ser 2, 3 ou 5 (tabela 3.1). Logo é necessário assegurar que a instrução $soma = soma + p\_x$ seja executada simultaneamente pelos dois processos.

Quando é necessário garantir que apenas um único processo executa uma sequência de instruções de cada vez significa que existe um problema de exclusão mútua e o conjunto de instruções é denominado de *secção crítica*. Uma abordagem simples para resolver o problema de exclusão mútua é denominada de *semáforos binários*. A ideia base consiste na existência de uma variável s cujo valor indica se a secção crítica está livre. Se s é 1 a secção está livre, se for 0 a secção não pode ser acedida. Logo cada processo deverá executar as seguintes instruções:

```
shared int s = 1;

while (s == 0);       /* Espera até o semáforo ser 1 */
s = 0;                /* Nega o acesso aos outros processos */
soma = soma + p_x;    /* Executa a secção crítica */
s = 1;                /* Liberta o acesso */
```

O problema das instruções descritas para a manipulação de s é de estas não serem atómicas. Isto é, enquanto um processo carrega o valor de $s = 1$ num registo para testar se pode aceder à secção crítica, outro processo pode estar a guardar o valor 0 em s. Torna-se então necessário dispor de uma forma de, assim que um processo inicia o acesso a s, nenhum outro processo possa aceder até ao primeiro o libertar, incluindo a alteração do seu valor. Logo, para além da variável partilhada, um semáforo binário contém de mais duas operações especiais: $test\_and\_set()$ e $reset()$.

A primeira função $test\_and\_set()$, apresenta um comportamento semelhante a $while(!s)$; $s = 0$; e previne que outros processos acedam a s assim que um processo sai do ciclo.

De forma semelhante *reset*() coloca o valor de s a 1 mas de forma atómica. Os mecanismos para atingir a atomicidade são dependentes do sistema em questão. Uma solução simples é a inclusão de instruções máquina para "fechar" e "abrir" variáveis. Quando uma variável é fechada apenas o processo que a fechou é que pode escrever nela.

Agora há que encontrar a forma de garantir que a soma correcta é impressa. Por outras palavras, se o processo 0 é responsável por imprimir a soma como é que este pode saber que os outros processos já adicionaram os seus p_x? Seguindo a mesma abordagem que tem vindo a ser apresentada, poderia ser criada uma nova variável partilhada para contar o número de processos que actualizou a variável *soma*. Contudo, esta operação é realizada normalmente por uma instrução de alto nível denominada *barrier*. A instrução *barrier* é normalmente implementada como uma função. Assim que um processo chama a função, esta não retorna até todos os outros processos a terem chamado. Assim, se for chamada uma *barrier*() depois da soma, o processo 0 saberá que a adição foi completamente realizada assim que retorna da chamada. O programa descrito neste exemplo terá então um aspecto semelhante ao programa seguinte:

```
int p_x;
shared int soma = 0;
shared int s = 1;

...
test_and_set(s);
soma = soma + p_x;
reset(s);
barrier();
if (sou o processo 0)
  print(soma);
```

O conceito de variáveis partilhadas e de *barrier* é bastante natural e apelativo para os programadores com experiência em sistemas convencionais. Contudo, a ideia de um semáforo binário já não é tão atraente. É uma fonte potencial de erros e obriga à execução série da região crítica. Este facto levou ao estabelecimento de alternativas. Os *monitores* providenciam uma alternativa de alto nível e estão disponíveis em muitos sistemas. Basicamente, estes encapsulam as estruturas de dados partilhadas e as operações são realizadas nos monitores e não directamente nos dados. Por outras palavras, as estruturas de dados são definidas dentro dos monitores e as regiões críticas são funções desses monitores. Quando um processo executa uma função de um monitor os outros processos são impedidos de a chamar. Infelizmente os monitores não resolvem o problema da serialização. Não é difícil imaginar soluções alternativas ao programa da soma apresentado que não forcem o acesso série às variáveis partilhadas. Contudo as soluções obvias tendem a ser extremamente complexas pois introduzem outras variáveis partilhadas e secções críticas [Pac97].

3.2.2 Memória Distribuída

O método mais comum de programar sistemas MIMD de memória distribuída é pela troca de mensagens, ou por alguma variante deste método. Na sua forma mais básica, os processos coordenam as suas actividades através do envio e recepção de mensagens. Logo, as operações fundamentais neste método de programação são o envio e a recepção de uma mensagem, normalmente disponíveis na forma de duas funções: $Send()$ e $Receive()$.

Supondo que o processo 0 deseja enviar uma mensagem ao processo 1. Então este executa uma chamada à função $Send()$. Para receber a mensagem, o processo 1 tem de realizar uma chamada correspondente à função $Receive()$. Existem algumas questões que devem ser realçadas neste ponto. Primeiro, as operações executadas pelo processo 0 ($Send()$) são diferentes das executadas pelo processo 1 ($Receive()$). Contudo, isto não significa que sejam utilizados programas diferentes para os dois processos. O programa pode incluir um ramo condicional semelhante a:

```
if (processo == 0)
  Send(...);
else if (processo == 1)
  Receive(...);
```

Esta abordagem à programação de sistemas MIMD é denominada de *Single Program Multiple Data* (SPMD). Nos programas SPMD o efeito de executar diferentes instruções em diferentes processos é conseguido pela utilização de ramos condicionais no código do programa. Esta é a abordagem mais comum à programação dos sistemas MIMD [Pac97].

Outra questão que deve ser realçada é a semântica do par Enviar / Receber. Suponhamos que o processo 0 chama a função $Send()$, mas o processo 1 não realiza a chamada correspondente à função $Receive()$. Ou, pior ainda, o processo 0 chama a função $Send()$ mas o processo 1 falha a chamada a $Receive()$. O programa deverá ser terminado com erro ou deverá ficar parado? A resposta depende do sistema utilizado, isto é, se este permite ou não o armazenamento de mensagens [Pac97].

3.2.2.1 Comunicação síncrona e assíncrona

Existem diversas formas para o tratamento do envio e recepção de mensagens. Suponhamos que o processo 0 envia uma mensagem ao processo 1. Antes de enviar a mensagem, o processo 0 pode enviar um "pedido de envio" ao processo 1 e permanecer em estado de espera até receber deste um "pronto a receber", após o qual é realizada a transmissão da mensagem. Uma outra forma de transmissão da mensagem pode ser realizada no caso do sistema permitir o armazenamento da mensagem. Isto é, quando o processo 0 envia a mensagem, o sistema guarda-a temporariamente num bloco de memória, permitindo ao processo 0 continuar a sua execução. Quando o processo 1 fica pronto a receber, o sistema simplesmente copia o conteúdo da mensagem armazenada para a posição de memória indicada pelo processo 1. A primeira abordagem em que o processo 0 espera pelo processo 1 é denominada de *comunicação síncrona*. A segunda abordagem descrita é referida como *comunicação assíncrona*. A vantagem óbvia da comunicação assíncrona é de o processo que envia a mensagem poder continuar a sua execução sem que o processo que a recebe esteja pronto para a receber. Esta abordagem apresenta a desvantagem de utilizar recursos do sistema que de outra forma não seriam necessários e de demorar mais tempo na comunicação pois esta implica a cópia da mensagem entre a área de memória do sistema e as áreas de memória de cada um dos processos (se os processos que recebem as mensagens estiverem prontos).

Grande parte dos sistemas providencia alguma forma de armazenamento, mas os detalhes da sua implementação variam bastante. Alguns sistemas tentam armazenar todas as mensagens. Outros apenas armazenam mensagens relativamente pequenas e utilizam protocolos síncronos para a transmissão das mensagens demasiado grandes. Outros deixam o utilizador decidir se as mensagens devem ser armazenadas e quanto espaço deve ser reservado para o seu armazenamento. Outros ainda, armazenam as mensagens no processo que as envia e outros nos processos que as recebem.

Num sistema que permita o armazenamento de mensagens, se um processo envia uma mensagem a outro processo, o programa não deve terminar enquanto não for executada a sua recepção. O conteúdo da mensagem deverá permanecer na memória do sistema. Se, por outro lado, o sistema não permitir o armazenamento, o processo que envia a mensagem deverá permanecer à espera do “pronto a receber” do outro processo [Pac97].

3.2.2.2 Comunicação bloqueante e não-bloqueante

É também necessário analisar o que acontece no lado da recepção das mensagens. Isto é, supondo que um processo executa uma chamada à função $Receive()$, mas a mensagem só é enviada mais tarde. Neste caso, o processo permanece em estado de espera até que a mensagem seja enviada. Este tipo de recepção de mensagens é conhecida como *comunicação bloqueante*. A comunicação bloqueante não deve ser confundida com a comunicação síncrona. Na comunicação síncrona, os dois processos comunicam directamente, mas o processo que envia a mensagem não inicia a transmissão até receber a permissão explicita do processo que a recebe. No caso da comunicação bloqueante não necessário que o processo que envia a mensagem receba a permissão prévia para a enviar. Isto é, a mensagem pode já ter sido armazenada na memória do sistema antes do processo que a recebe ficar pronto para a receber.

Grande parte dos sistemas oferece uma alternativa, uma operação de recepção de mensagens não bloqueante. Esta operação pode ser executada, por exemplo, por uma função $Imediate\_Receive()$. A função $Imediate\_Receive()$ apenas notifica o sistema operativo que o processo pretende receber uma mensagem. O processo retorna imediatamente da chamada à função, continuando a executar outras tarefas que não dependam da mensagem. Este volta mais tarde a verificar se a mensagem já se encontra disponível.

A utilização de comunicações não bloqueantes pode proporcionar melhorias significativas no desempenho de programas paralelos que utilizem a troca de mensagens. Se um nó de um sistema paralelo consegue comunicar e realizar cálculos em simultâneo, o tempo utilizado nas comunicações pode ser substancialmente reduzido. Por exemplo, se cada nó de um sistema possuir um processador dedicado para as comunicações, então é possível iniciar uma comunicação não bloqueante ($Imediate\_Receive()$), realizar cálculos que não dependam do resultado da comunicação e, após os cálculos estarem completos, terminar a comunicação iniciada. Enquanto os cálculos estão a ser realizados, o processador dedicado às comunicações pode realizar grande parte do trabalho requerido pela comunicação. Como as comunicações são relativamente mais penosas que os cálculos, a sobreposição dos cálculos e das comunicações pode resultar em ganhos de desempenho bastante significativos [Pac97, LER92].

3.2.2.3 Distribuição dos dados

Uma das abordagens mais simples à programação de sistemas paralelos é conhecida como *paralelismo nos dados*. Nesta abordagem os dados são distribuídos pelos vários processos, em que cada processo executa as mesmas instruções na sua parte dos dados. Sem dúvida esta abordagem é adequada às máquinas SIMD. Contudo, é muito comum ser utilizada em sistemas MIMD. Um dos seus aspectos mais atractivos é, para estruturas de dados muito regulares, ser possível ao programador indicar apenas que a estrutura deve ser distribuída pelos processos e o compilador substitui automaticamente esta indicação pelo código de distribuição e de operações paralelas sobre os dados [Pac97, LER92].

Mas a distribuição dos dados pelos processos é, em geral, bastante difícil, a não ser que estes se encontrem estruturados de forma estática e bastante regular (como por exemplo, uma matriz). Este pode ser um problema bastante sério em qualquer programa paralelo. Contudo é especialmente problemático nos programas em que é utilizado paralelismo nos dados e onde a distribuição é especificada em tempo de compilação.

O paralelismo nos dados é utilizado de forma diferente noutros contextos. Pode ser utilizado para descrever a metodologia para o desenho de um programa paralelo. Neste contexto é normalmente contrastado com a programação com paralelismo de controlo, em que o paralelismo é obtido pela divisão do controlo ou das instruções do programa em vez dos dados. Em geral, grande parte dos programas paralelos utilizam ambas as abordagens para a abordagem ao paralelismo [Pac97, LER92].

A questão da localização dos dados é uma questão crítica na programação de ambos os sistemas de memória distribuída e de memória partilhada de acesso não uniforme

(NUMA). Em geral, as operações de comunicação são bastante mais dispendiosas que as operações de processamento. Nos sistemas convencionais, é um facto que as instruções de acesso à memória são bastante mais lentas que as operações que envolvem apenas a CPU. Esta diferença de tempos é ainda mais dramática se a memória for remota, isto é, se for a memória local de outro nó num sistema de memória distribuída ou um módulo de memória "distante" num sistema de memória partilhada NUMA. Logo, um esforço considerável tem sido dedicado ao problema da localização dos dados, isto é, o problema da distribuição dos dados pelos processadores de forma a minimizar a comunicação. Uma solução, um tanto ao quanto extrema, é guardar os dados na memória de um único nó, num sistema distribuído, e deixar que esse nó realize todo o processamento. Um procedimento semelhante pode ser aplicado aos sistemas de memória partilhada NUMA. É claro que esta solução resultaria num enorme desperdício de recursos computacionais. Logo, existe um compromisso entre o problema da distribuição do processamento, normalmente referido como *balanceamento da carga* e o problema da localização dos dados. Isto é, pretende-se atribuir a mesma quantidade de trabalho a cada processador, caso contrário estão a ser desperdiçados recursos computacionais. Qualquer distribuição dos dados deve considerar quer o balanceamento da carga quer a localização dos dados [Pac97, LER92].

Talvez o caso mais simples para o problema da localização dos dados consiste na forma de distribuir um vector por um grupo de nós num sistema de memória distribuída. Supondo que se pretende distribuir um vector com n elementos por p nós e que a quantidade de processamento associada a cada elemento do vector é aproximadamente a mesma. Por outras palavras, se for atribuído o mesmo número de elementos do vector a cada nó é atingido o objectivo do balanceamento da carga. Se o número de nós, p, é igual ao número de elementos do vector, n, então só existe uma distribuição possível: a cada processador é atribuído um único elemento do vector, o que faz com que o problema pareça trivial.

Se n é múltiplo de p, então pode parecer, à primeira vista, que existem apenas duas formas de distribuição equitativa dos dados: a distribuição dos dados pode ser realizada através de um mapeamento por blocos em que os dados são divididos em blocos de entradas consecutivas e cada bloco é atribuído a nós distintos; ou a distribuição consiste num mapeamento cíclico onde o primeiro elemento do vector é atribuído ao primeiro nó, o segundo ao segundo nó e assim sucessivamente. Quando cada nó contém um elemento do vector, a atribuição recomeça novamente com os próximos p elementos. Este processo é repetido até todos os elementos do vector terem sido atribuídos a um dos nós. Outra distribuição possível consiste numa variante de um mapeamento cíclico. O vector é dividido em blocos de elementos consecutivos como no mapeamento por blocos. Contudo os blocos não são de tamanho igual a n/p, mas menores. Estes blocos são então atribuídos aos nós da mesma forma que os elementos são atribuídos no mapeamento cíclico.

Até agora tem sido considerado que o número de elementos n é múltiplo de p. Mas, se não for o caso, depressa se chega à conclusão que existem inúmeras formas diferentes de distribuição apenas para um vector de elementos. Se as estruturas de dados forem matrizes de várias dimensões, ou árvores e grafos, o problema torna-se astronomicamente complexo. Então como deve ser escolhida a distribuição de dados apropriada? Não surpreende que o mapeamento dos dados seja dependente do problema em questão, existindo inúmeras referências com análises e discussões para diferentes distribuições [Pac97, LER92].

3.3 Desempenho

O objectivo principal da computação paralela é o de desenhar programas paralelos que sejam capazes de resolver problemas de maiores dimensões em menos tempo. Se os programas série fossem suficientemente rápidos para resolver todos os problemas pretendidos e se conseguissem armazenar em memória todos os dados, a programação paralela seria apenas um exercício intelectual. Logo um dos pontos críticos no desenho de programas paralelos é o cálculo da estimativa do desempenho. Antes de se implementar um programa deve ser realizada uma estimativa do seu desempenho de forma a verificar se este cumpre os objectivos pretendidos [Pac97, LER92].

Embora seja dada especial atenção ao tempo de execução, é também importante considerar outros aspectos na avaliação do desempenho de um programa paralelo. Em particular, deve ser sempre considerado o custo de desenvolvimento do programa. Muitos programas paralelos são desenhados para obterem o melhor desempenho possível e, como consequência, demoram anos a ser desenvolvidos. Claramente, deve ser sempre ponderada a decisão de desenvolver um programa, rápido, complexo e caro, face ao custo de um programa mais simples embora mais lento.

3.3.1 Análise de desempenho de um programa série

A análise de desempenho é uma parte integrante do desenvolvimento de um programa. Logo, existem diversos graus de precisão implícitos na expressão *análise de desempenho*. Antes de ser escrito qualquer código, só é possível realizar uma estimativa de desempenho que envolve um número arbitrário de constantes simbólicas da estimativa do número de expressões executadas. Á medida que o desenvolvimento prossegue, algumas constantes podem ser substituídas por números que apenas são válidos para um sistema e compilador em particular. Logo, numa primeira análise de um programa, o seu tempo de execução é especificado pelo número de expressões executadas. À medida que os detalhes da implementação num determinado sistema emergem, o tempo de execução pode ser especificado em milisegundos ou microsegundos.

Quando o tempo de execução de um programa é discutido, pretende-se ser capaz de afirmar algo semelhante a "o tempo de execução deste programa é T(n) unidades de tempo se os dados de entrada forem de dimensão n". É claro que o tempo de execução de um programa - o intervalo de tempo desde o início da execução até ao fim - irá depender de outros factores para além da dimensão dos dados de entrada.

Por exemplo, irá depender do hardware utilizado, da linguagem de programação e do compilador, ou de alguns detalhes dos dados de entrada para além da sua dimensão.

Para evitar o primeiro factor (o hardware utilizado), na análise inicial o tempo de execução vai ser relativo ao número de "expressões executadas". Isto levanta uma outra questão. Se estão a ser contadas expressões, estas são expressões em assembler? Se forem, que tipo de assembler, RISC ou CISC? Ou, as expressões são numa linguagem de alto nível e, se forem, que linguagem? Também, será que contar expressões é correcto dado que, em geral, expressões diferentes apresentam tempos de execução diferentes? Por fim, é bastante fácil encontrar programas que apresentam comportamentos bastante diferentes com dados de entrada diferentes, mesmo que a sua dimensão seja a mesma. Por exemplo, o algoritmo de ordenação *Insertion Sort*, que ordena um vector de n inteiros por ordem crescente através de uma pesquisa linear, irá demorar menos tempo se os valores do vector forem $1, 2, ..., n$ do que se forem $n, n-1, ..., 1$. Este problema pode ser evitado se a análise de desempenho se referir ao tempo de execução na pior situação ou no caso médio.

De uma forma geral, o tempo de execução refere-se à pior situação. Contudo, quando são contadas expressões os dois primeiros casos não podem ser evitados sem introduzir alguma imprecisão nas medidas. Grande parte dos autores trata esta imprecisão pela utilização de uma *análise assimptótica*. Na análise assimptótica são especificados os limites do desempenho de um programa. Por exemplo, é normalmente mencionado que o algoritmo de ordenação conhecido como *Bubble Sort* é um algoritmo de ordem n^2, ou $O(n^2)$. Isto significa que, se o algoritmo Bubble Sort for utilizado para ordenar uma lista de valores de n elementos, o número de expressões executadas vai ser menor que uma constante múltipla de n^2, desde que n seja suficientemente grande. Este tipo de análise tem sido muito útil para a estimativa de desempenho de programas série.

Esta ideia é ilustrada no exemplo seguinte. Neste exemplo é estimado o tempo de execução de um programa de integração em que é utilizada a regra dos trapézios. Os dados de entrada do programa consistem no primeiro e no último ponto do intervalo de integração (a e b) e do número de trapézios (n) utilizados. O programa calcula o integral de uma função (f) pela soma das áreas dos vários trapézios. Assim que acaba a soma, o programa retorna o resultado. O código fundamental do programa é o seguinte:

```
h = (b - a) / n;
integral = (f(a) + f(b)) / 2;
x = a;
for (i = 1; i <= n-1; i++) {
    x = x + h;
    integral = integral + f(x);
}
integral = integral * h;
```

O número de expressões executadas fora do ciclo for não depende da dimensão dos dados de entrada. Pode dizer-se que existem c_1 expressões. Se for assumido que o cálculo de $f(x)$ requer um número constante de instruções, então o número de expressões executadas em cada passo do ciclo for é também constante, por exemplo, c_2. Logo, o número total de expressões executadas é de $T(n) = c_1 + c_2 * (n - 1)$. Isto é, $T(n)$ é linear em n. Depois de reagrupar e renomear as constates, obtém-se $T(n) = k_1 * n + k_2$. Para este programa simples é razoável assumir que o tempo de execução das várias expressões não é muito diferente. Ainda, é de esperar que o número total de expressões executadas dentro do ciclo for seja consideravelmente maior que o número de expressões executadas fora do ciclo. Isto é, $k_1 >> k_2$. Logo, $T(n)$ pode ser aproximado por: $T(n) = k_1 * n$. Logo é de prever que, se n for aumentado de um factor r, $T(n)$ também aumente pelo mesmo factor. Isto é, se $T(n) = t$, é de prever que o tempo de execução para uma dimensão dos dados de entrada $r * n$, $T(r * n)$ seja aproximadamente de $r * t$.

3.3.2 Análise de desempenho de um programa paralelo

Uma diferença clara entre a estimativa de desempenho entre um programa série e um paralelo é que o tempo de execução de um programa paralelo depende de duas variáveis: a dimensão dos dados de entrada e o número de processos. Logo, em vez de ser utilizado $T(n)$ para representar o tempo de execução, é utilizada uma função de duas variáveis, $T(n, p)$. Esta função representa o intervalo de tempo desde o início de execução do primeiro processo até à execução da última expressão pelo o último processo.

Quando se discute o desempenho de um programa paralelo, é normalmente levantada a questão de como comparar o seu desempenho com o do programa série equivalente. As medidas de comparação mais utilizadas são o *speedup* e a *eficiência*. Estas medidas são discutidas com um pouco mais de detalhe mais à frente. Resumidamente, o speedup representa a razão do tempo de execução de um programa série sobre o tempo de execução do programa paralelo. Isto é, se $T_s(n)$ representar o tempo de execução do programa série e $T_p(n, p)$ representar o tempo de execução da versão paralela com p processadores, então o speedup do programa paralelo é calculado como:

$$S(n, p) = T_s(n)/T_p(n, p) \tag{3.1}$$

Existe alguma ambiguidade nesta definição. Por exemplo, será que o programa série é apenas o programa paralelo a ser executado com um único processo, ou será que é o programa série equivalente mais rápido? Grande parte dos autores definem o speedup como sendo a razão entre o tempo de execução do programa série mais rápido, executado no sistema paralelo, sobre o tempo de execução do programa paralelo.

Para um valor fixo de p, normalmente dá-se o caso de $0 < S(n, p) < p$. Se $S(n, p) = p$, diz-se que o programa paralelo apresenta um *speedup linear*. É claro que esta situação é rara dado que grande parte dos programas paralelos introduzem algum processamento extra devido principalmente às comunicações entre os processos. Infelizmente, uma situação bastante mais comum é o *abrandamento* (ou *slowdown*). Isto é, o programa paralelo quando executado com mais do que um processo é mais lento que o programa série.

Uma medida de desempenho alternativa ao speedup é a eficiência. A eficiência mede a utilização dos processos num programa paralelo, relativamente ao programa série. A eficiência é definida da seguinte forma:

$$E(n,p) = S(n,p)/p = T_s(n)/(p * T_p(n,p)) \tag{3.2}$$

Como $0 < S(n,p) <= p$, $0 < E(n,p) < 1$. Se $E(n,p) = 1$, o programa exibe um speedup linear, enquanto se $E(n,p) < 1/p$, o programa exibe um abrandamento.

3.3.2.1 Custo das comunicações

As comunicações, assim como as operações de Entrada/Saída, são significativamente mais pesadas que os cálculos locais. Logo, para se poder realizar uma estimativa razoável do desempenho de programas paralelos deve ser tomado em conta o custo das comunicações separadamente do custo dos cálculos e das operações de Entrada/Saída. Isto é, $T(n,p) = T_{calc}(n,p) + T_{i/o}(n,p) + T_{comm}(n,p)$.

Para ser possível realizar uma estimativa do custo das comunicações devem ser analisadas as operações realizadas quando dois processos comunicam entre si. Quando um processo envia uma mensagem a outro processo, os detalhes do que acontece ao nível do hardware variam bastante de sistema para sistema. Diferentes sistemas utilizam protocolos de comunicação diferentes, e o mesmo sistema irá executar diferentes conjuntos de instruções dependendo se os dois processos residem no mesmo processador ou em processadores distintos. Pondo de parte o primeiro caso, quando os processos se encontram em processadores distintos podem ser realizadas algumas observações gerais sobre o que acontece na grande maioria dos sistemas [Pac97].

A execução do par enviar / receber pode ser dividida em duas fases: Uma fase de inicialização e outra de comunicação. Mais uma vez, os detalhes do que é realizado em cada uma das fases varia consoante o sistema. Contudo, tipicamente durante a fase de inicialização a mensagem é copiada para uma área de memória controlada pelo sistema juntamente com alguma informação adicional como a origem e o destino da mensagem. Durante a fase de comunicação os dados são transmitidos entre os processadores. Pode existir ainda uma outra fase, análoga à fase de inicialização, no processo receptor, na qual a mensagem é copiada de uma área de memória controlada pelo sistema para a área de memória reservada pelo processo.

Os custos de cada uma destas fases variam de sistema para sistema, logo irão ser representados por constantes. É utilizado t_s para representar o tempo de execução da fase de inicialização, incluindo também o tempo utilizado na cópia da mensagem da área de memória controlada pelo sistema para a área de memória reservada pelo processo receptor. A constante t_c representa o tempo necessário para transmitir uma única unidade de dados de um processo para outro. A unidade de dados pode ser um byte, uma palavra de memória ou qualquer outra unidade maior. Utilizando esta notação, o custo de transmitir uma mensagem de k unidades de dados é de $t_s + k * t_c$.

O tempo t_s é por vezes denominado de *latência*, e o inverso de t_c é denominado de *largura de banda*. É muito difícil obter estimativas fiáveis de t_s e t_c. Para um sistema em particular, a melhor forma de o realizar é escrever programas que trocam mensagens e extraiam tempos. Em grande parte dos sistemas, t_c é da mesma ordem de grandeza que o custo de uma operação aritmética, t_a, e t_s encontra-se entre uma a três ordens de grandeza superiores a t_c.

O programa do exemplo seguinte é uma versão paralela do programa de integração utilizado no exemplo anterior. Para este programa é estimado o seu tempo de execução, incluindo o custo das comunicações. É ainda realizada uma estimativa do seu speedup.

```
h = (b - a) / n;      /* h é o mesmo para todos os processos */
local_n = n / p;      /* assim como o número de intervalos */

/* O tamanho do intervalo de integração de cada processo = local_n * h.
   Logo o meu intervalo começa em: */
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;

/* Chama a função de integração série */
integral = Serial_Integral(local_a, local_b, local_n, h);

/* Adiciona os integrais calculados por cada processo */
Soma_Paralela(&integral);
```

Á excepção das chamadas às funções $Serial\_Integral()$ e $Soma\_Paralela()$, as restantes expressões são executadas num tempo constante e apenas adicionam uma constante, c_1, ao tempo total de execução. A função $Serial\_Integral()$ executa o código (série), apresentado no exemplo anterior, para o cálculo do integral no intervalo de cada processo. Como cada processo soma as áreas de n/p trapézios, esta secção do código apresenta um tempo de execução igual a:

$c_2 * (n/p - 1) + c_3$

O tempo de execução estimado para a função $Soma\_Paralela()$ é de:

$c_4 * log_2(p) * (t_s + t_c + t_a) + c_5$

Esta estimativa é baseada na suposição da $Soma\_Paralela()$ ser implementada através da função $MPI\_Reduce()$ providenciada pela *Message Passing Interface* - MPI, discutida no final do capítulo, e para a qual é utilizado um padrão de comunicação estruturado em árvore. Na estimativa encontrada, t_a representa o tempo necessário para o cálculo da soma. Logo o tempo total de execução estimado para o programa paralelo é igual a $T_p(n,p) = k_1 * (n/p) + k_2 log_2(p) + k_3$. Da discussão anterior, é esperado que k_3 seja desprezável. Logo, o speedup previsto pode ser calculado como:

$S(n,p) = \frac{k_1 * n}{k_1 * (n/p) + k_2 log_2(p)}$.

3.3.3 Speedup

Já foi referido que o speedup é uma medida utilizada para a avaliação do desempenho de um programa paralelo. Também já foi referida a forma como este é calculado (equação 3.1). Resumidamente, o speedup é obtido pela razão entre o tempo de execução do programa série e o tempo de execução do programa paralelo ($T_s(n)/T_p(n,p)$). Mas o que é que esta medida realmente representa?

3.3.3.1 A lei de Amdahl

Desde o início da computação, os construtores têm-se apresentado um pouco cépticos em relação ao efeito multiplicativo de vários processadores a trabalhar em conjunto na resolução de um único problema. Em 1967, Gene Amdahl, um projectista da IBM afirmou [Amd67]:

"Por mais de uma década os profetas tinham proclamado a controvérsia que a organização de um único computador tinha chegado aos seus limites e que avanços verdadeiramente significativos só poderiam ser atingidos por uma interligação de uma multiplicidade de computadores de tal forma a permitir uma solução cooperativa... A natureza deste overhead (introduzido pelo paralelismo) parece ser sequencial de forma a que é pouco provável de ser agradável às técnicas de processamento paralelo. O overhead sozinho irá estabelecer um limite máximo à resposta do sistema de cinco a sete vezes a resposta do processamento série, mesmo se a lida da casa for realizada por um processador separado... A dado ponto no tempo é difícil prever como os bottlenecks anteriores serão ultrapassados eficientemente."

Conhecida como a *lei de Amdahl*, este aviso foi traduzido numa equação que resultou num limite estimado para o ganho de desempenho dos programas paralelos. A seguir é apresentado um exemplo onde é utilizada a lei de Amdahl para estimar o desempenho de um programa paralelo genérico [Pac97, LER92].

Imagine-se que um determinado programa paralelo requer um tempo T para resolver uma instância de um problema utilizando um único processo. Uma instância de um problema é definida pelo problema mais um determinado conjunto de dados de entrada. Por exemplo, integrar a função e^{-x^2} no intervalo $[0,1]$ utilizando a regra dos trapézios com 512 trapézios. Imagine-se também que o programa é formado por uma fracção r de expressões ($0 <= r <= 1$) que são puramente paralelas. Isto é, independentemente do número de processadores utilizados (p) o programa apresenta um speedup linear e um tempo de execução igual a $r * T/p$ quando é executado em paralelo com p processos. Imagine-se também que a restante fracção do programa, $1-r$, é inerentemente série. Ou seja, independentemente do número de processadores p utilizado o tempo de execução desta fracção é de $(1-r)*T$ quando é executada em paralelo com p processos. Um exemplo do que poderá ser uma expressão de código inerentemente série é uma expressão de leitura de dados, onde um único valor é necessário para todos os processos. Sem ser discutido, de momento, até que ponto estas suposições são válidas, o speedup do programa é dado por:

$S(p) = \frac{T}{(1-r)*T+r*T/p}$

Ao derivar S em relação a p obtém-se:

$\frac{dS}{dp} = \frac{r}{((1-r)*p+r)^2} >= 0$

Logo $S(p)$ é uma função crescente em p e, á medida que p tende para infinito $S(p)$ tende para $1/(1-r)$. Logo o speedup encontra-se limitado superiormente por $(1-r)^{-1}$.

Por exemplo, se $r = 0.5$, o speedup máximo possível é 2. Se $r = 0.75$, o speedup máximo possível é igual a 4, e se $r = 0.99$, o speedup máximo possível é 100. Em particular, se $r = 0.99$, mesmo que sejam utilizados 10000 processos, não é possível obter um speedup melhor do que 100, que corresponde a uma eficiência de 0.01.

Será que a lei de Amdahl está correcta? Se sim, então parece implicar que não existe nenhuma razão para se tentar usar paralelismo massivo (isto é, centenas de milhares de processos). A matemática está claramente correcta. Logo, desde que se confie nas suas hipóteses, a lei também está correcta. Será então razoável aceitar as hipóteses por de trás da lei de Amdahl?

Dada uma instância de um problema e um programa para a sua resolução, parece razoável assumir que uma determinada fracção das expressões do programa não possam ser executadas em paralelo. Parece mesmo demasiado optimista assumir que as restantes expressões sejam executadas "perfeitamente" em paralelo. Contudo, o que não parece ser razoável é a utilização de um número arbitrário de processos para a resolução de uma determinada instância de um problema. Por outras palavras, o parâmetro n, a dimensão do problema, encontra-se notavelmente ausente de todas as equações derivadas [Pac97, LER92].

Parece óbvio, por exemplo, que, ao tentar utilizar mais de 500 processos para resolver uma instância de um problema que apenas necessite de 500 cálculos, se estejam a desperdiçar recursos. Antes, devem ser utilizados mais processos para a resolução de instâncias maiores do problema, e talvez, ao serem utilizadas instâncias de maiores dimensões, a fracção série tornar-se-á mais pequena. Esta foi uma das ideias chave no desenvolvimento dos primeiros programas que apresentaram speedups superiores a 1000. Por exemplo, no programa de integração apresentado, existem essencialmente duas expressões série: a leitura dos dados de entrada e a saída de resultados. Logo, à medida que n, o número de trapézios utilizados na integração, aumenta, a parte série do programa terá uma contribuição cada vez mais insignificante no computo global do programa.

3.3.3.2 Trabalho e overhead

A observação de que em muitos programas paralelos a importância da fracção inerentemente série do programa decresce, à medida que a dimensão do problema cresce, aponta uma visão mais optimista da programação paralela que a lei de Amdahl.

Um segmento inerentemente série de um programa pode ser executado em paralelo essencialmente de duas formas: um único processo executa o segmento enquanto que os restantes processos permanecem em estado de espera, ou todos os processos executam o mesmo segmento. Ambas as abordagens são fontes de *overhead* paralelo, isto é, ambas as abordagens aumentam a quantidade total de trabalho realizada pelo programa em paralelo em relação à quantidade realizada pelo programa série.

De uma forma mais precisa, a quantidade de trabalho realizada por um programa série pode ser representada simplesmente pelo tempo de execução: $W_s(n) = T_s(n)$. A quantidade de trabalho realizada por um programa em paralelo pode ser calculada como soma da quantidade de trabalho realizada por cada processo, ou seja, $W_p(n) = \sum_{q=0}^{p-1}(W_q(n,p))$, onde $W_q(n,p)$ representa a quantidade de trabalho realizada pelo programa executado no processo q. É importante observar que, de forma diferente do que acontece no mundo real, o trabalho inclui o tempo de espera. Se ao medir o tempo de execução de um programa série este permanece em estado de espera durante algum tempo por alguma razão (por exemplo, à espera dos dados de entrada), então este tempo de espera é incluído no tempo total de execução ou no trabalho. Logo, cada $W_q(n,p)$ deverá incluir o tempo que um processo q permanece em estado de espera. Mas, o facto de este ser considerado, simplifica bastante os cálculos realizados pois implica que, para cada processo q, $W_q(n,p) = T_p(n,p)$. Isto é, $W_q(n,p)$ é simplesmente o tempo de execução do programa paralelo pois o processo q ou está a realizar trabalho útil ou encontra-se em estado de espera durante a execução do programa. Logo a equação para $W_p(n)$ é simplesmente igual a $W_p(n) = p * T_p(n,p)$.

Baseado no trabalho, uma definição alternativa para eficiência é: $E_p(n) = T_s(n)/(p * T_p(n,p)) = W_s(n)/W_p(n)$. A partir desta definição, dizer que um programa paralelo apresenta um speedup linear, ou $E_p(n) = 1$, é equivalente a afirmar que "A quantidade de trabalho realizada por um programa paralelo é a mesma que a quantidade de trabalho realizada pelo programa série equivalente". Pode-se então formalizar a definição de *overhead*. Este representa a quantidade de trabalho realizada pelo programa paralelo que não é realizada pelo programa série, e pode ser definida simplesmente como: $T_o(n,p) = W_p(n) - W_s(n) = p * T_p(n,p) - T_s(n)$. Ou ainda, pode definir-se overhead por processo como a diferença entre o tempo de execução do programa paralelo com o tempo de execução ideal obtido com speedup linear: $T'_o(n,p) = T_p(n,p) - T_s(n)/p$. A partir desta última definição pode-se pensar na definição de overhead como "overhead total": $T_p(n,p) = p * T'_o(n,p)$.

Então, para o programa de integração utilizado nos exemplos anteriores o overhead $T_o(n,p)$ é calculado facilmente. Da análise do tempo de execução, o tempo estimado para o programa série é igual a $T_s(n) = k_1 * n$, e para o programa paralelo, igual a $T_p(n,p) = k_1 * n/p + k_2 * log_2(p)$. Logo, o trabalho realizado pelo programa paralelo é igual a $W_p(n) = k_1 * n + k_2 * p * log_2(p)$. A equação de overhead encontrada para o programa de integração é igual a $T_o(n,p) = k_2 * p * log_2(p)$. Por outras palavras, o trabalho total realizado pelo programa paralelo irá diferir do trabalho total realizado pelo programa série de $k_2 * p * log_2(p)$. Isto é verdade independente do valor de n. Logo se n for relativamente grande em relação a p, é de esperar que a quantidade de trabalho realizada pelo programa paralelo seja muito semelhante à quantidade de trabalho realizada pelo programa série.

Fontes de overhead: Pode-se afirmar, das conclusões anteriores, que existem três fontes principais de overhead: comunicação, tempo de espera e comunicação extra [Pac97, LER92].
É claro que a comunicação entre processos não é uma questão levantada para um programa série. Logo, qualquer comunicação entre processos vai contribuir para T_o.
O tempo de espera pode ou não contribuir para o overhead. Infelizmente, geralmente contribui. Um ou mais processos permanecem em estado de espera enquanto aguardam por informação de outro processo. Por exemplo, no programa de integração, os processos 1, 2, ..., p-1 permanecem em estado de espera enquanto o processo 0 lê os dados de entrada.
Por fim, existem numerosas oportunidades para um programa paralelo realizar cálculos extra. Algumas das mais óbvias são cálculos que não realizados pelo programa série e cálculos repetidos. Um exemplo da primeira é o cálculo por cada processo, no programa de integração, do subintervalo de integração. Claramente este cálculo não é necessário no programa série. Um exemplo da segunda razão ocorre quando cada processo calcula o número de trapézios utilizados para o cálculo do integral. Como o programa assume que n é múltiplo de p, este valor é igual em todos os processos (este é também um exemplo de um cálculo não necessário no programa série).
A equação de overhead encontrada para o programa paralelo do integral é simplesmente igual *p * (o tempo de execução da função Soma_Paralela())*. Isto é, se forem ignoradas as operações de leitura e escrita, o overhead deve-se à chamada da função $Soma\_Paralela()$. Grande parte do tempo utilizado pela chamada a esta função advém das comunicações. Contudo, também existe tempo de espera e computação extra. Ao ser utilizada uma comunicação estruturada em árvore, cada processo, para além do principal (neste caso o processo 0), encontra-se em estado de espera antes de se completar o cálculo. Também, cada processo tem de realizar alguns cálculos a cada passo para determinar qual o processo com quem vai comunicar.

3.3.3.3 Escalonabilidade

Entre 1967 e 1988, a lei de Amdahl assombrou os investigadores da computação paralela. Esta tornou-se um tal "incómodo" para computação paralela que Gordon Bell ofereceu uma recompensa de $1.000 a quem conseguisse aplicar com sucesso a computação paralela a problemas práticos reais. Bell estava particularmente interessado em executar "problemas de produção" em computadores paralelos para demonstrar que estes poderiam ser utilizados por pessoas comuns nos problemas do dia a dia.

Em 1988 uma equipa de investigadores dos Sandia Labs ganhou o prémio de Gordon Bell para o processamento paralelo com um nCUBE com 1024 processadores. Aparentemente, conseguiram ultrapassar a lei de Amdahl apresentando speedups na ordem das centenas para um problema com uma parte inerentemente sequencial, r, entre 0.004 e 0.008, para o qual a lei de Amdahl prevê um speedup entre 125 e 250. Enquanto os investigadores dos Sandia Labs não eram bem pessoas que resolviam problemas de todos os dias, nos anos seguintes o prémio de Gordon Bell foi ganho por outras pessoas que resolveram alguns problemas extremamente práticos através da computação paralela.

Este resultado deu origem a uma séria revisão da lei de Amdahl. Um dos principais resultados dessa revisão foi a ideia de *escalonabilidade*. Essencialmente, um programa é *escalonável* se é possível manter um dado nível de eficiência à medida que é aumentado o número de processos e, simultaneamente, a dimensão do problema [Pac97, LER92].

A lei de Amdahl afirma que existe um limite superior bem definido para o speedup que pode ser atingido por um programa paralelo. Do ponto de vista da eficiência, se r é a fracção do programa que pode ser executada em paralelo, então

$$E = \frac{T_s}{p*(1-r)*T_1+r*T_1)} = \frac{1}{p*(1-r)+r}$$

Desta forma, a partir da lei de Amdahl, pode-se concluir que a eficiência de um programa paralelo, aplicado a uma instância de um problema, aproxima-se de zero à medida que o número de processos utilizados aumenta. Reformulando esta conclusão em termos de overhead, $T_o(n,p) = p * T_p(n,p) - T_s(n)$, a equação da eficiência pode ser rescrita como

$$E(n,p) = \frac{T_s(n)}{p*T_p(n,p)} = \frac{T_s(n)}{T_o(n,p)+T_s(n)} = \frac{1}{T_o(n,p)/T_s(n)+1}$$

A partir desta última equação pode-se afirmar que o comportamento da eficiência em relação a p é completamente determinado pela função de overhead, $T_o(n,p)$.

Para o programa de integração, que tem vindo a ser utilizado como exemplo, foi determinado que $T_o(n,p) = k_2 * p * log_2(p)$. Logo, a eficiência obtida, para uma dada instância de um problema, aproxima-se de zero à medida que o número de processos utilizados aumenta. Este comportamento é completamente esperado. As estimativas mostram que, para um determinado valor de n, a eficiência decresce à medida que p aumenta. Contudo, esta equação também mostra que, para um número fixo de processos p, à medida que n aumenta é possível obter uma eficiência tão próxima de

1 quanto o desejado.

Mas, e o que é que acontece quando ambos p e n são aumentados em simultâneo? Se forem substituídos T_s e T_o pelas equações correspondentes na equação de eficiência encontrada obtém-se:

$E(n,p) = \frac{1}{k_2 * p * log_2(p)/(k_1 * n) + 1}$

Se n e p forem aumentados na mesma proporção, a eficiência irá diminuir dado que $p * log_2(p)$ irá crescer ligeiramente mais rápido que n. Contudo, se n crescer na mesma proporção que $p * log_2(p)$, então é possível manter uma eficiência constante. Esta última observação é a chave para a escalonabilidade.

Um programa paralelo é *escalonável* se, à medida que o número de processos (p) aumenta, for possível encontrar uma taxa de crescimento para a dimensão do problema (n) de tal forma a que a sua eficiência permaneça constante. É claro que esta definição sugere a existência de graus variáveis de escalonabilidade. Se for possível resolver a equação de eficiência para n em ordem a E e p, obtém-se uma formula explicita para a taxa de crescimento da dimensão do problema. Mas, por vezes pode tornar-se muito difícil resolver a equação para n em ordem a E e p. Por exemplo, se $T_s(n)$ for de grau 5 em n, não existe nenhuma abordagem geral que permita resolver a equação.

Convém não perder uma visão das considerações práticas ao manipular as equações. Por exemplo, supondo que foi desenvolvido um programa paralelo com speedup linear. Então, $T_o(n,p) = p * T_p(n,p) - T_s(n) = p * T_s(n)/p - T_s(n) = 0$ e, teoricamente, $E = 1$ independentemente de n e p. Contudo, num problema real estas igualdades depressa irão falhar. Supondo que é pretendido adicionar dois vectores distribuídos que devem permanecer distribuídos após a adição, então cada processo simplesmente adiciona as suas componentes locais dos vectores. Se n, a dimensão dos vectores, for múltipla de p, e a cada processo é atribuído n/p elementos de cada vector, então o programa apresenta um speedup linear. A chave do speedup ser linear para este problema resulta de n ser múltiplo de p. Em particular, se $p > n$, alguns processadores irão permanecer em estado de espera e o overhead não será 0. Por outras palavras, para se manter uma eficiência igual a 1 para um número arbitrário de processos p, n deverá crescer de forma proporcional a p.

3.4 Message Passing Interface - MPI

Dado que o principal objectivo deste trabalho ser o de desenvolver uma aplicação para ser executada num sistema paralelo, não é possível ignorar as questões relacionadas com o hardware e software do sistema bem como as decisões realizadas a esse nível que irão afectar o desempenho da aplicação. De facto, há pouco tempo, poderia ser mesmo necessário aprender duas linguagens de programação completamente novas se fosse pretendido programar duas máquinas paralelas diferentes. Contudo, esta situação foi bastante melhorada. Fundamentalmente, dois grupos iniciaram a tarefa de desenvolver padrões para a programação de sistemas paralelos.

O primeiro grupo, *The High Performance Fortran Forum*, desenvolveu um conjunto de extensões à linguagem de programação Fortran 90 que permite desenvolver, de forma rápida e eficiente, programas com paralelismo nos dados. Um programa com paralelismo nos dados consegue o paralelismo pela distribuição dos dados (por exemplo, vectores ou matrizes) pelos processadores, em que cada processador realiza (mais ou menos) as mesmas operações sobre a parte dos dados que lhes foi atribuída. O sistema *High Performance Fortran* (HPF) proporciona um conjunto de directivas para distribuir convenientemente listas de elementos de dados e operar sobre estas em paralelo. O HPF é um sistema de programação extremamente poderoso e bem desenhado. Contudo, existem muitos algoritmos paralelos que não podem ser convertidos em linguagem máquina de uma forma eficiente pela geração corrente de compiladores HPF [Pac97].

O segundo grupo, o *Message-Passing Interface Forum (MPI)* [Pac97], realizou uma abordagem diferente para o desenvolvimento de um padrão para a programação de um sistema paralelo. Ao contrário de especificar uma nova linguagem (e desta forma um novo compilador), especificou uma biblioteca de funções que podem ser utilizadas num programa em C ou Fortran. Esta biblioteca consiste de um conjunto de definições e funções que podem ser utilizadas para implementar programas paralelos que utilizam a troca de mensagens.

A troca de mensagens pode ser vista, na sua forma mais simples, como a passagem de informação entre dois processos. Este método é uma abordagem bastante geral para a implementação de programas paralelos extremamente eficientes e é actualmente um dos métodos mais utilizados para a programação de muitos sistemas paralelos.

O principal inconveniente na utilização deste método é a dificuldade em desenhar e desenvolver programas paralelos através da troca de mensagens. De facto, a troca de mensagens tem sido por vezes referida como a "linguagem máquina da computação paralela" pois obriga os programadores a lidar com bastantes detalhes de baixo nível. Contudo, à medida que mais e mais software onde é utilizado o MPI é desenvolvido, algoritmos mais complexos e sofisticados vão sendo incluídos em bibliotecas compatíveis, em que a sua utilização em novos programas passa simplesmente pela chamada a uma função. De facto, após um ano do fim do desenvolvimento do MPI, quase todos os construtores de sistemas paralelos incluíram uma versão nos seus sistemas. Também, por esta altura, encontram-se já disponíveis pelo menos três versões livres do MPI, e várias bibliotecas de algoritmos excelentes que foram desenvolvidas utilizando este padrão. O MPI não é nenhuma linguagem de programação nova. É simplesmente uma biblioteca de definições e funções que podem ser utilizadas em programas desenvolvidos em C e Fortran. Logo, para se compreender o MPI, é apenas necessário compreender um conjunto de funções e definições próprias.

Nesta secção, não se pretende realizar uma descrição pormenorizada do MPI ou descrever o seu funcionamento com muito detalhe, até porque os pormenores do seu funcionamento dependem do sistema paralelo onde se encontra. A intenção é apenas descrever algumas das suas principais características e do funcionamento de algumas das suas funções. Apenas descrever o suficiente para ser possível compreender o funcionamento do programa paralelo implementado neste trabalho.

3.4.1 Envio e recepção de mensagens

A versão actual do MPI apenas permite que os processos sejam criados estaticamente no início da execução de um programa paralelo. Nenhum outro processo, que utilize alguma função do MPI, pode ser criado durante a execução do programa. Desta forma, o ambiente de execução de um programa paralelo é formado por um conjunto de processos definido no início da sua execução e as comunicações entre os processos são implementadas por chamadas às funções disponíveis no MPI. Este ambiente de execução é denominado no MPI por *comunicador*. Resumidamente, um *comunicador* é uma colecção de processos que trocam mensagens entre si. Para além do comunicador geral, formado por todos os processos criados no início da execução e definido automaticamente pelo MPI, é possível criar novos comunicadores formados por subconjuntos de processos do conjunto inicial. A criação de novos comunicadores permite realizar operações de comunicações colectivas mais eficientes, estruturadas em subconjuntos de processos, nas quais participam simultaneamente todos os processos de um comunicador. Os comunicadores são um dos principais factores que diferenciam o MPI de grande parte dos sistemas de troca de mensagens. Num comunicador os processos são identificados pela sua posição entre 0 e $p-1$ dentro do grupo, onde p é o número de processos.

No MPI a troca de uma mensagem entre dois processos é realizada pela chamada às funções $MPI\_Send()$ e $MPI\_Recv()$. Como os próprios nomes das funções sugerem, a função $MPI\_Send()$ é utilizada para enviar uma mensagem a um processo e a função $MPI\_Recv()$ para receber uma mensagem. Supondo que o processo 0 pretende enviar uma mensagem ao processo 1, então, muito sucintamente, o processo 0 chama a função $MPI\_Send()$ onde especifica o conteúdo e o tamanho da mensagem e o processo a que esta se destina. Para a receber, o processo 1 tem que chamar a função $MPI\_Recv()$ indicando o endereço de memória onde deve ser guardada a mensagem, o tamanho máximo reservado para a mensagem e o processo que a envia. Uma chamada à função $MPI\_Send()$ tem que corresponder sempre a uma chamada à função $MPI\_Recv()$ no processo indicado na chamada a $MPI\_Send()$

3.4.2 Comunicações colectivas

O principal objectivo da computação paralela é dispor de vários processos a trabalharem em conjunto para a resolução de um determinado problema. Se um dos processos realizar grande parte do trabalho, não faz sentido a utilização do paralelismo, sendo preferível utilizar apenas uma máquina convencional com um único processador. Uma *operação de comunicação colectiva* é uma operação de comunicação que envolve todos

Time	Processo A	Processo B	Processo C
1	$Send(x)$	Processamento Local	Processamento Local
2	$Send(y)$	Processamento Local	Processamento Local
3	Processamento Local	$Receive(y)$	$Receive(x)$
4	Processamento Local	$Receive(x)$	$Receive(y)$

Tabela 3.2: Exemplo de uma sequência de eventos numa comunicação colectiva.

os processos que pertencem ao mesmo comunicador. Estas surgem como uma forma de melhorar a distribuição do processamento nas comunicações em que participam todos os processos.

Outro aspecto importante a realçar nas comunicações colectivas é ilustrado pelo seguinte exemplo. Imagine-se três processos A, B e C a realizarem uma comunicação colectiva, em que o processo A envia dois valores, x e y, simultaneamente para B e C como representado na tabela 3.2. Ao receber os valores, o processo B troca os valores x e y, enquanto o processo C recebe-os pela ordem correcta.

No início da programação paralela as operações de comunicações colectivas eram pontos de sincronização dos processos. Uma função de comunicação colectiva não terminava até que todos os processos tivessem realizado a operação. Logo, as chamadas às funções eram feitas pela ordem com que eram executadas. Nos sistemas paralelos actuais não existe esta restrição. Se, por exemplo, o sistema permite o armazenamento de mensagens, é possível a A enviar os dois valores e continuar o seu processamento mesmo antes dos outros dois processos iniciarem a recepção. Contudo, do ponto de vista das comunicações, o resultado tem de ser igual ao obtido no caso de haver sincronização dos processos. Desta forma, o processo B deverá continuar a receber os valores de x e y trocados.

Todas as funções de comunicações colectivas especificadas pelo MPI comportam-se, em termos dos dados transmitidos, como pontos de sincronização. As chamadas às funções de comunicações colectivas em processos distintos são feitas pela ordem com que são executadas.

3.4.2.1 Difusão de dados

A *difusão* é uma operação de comunicação colectiva onde um único processo envia os mesmos dados para todos os processos que pertencem ao mesmo comunicador.

Suponhamos que um programa paralelo está a ser executado, por exemplo, num ambiente formado por 8 processos. Como na generalidade das aplicações paralelas, depois de realizadas as tarefas básicas de inicialização, um dos processos, por exemplo o processo 0, lê os dados de entrada do programa, enquanto os restantes processos permanecem em estado de espera. Esta é uma situação indesejada mas, muitas vezes apenas um dos processos pode ler os dados de entrada. Contudo, após o processo 0

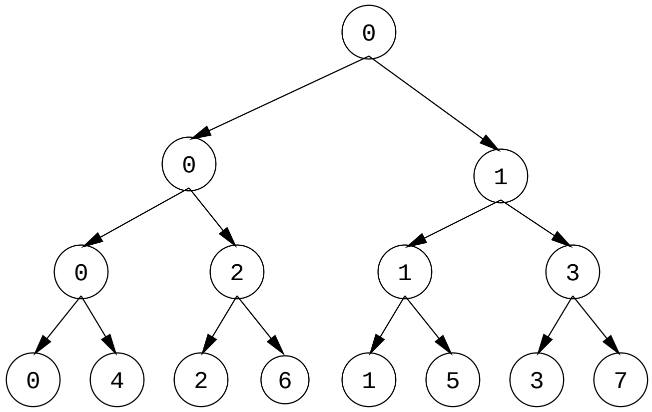


Figura 3.11: Comunicação estruturada em árvore.

ter reunido os dados de entrada estes tem de ser distribuídos pelos vários processos. A forma mais simples de distribuição dos dados é fazer com que o processo 0 envie os dados a cada um dos restantes processos. Mas, enquanto o processos 0 envia os dados a um dos processos, os restantes permanecem em estado de espera. Esta situação é também indesejada dado que a quantidade de dados a distribuir pode dar origem a tempos de comunicação elevados. Seria preferível dispor de uma forma de distribuição mais uniforme do trabalho pelos processos.

Uma outra solução para a distribuição dos dados é imaginar que se dispõe de uma árvore de processos, com o processo 0 na raiz. Durante uma primeira fase, o processo 0 envia os dados, por exemplo, para o processo 1. Na fase seguinte, o processo 0 envia os dados ao processo 2, enquanto que o 1 envia os dados a 3. Na última fase, o processo 0 envia os dados ao 4, enquanto que o 1 envia os dados ao 5, o 2 ao 6, e o 3 ao 7 (figura 3.11). A distribuição dos dados implementada desta forma reduz as fases de comunicação de 7 para 3.

Generalizando a solução anterior, se existirem p processos, a comunicação estruturada em árvore permite a distribuição dos dados em $Ceiling(log_2(p))$ fases, e não em $p-1$ fases. Este procedimento pode levar a ganhos significativos. Por exemplo, se forem utilizados 1024 processos a distribuição dos dados de entrada é realizada em apenas $log_2(1024) = 10$ fases de comunicação caso seja utilizado um esquema de comunicação em árvore. Desta forma, o tempo necessário para a distribuição é reduzido 100 vezes.

A difusão dos dados estruturada em árvore pode ser bastante mais eficiente do que uma difusão baseada numa sequência de envios, todos com origem no processo raiz. Contudo, os detalhes de implementação da difusão em árvore são relativamente complexos. E, se não for conhecida a topologia do sistema utilizado, não é possível ter a certeza que a solução implementada seja a mais eficiente. Logo, é preferível dispor de uma função para a difusão de valores já implementada por alguém com um conhecimento mais aprofundado do sistema utilizado. A biblioteca de funções do MPI dispõe de uma função de difusão - $MPI\_Bcast()$. Dado que o MPI apenas especifica a sintaxe da chamada da função e o resultado devolvido pela sua execução, é possível para os programadores implementar uma função "optimizada" para o sistema em questão.

3.4.2.2 Operações de redução

Muitas vezes durante a implementação de programas paralelos dá-se o caso de ser necessário realizar cálculos em que os valores se encontram distribuídos pelos vários processos. Por exemplo, consideremos um programa paralelo para realizar a soma de todos os elementos de uma lista de valores em que, a cada processo, foi atribuído um conjunto distinto de valores. Após cada processo encontrar a soma dos seus valores locais é necessário calcular a soma global das somas de cada processo. Esta "soma global" é um exemplo de uma classe geral de operações de comunicação denominadas de "operações de redução".

Numa operação de redução global, todos os processos num comunicador contribuem com dados que são processados utilizando uma operação binária. São operações binárias típicas, entre outras, a adição, o valor máximo e o valor mínimo. A função MPI para a realizar operações de redução é a *MPI_Reduce*(). Apesar de esta função de comunicação colectiva ser chamada por todos os processos que pertencem ao comunicador, o resultado da operação é guardado apenas num dos processos, o processo definido como raiz da operação.

Por exemplo, se o objectivo for simplesmente imprimir o resultado da operação, isto não apresenta problemas. Mas, se o resultado da operação for necessário para realizar outros cálculos no programa, pode ser conveniente guarda-lo em todos os processos. Uma abordagem obvia é executar a chamada a *MPI_Reduce*() seguida de uma chamada a *MPI_Bcast*(). Mas é possível utilizar uma implementação mais eficiente. Se, por exemplo, o programa paralelo para a soma dos valores de uma lista fosse executado com 8 processos, uma abordagem mais eficiente para o cálculo da soma, em que o resultado é guardado em todos os processos, poderia ser conseguida através dos seguintes passos:

1. (a) Os processos 0 e 4 trocam os seus resultados locais, os processos 1 e 5 trocam os seus, o 2 e o 6 trocam, e o 3 e o 7 trocam.
   (b) Cada processo adiciona o seu resultado com o valor recebido.
2. (a) Os processos 0 e 2 trocam os seu resultados intermédios, os processos 1 e 3 trocam os seus, o 4 e o 6 trocam, e o 5 e o 7 trocam.
   (b) Cada processo adiciona os valores.
3. (a) Os processos 1 e 3 trocam, o 2 e o 3 trocam, o 4 e o 5 trocam, e o 6 e o 7 trocam.
   (b) Cada processo adiciona.

Essencialmente, os passos anteriores descrevem uma operação de redução estruturada em árvore com raiz em cada um dos processos simultaneamente. A figura 3.12 ilustra

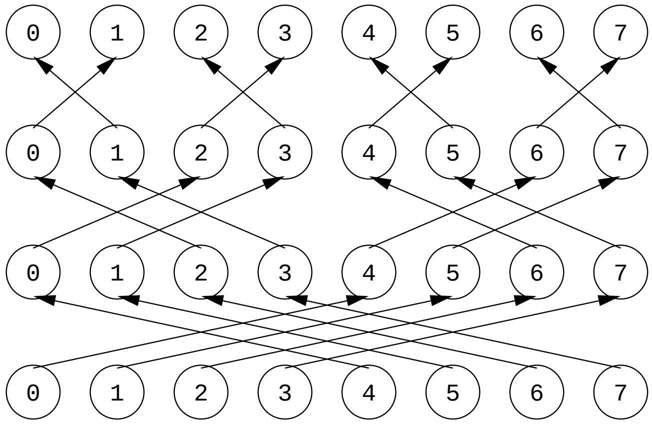


Figura 3.12: Estrutura de comunicação butterfly.

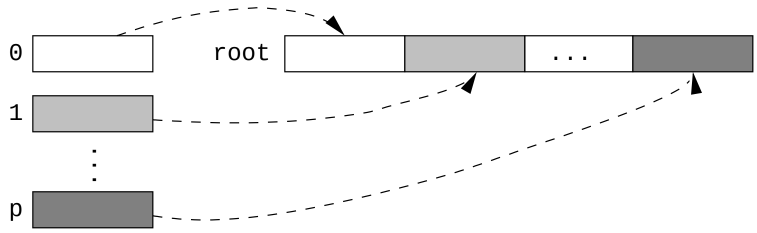


Figura 3.13: Lista de valores obtida pelo processo raiz após a chamada a $MPI\_Gather()$.

esta estrutura. Esta estrutura de comunicação é normalmente conhecida como *butterfly*. O MPI também dispõe de uma função para realizar uma operação de redução em que o resultado é devolvido em cada processo - $MPI\_Allreduce()$.

3.4.2.3 Reunião de dados

Uma outra situação bastante frequente, é a necessidade de reunir dados dos vários processos num único processo. Assim como para as outras operações, o MPI providencia uma função para reunião de valores num único processo - $MPI\_Gather()$. Esta função reúne os dados fornecidos por cada processo num dos processos especificado como raiz da comunicação. Os dados são reunidos num lista de valores no processo raiz em que as primeiras posições correspondem aos valores enviados pelo processo 0, as posições seguintes aos valores enviados pelo processo 1, e assim sucessivamente (figura 3.13).

Mas, e se for necessário reunir os valores em todos os processos? Uma abordagem possível seria chamar a função $MPI\_Gather()$ em cada processo, tantas vezes quanto o número de processos. Na primeira chamada à função os valores seriam reunidos no processo 0, na segunda chamada no processo 1 e assim sucessivamente. Contudo, uma forma mais eficiente seria utilizar o esquema de comunicação butterfly já mencionado (figura 3.12). Para esta comunicação colectiva o MPI também oferece uma função

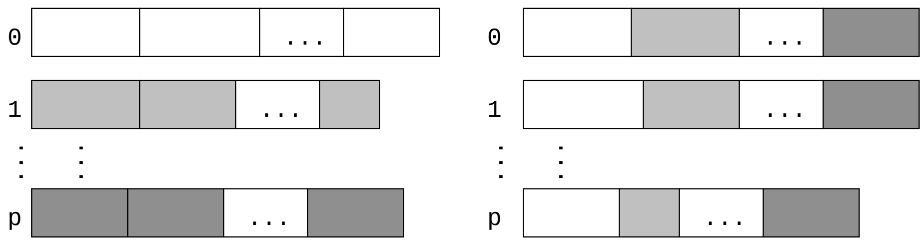


Figura 3.14: Distribuição dos dados realizada pela função *MPI_Alltoall*().

- *MPI_Allgather*() - que reúne os dados enviados por cada processo em todos os processos. Os dados recebidos são também guardados numa lista de valores em cada um dos processos de forma semelhante à função *MPI_Gather*() (figura 3.13).

3.4.2.4 Troca total de dados

Uma comunicação denominada *troca total* ou de distribuição *todos para todos*, pode ser vista como uma série de distribuições, em que cada processo envia um conjunto de dados distintos para cada um dos outros processos (figura 3.14). É uma comunicação colectiva semelhante à descrita no paragrafo anterior.

O MPI providencia uma função para realizar uma troca total denominada de *MPI_Alltoall*().

Capítulo 4

Algoritmos de construção de árvores de decisão em paralelo

A classificação tem sido identificada como um problema importante em várias áreas como a extracção do conhecimento e a aprendizagem automática. Para as tarefas de classificação é fornecido um conjunto de observações, ou exemplos, denominado de conjunto de treino, onde cada observação é descrita por um conjunto de atributos e por um identificador de uma classe. Os atributos podem ser contínuos, formados por valores ordenados, ou discretos, formados por um conjunto de valores não ordenados (normalmente pequeno). O objectivo da classificação é construir um modelo capaz de prever a classe de uma observação baseado nos seus atributos. Assim que um modelo é construído, este pode ser utilizado para identificar a classe de novas observações não classificadas. Possíveis aplicações para tarefas de classificação surgem de diversas áreas como markting de vendas, identificação de fraudes e diagnósticos médicos [MST94].

Ao longo dos anos foram propostos diversos modelos para a representação do modelo utilizado na classificação tais como redes neuronais, modelos estatísticos (como discriminantes lineares ou quadráticos), árvores de decisão, e algoritmos genéticos [MST94, Qui92]. Dentre estes modelos, as árvores de decisão são os que mais se adequam para a extracção do conhecimento e aprendizagem automática. As árvores de decisão são relativamente rápidas de construir e atingem uma precisão na classificação semelhante, ou por vezes superior, aos outros métodos de classificação [MST94, Qui92, BFOS84].

Nas áreas da extracção do conhecimento e da aprendizagem automática a quantidade de dados a ser processada pode ser bastante grande dado que a utilização de conjuntos de dados maiores pode dar origem a uma maior precisão na classificação. Desta forma, é bastante importante dispor de algoritmos de classificação computacionalmente eficientes e escalonáveis. Uma forma possível de reduzir a complexidade computacional na construção de árvores de decisão a partir de grandes conjuntos de dados, é utilizar apenas uma pequena amostra do conjunto de treino. Contudo, as árvores de decisão obtidas desta forma não apresentam o mesmo nível de precisão obtido por uma árvore de decisão construída a partir da totalidade dos dados [Cat91].

Uma outra forma possível de reduzir a quantidade de tempo utilizada na construção das árvores de decisão a partir de grandes conjuntos de dados e, desta forma, manter a precisão da classificação, é a utilização de uma abordagem paralela [DGST97, FL98, SAM96, SHKS99, AGS01].

As árvores de decisão são normalmente construídas em duas fases. Na primeira, a árvore de decisão é construída até todos os exemplos associados a um nó pertencerem à mesma classe. Na segunda fase, alguns nós são simplificados, de acordo com um determinado critério, e substituídos por folhas de forma a eliminar alguma dependência de ruído de natureza estatística ou de algum conceito presente apenas nos dados de treino. Sem qualquer dúvida, a fase computacionalmente mais dispendiosa do algoritmo é a fase de construção da árvore de decisão. À excepção dos conjuntos de treino muito pequenos, o tempo utilizado na simplificação da árvore é apenas uma pequena fracção do tempo total de execução do algoritmo [Cat91, Qui92, Rug00, SHKS99]. Desta forma, para descrição neste capítulo dos algoritmos de construção de árvores de decisão em paralelo é apenas referida a primeira fase de construção da árvore e omitida a fase de simplificação.

Como foi descrito no segundo capítulo para a construção das árvores de decisão é normalmente utilizada uma técnica de "dividir para conquistar". Este processo pode ser resumido nos passos seguintes para a construção recursiva de uma árvore de decisão a partir de um conjunto de treino T:

```
Constrói Árvore(T):
  1. Se todos os exemplos em T pertencem à mesma classe:
     1.1. Retorna uma folha.
  2. Se não:
     2.1. Para cada atributo A:
          2.1.1. Avalia a possível divisão do conjunto de exemplos baseada no
                 atributo A.
     2.2. Divide o conjunto de exemplos T baseado no melhor atributo e
          constrói o nó de decisão N.
     2.3. Para cada subconjunto Ti resultante da divisão:
          2.3.1. Filho i de N = Constrói Árvore(Ti).
     2.4. Retorna nó de decisão N.
```

O processo de construção de uma árvore de decisão apresenta uma concorrência natural. Assim que o conjunto de exemplos associado a um nó é dividido, os seus filhos podem ser criados concorrentemente (passo 2.3.1). Ainda, o processo de construção de um nó pode ser decomposto pelo processamento dos dados no conjunto de treino. Estas duas formas de decomposição do processamento resumem as abordagens para a implementação de algoritmos de construção de árvores de decisão em paralelo. Contudo, a implementação de tais algoritmos continua a ser um desafio devido fundamentalmente a duas razões [SHKS99, FL98]:

- Primeiro, a forma da árvore é bastante irregular e só é determinada em tempo de execução. Ainda, a quantidade de trabalho associada a cada nó varia e é dependente do conjunto de exemplos cobertos. Logo, qualquer alocação estática do processamento dos nós dá origem a uma má distribuição do processamento.

- Segundo, apesar dos sucessores de um nó poderem ser processados concorrentemente, estes utilizam um subconjunto dos exemplos cobertos pelo nó pai. Se os exemplos são distribuídos dinamicamente e atribuídos a diferentes processos, responsáveis por diferentes nós, então esta distribuição obriga à implementação de mecanismos de troca de dados. Se os exemplos não forem distribuídos de forma adequada, o desempenho do algoritmo pode ser prejudicado devido a uma má localização dos dados.

Pretende-se neste capítulo fornecer uma visão geral de algumas estratégias para a implementação de algoritmos de construção de árvores de decisão em paralelo, onde é utilizado paralelismo de controlo, paralelismo nos dados e paralelismo híbrido. No paralelismo de controlo, diferentes nós da árvore de decisão são construídos por processos diferentes enquanto que numa abordagem ao paralelismo nos dados, todos os processos participam da construção do mesmo nó da árvore mas utilizando partes dos dados distintas. Numa abordagem ao paralelismo híbrido estas duas estratégias são combinadas de forma a tirar partido das vantagens oferecidas por cada uma.

As estratégias apresentadas para a implementação de algoritmos de construção de árvores de decisão em paralelo apresentadas assumem um ambiente de execução de memória distribuída em que cada processo dispõe da sua própria memória privada. Existem também outras implementações de algoritmos de construção de árvores de decisão em paralelo desenhados para sistemas de memória partilhada [KR88, ZnRA99], mas que não se enquadram dentro do objectivo deste trabalho. Apesar do modelo para a execução dos algoritmos ser um sistema de memória distribuída estes algoritmos podem também ser implementados em sistemas de memória partilhada, por exemplo, através da utilização de bibliotecas como o MPI que tornam as aplicações compatíveis com diversas arquitecturas paralelas.

4.1 Paralelismo de controlo

A construção de árvores de decisão em paralelo para qual é utilizada uma abordagem ao paralelismo de controlo é caracterizada por uma distribuição dinâmica dos nós da árvore pelos processos. De uma forma geral, um dos processos inicia a construção da árvore a partir do conjunto de treino inicial. Este processo continua a exploração da árvore de decisão até o número de nós ainda por explorar igualar o número de processos disponíveis. Neste ponto os nós são distribuídos pelos processos em que, cada processo fica responsável pela exploração da sub-árvore com raiz no nó que lhe foi atribuído (figura 4.1). Mas, para um determinado processo continuar a construção da árvore de

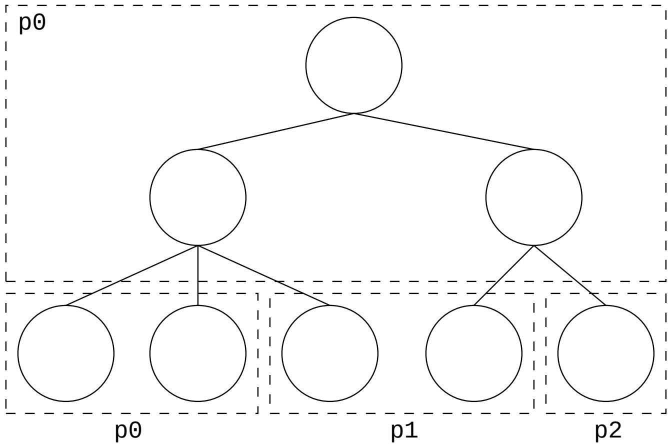


Figura 4.1: Distribuição dos nós pelos processos no paralelismo de controlo.

decisão a partir do nó que lhe foi atribuído, este necessita de conhecer o conjunto de exemplos coberto pelo nó. Levanta-se então a questão da localização dos dados. Isto é, qual deverá ser a melhor forma de distribuir o conjunto de treino pelos vários processos para que estes tenham acesso aos exemplos cobertos pelos seus nós. Se, por exemplo, o conjunto de treino for distribuído de uma forma dinâmica em que, a um determinado processo são distribuídos os exemplos cobertos por um nó atribuído a outro processo, então esta distribuição vai originar elevados custos de comunicação. A distribuição do conjunto de treino é um factor crítico para o desempenho dos algoritmos de construção de árvores de decisão em paralelo tanto para o paralelismo de controlo como para o paralelismo nos dados.

Em [DGST97], são descritas três estratégias para a implementação do algoritmo de construção da árvore de decisão do C4.5 em paralelo utilizando uma abordagem ao paralelismo de controlo. As abordagens descritas diferem entre si essencialmente na forma como os dados são distribuídos pelos processos.

Na primeira estratégia apresentada, cada processo dispõe de uma cópia da totalidade do conjunto de treino na sua memória. Desta forma, quando os nós são distribuídos pelos processos a única comunicação necessária é a indicação pelo processo que iniciou a construção da árvore do primeiro e do último exemplo cobertos por cada nó. Contudo, durante a construção da árvore de decisão a ordem dos exemplos no conjunto de treino é alterada quando este é dividido para a construção de novos nós. Sempre que a ordem dos exemplos é alterada pelo processo que inicia a construção, a nova ordem é comunicada a todos os outros processos de forma a manter o conjunto de exemplos coerente na memória de cada um.

A principal desvantagem desta estratégia é a de que cada processo guardar na sua memória a totalidade do conjunto de treino, limitando a quantidade de exemplos que podem ser utilizados na construção da árvore. Contudo, dado que cada processo tem acesso à totalidade do conjunto de treino, durante a construção da árvore de decisão em paralelo não é necessário existir qualquer comunicação entre os processos, possibilitando um bom desempenho do algoritmo. As comunicações são apenas necessárias antes da distribuição dos nós, para indicar a ordem dos exemplos no conjunto de treino, e durante a distribuição dos nós.

Na segunda e na terceira estratégia apresentadas em [DGST97] o conjunto de treino é distribuído pelos processos, ao contrário de ser replicado na memória de cada um. Quando um nó é atribuído a um processo, o conjunto de exemplos associados ao nó é enviado pelo processo que inicia a construção da árvore. Estas duas estratégias diferem entre si na forma como são distribuídos os valores dos atributos contínuos, dado que o ponto de divisão é escolhido de entre todos os valores do atributo, quando o conjunto de exemplos num nó é dividido por um atributo contínuo.

Na segunda estratégia apresentada, quando um nó é atribuído a um processo, a totalidade dos valores dos atributos contínuos são copiados para o processo que recebe o nó. Esta estratégia apresenta a vantagem em relação à anterior de não ser necessário manter uma cópia da totalidade do conjunto de treino na memória de cada processo. Também durante a fase inicial da construção da árvore não é necessário existir qualquer comunicação entre os processos pois os restantes processos não contém nenhum conjunto de exemplos que deve ser mantido. Contudo, durante a distribuição dos nós o volume de comunicações é bastante maior dado que os subconjuntos cobertos pelos nós e os restantes valores dos atributos contínuos têm de ser copiados para a memória de cada processo. Numa situação extrema, se todos os atributos forem contínuos, esta estratégia é bastante idêntica à anterior mas com a desvantagem acrescida de, durante a distribuição dos nós, todo o conjunto de treino ser enviado para cada processo.

Por fim, na terceira estratégia apresentada em [DGST97], os valores dos atributos contínuos são distribuídos de forma igual aos atributos discretos. Quando os nós são distribuídos pelos processos, apenas os subconjuntos de exemplos cobertos pelos nós, são enviados. Contudo, durante a construção da árvore de decisão em paralelo, sempre que um atributo contínuo é escolhido para a divisão do conjunto de exemplos associados a um nó, o processo responsável por esse nó é obrigado a comunicar com os outros processos para encontrar o ponto de divisão.

Uma outra desvantagem comum às três estratégias anteriores é a do paralelismo não ser explorado logo no início da construção da árvore de decisão quando é mais necessário. A construção da árvore em paralelo só é iniciada assim que o número de nós iguala o número de processos disponíveis.

Uma outra implementação de um algoritmo de construção de árvores de decisão em paralelo com uma abordagem ao paralelismo de controlo é apresentada em [FL98]. Nesta implementação o paralelismo é explorado logo no início da construção da árvore. Assim que novos nós são criados, estes são imediatamente atribuídos a outros processos (figura 4.2), explorando o paralelismo logo no início da construção da árvore de decisão quando é mais necessário.

Apesar da construção de árvores de decisão em paralelo com uma abordagem ao paralelismo de controlo ser bastante simples e se enquadrar com a concorrência natural

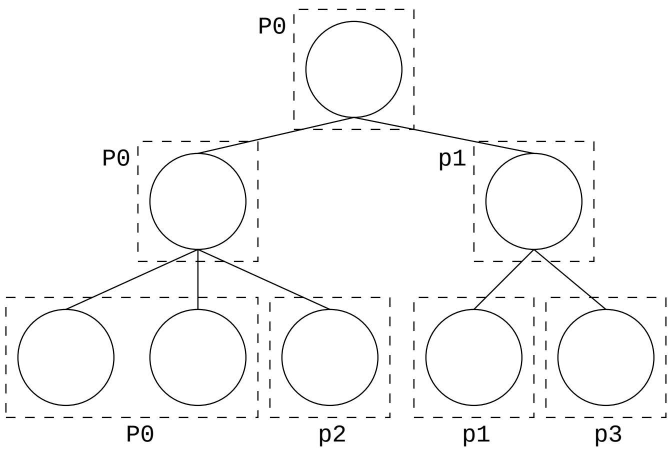


Figura 4.2: Distribuição dos nós na implementação descrita em [FL98].

do algoritmo, esta sofre de um grave problema de balanceamento da carga. O mau balanceamento da carga pelos processos deve-se principalmente à estrutura bastante irregular das árvores de decisão que só é determinada durante a execução do algoritmo (figura 4.3). Isto é, a exploração de alguns nós pode resultar em árvores de decisão pequenas enquanto que outros podem dar origem a árvores bastante grandes. Esta diferença de tamanho das árvores dá origem a que alguns processos terminem a construção das suas árvores mais rapidamente e permaneçam em estado de espera até que todos os processos terminem a construção das suas árvores, contribuindo, desta forma, para um pior desempenho do algoritmo.

4.2 Paralelismo nos dados

Assim como para o paralelismo de controlo, a utilização de paralelismo nos dados para o desenho de algoritmos de construção de árvores de decisão pode ser descrita genericamente como a execução do mesmo conjunto de instruções (algoritmo) por todos os processos. Contudo, nesta abordagem ao paralelismo todos os processos cooperam para construção dos mesmos nós da árvore de decisão, um nó de cada vez. O paralelismo é conseguido pela distribuição do conjunto de treino pelos processos onde, cada processo, é responsável por um subconjunto de exemplos distinto do conjunto de treino. Isto é, a construção de um nó da árvore de decisão é realizada simultaneamente por todos os processos em que, cada processo, processa uma parte distinta dos dados associados ao nó.

A distribuição do conjunto de treino pode ser realizada de duas formas diferentes: verticalmente, em que cada processo fica responsável por um grupo distinto de atributos, ou horizontalmente, em que um subconjunto distinto de exemplos é atribuído a cada processo (figuras 4.4 e 4.5 respectivamente).

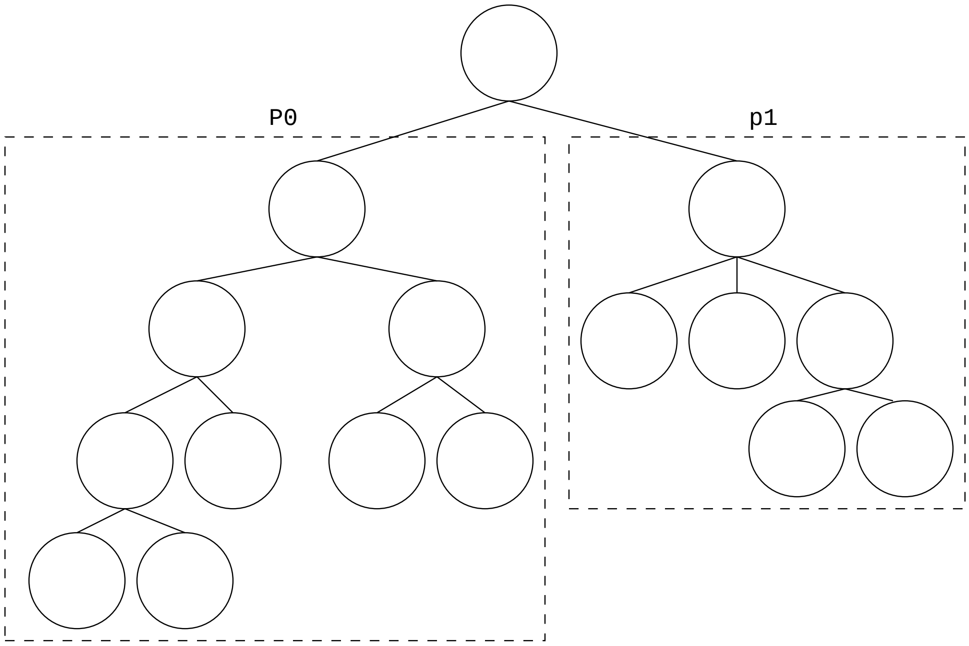


Figura 4.3: Abordagem ao paralelismo de controlo na construção de árvores de decisão.

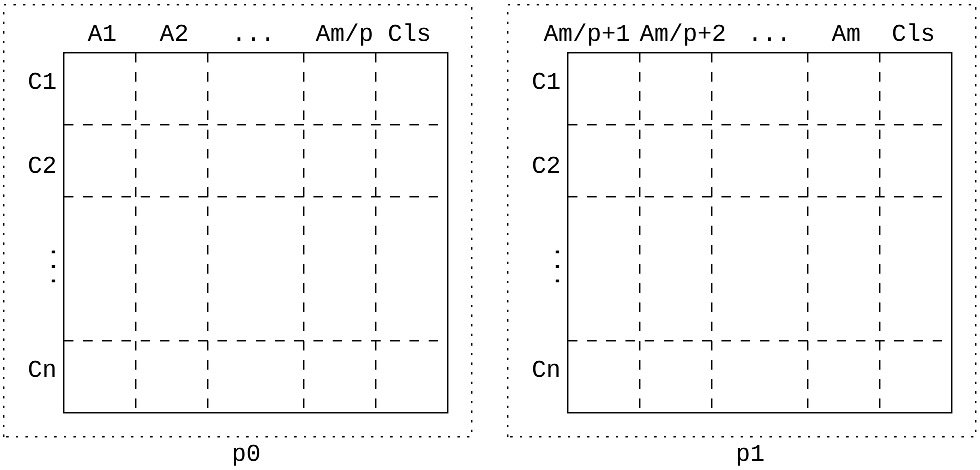
4.2.1 Distribuição vertical dos dados

De uma forma geral, a abordagem ao paralelismo com distribuição vertical dos dados consiste em distribuir o conjunto de treino verticalmente, atribuindo a cada processo um conjunto distinto de atributos (figura 4.4). Cada processo guarda na sua memória apenas os valores dos atributos que lhe foram atribuídos e os valores das classes que, embora sejam repetidos na memória de cada processo, evita um grande volume de comunicações entre os processos dado que esta informação é constantemente necessária durante a construção da árvore.

Embora todos os processos executem, mais ou menos, o mesmo algoritmo para a construção de um nó da árvore de decisão, cada processo é apenas responsável pela a avaliação das possíveis divisões do conjunto de exemplos para o seu conjunto de atributos. Após a avaliação de todos os atributos, os processos comunicam entre si para encontrarem a melhor divisão.

Para a divisão do conjunto de exemplos cobertos pelo nó, o processo responsável pelo atributo escolhido, envia aos outros processos a atribuição dos exemplos aos novos nós para que estes possam dividir as suas listas de atributos. Cada processo fica então responsável por dividir localmente o seu conjunto de atributos.

A distribuição vertical do conjunto de treino e, de uma forma geral, as abordagens ao paralelismo nos dados permitem construir árvores de decisão a partir de conjuntos de treino maiores, pois este encontra-se distribuído pela memória dos vários processos. E, como a árvore é construída de forma síncrona, isto é, os nós da árvore são construídos um de cada vez por todos os processos, esta abordagem não sofre do problema de mau



Ai – Atributo i de 1 a m
Ci – Caso i de 1 a n
Cls – Classes

Figura 4.4: Paralelismo nos dados com distribuição vertical.

balanceamento da carga causado pela estrutura irregular das árvores de decisão.

Contudo, para a construção da árvore é necessário um maior volume de comunicações quer durante a avaliação dos atributos quer na divisão do conjunto de exemplos associados a um nó. Além disso, o paralelismo com distribuição vertical dos dados pode levar a uma má distribuição do processamento devido à avaliação dos atributos contínuos exigir um maior processamento que a avaliação dos atributos discretos. Se os atributos não são distribuídos de forma adequada pelos processos então, durante a avaliação das possíveis divisões, alguns processos podem terminar a avaliação dos seus atributos mais cedo que os outros e permanecer em estado de espera até todos os atributos terem sido avaliados, prejudicando o desempenho do algoritmo.

Em [FL98] são descritos dois algoritmos de construção de árvores de decisão em paralelo com distribuição vertical do conjunto de treino onde são utilizadas duas abordagens diferentes para evitar o mau balanceamento da carga devido à avaliação dos atributos contínuos.

No primeiro algoritmo a má distribuição do processamento é evitada pela utilização de uma representação binária para os valores dos atributos. Os atributos com valores não binários são convertidos numa fase de pré-processamento. Esta representação dos valores dos atributos permite a utilização de operações binárias muito rápidas para a avaliação das divisões e torna o tempo necessário para a avaliação dos atributos discretos e contínuos aproximadamente o mesmo, o que reduz substancialmente o problema da distribuição do processamento. Contudo, a representação binária para os valores dos atributos pode dar origem a árvores de decisão com menor precisão na classificação.

Como foi referido no segundo capítulo durante a descrição do algoritmo de construção de árvores de decisão do C4.5, um dos pontos críticos para o desempenho do algoritmo é precisamente a avaliação dos atributos contínuos. Para a avaliação de um atributo contínuo o conjunto de exemplos associados a um nó é ordenado pelos valores do atributo. É a operação de ordenação dos exemplos que torna a avaliação dos atributos contínuos numa operação tão dispendiosa [Rug00].

No segundo algoritmo com distribuição vertical do conjunto de treino descrito em [FL98], os exemplos associados a um nó não são ordenados durante a avaliação dos atributos contínuos, mas os valores dos atributos contínuos são agrupados pelos valores das classes. O algoritmo encontra a melhor divisão baseado nestes grupos. A avaliação realizada desta forma introduz uma aproximação que pode levar a uma perda de precisão na classificação. Mas, também torna o processamento necessário para a avaliação dos atributos contínuos muito semelhante ao necessário para os atributos discretos evitando desta forma uma má distribuição do processamento.

Uma outra forma para evitar a má distribuição do processamento devido à diferença de processamento entre a avaliação dos atributos contínuos e discretos é distribuir os atributos contínuos em igual número pelos processos. Desta forma, cada processo realiza aproximadamente as mesmas operações na construção de um nó da árvore de decisão. Esta é uma solução bastante simples e que não introduz nenhuma aproximação na avaliação dos atributos e, desta forma, não existe perda de precisão na classificação [Cha97, SHKS99].

As abordagens ao paralelismo com distribuição vertical dos dados apresentam as vantagens de uma boa distribuição do processamento e a capacidade de construir árvores de decisão a partir de conjuntos de dados maiores. Contudo, estas abordagens não escalonam bem com o aumento do número de processos [Cha97, SHKS99]. Esta situação deve-se a que, com o aumento do número de processos, o número de atributos em cada processo diminui. Como, normalmente, o número de atributos não é significativamente grande quando comparado com o número de processos, facilmente é atingido o ponto em que cada processo contém um número pequeno de atributos. Mesmo com um aumento significativo do número de exemplos, neste ponto o tempo utilizado nas comunicações sobrepõe-se ao tempo de processamento contribuindo, desta forma, para um pior desempenho.

4.2.2 Distribuição horizontal dos dados

A abordagem à construção de árvores de decisão em paralelo com distribuição horizontal dos dados consiste em atribuir a cada processo um subconjunto de exemplos distinto do conjunto de treino (figura 4.5). Assim como na distribuição vertical dos dados, cada processo apenas guarda em memória o subconjunto de exemplos atribuído.

De uma forma genérica, para a construção de um nó da árvore de decisão, os vários processos avaliam localmente os seus subconjuntos de exemplos e comunicam entre si para encontrar a melhor divisão "global" dos exemplos para o nó. Após encontrada a

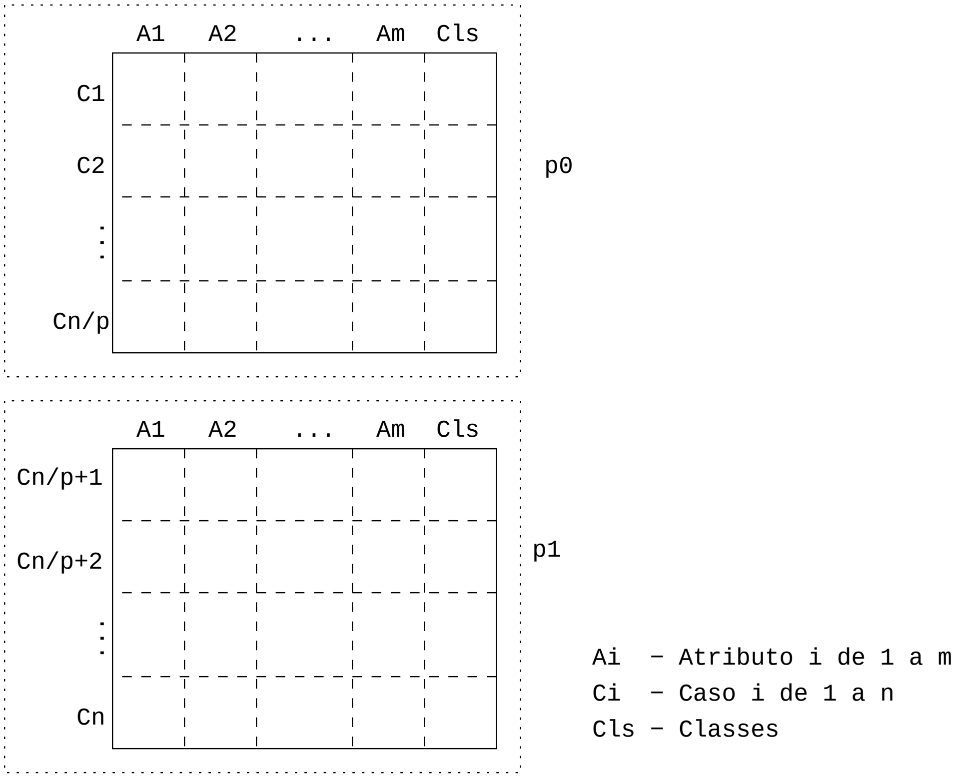


Figura 4.5: Paralelismo nos dados com distribuição horizontal.

melhor divisão, cada processo fica responsável por dividir localmente o seu subconjunto de exemplos.

À primeira vista, esta abordagem para construção de árvores de decisão em paralelo parece bastante simples. Mas, a avaliação dos atributo contínuos e, em particular, a ordenação dos exemplos realizada cada vez que um atributo contínuo é avaliado, é uma das operações que mais influencia o desenho de um algoritmo paralelo com distribuição horizontal dos dados.

Uma abordagem simples, semelhante à do algoritmo série, seria realizar uma ordenação paralela dos exemplos sempre que um atributo contínuo é avaliado em cada nó da árvore de decisão. Após a ordenação paralela dos exemplos, cada processo pode encontrar o melhor ponto de divisão local para o seu subconjunto de exemplos. O ponto de divisão escolhido para o conjunto de exemplos cobertos pelo nó é o melhor ponto de divisão encontrado localmente por cada um dos processos. Contudo a ordenação global dos exemplos requer um volume demasiado elevado de comunicações entre os processos comprometendo o desempenho de qualquer algoritmo que utilize uma tal abordagem [SHKS99, Kuf97, SAM96].

Em [SAM96], são descritas as implementações de dois algoritmos para a construção de árvores de decisão em paralelo com distribuição horizontal dos dados: os sistemas SLIQ e SPRINT. Embora nestes sistemas seja utilizado um critério de avaliação das

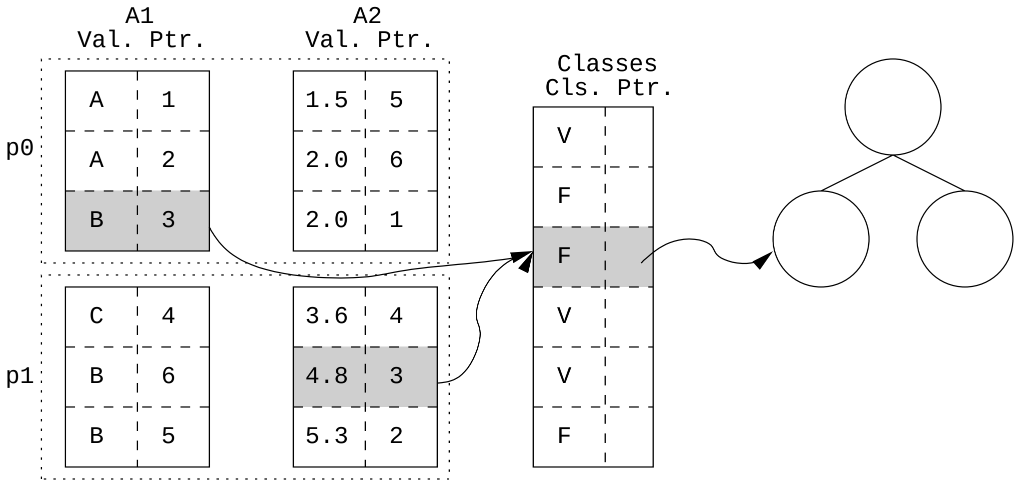


Figura 4.6: Listas de atributos e de classes utilizadas no SLIQ.

divisões diferente do sistema C4.5, o *Gini Index*, o algoritmo para a construção das árvores de decisão é bastante semelhante. Também nestes sistemas, para a avaliação dos atributos contínuos é necessário que os seus valores se encontrem ordenados. De forma a evitar uma ordenação paralela dos exemplos sempre que um atributo contínuo é avaliado, nos dois algoritmos são utilizadas listas separadas de valores para cada atributo que são ordenadas apenas uma vez no início do algoritmo de construção da árvore de decisão.

No sistema SLIQ os atributos são guardados em listas separadas de valores. A utilização de listas separadas para os valores dos atributos permite que a ordem dos valores numa lista de um atributo seja independente da ordem dos valores na lista de um outro atributo. Desta forma os valores dos atributos contínuos podem ser ordenados globalmente apenas uma única vez no início da construção da árvore de decisão e mantidos ordenados durante a execução do algoritmo. O SLIQ utiliza ainda uma outra lista de valores, denominada de lista de classes, que contém os valores da classe para cada exemplo. Cada entrada nas listas de atributos consiste no valor do atributo e de um apontador para uma entrada na lista de classes que identifica a classe do exemplo correspondente. As entradas da lista de classes, para além do valor da classe, contêm também um apontador para o nó da árvore ao qual o exemplo se encontra atribuído (figura 4.6).

Na implementação paralela do SLIQ, e após a distribuição do conjunto de treino, cada processo fica responsável pela criação das suas listas locais de atributos. As listas dos atributos discretos não necessitam de mais nenhum processamento adicional. Contudo, as listas dos atributos contínuos têm de ser ordenadas o que acontece apenas uma vez antes do início da construção da árvore de decisão.

As listas dos atributos discretos contêm apenas os valores dos exemplos que foram inicialmente atribuídos a cada processo. Contudo, devido à ordenação paralela, as listas dos atributos contínuos num determinado processo podem conter valores de exemplos atribuídos a outro processo. Para ser possível conhecer os valores das classes correspondentes a estes exemplos em [SAM96] foram implementadas duas estratégias para o tratamento da lista de classes:

- Numa primeira abordagem a lista de classes é dividida pelos processos da mesma forma que os atributos. Isto é, em cada processo a lista de classes apenas contém os valores das classes correspondestes aos exemplos que lhes foram atribuídos inicialmente. Os processos comunicam entre si para conhecerem a classe de um exemplo que não se encontra na sua lista de classes local. Mas, como o valor da classe é necessário constantemente durante a construção da árvore de decisão, esta abordagem apresenta elevados custos de comunicações piorando o desempenho do algoritmo.

- Na segunda abordagem, em cada processo a lista de classes contém os valores das classes para todo o conjunto de treino. Isto é, a lista de classes é inteiramente replicada na memória de cada processo.

No SLIQ a avaliação dos atributos não é muito diferente da avaliação realizada no algoritmo do C4.5, à excepção do critério utilizado. Para a avaliação dos atributos discretos cada processo percorre a lista de valores do atributo e reúne as distribuições necessárias para a sua avaliação. A seguir, os vários processos enviam as suas distribuições para um único processo que as reune e encontra o valor "global" da avaliação. A avaliação dos atributos contínuos também é bastante semelhante à realizada pelo C4.5. Cada processo percorre a sua lista local de valores do atributo avaliando cada possível ponto de divisão. No final os processos enviam para um dos processos os seus melhores pontos de divisão locais o qual escolhe o melhor ponto de divisão "global" para o atributo. Contudo, antes de iniciar a avaliação do atributo cada processo tem de conhecer as distribuições dos valores do atributo pelas classes nos outros processos. As distribuições locais dos valores dos atributos contínuos pelas classes são partilhadas por todos os processos antes do início da avaliação dos atributos.

Por fim, após escolhida a melhor divisão do conjunto de exemplos cada processo é responsável pela divisão das suas listas. A divisão do conjunto de exemplos associados a um nó é realizada simplesmente actualizando o apontador para o nó da árvore correspondente na lista de classes. Dado que a lista de classes deve permanecer consistente na memória de cada processo, a atribuição dos exemplos aos nós resultantes da divisão é partilhada por todos os processos que actualizam as suas listas de classes locais.

O facto de, no SLIQ, a lista de classes ser replicada na memória de cada processo, pode limitar o tamanho do conjunto de treino utilizado para a construção da árvore.

O segundo algoritmo apresentado em [SAM96], o sistema SPRINT, foi desenhado de forma a evitar a utilização de estruturas de dados centralizadas, como a lista de classes no SLIQ. Este algoritmo também utiliza listas separadas para os valores dos atributos (figura 4.7). A lista de classes é eliminada pela inclusão do valor da classe nas listas

	A1			A2		
	Val.	Cls.	Ind.	Val.	Cls.	Ind.
p0	A	V	1	1.5	V	5
	A	F	2	2.0	F	6
	B	F	3	2.0	V	1
p1	C	V	4	3.6	V	4
	B	V	6	4.8	F	3
	B	F	5	5.3	F	2

Figura 4.7: Listas de atributos utilizadas no SPRINT.

de atributos. Também o apontador para a lista de classes nas listas de atributos do SLIQ é substituindo pelo o índice do exemplo no conjunto de treino. Desta forma, as listas de atributos são completamente independentes e a única forma possível de correlacionar os valores dos atributos é pelo valor do índice do exemplo em cada uma das listas.

No SPRINT a avaliação dos atributos é realizada exactamente da mesma forma que no SLIQ. Para os atributos discretos, cada processo encontra localmente as distribuições dos exemplos pelas classes que são depois reunidas num único processo para o cálculo da avaliação. Na avaliação dos atributos contínuos os processos também encontram o melhor ponto de divisão local que é enviado posteriormente para um único processo que escolhe o melhor ponto de divisão "global". Antes da avaliação dos atributos contínuos é também necessário que os processos troquem entre si as distribuições dos valores dos atributos contínuos pelas classes.

Após encontrada a melhor divisão, a divisão da lista de valores do atributo escolhido é imediata. Cada processo percorre a sua lista e, dependendo do valor do atributo, atribuí a entrada a um dos nós resultantes da divisão. Como não existe nenhuma estrutura adicional para determinar a que nó é que cada exemplo pertence (como a lista de classes no SLIQ), durante a divisão da lista do atributo escolhido cada processo reune numa tabela de *Hash* os índices dos exemplos atribuídos a cada nó. Esta tabela é depois partilhada por todos os processos e utilizada por cada um na divisão das restantes listas de atributos.

Em [Kuf97] é apresentado um algoritmo de construção de árvores de decisão em paralelo em que é utilizada uma abordagem ao paralelismo com distribuição horizontal dos dados diferente da adoptada nos sistemas anteriores. Neste algoritmo os atributos não são separados em listas de valores independentes nem é realizada nenhuma ordenação global para os valores dos atributos contínuos.

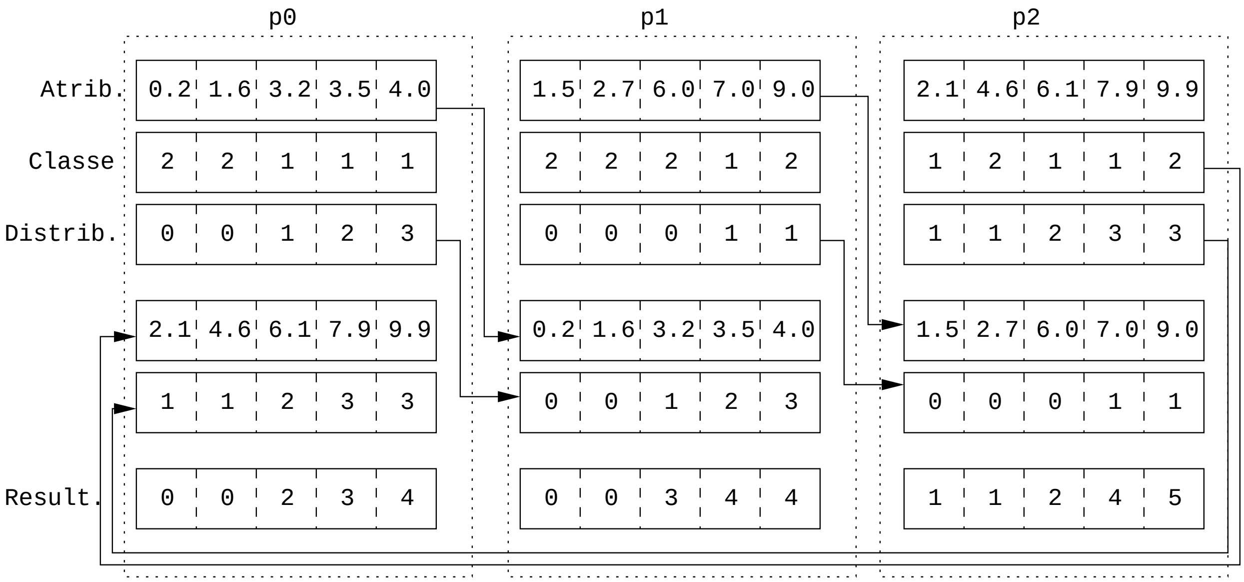


Figura 4.8: Exemplo da segunda fase de avaliação dos atributos contínuos no algoritmo apresentado em [Kuf97].

Dado que os valores dos atributos contínuos não são ordenados globalmente, a avaliação deste atributos é um pouco diferente. Esta é implementada em três fases:

1. Cada processo ordena localmente os seus exemplos pelos valores do atributo e encontra as distribuições dos exemplos para cada ponto de divisão do seu conjunto de exemplos.

2. As distribuições encontradas na fase anterior, juntamente com os valores do atributo, são trocadas entre processos vizinhos $p-1$ vezes (em que p é igual ao número de processos). Sempre que um processo recebe as distribuições de um processo vizinho este junta-as às suas, de forma a que, após as $p-1$ trocas, todos os processos conheçam as distribuições globais necessárias para o cálculo da avaliação dos ponto de divisão (figura 4.8).

3. Após cada processo encontrar a melhor divisão local do seu conjunto de exemplos, os vários pontos de divisão são enviadas para um único processo que escolhe a melhor divisão "global" do conjunto de exemplos.

A avaliação dos atributos discretos não levanta quaisquer problemas. À semelhança dos outros algoritmos, cada processo reune as distribuições locais dos valores pelas classes que são reunidas num dos processos para encontrar o valor "global" da avaliação.

Após a avaliação de todos os atributos e escolhida a melhor divisão do conjunto de exemplos, esta é realizada da mesma forma que no algoritmo sequencial, embora neste caso, cada processo é responsável pela divisão dos seu subconjunto de exemplos.

Em geral as abordagens à construção de árvores de decisão com distribuição horizontal dos dados apresentam bons resultados tanto em desempenho como em escalonabilidade, sendo também capazes de processar conjuntos de dados bastante grandes. Contudo, estas abordagens apresentam custos de comunicações bastante elevados, principalmente durante a divisão dos exemplos ou, no algoritmo algoritmo apresentado em [Kuf97], durante a avaliação dos atributos contínuos [SHKS99, Kuf97, FL98].

4.3 Paralelismo Híbrido

Os algoritmos de construção de árvores de decisão com uma abordagem ao paralelismo híbrido podem ser caracterizados por utilizarem ambas as abordagens ao paralelismo nos dados, com distribuição vertical ou horizontal, e ao paralelismo de controlo. A implementação de algoritmos de construção de árvores de decisão com uma abordagem ao paralelismo híbrido é fortemente motivada pelo compromisso entre a distribuição do processamento em cada nó da árvore e o volume de comunicações necessárias. Os nós próximos da raiz, que cobrem um número significativo de exemplos, são construídos utilizando uma abordagem ao paralelismo nos dados, com distribuição vertical ou horizontal, de forma a evitar os problemas já referidos da má distribuição do processamento e do mau aproveitamento do paralelismo associados ao paralelismo de controlo (figura 4.9). Mas, para os nós que cobrem um número pequeno de exemplos, o overhead associado às comunicações pode sobrepor-se ao tempo utilizado no processamento dos exemplos nos nós. Para evitar uma perda de desempenho devido ao overhead das comunicações, quando o número de exemplos cobertos por um nó é inferior a um determinado valor, um dos processos continua sozinho a construção da sub-árvore com raiz nesse nó (paralelismo de controlo).

Em [SHKS99] é apresentado um outro algoritmo de construção de árvores de decisão em paralelo para o qual é utilizada uma abordagem ao paralelismo híbrida. Este algoritmo inicia a construção da árvore de decisão com uma abordagem ao paralelismo nos dados com distribuição horizontal. Como já foi referido, nesta abordagem cada nó da árvore de decisão é construído, um de cada vez, por todos os processos. Estes trocam entre si as suas distribuições dos exemplos locais para o cálculo dos valores "globais" da avaliação dos atributos. Os passos realizados por todos os processos na construção de um nó da árvore são descritos a seguir de uma forma bastante compacta:

1. Escolhe um nó da árvore para explorar de acordo com uma regra de construção da árvore (por exemplo, primeiro em largura ou primeiro em profundidade). No início é sempre seleccionada a raiz da árvore.

2. Reune, para cada atributo, as distribuições locais dos exemplos pelas classes.

3. Troca as distribuições encontradas entre os processos através de uma operação de redução global.

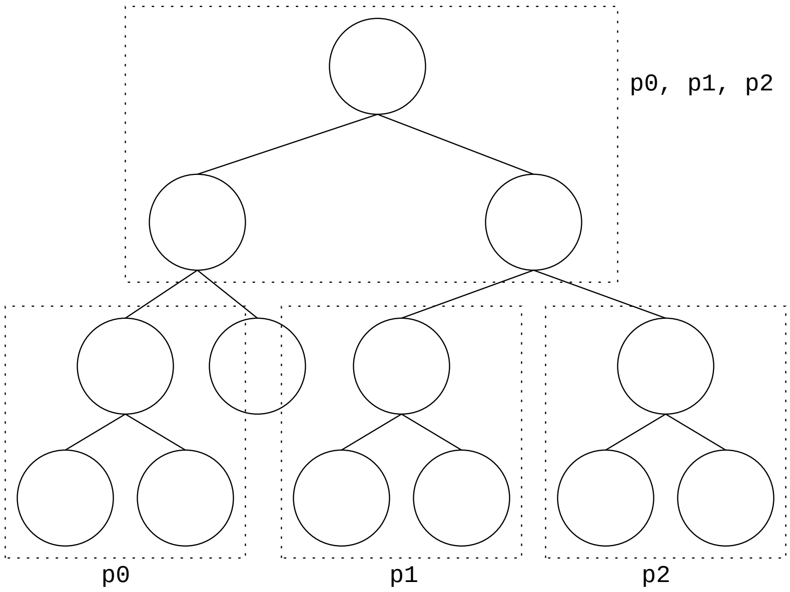


Figura 4.9: Paralelismo híbrido.

4. Calcula a avaliação de cada atributo simultaneamente em cada processo e selecciona a melhor divisão dos exemplos.

5. Cria os novos nós e divide localmente os exemplos associados ao nó pelos nós filho.

6. Repete os passos anteriores até não existir mais nenhum nó por explorar.

Como já foi referido diversas vezes, a construção da árvore de decisão em paralelo com distribuição horizontal dos dados necessita de um grande volume de comunicações para a construção de um nó. Neste algoritmo a árvore de decisão é construída em paralelo com distribuição horizontal dos exemplos enquanto o custo associado às comunicações não for demasiado elevado. Quando o custo das comunicações atinge um determinado limite a construção da árvore de decisão é realizada adoptando uma abordagem ao paralelismo de controlo.

A implementação do paralelismo de controlo utilizada neste algoritmo é um pouco diferente da descrita no início deste capítulo. Muito sucintamente, neste ponto os processos são divididos em dois grupos (figura 4.10). Os nós da árvore ainda por explorar são distribuídos pelos grupos de processos de forma a que, em cada grupo, o número de exemplos cobertos pelos nós seja aproximadamente o mesmo. Após a criação dos dois grupos de processos a construção dos nós dentro de cada grupo é novamente realizada adoptando a mesma abordagem ao paralelismo nos dados (com distribuição horizontal) apresentada. Contudo a exploração dos nós num dos grupos é completamente independente da exploração dos nós no outro grupo (paralelismo de controlo). Este processo de divisão da construção da árvore por grupos de processos é repetido sempre que, dentro de um determinado grupo, o custo associado às co-

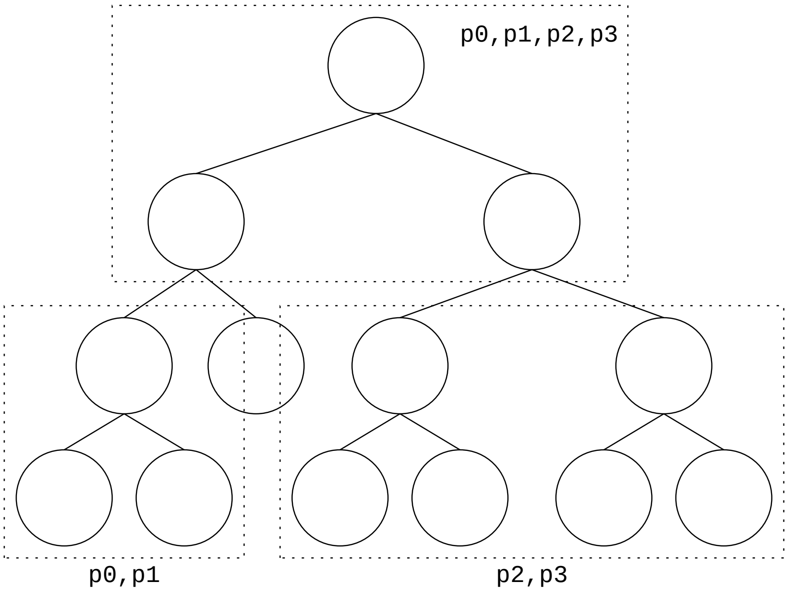


Figura 4.10: Abordagem ao paralelismo híbrido utilizada em [SHKS99].

municações na exploração dos nós com paralelismo nos dados atinge um determinado limite.

Para evitar uma má distribuição do processamento devido à estrutura irregular das árvores de decisão típica do paralelismo de controlo, neste algoritmo, quando um grupo de processos termina a exploração de todos os seus nós, junta-se a um outro grupo com nós ainda por explorar e com igual número de processos. Os processos que se juntam ao grupo recebem metade dos exemplos cobertos pelos nós ainda não explorados.

Um elemento chave deste algoritmo e, em geral, de qualquer implementação híbrida é o critério utilizado para decidir quando deve ser realizada a divisão de um conjunto de processos, isto é, quando se deve passar do paralelismo nos dados para o paralelismo de controlo. O desempenho de uma implementação híbrida depende fortemente deste critério. Neste algoritmo em particular, se a divisão é realizada muitas vezes, então a abordagem híbrida ao paralelismo aproxima-se da construção da árvore com uma abordagem ao paralelismo de controlo, o que dá origem a grandes custos de transferência de dados entre os processos. Se, pelo contrário, a divisão é realizada muito tarde, então o algoritmo vai apresentar custos de comunicações bastante elevados associados à construção dos nós com poucos exemplos com um abordagem ao paralelismo nos dados. Neste algoritmo a divisão dos processos em dois grupos é realizada sempre que o custo associado às comunicações iguala o custo de transferir os dados entre os processos para a distribuição dos nós pelos grupos. Ou seja, a divisão é realizada quando

$\sum$*(Custos de comunicações) >= Custo de transferência + Custo do balanceamento da carga*

De uma forma geral, num algoritmo de construção de árvores de decisão em paralelo híbrido, a passagem da construção da árvore com paralelismo nos dados para a construção com paralelismo de controlo é realizada quando o custo das comunicações começa a dominar o tempo de processamento e iguala o custo da passagem para o paralelismo de controlo.

4.4 Principais características das diferentes abordagens

Apresenta-se agora um resumo das principais características das diferentes abordagens descritas neste capítulo. Um resumo dos pontos fracos e dos pontos fortes de cada uma das abordagens. É ainda referido, muito sucintamente, a forma como algumas das estratégias apresentadas resolveram alguns dos problemas associados a cada uma das abordagens.

4.4.1 Paralelismo de controlo

De uma forma geral, os algoritmos de construção de árvores de decisão em paralelo com uma abordagem ao paralelismo de controlo são bastante simples. Grande parte dos algoritmos que se enquadram dentro desta abordagem aproveitam a concorrência natural da estratégia "dividir para conquistar" dos algoritmos de construção de árvores de decisão para a sua implementação em paralelo [SHKS99, FL98, DGST97].

Numa abordagem genérica ao paralelismo de controlo, um dos processos inicia a construção da árvore a partir do conjunto de treino inicial. Á medida que vão sendo criados novos nós, estes são distribuídos pelos processos em que cada processo fica responsável pela construção das sub-árvores com raiz nos nós que lhe foram atribuídos (figura 4.1). Como a construção de uma sub-árvore com raiz num determinado nó é independente da construção de uma outra sub-árvore com raiz num outro nó distinto, esta abordagem apresenta custos de comunicações bastante baixos. As únicas comunicações necessárias dão-se no início da construção da árvore antes da distribuição dos nós e durante a distribuição dos nós para a cópia dos exemplos associados a cada nó.

Contudo, esta abordagem ao paralelismo apresenta um mau balanceamento da carga pelos processos. A má distribuição do processamento deve-se principalmente à estrutura irregular das árvores de decisão. Como a estrutura de uma árvore de decisão só é determinada em tempo de execução, ao serem distribuídos os nós pelos processos não é possível saber o tamanho das árvores obtidas a partir de cada um dos nós. Logo, alguns processos podem terminar a construção das suas árvores mais cedo permanecendo em estado de espera até todos os processos terminarem a construção das suas árvores contribuindo, desta forma, para um pior desempenho do algoritmo [SHKS99, FL98, DGST97].

Outra desvantagem apresentada por esta abordagem é a de ser necessário manter uma cópia do conjunto de treino na memória de cada processo ou, pelo menos, todo conjunto de treino deve estar residente na memória do processo que inicia a construção da árvore, o que limita o número de exemplos que é possível utilizar para a construção da árvore. Como alternativa, o conjunto de treino pode ser distribuído pela memória dos processos. Contudo, esta solução apresenta um elevado volume de comunicações necessário para a partilha dos dados guardados na memória de cada processo [DGST97, FL98].

4.4.2 Paralelismo nos dados

A abordagem ao paralelismo nos dados apresenta, de uma forma geral, uma boa distribuição do processamento. Contudo o volume de comunicações é bastante superior ao do paralelismo de controlo [SHKS99, FL98].

4.4.2.1 Distribuição vertical dos dados

Como foi referido, apesar da abordagem ao paralelismo com distribuição vertical dos dados apresentar uma distribuição do processamento melhor que no paralelismo de controlo, ainda podem existir alguns problemas no balanceamento da carga. Estes problemas devem-se principalmente à diferença de processamento exigido entre a avaliação dos atributos discretos e contínuos [FL98].

Nos algoritmos descritos que utilizam o paralelismo com distribuição vertical dos exemplos a diferença de processamento na avaliação dos diferentes tipos de atributos é substancialmente reduzida pela utilização de aproximações na avaliação dos atributos contínuos ou pela transformação dos valores dos atributos. Contudo, estas aproximações podem dar origem a uma perda na precisão de classificação das árvores de decisão obtidas por estes algoritmos [FL98].

Uma outra solução para o problema do balanceamento da carga passa pela distribuição dos atributos contínuos o mais equitativa quanto possível pelos processos [SHKS99]. Neste caso, como os atributos contínuos são sempre avaliados em cada nó, cada processo realiza, mais ou menos, a mesma quantidade de processamento.

A distribuição vertical dos dados apresenta, de uma forma geral, um óptimo balanceamento da carga pelos processos e o volume de comunicações necessário para a construção dos nós é menor que o necessário numa abordagem com distribuição horizontal. Contudo os algoritmos de construção de árvores de decisão em paralelo com distribuição vertical dos dados não escalonam bem [SHKS99, Cha97].

4.4.2.2 Distribuição horizontal dos dados

Os algoritmos de construção de árvores de decisão em paralelo com distribuição horizontal dos exemplos apresentam, de uma forma geral, uma boa distribuição do processamento. Contudo, pode dar-se o caso de, em alguns nós da árvore, parte dos processos realizar todo o trabalho enquanto que os restantes permanecem em estado de espera.

Supondo que, a dado ponto na construção da árvore, a exploração de um nó dá origem a dois novos nós, então cada processo vai dividir o seu conjunto de exemplos em dois. Se, com a divisão dos exemplos, uma parte dos processos contém apenas os exemplos associados ao primeiro nó, e os restantes contêm os exemplos que pertencem apenas ao segundo nó, quando os novos nós forem explorados, apenas uma parte dos processos (os que contêm os exemplos associados aos nós), vão realizar processamento útil enquanto que os restantes processos permanecem em estado de espera. Este pequeno exemplo ilustra como, em alguns casos, o paralelismo com distribuição horizontal pode levar a uma má distribuição do processamento. Uma solução possível para este problema, utilizada no algoritmo descrito em [SHKS99], é construir de uma só vez todos os nós ainda por explorar. Isto é, numa só passagem pelo conjunto de exemplos local, cada processo encontra as distribuições locais e realiza os cálculos necessários para cada nó ainda por explorar.

A abordagem ao paralelismo com divisão horizontal dos dados é também caracterizada por apresentar um grande volume de comunicações na construção de um nó de decisão. Para os nós perto da raiz da árvore o overhead associado às comunicações é relativamente pequeno dado que a quantidade de exemplos a processar em cada nó é relativamente grande. Contudo, à medida que a árvore vai sendo construída, o número de exemplos em cada nó diminui. Mas, o volume de comunicações não diminui na mesma proporção que o número de exemplos. Logo, para os nós mais baixos da árvore, o overhead associado às comunicações domina o tempo de processamento.

Apesar das desvantagens apresentadas, os algoritmos com distribuição horizontal dos dados escalonam bastante bem [SAM96, Kuf97, SHKS99, FL98].

4.4.3 Paralelismo híbrido

As implementações de algoritmos de construção de árvores de decisão com uma abordagem ao paralelismo híbrida tentam combinar as vantagens de uma boa distribuição do processamento,típica do paralelismo nos dados, com um baixo overhead de comunicações associado ao paralelismo de controlo.

O algoritmo descrito neste capitulo em que é utilizado o paralelismo híbrido soluciona o problema do overhead das comunicações, típico de uma abordagem ao paralelismo com distribuição horizontal dos dados, pela divisão do processamento em grupos distintos de processos. Ao dividir o processamento em grupos mais pequenos de processos diminui o overhead das comunicações dado que a quantidade de informação trocada é menor e envolve um número menor de processos.

As abordagens à construção de arvores de decisão com paralelismo híbrido apresentam, de uma forma geral, melhores resultados ao nível do speedup e eficiência que as outras duas abordagens apresentadas. Os algoritmos com paralelismo híbrido escalonam bastante bem [SHKS99, Cha97, FL98].

Como nota final, é importante referir que o comportamento de um algoritmo de construção de árvores de decisão em paralelo depende fortemente do conjunto de treino utilizado para a construção da árvore. Isto é, a quantidade de processamento necessária em cada nó assim como a estrutura da árvore são determinadas apenas em tempo de execução e diferem bastante de caso para caso. Logo, é possível que um determinado algoritmo de construção de árvores de decisão apresente resultados bastante bons quando utilizado com um determinado conjunto de treino, mas quando é utilizado outro conjunto de treino diferente apresente piores resultados [FL98, Cha97].

Capítulo 5

Algoritmo de Construção de Árvores de Decisão em Paralelo

Neste capítulo descreve-se o algoritmo de construção de árvores de decisão em paralelo desenvolvido como objectivo principal deste trabalho. O algoritmo paralelo foi desenvolvido tendo como base o algoritmo de construção de árvores de decisão do sistema C4.5 [AGS01]. À semelhança de outros sistemas de indução de árvores de decisão, como o CART [BFOS84], o C4.5 [Qui92] representa o resultado de vários anos de investigação nas áreas da aprendizagem automática e extracção do conhecimento. Este tem sido tomado como ponto de referência para o desenvolvimento e análise de novos algoritmos [Rug00, FL98, DGST97]. Também, os resultados de diversos testes realizados mostram que o algoritmo de indução de árvores de decisão do C4.5 apresenta uma boa precisão na classificação e é um dos mais rápidos entre os algoritmos comparados para a extracção do conhecimento e aprendizagem automática [MST94].

No desenvolvimento do algoritmo paralelo, o algoritmo do sistema C4.5 foi substancialmente modificado de forma a resolver alguns dos seus principais problemas e que se tornariam especialmente preocupantes na implementação paralela, como por exemplo, as sucessivas ordenações do conjunto de exemplos realizadas sempre que um atributo contínuo é avaliado. Desta forma, no algoritmo paralelo desenvolvido, o algoritmo do C4.5 foi modificado para utilizar uma abordagem semelhante à dos sistemas SLIQ e SPRINT apresentados no capítulo anterior. Esta abordagem consiste na implementação de uma algoritmo paralelo com distribuição horizontal dos dados em que se utilizam listas separadas para os valores dos atributos que, como se verá mais, permitem que os valores dos atributos contínuos sejam ordenados apenas uma única vez no início do processo de construção da árvore de decisão.

A descrição do algoritmo segue uma estrutura com uma correspondência directa ao fluxo de execução do algoritmo. Primeiro descreve-se o ambiente de execução para o qual foi desenhado, detalhando também a forma como são distribuídos os dados e o seu processamento. Na secção seguinte descrevem-se as listas de atributos e de classes utilizadas pelo algoritmo e a ordenação paralela utilizada para a ordenação dos valores dos atributos contínuos. Por fim descreve-se o processo de construção de árvores de decisão em paralelo dividido nas fases de crescimento e simplificação da árvore.

5.1 Modelo de implementação

O algoritmo de construção de árvores de decisão em paralelo desenvolvido neste trabalho foi desenhado para ser executado num sistema paralelo de memória distribuída. Um sistema de memória distribuída é formado por um conjunto de processadores, cada um com a sua própria memória privada. Cada par processador / memória, ou nó, encontra-se ligado aos outros nós através de uma rede de interligação que permite a comunicação entre os vários processadores. O método mais utilizado para a implementação das comunicações nestes sistemas consiste na troca de mensagens entre os processadores. Desta forma, o algoritmo desenvolvido foi desenhado para ser executado por um conjunto de processos, cada um com acesso à sua própria memória privada. As comunicações entre os processos foram implementadas através da troca de mensagens. Para a implementação do algoritmo foi considerada a utilização da biblioteca de funções do *Message Passing Interface* (MPI) [Pac97] que, para além de proporcionar mecanismos bastante simples para execução de um algoritmo paralelo num ambiente distribuído, dispõe de funções de troca de mensagens de mais alto nível como a difusão ou reunião dos dados.

Como no desenvolvimento de qualquer algoritmo paralelo, no desenvolvimento do algoritmo de construção de árvores de decisão em paralelo foram consideradas as questões fundamentais da distribuição do processamento e dos dados pelos processos. Resumidamente, uma distribuição uniforme dos dados foi conseguida através da distribuição horizontal do conjunto de treino (utilizado na construção da árvore) equitativamente pelos vários processos. Desta forma, cada processo é apenas responsável pelo processamento de um subconjunto de exemplos distintos. Também, durante a execução do algoritmo, cada processo realiza as mesmas operações sobre o seu conjunto de dados, conseguindo-se assim uma distribuição uniforme do processamento.

Como, durante a construção da árvore de decisão, cada processo executa o mesmo algoritmo mas sobre conjuntos de dados distintos, o algoritmo desenvolvido enquadra-se dentro dos algoritmos de construção de árvores de decisão em paralelo com distribuição horizontal dos dados.

5.2 Leitura dos dados

De forma semelhante a outros algoritmos de construção de árvores de decisão em paralelo, como o SLIQ ou o SPRINT [SAM96], o conjunto de treino utilizado na construção da árvore é distribuído horizontalmente pelos processos, em que, cada processo fica responsável por um conjunto de exemplos distintos.

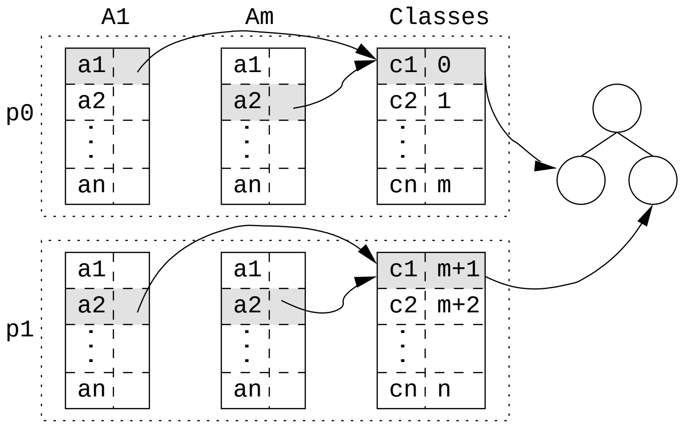


Figura 5.1: Listas de atributos e de classes.

Para a representação dos dados, o algoritmo desenvolvido utiliza estruturas semelhantes às utilizadas no sistema SLIQ apresentado capítulo anterior. Os dados são guardados em duas estruturas diferentes (figura 5.1):

- A lista de classes, com uma entrada para cada exemplo do conjunto de treino. Cada entrada contém o valor da classe, o índice global no conjunto de treino e um apontador para o nó da árvore ao qual o exemplo está associado.
- As listas de atributos, uma lista independente para cada atributo, e também com uma entrada para cada exemplo no conjunto de treino. Cada entrada das listas de atributos contém o valor do atributo para um determinado exemplo e um apontador para a entrada correspondente na lista de classes.

Durante a leitura do conjunto de exemplos atribuído, cada processo é responsável pela criação das suas próprias listas de atributos e de classes.

Para a ilustração dos vários passos do algoritmo utiliza-se o mesmo conjunto de exemplos já apresentado no segundo capítulo durante a descrição das árvores de decisão e agora novamente retomado (tabela 5.1). Cada exemplo deste conjunto é descrito por quatro atributos (estado do céu, temperatura, humidade e vento) e indica se é um bom dia ou não para jogar futebol no parque.

Supondo que este conjunto de exemplos é utilizado para a construção da árvore com dois processos. Então, cada um dos processos fica responsável por um subconjunto de 7 exemplos e as listas de atributos e de classes construídas por cada um são idênticas às representadas na figura 5.2.

Após a leitura do conjunto de treino e da criação das listas de atributos e de classes, as listas dos atributos discretos não necessitam de mais nenhum processamento adicional. Contudo, antes do início da construção da árvore de decisão, as listas dos atributos contínuos têm de ser ordenadas globalmente pelos valores do atributo. A

Céu	**Temperatura**	**Humidade**	**Vento**	**Classe**
Limpo	75	70	Sim	Jogar
Limpo	69	70	Não	Jogar
Limpo	85	85	Não	Não Jogar
Limpo	80	90	Sim	Não Jogar
Limpo	72	95	Não	Não Jogar
Enevoado	72	90	Sim	Jogar
Enevoado	83	78	Não	Jogar
Enevoado	64	65	Sim	Jogar
Enevoado	81	75	Não	Jogar
Chuva	71	80	Sim	Não Jogar
Chuva	65	70	Sim	Não Jogar
Chuva	75	80	Não	Jogar
Chuva	68	80	Não	Jogar
Chuva	70	96	Não	Jogar

Tabela 5.1: Conjunto de exemplos utilizados para a ilustração dos vários passos do algoritmo.

ordenação global dos valores dos atributos contínuos é realizada adaptando o algoritmo de ordenação paralela descrito em [DNS91].

5.2.1 Ordenação paralela dos atributos contínuos

Para a ordenação global dos valores dos atributos contínuos utiliza-se o mesmo algoritmo de ordenação paralela que o utilizado no sistema SPRINT [SAM96], e descrito em [DNS91]. O problema considerado pelos autores é um problema de ordenação externa num sistema de base de dados paralelo sem partilha. Um sistema de base de dados "sem partilha" significa que o sistema encontra-se implementado num sistema de multiprocessadores em que cada processador dispõe da sua própria memória e disco locais. Todas as comunicações entre os processos são realizadas através de uma rede de interligação. Este sistema vai de encontro com o sistema de memória partilhada assumido para a execução do algoritmo desenvolvido neste trabalho. O problema de ordenação é um problema de "entradas-múltiplas, saídas-múltiplas". Os dados a ordenar encontram-se distribuídos pelos processos e não ordenados. No final do algoritmo de ordenação, os dados devem encontrar-se novamente distribuídos pelos processos e cada processo deve conter aproximadamente o mesmo número de valores ordenados e sem sobreposição.

O algoritmo genérico para o problema de ordenação considerado pode ser descrito como que dividido nas três fases seguintes:

1. Encontrar um "vector de divisão" $v[i]$, onde $1 \leq i < k$, de forma a que, na ordem final, todos os valores no processo p_1 sejam menores que $v[1]$, todos os valores

	Céu	Temp.	Humidade	Vento	Classes	
p0	Limpo	75	70	Sim	Jogar	0
	Limpo	69	70	Não	Jogar	1
	Limpo	85	85	Não	Não Jogar	2
	Limpo	80	90	Sim	Não Jogar	3
	Limpo	72	95	Não	Não Jogar	4
	Enevoado	72	90	Sim	Jogar	5
	Enevoado	83	78	Não	Jogar	6
p1	Enevoado	64	65	Sim	Jogar	7
	Enevoado	81	75	Não	Jogar	8
	Chuva	71	80	Sim	Não Jogar	9
	Chuva	65	70	Sim	Não Jogar	10
	Chuva	75	80	Não	Jogar	11
	Chuva	68	80	Não	Jogar	12
	Chuva	70	96	Não	Jogar	13

Figura 5.2: Listas de atributos e de classes criadas a partir do conjunto de treino representado na tabela 5.1.

em p_2 sejam maiores que $v[1]$ mas menores que $v[2]$, e assim sucessivamente, até todos os valores em p_k sejam maiores que $v[k-1]$.

2. Com base no vector de divisão encontrado, redistribuir os valores pelos processos.

3. Após a distribuição dos valores, ordenar localmente os valores em cada processo.

O primeiro passo do algoritmo anterior é um passo crítico para o seu desempenho. Em [DNS91] foram consideradas duas abordagens diferentes para determinar o vector de divisão. Para a ordenação dos valores dos atributos contínuos, no algoritmo desenvolvido foi utilizada a primeira abordagem, denominada de "divisão probabilística" e onde o vector de divisão é obtido através de um subconjunto de valores retirados aleatoriamente do conjunto total de valores não ordenados do atributo. Os passos seguintes resumem o algoritmo de ordenação paralela utilizado na ordenação global de cada uma das listas de valores dos atributos contínuos:

1. Em paralelo, cada processo retira aleatoriamente um conjunto de valores (amostras) a partir da lista local de valores do atributo.

2. Em paralelo, cada processo ordena localmente o conjunto de amostras e envia-o para um único processo onde os vários conjuntos são combinados para formar um único conjunto ordenado de amostras. A partir do conjunto de amostras é encontrado um vector de divisão que é comunicado a todos os processos.

3. Em paralelo, cada processo percorre a sua lista de valores do atributo e, utilizando o vector de divisão, redistribui os valores pelos vários processos.

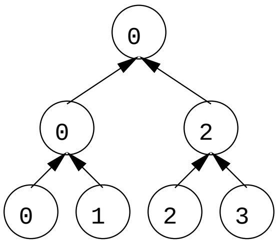


Figura 5.3: Comunicação estruturada em árvore.

4. Em paralelo e após a redistribuição dos valores, cada processo ordena localmente a sua lista de valores do atributo.

No segundo passo do algoritmo a reunião dos conjuntos de amostras é realizada imaginado os processos numa árvore binária (figura 5.3). Desta forma, os processos nas folhas enviam o seu conjunto de amostras para os pais, que o junta ao seu conjunto para formar um novo conjunto ordenado com as amostras dos dois processos. A seguir, os pais enviam os novos conjuntos para os seus pais, e assim sucessivamente, até que o processo na raiz da árvore reune o conjunto completo de amostras ordenado. A partir deste conjunto, o processo na raiz determina um vector de divisão com tantos elementos como o número de processos menos um. Cada elemento deste vector representa o maior valor atribuído ao processo correspondente. Por fim, o vector divisão é enviado a todos os processos utilizando, por exemplo, uma comunicação colectiva de difusão também estruturada em árvore.

Para a redistribuição dos valores (terceiro passo), cada processo percorre a sua lista local de valores do atributo e, para cada valor v, determina a posição r no vector de divisão V_s tal que $V_s[r-1] < v \leq V_s[r]$. Este valor é atribuído ao processo $k = r$.

Durante a distribuição dos valores dos atributos contínuos, cada processo pode receber valores que correspondem a exemplos que não se encontram na sua lista de classes, construída a partir da distribuição inicial do conjunto de treino. No terceiro passo, para além de ser enviado o valor do atributo, são também enviados o valor da classe e o índice global correspondente. Ao receber os valores, cada processo actualiza a sua lista de classes com a informação dos novos exemplos recebidos.

O algoritmo de ordenação utilizado não garante que, no final da ordenação, os processos contenham um número igual de valores do atributo. O número de valores em cada processo depende fortemente do número de amostras utilizado. Quanto maior for o número de amostras maior é a probabilidade de uma melhor distribuição dos valores. Contudo, a utilização de um maior número de amostras também aumenta o custo de execução dos primeiros passos do algoritmo. Logo, existe um compromisso entre o número de amostras utilizado e uma distribuição uniforme dos valores dos atributos pelos processos, o que torna a escolha de um número adequado de amostras num dos factores críticos para o desempenho do algoritmo. A forma como é determinado o número de amostras é abordada no capítulo seguinte, durante a análise de desempenho

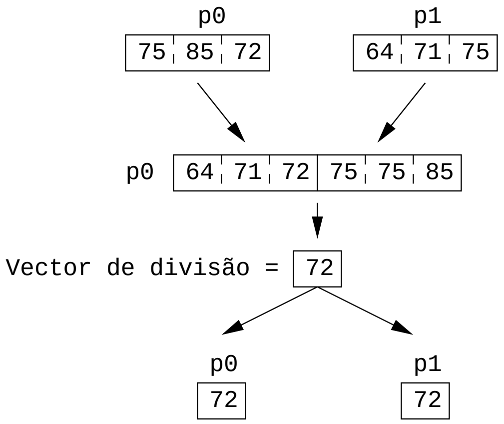


Figura 5.4: Construção e ordenação do vector de divisão.

do algoritmo.

Na figura 5.4 ilustra-se o processo para determinar o vector de divisão utilizado para a ordenação global dos valores do atributo *Temperatura* do conjunto de treino utilizado no exemplo anterior (tabela 5.1). Neste exemplo, cada processo reúne um conjunto de 3 amostras a partir das suas listas locais do atributo. Estes dois conjuntos de amostras são reunidos no processo 0 que determina um vector de divisão de um elemento. O vector de divisão é comunicado aos dois processos e utilizado posteriormente para a redistribuição dos valores do atributo. Na figura 5.5 é ilustrado o resultado final da ordenação do atributo e da actualização das listas de classes com a informação dos novos exemplos recebidos por cada processo.

5.3 Construção da árvore de decisão

O processo de construção de uma árvore de decisão pode ser dividido em duas fases. A primeira fase, a fase de crescimento da árvore, a árvore de decisão é construída até todos os exemplos cobertos por um nó pertencerem à mesma classe (ou que o número de exemplos seja inferior a um valor pré determinado). Na segunda fase, alguns nós são simplificados de acordo com um determinado critério, e substituídos por folhas de forma a eliminar alguma dependência a ruído de natureza estatística ou a algum conceito presente apenas nos dados de treino.

Sem qualquer dúvida, a fase computacionalmente mais dispendiosa do algoritmo é a fase de construção da árvore de decisão. À excepção dos conjuntos de treino muito pequenos, o tempo utilizado na simplificação da árvore é apenas uma pequena fracção do tempo total de execução do algoritmo [Cat91, Qui92, Rug00, SHKS99]. Logo, a fase onde o paralelismo é mais necessário é a fase de crescimento da árvore. Embora o paralelismo não seja tão essencial para a simplificação, no algoritmo desenvolvido esta fase foi também implementada em paralelo aproveitando o facto de os dados já se encontrarem distribuídos pelos processos, usufruindo desta forma, da vantagem de

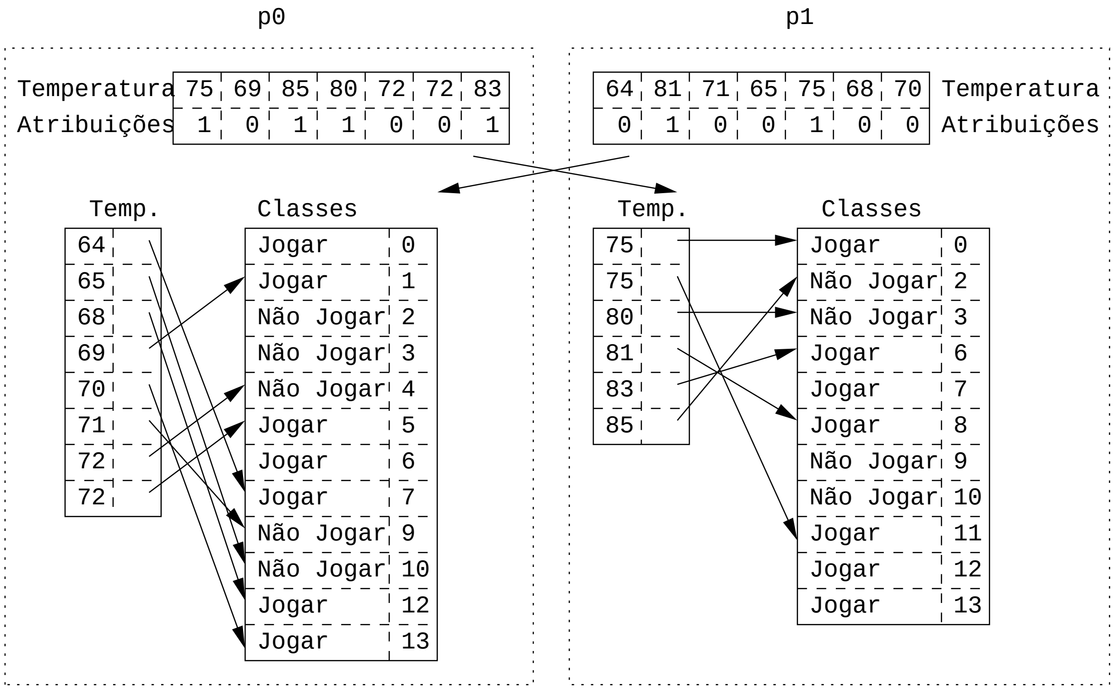


Figura 5.5: Resultado final da ordenação paralela do atributo *Temperatura*.

dispor de k processos a processar $1/k$ do total dos dados

5.3.1 Crescimento da árvore de decisão

Assim como no sistema C4.5, no algoritmo paralelo também é utilizada uma estratégia de "dividir para conquistar" na construção da árvore, em que os vários processos cooperam entre si na construção de cada nó da árvore, um nó de cada vez. Em cada nó o conjunto de exemplos é dividido e os subconjuntos resultantes da divisão são associados a novos nós até todos os subconjuntos cobrirem exemplos que pertençam a apenas uma única classe.

No algoritmo de construção de árvores de decisão do C4.5 a árvore de decisão é construída "primeiro em profundidade". Isto é, cada ramo da árvore é explorado até ao fim antes de se dar início à exploração de outros ramos (figura 5.6a). Contudo, a utilização deste método na construção da árvore de decisão em paralelo pode dar origem a uma má distribuição do processamento. Por exemplo, se apenas uma parte dos processos contêm os exemplos cobertos por um nó, apenas esses processos vão realizar processamento útil na exploração da sub-árvore com raiz nesse nó, enquanto que os restantes permanecem em estado de espera. No algoritmo paralelo, a árvore de decisão é construída adoptando um método diferente. Esta é construída em "primeiro em largura" em que, para um determinado nível da árvore, são construídos todos os nós ainda por explorar antes de se iniciar a construção dos nós do nível seguinte (figura 5.6b). Desta forma, é evitada uma má distribuição do processamento em que, para

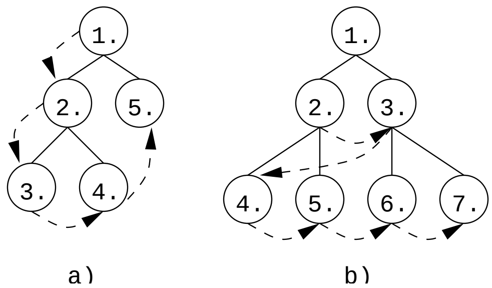


Figura 5.6: Métodos de construção de uma árvore: a) Primeiro em profundidade; b) Primeiro em largura.

cada nível da árvore, cada processo processa a totalidade dos exemplos pelos quais é responsável, independentemente se é, ou não, responsável pelos exemplos cobertos por um determinado nó.

Independentemente do método de construção da árvore utilizado, os passos executados para a construção de um único nó são iguais em todos os processos. A cada nó da árvore de decisão são associadas as listas com os valores dos atributos para os exemplos cobertos pelo nó. Quando um conjunto de exemplos é divido, as listas de atributos associadas ao nó são também divididas e as sub-listas são associadas aos nós filho resultantes da divisão.

A cada nó são também associadas outras estruturas que contêm as distribuições globais dos exemplos pelas classes para cada atributo e que dependem do tipo de atributo. Estas estruturas são utilizadas posteriormente durante a fase de avaliação dos atributos. Por enquanto não é necessário conhecer com demasiado detalhe qual a informação guardada nestas estruturas ou modo como estas são determinadas. Apenas é importante referir que, a partir desta informação, quando se inicia a construção de um nó já se conhece o número total de exemplos associados ao nó e a sua distribuição pelas classes.

Em resumo, a construção de um nó da árvore de decisão é realizada por todos os processos que executam os mesmos passos, ou algoritmo, mas sobre um subconjunto de exemplos distintos do conjunto coberto pelo nó. O algoritmo para a construção de um nó da árvore de decisão, executado por cada um dos processos, pode ser resumido nos seguintes passos:

```
Constrói nó N:
  1. Se todos os exemplos em N pertencem à mesma classe Ci:
     1.1. O nó N é uma folha que representa a classe Ci.
  2. Se não:
     2.1. Para cada atributo A:
          2.1.1 Calcula o ganho da divisão pelo atributo.
     2.2. Cria tantos filhos N como os resultados possíveis da melhor
          divisão encontrada.
```

```
2.3. Divide a lista do atributo Ai com melhor ganho, associando
     cada sub-lista a um filho correspondente de N
2.4. Determina a atribuição global dos exemplos aos nós.
2.5. Divide as restantes listas dos atributos associando cada
     sub-lista ao filho de N correspondente.
2.6. Para cada filho Si de N:
     2.6.1. Encontra as distribuições globais para cada atributo
            no conjunto de exemplos cobertos por Si.
```

A avaliação em paralelo das possíveis divisões do conjunto de exemplos associados a um nó é bastante semelhante à avaliação das divisões realizada no algoritmo do sistema C4.5. O algoritmo do C4.5 utiliza o *critério de razão ganho* para a avaliação das divisões do conjunto de exemplos cobertos por um nó, determinado a partir das distribuições dos exemplos pelas classes. Não existe qualquer diferença na versão paralela do algoritmo. Nenhum processamento adicional ou comunicação é necessária quando os processos estão a avaliar os seus conjuntos de exemplos locais. Durante a avaliação das possíveis divisões obtém-se a vantagem de dispor de k processos a processar de forma independente $1/k$ dos exemplos cobertos pelo nó. As versões paralela e série diferem entre si apenas antes e depois da fase de avaliação dos atributos.

5.3.1.1 Avaliação dos atributos discretos

A avaliação de uma divisão de um conjunto de exemplos é realizada através do critério de razão de ganho associado à divisão. A razão de ganho para a divisão de um conjunto de exemplos T em T_i subconjuntos $(1 \leq i \leq m)$ é determinada a partir das seguintes equações:

$$Razao\ de\ Ganho = \frac{Ganho(T)}{\sum_{i=1}^{m} \frac{|T_i|}{|T|} * log_2(\frac{|T_i|}{|T|})}\ bits \tag{5.1}$$

Em que

$$Ganho(T) = Info(T) - \sum_{i=1}^{m} \frac{|T_i|}{|T|} * Info(T_i) \tag{5.2}$$

e

$$Info(T) = -\sum_{j=1}^{n} \frac{freq(C_j, T)}{|T|} * log_2(\frac{freq(C_j, T)}{|T|})\ bits \tag{5.3}$$

Para o cálculo da razão de ganho associada a uma divisão de um conjunto de exemplos é suficiente conhecer as distribuições dos exemplos pelas classes para cada subconjunto resultante da divisão. Como, a divisão do conjunto de exemplos baseada num atributo discreto tem como resultado tantos subconjuntos como os valores possíveis do atributo (um subconjunto para cada valor), a única informação necessária para a avaliação da divisão consiste apenas nas distribuições dos exemplos pelas classes para cada valor possível do atributo.

Céu	**Jogar**	**Não Jogar**
Limpo	2	3
Enevoado	4	0
Chuva	3	2

Tabela 5.2: Distribuições globais dos exemplos pelas classes para os valores possíveis do atributo *Céu*

Quando um nó é criado, são-lhe associadas estruturas com as distribuições globais dos exemplos para cada atributo. Para os atributos discretos estas estruturas contêm as distribuições globais dos exemplos pelas classes para cada valor possível do atributo. As distribuições globais dos exemplos são determinadas durante a fase de divisão do conjunto de exemplos do nó antecessor descrita mais à frente, sendo apenas necessário, por enquanto, saber que estas contêm os valores globais das distribuições. Isto é, através destas distribuições cada processo conhece as distribuições para os exemplos guardados em todos os processos. Com base nesta informação cada processo calcula, em paralelo, a razão de ganho obtida pela divisão do conjunto de exemplos para cada atributo discreto. Os valores obtidos para a razão de ganho são iguais em todos os processos dado que todos têm acesso aos valores globais das distribuições.

Considerando o mesmo conjunto de exemplos que tem vindo a ser utilizado ao longo deste capítulo (tabela 5.1), a avaliação da divisão deste conjunto baseada no atributo *Céu* é determinada por cada processo com base nas distribuições globais dos exemplos pelas classes para cada um dos três valores possíveis do atributo e representadas na tabela 5.2. Desta forma, cada um dos processos apenas substituí as variáveis referentes às distribuições nas equações 5.1, 5.2 e 5.3 pelos valores correspondentes:

$$Info(T) = -9/14 * log_2(9/14) - 5/14 * log_2(5/14) = 0,940$$

$$Info(T_{Limpo}) = -2/5 * log_2(2/5) - 3/5 * log_2(3/5) = 2,170$$

$$Info(T_{Enevoado}) = 0$$

$$Info(T_{Chuva}) = -3/5 * log_2(3/5) - 2/5 * log_2(2/5) = 2,170$$

$$Ganho(Ceu) = 0,940 - 5/14 * 2,170 - 4/14 * 0 - 5/14 * 2,170 = -0,610$$

$$Razao\ de\ Ganho(Ceu) = \frac{-0,610}{5/14 * log_2(5/14) + 4/14 * log_2(4/14) + 5/14 * log_2(5/14)} = 0,387$$

5.3.1.2 Avaliação dos atributos contínuos

A avaliação dos atributos contínuos é também bastante semelhante à realizada pelo algoritmo do sistema C4.5. A divisão do conjunto de exemplos baseada num atributo contínuo tem como resultado dois subconjuntos: o primeiro com os valores do atributo menores ou iguais a um determinado valor (ponto de divisão) e o segundo com os valores maiores. A avaliação da divisão é realizada com base nestes dois subconjuntos. Para um atributo contínuo com n valores ordenados, o algoritmo avalia as $n-1$ divisões

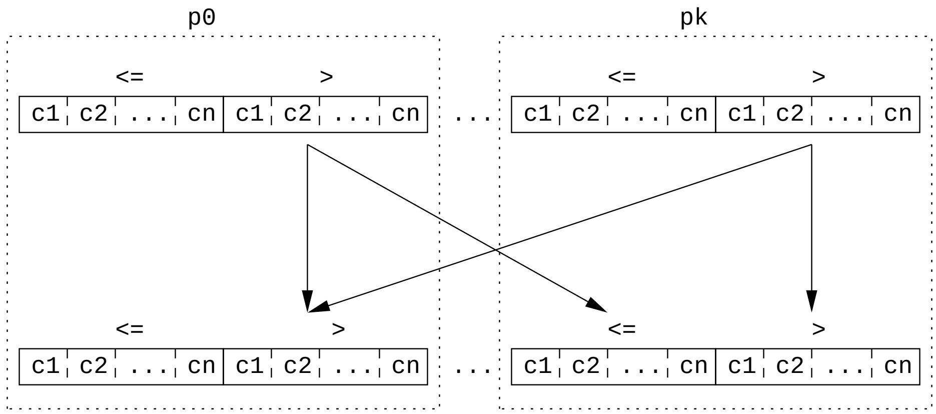


Figura 5.7: Inicialização das distribuições dos atributos contínuos.

possíveis e escolhe aquela que apresentar a melhor razão de ganho. O ponto de divisão escolhido é o maior valor, de entre todos os valores do atributo, inferior ao valor médio dos limites da divisão.

Ao contrário do que acontece no algoritmo do C4.5, no algoritmo paralelo não é necessário ordenar o conjunto de exemplos pelos valores do atributo sempre que um atributo contínuo é avaliado. Os valores dos atributos encontram-se em listas separadas de valores que, para os atributos contínuos, foram já ordenadas no início da construção da árvore e a sua ordem mantida ao longo da execução do algoritmo.

Durante a avaliação de um atributo contínuo cada processo percorre a sua lista local de valores para determinar a melhor divisão para o seu conjunto de valores. Contudo, antes da avaliação de um atributo contínuo, cada processo necessita de conhecer as distribuições globais dos exemplos pelas classes que, de forma semelhante aos atributos discretos, são determinadas durante a fase de divisão do conjunto de exemplos do nó antecessor, descrita mais à frente. As estruturas associadas ao nó com as distribuições para os atributos contínuos contêm também as distribuições dos exemplos pelas classes para ambos os subconjuntos resultantes da divisão. Como, antes da avaliação do atributo ainda não é conhecido nenhum ponto de divisão, estas estruturas são inicializadas por cada processo como se o ponto de divisão fosse o primeiro valor do atributo conhecido pelo processo. As distribuições locais de cada processo são posteriormente partilhadas por todos os processos em que, cada processo, inicializa as distribuições com os valores menores ou iguais que o ponto de divisão com as distribuições locais dos processos de ordem inferior e as distribuições com os valores maiores que ponto de divisão com as suas distribuições locais e com as distribuições dos processos de ordem superior (figura 5.7). Assim como acontece para os atributos discretos, a inicialização das estruturas dos atributos contínuos é realizada durante a fase de divisão dos exemplos, encontrando-se os seus valores já inicializados antes da avaliação dos atributos.

Durante a avaliação de um atributo contínuo cada processo percorre a sua lista local de valores do atributo em que, a cada passo, actualiza as suas distribuições para o ponto de

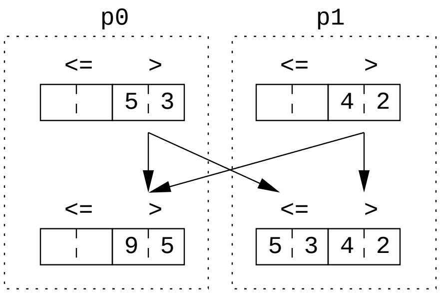


Figura 5.8: Inicialização das distribuições associadas ao atributo *Temperatura*.

p0

Temperatura

64	Jogar
65	Não Jogar
68	Jogar
69	Jogar
70	Jogar
71	Não Jogar
72	Não Jogar
72	Jogar

<=		>		Razão
1	0	8	5	0
1	1	8	4	-0,017
2	1	7	4	-0,001
3	1	6	4	-0,017
4	1	5	4	-0,098
4	2	5	3	-0,001
4	3	5	2	-0,016
5	3	4	2	-0,001

p1

Temperatura

75	Jogar
75	Jogar
80	Não Jogar
81	Jogar
83	Jogar
85	Não Jogar

<=		>		Razão
6	3	3	2	-0,003
7	3	2	2	-0,029
7	4	2	1	-0,001
8	4	1	1	-0,017
9	4	0	1	0

Figura 5.9: Avaliação do atributo *Temperatura*.

divisão actual e calcula a razão de ganho para essa divisão. Após a avaliação de todas as possíveis divisões dos seus conjuntos de valores locais, os processos comunicam entre si para determinarem a melhor divisão a partir das k melhores divisões encontradas localmente por cada um.

Para ilustrar o processo de avaliação dos atributos contínuos considera-se a avaliação da divisão do mesmo conjunto de exemplos que tem vindo a ser utilizado (tabela 5.1) pelo atributo *Temperatura*. Na figura 5.8 representam-se as estruturas, já inicializadas, com as distribuições dos exemplos pelas classes para os valores menores e maiores que o ponto de divisão em cada processo. Estas estruturas são utilizadas, por cada processo, para o cálculo da razão de ganho para cada ponto de divisão, como ilustrado na figura 5.9. Ao percorrer a sua lista local dos valores do atributo, cada processo actualiza as distribuições dos exemplos pelas classes e substitui os valores correspondentes nas equações 5.1, 5.2 e 5.3 para determinar o valor da razão de ganho para cada ponto de divisão. Neste caso, o atributo não é considerado para a divisão do conjunto de exemplos dado que os ganhos obtidos são todos inferiores ou iguais a 0.

5.3.1.3 Divisão do conjunto de exemplos

Após cada processo ter avaliado todas as possíveis divisões “locais” do conjunto de exemplos por cada atributo, os resultados são partilhados para encontrar a melhor divisão. Os resultados das avaliações para os atributos discretos são iguais em todos os processos uma vez que estes são calculados com base na mesma informação global. Contudo, para os atributos contínuos, cada processo encontra um ponto de divisão local que melhor divide o seu conjunto de valores. Os melhores pontos de divisão encontrados localmente por cada processo são partilhados e, para cada atributo contínuo, é escolhido o melhor ponto de divisão de entre os melhores pontos de divisão encontrados. Assim como no algoritmo do C4.5, o atributo escolhido para a divisão do conjunto de exemplos cobertos pelo nó é aquele que apresenta a maior razão de ganho (equação 5.1) e em que a média do ganho de informação (equação 5.2) é superior à média dos ganhos de informação encontrados para todos os atributos.

Após ser determinada a melhor divisão do conjunto de exemplos cobertos pelo nó, são criados, em cada processo, tantos nós filho como os resultados da divisão. A divisão do conjunto de exemplos cobertos pelo nó é realizada, em cada processo, pela divisão das listas de atributos, onde cada sub-lista é associada ao nó correspondente, e pela actualização dos apontadores para os nós na lista de classes fazendo corresponder, desta forma, cada exemplo ao novo nó ao qual é associado.

A divisão da lista do atributo escolhido é bastante simples. Cada processo percorre a sua lista local do atributo e, dependendo do valor do atributo, atribui a entrada à sub-lista correspondente.

Como não existe nenhuma relação directa entre as listas de atributos os apontadores para os nós na lista de classes são utilizados na divisão das restantes listas e que foram actualizados durante a divisão da lista do atributo escolhido. Contudo, a lista de valores do atributo escolhido num determinado processo pode não conter os valores de todos os exemplos presentes nas restantes listas. Torna-se então necessário que os vários processos troquem entre si a informação das suas atribuições locais dos exemplos aos novos nós. A partir das entradas da lista de classes actualizadas, cada processo reúne a informação dos índices globais dos exemplos e a que nó filho da árvore foram associados. Esta informação é então partilhada entre todos os processos. Para a partilha das atribuições é utilizada uma comunicação colectiva estruturada num esquema em *butterfly* como ilustrado na figura 5.10. Em cada fase da comunicação, quando um processo recebe as atribuições de um outro processo, junta-as às suas através de uma operação de reunião que tem como resultado uma lista das atribuições dos dois processos ordenada pelo índice global dos exemplos.

Depois de receber a informação das atribuições dos exemplos aos nós de todos os processos, cada processo divide localmente as listas de valores dos restantes atributos.

Para a divisão da lista de um atributo, que não o escolhido para a divisão, a lista é percorrida e, para cada valor, é verificado através da lista de classes se o exemplo

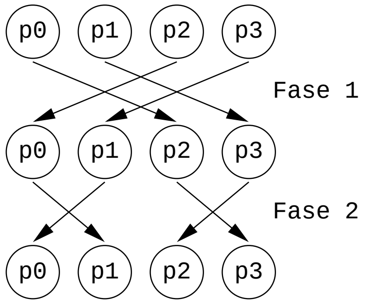


Figura 5.10: Estrutura de comunicação *Butterfly*.

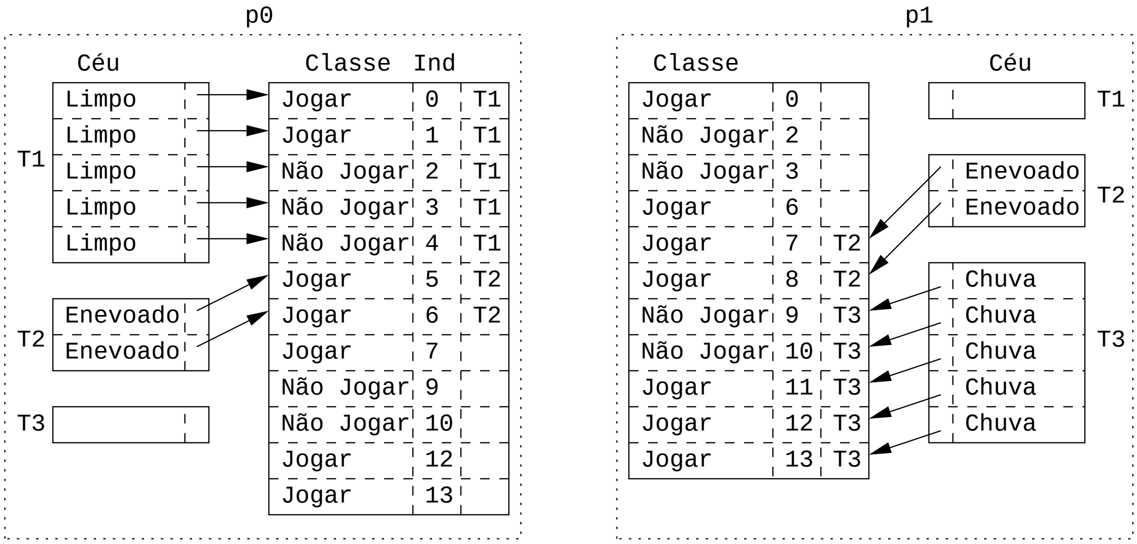


Figura 5.11: Divisão das listas do atributo *Céu*.

correspondente já foi atribuído a um dos novos nós criados com a divisão. Se o exemplo já foi atribuído, então a entrada é atribuída à sub-lista correspondente. Se o exemplo ainda não foi atribuído, a atribuição é procurada na lista de atribuições global, utilizando uma pesquisa binária. A entrada é então atribuída à sub-lista correspondente e é actualizado o apontador para o nó da árvore na lista de classes.

Por exemplo, suponhamos que o atributo escolhido para a divisão do conjunto de exemplos que tem vindo a ser utilizado é o atributo *Céu*. Com base neste atributo são criados três novos nós na árvore e o conjunto de exemplos dividido em três subconjuntos, um para cada valor do atributo. A figura 5.11 ilustra o resultado da divisão das listas do atributo *Céu* em cada um dos processos e a respectiva actualização das listas de classes. Para tornar possível a divisão das restantes listas de atributos, os dois processos trocam entre si a informação das atribuições dos exemplos aos nós, reunida na tabela 5.3. Na figura 5.12 ilustra-se o resultado da divisão das listas do atributo *Temperatura* nos dois processos e da respectiva actualização das entradas das listas de classes com base na lista de atribuições partilhada.

Durante a divisão das listas de atributos, cada processo encontra as distribuições locais

Índice	Nó
0	0
1	0
2	0
3	0
4	0
5	1
6	1
7	1
8	1
9	2
10	2
11	2
12	2
13	2

Tabela 5.3: Lista de atribuições dos exemplos aos nós.

dos exemplos pelas classes para cada atributo e para cada novo nó. Para os atributos discretos são encontradas as distribuições dos exemplos pelas classes para cada valor possível do atributo. Para os atributos contínuos são encontradas as distribuições locais de todos os exemplos pelas classes, guardadas nas distribuições para os valores do atributo maiores que o ponto de divisão.

Em seguida, as distribuições são partilhadas entre todos os processos. Para os atributos discretos as distribuições globais são obtidas pela soma das distribuições locais de cada processo. No caso dos atributos contínuos, as distribuições dos exemplos com os valores do atributo menores e maiores que o ponto de divisão são inicializadas da forma referida anteriormente e ilustrada na figura 5.7.

5.3.2 Simplificação da árvore de decisão

Para a simplificação da árvore de decisão o conjunto de treino é redistribuindo pelos processos de acordo com a distribuição inicial, antes da ordenação dos atributos contínuos. A redistribuição dos exemplos é realizada ou por uma nova leitura do conjunto de treino, ou pela troca dos exemplos entre os processos. Na segunda hipótese, os processos determinam quais os exemplos que têm de enviar a cada processo através do índice global do exemplo. No final da redistribuição dos exemplos, as listas dos atributos estão perfeitamente correlacionadas umas com as outras e com a lista de classes. Isto é, o valor na i-ésima entrada da lista de um atributo corresponde ao exemplo cujos valores dos restantes atributos encontram-se nas i-ésimas entradas das outras listas e o valor da classe encontra-se também na i-ésima entrada da lista de classes.

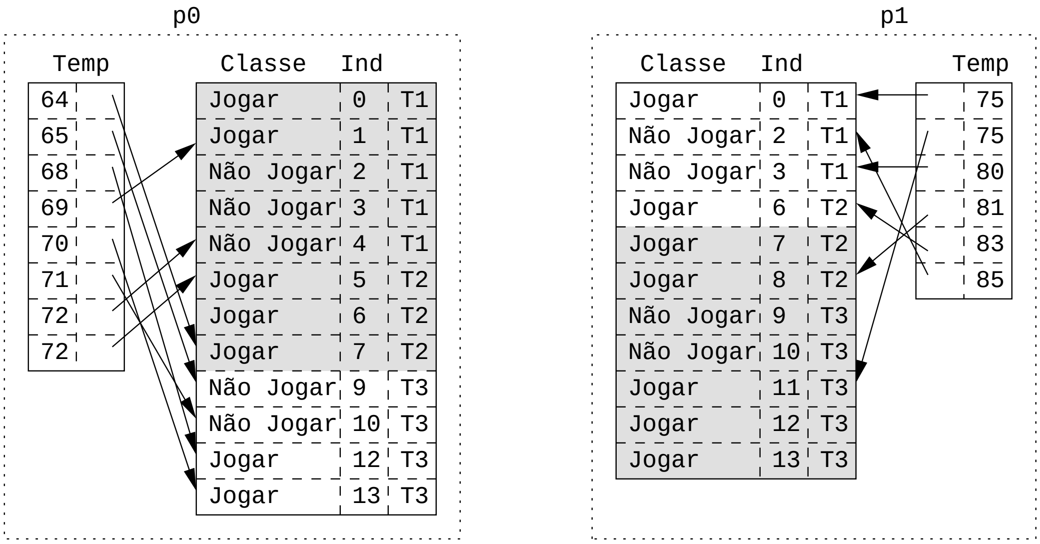


Figura 5.12: Divisão das listas do atributo *Temperatura*.

O processo de simplificação da árvore de decisão no algoritmo paralelo é bastante idêntico ao do sistema C4.5. O C4.5 utiliza uma variação de um método de simplificação posterior da árvore denominado de simplificação pessimista. Neste método é utilizado um ajuste estatístico para a estimativa do erro de classificação esperado para cada sub-árvore da árvore de decisão.

A estimativa do erro para uma folha que cubra N exemplos, com E exemplos mal classificados, é dada por $N * U_{CF}(E, N)$ exemplos mal classificados, onde $U_{CF}(E, N)$ representa o limite superior da probabilidade obtida pela distribuição binomial para um determinado nível de confiança CF. O erro previsto para um nó de decisão é calculado como a soma dos erros previstos para cada um dos seus filhos.

Como na fase de crescimento, a simplificação da árvore de decisão é realizada de forma síncrona. Isto é, cada nó da árvore de decisão é simplificado, um de cada vez, em que os vários processos executam o mesmo conjunto de instruções (algoritmo) para a sua simplificação. O algoritmo executado por cada processo para a simplificação de um nó da árvore pode ser resumido nos seguintes passos:

```
Simplifica_Nó(N):
  1. Encontra a distribuição local dos exemplos cobertos por N pelas
     classes.
  2. Determina a distribuição global dos exemplos.
  3. Encontra a classe Ci com o maior número de exemplos.
  4. Erro_Folha = Estimativa do erro considerando que N é uma folha
                  que representa a classe Ci.
  5. Se N é uma folha:
     5.1. Retorna Erro_Folha.
  6. Se não:
```

```
6.1. Erro_Nó = 0.
6.2. Para cada filho Ni de N:
     6.2.1. Erro_Nó = Erro_Nó + Simplifica_Nó(Ni).
6.3. Erro_Filho = Estimativa do erro para o filho Nj que, cobre
                  o maior número de exemplos se este
                  classificasse os exemplos cobertos por N.
6.4. Se Erro_Folha < Erro_Nó e Erro_Folha < Erro_Filho:
     6.4.1. Simplifica N por uma folha que representa a classe Ci.
     6.4.2. Retorna Erro_Folha.
6.5. Se Erro_Filho < Erro_Nó:
     6.5.1. Substitui N pelo filho Nj.
     6.5.2. Retorna Erro_Filho.
6.6. Se não:
     6.6.1. Retorna Erro_Nó.
```

Durante a simplificação da árvore de decisão cada processo executa o algoritmo anterior apenas sobre o seu conjunto local de exemplos cobertos pelo nó. O único passo onde é necessária a comunicação entre os processos é o passo 2 onde os processos partilham as suas distribuições locais de forma a que cada processo tenha acesso à distribuição global dos exemplos cobertos pelo nó.

5.3.3 Valores desconhecidos

Até agora na descrição do algoritmo de construção de árvores de decisão em paralelo foi sempre assumido que todos os valores dos atributos eram conhecidos. Infelizmente os dados muitas vezes contêm valores de atributos desconhecidos. A existência de valores desconhecidos nos dados pode dever-se a que um dado valor não seja importante para um determinado exemplo, ou que não tenha sido registado quando os dados foram reunidos, ou ainda que tenha sido mal introduzido pela pessoa encarregada da introdução dos dados.

Os valores desconhecidos nos dados levantam duas questões fundamentais para a construção da árvore de decisão:

- Como avaliar diferentes divisões do conjunto de exemplos com número diferente de valores desconhecidos nos subconjuntos resultantes;
- Ao realizar a divisão do conjunto de exemplos como tratar os exemplos com valores desconhecidos no atributo escolhido para a divisão;

A avaliação de uma divisão baseada num determinado atributo é calculada considerando a divisão do conjunto de exemplos obtida com base nos valores desse atributo. É obvio que não é possível determinar a que subconjunto um determinado exemplo deve pertencer se não for conhecido o seu valor para o atributo.

Para a avaliação de uma possível divisão, os exemplos com valores desconhecidos são considerados como pertencerem a um outro subconjunto para além dos subconjuntos criados com a divisão. Desta forma, a equação do critério de razão de ganho utilizada para a avaliação de uma divisão (equações 5.1, 5.2 e 5.3) é modificada da seguinte forma:

$$Razao\ de\ Ganho = \frac{Ganho(T)}{\sum_{i=1}^{m} \frac{|T_i|}{|T|} * log_2(\frac{|T_i|}{|T|}) - \frac{|T_u|}{|T|} * log_2(\frac{|T_u|}{|T|})}\ bits \tag{5.4}$$

Em que

$$Ganho(T) = F * (Info(T) - \sum_{i=1}^{m} \frac{|T_i|}{|T|} * Info(T_i))\ bits \tag{5.5}$$

e

$$Info(T) = -\sum_{j=1}^{n} \frac{freq(C_j, T)}{|T|} * log_2(\frac{freq(C_j, T)}{|T|})\ bits \tag{5.6}$$

Para o cálculo de $Info(T)$ (equação 5.6) apenas são considerados os exemplos com valores conhecidos no atributo. No cálculo do $Ganho(T)$ (equação 5.5) também são apenas considerados os exemplos com valores conhecidos e F representa a fracção dos exemplos conhecidos sobre o número total de exemplos. Por fim, no cálculo da *Razao de Ganho* (equação 5.4), $|T_u|$ representa o número de exemplos com valores desconhecidos.

Supondo que um conjunto de exemplos T associados a um nó é dividido, por exemplo, em $T_1, T_2, ..., T_n$ subconjuntos. Então ao atribuir um determinado exemplo a um subconjunto T_i pode-se afirmar que o exemplo pertence a T_i com uma probabilidade 1 e aos restantes subconjuntos com uma probabilidade 0. Assim como no C4.5, para o tratamento dos valores desconhecidos nos valores dos atributos é associado a cada exemplo um peso que é guardado na lista de classes juntamente com os valores das classes e os apontadores para os nós correspondentes. O peso de um exemplo representa a probabilidade de esse exemplo pertencer ao conjunto de exemplos onde se encontra.

No início do processo de construção da árvore de decisão todos os exemplos têm um peso igual a 1, ou seja, todos os exemplos pertencem ao conjunto de treino associado à raiz da árvore, com probabilidade 1. Quando, ao dividir um conjunto de exemplos T associados a um nó, não é conhecido o valor do atributo para um determinado exemplo, este é atribuído a todos os subconjuntos T_i resultantes da divisão. O seu peso também é alterado de forma a representar a probabilidade do exemplo pertencer a cada um dos subconjuntos. Este é calculado como:

$w' = w * prob(T_i)$

em que w representa o peso anterior do exemplo dado que este pode já ter sido alterado na divisão dos exemplos num nó antecessor. $prob(T_i)$ representa a probabilidade do exemplo pertencer ao subconjunto T_i calculada pela razão entre o número de exemplos conhecidos em T_i sobre o número de exemplo conhecidos em T.

A principal dificuldade no tratamento dos valores desconhecidos nos atributos reside na forma como o mesmo exemplo é atribuído a todos os subconjuntos resultantes da divisão.

No sistema C4.5 a árvore de decisão é construída em "primeiro em profundidade" em que cada ramo de um nó é explorado até ao fim antes de se dar início à exploração dos outros ramos. Assim, os exemplos com valores desconhecidos são atribuídos ao primeiro subconjunto e os seus pesos modificados. Este subconjunto é então utilizado para a construção da sub-árvore com raiz no primeiro filho. Após esta sub-árvore ter sido construída, os exemplos são atribuídos ao segundo subconjunto e os seus pesos novamente alterados. Este subconjunto é então utilizado para a construção da sub-árvore com raiz no segundo filho, e assim sucessivamente.

Contudo, no algoritmo paralelo a árvore é construída em "primeiro em largura". Isto implica que, antes da sub-árvore com raiz no primeiro filho ser completamente construída, os exemplos com valores desconhecidos são necessários para a construção dos restantes filhos e com os seus pesos alterados. Desta forma, quando não é conhecido o valor do atributo escolhido para um determinado exemplo, é atribuída uma cópia desse exemplo a cada um dos subconjuntos criados com a divisão. São criadas novas entradas na lista de classes, com os pesos do exemplo alterados para cada subconjunto, e associadas aos novos nós. Os valores dos atributos também são copiados para novas entradas nas listas de atributos em cada um dos nós e que apontam para as entradas correspondentes da lista de classes. Desta forma, os filhos de um nó cobrem subconjuntos distintos do conjunto de exemplos cobertos pelo nó.

Por fim, os pesos devem também ser considerados para a avaliação das divisões do conjunto de exemplos cobertos por um nó. A avaliação das divisões é realizada com base nas distribuições dos exemplos pelas classes para cada atributo. Para considerar os valores desconhecidos, estas distribuições são obtidas, não considerando o número de exemplos, mas a soma dos seus pesos. Também, as estruturas associadas aos nós da árvore onde são guardadas as distribuições dos exemplos guardam, para cada atributo e separadas das restantes distribuições, as distribuições dos exemplos pelas classes com valores desconhecidos no atributo.

Capítulo 6

Avaliação do Algoritmo

Neste capítulo realiza-se uma análise do desempenho do algoritmo de construção de árvores de decisão em paralelo desenvolvido. Inicia-se o capítulo pela descrição do ambiente onde foram realizados os diversos testes ao algoritmo assim como o conjunto de dados utilizados. A seguir apresentam-se os resultados obtidos com a execução do algoritmo. Para além dos resultados do desempenho global do algoritmo, apresentam-se também os resultados das suas fases mais importantes, nomeadamente, a ordenação paralelo dos atributos contínuos, a fase de crescimento e a fase de simplificação da árvore de decisão. Estes resultados são também comparados com as estimativas de desempenho realizados para cada uma das fases do algoritmo.

6.1 Metodologia utilizada

Nesta secção descreve-se o ambiente no qual foram conduzidas as várias experiências com o objectivo de avaliar o comportamento do algoritmo. Apresenta-se o ambiente de execução onde foi implementado o algoritmo e o conjunto de dados utilizado nos testes realizados. São também apresentados, de forma sucinta, as medidas utilizadas na avaliação do desempenho, como estas foram determinadas e que conclusões se podem tirar a partir dos seus resultados.

6.1.1 Ambiente de execução

O algoritmo de construção de arvores de decisão em paralelo foi implementado utilizando as primitivas standard do *Message Passing Interface (MPI)* [Pac97]. A utilização do MPI permite que a implementação do algoritmo seja facilmente utilizada numa grande variedade de máquinas e ambientes paralelos de memória distribuída. Os resultados apresentados neste capítulo foram obtidos a partir de experiências realizadas num PC servidor com quatro CPUs Pentium Pro, cada um à velocidade de 200MHz

Atributo	**Descrição**	**Valor**
Salary	Salário	Uniformemente distribuído entre 20000 e 150000
Commission	Comissão	$salary \geq 75000 => salary = 0$ senão uniformemente distribuído entre 10000 e 75000
Age	Idade	Uniformemente distribuído entre 20 e 80
Elevel	Nível de educação	Uniformemente escolhido entre 0 e 4
Car	Idade do carro	Uniformemente escolhido entre 1 e 20
Zipcode	Código postal	Uniformemente escolhido entre 9 códigos disponíveis
Hvalue	Valor da casa	Uniformemente distribuído entre 0.5k100000 e 1.5k100000 onde $k \in 0...9$ dependo do *Zipcode*
Hyears	Idade da casa actual	Uniformemente distribuído entre 1 e 30
Loan	Total do empréstimo	Uniformemente distribuído entre 0 e 500000

Tabela 6.1: Descrição dos atributos do conjunto de dados *synthetic*.

e com 256Kb de memória cache. O servidor dispõe também de 256MB de memória e corre com o sistema operativo Linux 2.2.12 fornecido pela distribuição standard da RedHat Linux realease 6.0.

6.1.2 Dados utilizados

Com o objectivo de avaliar experimentalmente o desempenho do algoritmo desenvolvido foi utilizado o conjunto de dados *synthetic* [AIS93]. Este conjunto de dados foi utilizado no teste de vários sistemas de construção de árvores de decisão em paralelo para os quais era necessário um grande conjunto de dados, como nos sistemas propostos em [SAM96] e [SHKS99].

O conjunto de dados synthetic é um conjunto de dados artificial em que os valores dos atributos, que descrevem cada exemplo, são gerados aleatoriamente. Cada exemplo é descrito por nove atributos, apresentados na tabela 6.1, em que os atributos *elevel*, *car* e *zipcode* são atributos discretos e os restantes contínuos. A classe de cada exemplo é determinada através de uma função dos seus atributos. Em [AIS93] são propostas dez funções diferentes para a classificação dos exemplos e que variam na sua complexidade. Para os conjuntos de exemplos utilizados nos testes apresentados neste capítulo foi considerada a mesma função utilizada para os testes dos sistemas apresentados em [SAM96] e [SHKS99]. Esta função é denominada em [AIS93] por *Função 2*:

Função 2:

$$
\begin{array}{ll}
GrpA: & ((age < 40) \wedge (50K \leq salary \leq 100K)) \vee \\
& ((40 \geq age < 60) \wedge (75K \leq salary \leq 125K)) \vee \\
& ((age \geq 60) \wedge (25K \leq salary \leq 75K))
\end{array}
$$

Esta função classifica cada exemplo num de dois grupos possíveis: *Grupo A* e *Grupo B*. Se os valores dos atributos que descrevem um exemplo satisfazem as condições da função então este é classificado no *Grupo A*, se não, é classificado no *Grupo B*. As árvores de decisão construídas a partir dos conjuntos de treino gerados utilizando esta função são relativamente pequenas.

6.1.3 Medidas de desempenho

Um algoritmo de indução de árvores de decisão é normalmente avaliado segundo três parâmetros:

- A **Precisão** que representa o grau de confiança de um procedimento de classificação e que, normalmente, é medida como a razão entre o número de exemplos correctamente classificados sobre o número total de exemplos.
- O **Tamanho** da árvore de decisão que está directamente relacionado com a facilidade de compreensão da árvore e medido como o número total de nós da árvore.
- O **Tempo de Aprendizagem** que representa o tempo necessário para o algoritmo construir a árvore de decisão.

No algoritmo desenvolvido neste trabalho consideram-se o mesmo critério de avaliação e os mesmos tipos de divisões dos conjuntos de exemplos que os considerados no sistema C4.5. Desta forma as árvores de decisão obtidas apresentam a mesma precisão de classificação e o mesmo tamanho que as obtidas com o sistema C4.5. O algoritmo de indução de árvores de decisão do C4.5 já foi alvo de diversos testes que demonstraram que as árvores obtidas com este sistema apresentam uma boa precisão de classificação e tamanho idêntico às árvores obtidas com outros algoritmos [Qui92, MST94, Rug00]. Desta forma, a análise do desempenho do algoritmo desenvolvido neste trabalho incide apenas sobre o tempo de aprendizagem, isto é, sobre o seu tempo de execução.

Com base nos tempos de execução do algoritmo são determinadas outras duas medidas, o *speedup* e a *escalonabilidade*, utilizadas na análise de desempenho de algoritmos paralelos. A análise de desempenho de um algoritmo paralelo foi já discutida no terceiro capítulo (secção 3.3) e onde estas duas medidas são descritas com maior detalhe. Neste capítulo é apenas fornecida uma descrição muito resumida das três medidas utilizadas na avaliação do algoritmo (tempo de execução, speedup e escalonabilidade) e o que estas representam.

6.1.3.1 Tempos de execução

Como o próprio termo indica, o tempo de execução representa a quantidade de tempo despendido na execução de todo ou de uma parte do algoritmo. Os intervalos de tempo foram medidos através de chamadas à função *gettimeofday()* disponível nas bibliotecas *standard* de linguagem C, e que retorna a hora do sistema com uma precisão até ao milésimo de segundo. Para além do tempo de execução total, foram também medidos os tempos de execução para diversas fases do algoritmo como, por exemplo, o tempo despendido na fase de crescimento da árvore de decisão ou o tempo despendido nas comunicações entre os processos.

O tempo de execução permite verificar quais as fases do algoritmo que mais influenciam a sua execução e, desta forma, o seu desempenho. Ainda, a partir dos tempos de execução é possível determinar-se outras medidas utilizadas na análise do desempenho do algoritmo como o speedup e a escalonabilidade.

6.1.3.2 Speedup

O *Speedup* é uma medida utilizada na avaliação do desempenho de algoritmos paralelos. Este representa o ganho obtido, em relação ao tempo de execução do algoritmo série, quando o algoritmo é executado com mais do que um processo. O speedup é determinado pela razão entre o tempo de execução do algoritmo série sobre o tempo de execução do algoritmo executado com mais do que um processo. Isto é, se $T_s(n)$ representar o tempo de execução do algoritmo série e $T_p(n,p)$ representar o tempo de execução do algoritmo executado com p processos então o speedup é determinado como:

$$S(n,p) = \frac{T_s(n)}{T_p(n,p)} \tag{6.1}$$

Para o tempo de execução do algoritmo série - $T_s(n)$ - é usual considerar-se o tempo de execução do algoritmo paralelo executado com apenas um processo. Na avaliação do algoritmo desenvolvido, para além de ser considerado o tempo de execução com apenas um processo, para o calculo do speedup considera-se também para o tempo de execução do algoritmo sequencial o tempo de execução do sistema C4.5, permitindo a comparação entre os dois algoritmos.

Idealmente espera-se que, para um determinado conjunto de dados, ao executar o algoritmo com dois processos, o seu tempo de execução seja metade do tempo de execução com apenas um processo. Isto é, o speedup ideal é igual ao número de processos utilizados na execução do algoritmo - $S(n,p) = p$. Contudo, raramente se obtêm speedups ideais. Normalmente, numa implementação paralela existe algum processamento adicional que não é realizado no algoritmo série como, por exemplo, as comunicações entre processos. Este processamento adicional é denominado de *overhead* e, tipicamente, o seu custo cresce com o aumento do número de processos utilizados provocando uma diminuição dos speedups obtidos.

Desta forma, para além de medir o ganho obtido pela execução do algoritmo com um número crescente de processos, o speedup pode também ser utilizado para avaliar o peso do overhead no ganho obtido.

6.1.3.3 Escalonabilidade

Uma outra medida bastante utilizada na avaliação do desempenho de programas paralelos é a escalonabilidade. Um programa é escalonável se for possível manter o seu tempo de execução aproximadamente constante à medida que a dimensão do problema é aumentada.

Num algoritmo de indução de árvores de decisão, uma estimativa da escalonabilidade obtém-se calculando a razão entre o tempo de execução do algoritmo com k processos sobre o tempo de execução com um único processo, mantendo constante o conjunto de dados em cada processo. Isto é, se $T_p(m,p)$ representa o tempo de execução do algoritmo em que cada processo é responsável pelo processamento de m exemplos, então a escalonabilidade do algoritmo, para um determinado número de processos, é determinada pela equação:

$$E(m,p) = \frac{T_p(m,p)}{T_p(m,1)} \tag{6.2}$$

Na situação ideal, a escalonabilidade apresenta um valor igual a 1. Isto é, o tempo de execução do algoritmo executado com k processos é igual ao tempo de execução com 1 processo mas com $1/k$ dos dados. Nesta situação, é possível manter constante o tempo de execução do algoritmo para qualquer dimensão do conjunto de dados utilizando, para tal, um número adequado de processos. Contudo, o overhead introduzido pela implementação paralela provoca um aumento do tempo de execução ao ser utilizado um maior número de processos. Quanto maior for overhead maior é o valor da escalonabilidade. Desta forma, para além de se avaliar a capacidade do algoritmo de lidar com problemas de maiores dimensões, a escalonabilidade permite também avaliar a taxa de crescimento do overhead com o aumento do número de processos.

6.2 Análise de resultados

Nesta secção apresenta-se uma síntese dos resultados obtidos nas várias experiências realizadas com algoritmo desenvolvido. O desempenho do algoritmo é avaliado através da análise da variação dos tempos de execução obtidos para diferentes dimensões do problema (utilizando conjuntos de treino de diferentes tamanhos e um número variável de processos). A análise do desempenho inclui também uma avaliação das características de speedup e escalonabilidade apresentadas pelo algoritmo.

Na figura 6.1 estão representados os gráficos com os tempos de execução obtidos para a execução do algoritmo com o conjunto de dados synthetic com 100 000, 200 000 e 400 000 exemplos, e com 1, 2, 3 e 4 processos. Como esperado, a partir dos gráficos

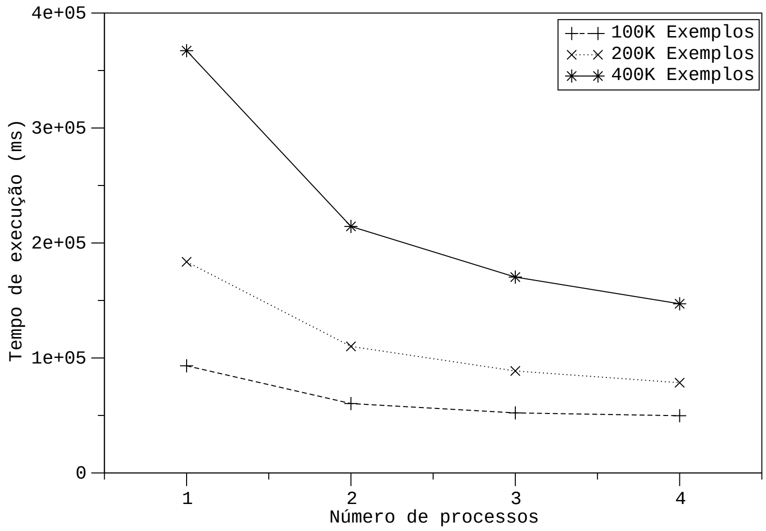


Figura 6.1: Tempos de execução obtidos para o algoritmo de construção de árvores de decisão em paralelo.

da figura 6.1 observa-se uma diminuição do tempo de execução do algoritmo à medida que se utiliza um maior número de processos.

A partir dos tempos de execução e da equação 6.1 determinaram-se os speedups obtidos pelo algoritmo para cada um dos testes realizados, representados nos gráficos da figura 6.2. Como se observa, o algoritmo apresenta bons speedups em que os melhores resultados são obtidos para a execução do algoritmo com o maior conjunto de treino (400 000 exemplos). A diminuição do speedup observada para a execução do algoritmo com 3 e 4 processos é explicada pelo overhead associado às comunicações entre processos. Como se verá mais à frente, este overhead deve-se principalmente à partilha das atribuições dos exemplos aos nós durante a fase de crescimento da árvore de decisão.

Na figura 6.3 são representados os speedups obtidos para o algoritmo mas considerando para o tempo de execução do algoritmo série os tempos de execução obtidos com o sistema C4.5. Os tempos de execução foram também obtidos para a execução dos algoritmos com o mesmo conjunto de dados "synthetic" com 100 000, 200 000 e 400 000 exemplos. Como se verifica, os resultados obtidos são inferiores aos da figura 6.2 em que é considerado para o tempo do algoritmo série o tempo de execução do algoritmo paralelo executado com apenas um processo. Estes resultados devem-se principalmente aos elevados tempos de execução verificados na fase de simplificação da árvore. Como foi referido no capítulo anterior, a simplificação da árvore poderia ser realizada através de um algoritmo série idêntico ao utilizado pelo C4.5. Contudo, foi implementada uma solução paralela de forma a aproveitar a distribuição dos dados e do processamento, obtendo-se a vantagem de dispor de p processos responsáveis, cada um, pelo processamento de $1/p$ dos dados. Mas, como é discutido mais à frente, nesta fase do algoritmo o overhead em relação ao processamento realizado pelo algoritmo do C4.5 é bastante elevado o que não justifica a utilização do algoritmo paralelo para a

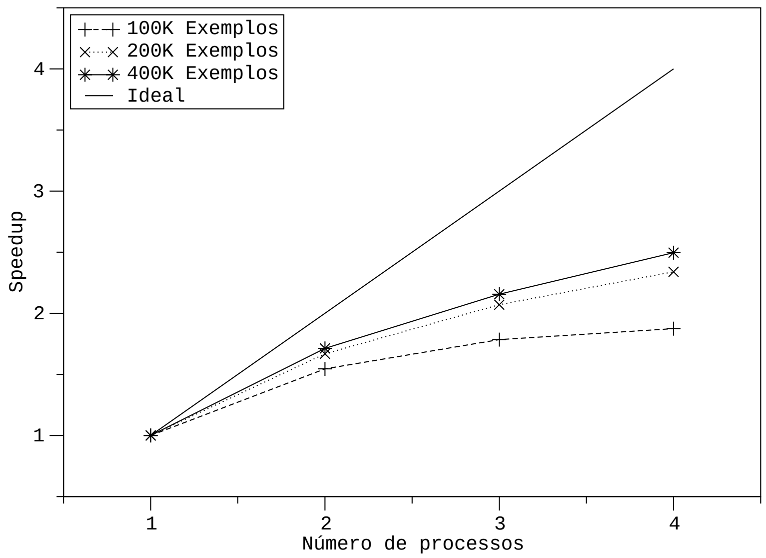


Figura 6.2: Speedups obtidos para o algoritmo de construção de árvores de decisão em paralelo.

dimensão dos conjuntos de dados utilizados.

Contudo, os speedups obtidos mostram melhores tempos de execução para o algoritmo paralelo para os conjuntos de dados com 200 000 e 400 000 exemplos indicando que poderão ser obtidos melhores resultados para conjuntos de dados ainda maiores, para os quais já se justifica realizar a simplificação da árvore em paralelo.

Os resultados representados nos gráficos da figura 6.4 mostram a escalonabilidade calculada para o algoritmo a partir da equação 6.2. Para determinar a escalonabilidade, foram medidos os tempos de execução em que, para um de número de exemplos constante em cada processo de 50 000, 75 000 e 100 000 exemplos, o algoritmo foi executado com 1, 2, 3 e 4 processos. A partir dos resultados obtidos verifica-se que o algoritmo escalona bem. Como é discutido mais à frente, a subida dos valores da escalonabilidade à medida que é aumentado o número de processos deve-se principalmente à partilha das distribuições dos exemplos durante a fase de crescimento da árvore de decisão.

Na figura 6.5 são representadas as curvas de speedup para o conjunto de treino com 400 000 exemplos relativamente ao tempo de execução total do algoritmo e das suas três fases mais importantes (a ordenação dos atributos contínuos, o crescimento e a simplificação da árvore de decisão). Como se pode verificar, o comportamento do algoritmo é influenciado principalmente pelo comportamento da fase de crescimento da árvore de decisão.

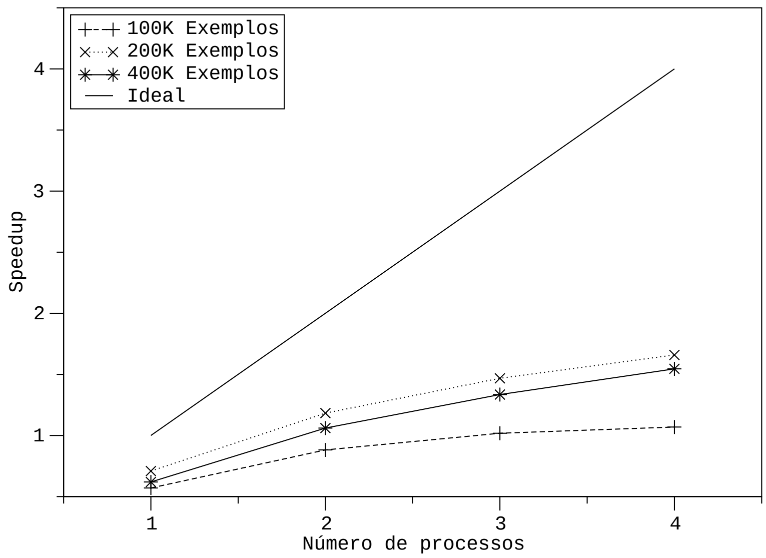


Figura 6.3: Speedups obtidos para o algoritmo relativamente aos tempos de execução do sistema C4.5.

6.2.1 Ordenação dos atributos contínuos

O algoritmo de ordenação paralela utilizado na ordenação dos valores dos atributos contínuos foi desenvolvido a partir do algoritmo apresentado em [DNS91]. Este algoritmo pode ser resumido nos passos seguintes:

1. Em paralelo, cada processo reúne um conjunto aleatório de valores (amostras) a partir da sua lista local de valores do atributo.

2. Em paralelo, cada processo ordena o seu conjunto local de amostras. Em seguida, os vários conjuntos são reunidos num único processo que tem como resultado um novo conjunto ordenado com as amostras de todos os processos. A partir deste conjunto é determinado o vector de divisão que é comunicado a todos os processos.

3. Em paralelo, e após a redistribuição dos valores do atributo pelos processos baseada no vector de divisão encontrado, cada processo ordena localmente a sua lista de valores do atributo.

O factor que mais influencia o desempenho do algoritmo de ordenação é o número de amostras utilizado para a construção do vector de divisão. Existe um compromisso entre o custo de processar um maior número de amostras e uma distribuição mais uniforme dos valores do atributo pelos processos. Uma má amostragem dos valores do atributo traduz-se directamente na diferença (ou desvio) do número de valores atribuídos a cada processo.

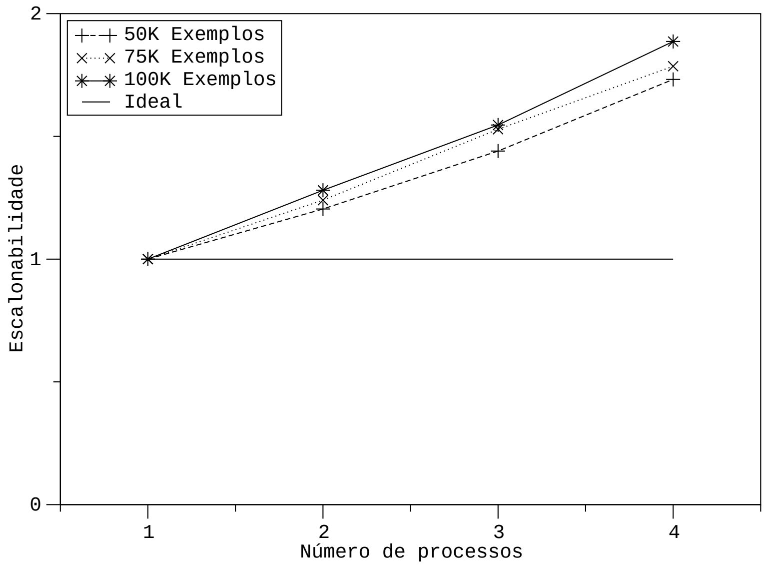


Figura 6.4: Escalonabilidade obtida para o algoritmo de construção de árvores de decisão em paralelo.

O *desvio*, s, pode ser definido como a razão entre o número máximo de valores por processo e o número médio de valores também por processo. Se existirem p processos, e N valores, então o número médio de valores por processo é dado por N/p. Então, pela definição de s, o número máximo de valores num processo é dado por $s*N/p$. Em [DNS91] é derivado o número de amostras, N_s, que deve ser utilizado para garantir um desvio s com uma probabilidade q. O número de amostras N_s em função de p, s e q é dado por:

$$N_s = \frac{2 * ln(p/q)}{s * (1 - 1/s)^2} \tag{6.3}$$

Considerando o desvio s, o custo de execução do algoritmo de ordenação pode ser aproximadamente estimado como:

- $N_s * C_1$ para obter o conjunto aleatório de amostras (passo 1).
- $N_s * log_2(N_s) * C_2$ para ordenar localmente o conjunto de amostras (passo 2).
- A reunião do conjunto de amostras num único processo (passo 2) é realizada utilizando um esquema de comunicação em árvore onde, em cada fase de comunicação, os processos pais combinam os conjuntos dos processos filhos num só conjunto ordenado de valores. Logo, é necessário, para a reunião das amostras, um total de $(p-1) * N_s * C_3$ para os processos enviarem os seus conjuntos para os pais e um total de $2*(p-1)*N_s*C_4$ para a junção, pelos pais, dos conjuntos recebidos num conjunto ordenado de amostras.

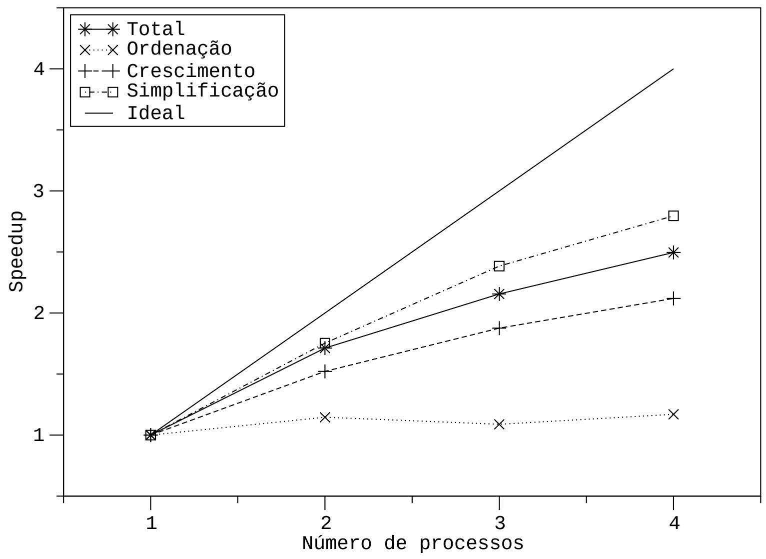


Figura 6.5: Speedups obtidos para as três fases principais do algoritmo.

- $(p-1) * log_2(p) * C_5$ para a comunicação do vector de divisão V_s a todos os processos, assumindo que é utilizada uma operação colectiva de difusão estruturada em árvore (passo 3).
- um total de $N/p * log_2(p) * C_6$ para determinar a atribuição aos processos dos valores das listas locais do atributo, em que N é o número total de exemplos do conjunto de treino (passo 4).
- um total de $3 * N/p * log_2(p) * C_7$ para a redistribuição dos valores pelos processos assumindo que é utilizada uma comunicação todos para todos (passo 4).
- um total de $s * N/p * (log_2(N/p) + 1) * C_8$ para a inserção dos valores recebidos nas listas locais do atributo e para a actualização das entradas correspondentes da lista de classes (passo 4).
- $s * N/p * log_2(s * N/p) * C_9$ para a ordenação local dos valores do atributo (passo 4).

As constantes C_x nas estimativas anteriores representam o custo unitário de execução de cada passo do algoritmo. Por exemplo, para primeiro passo $N_s * C_1$, a constante C_1 representa o custo de se obter uma única amostra. Nas estimativas de desempenho realizadas as constantes C_3, C_5 e C_7 foram aproximadas pelo valor 10 e as restantes pelo valor 1, uma vez que as operações de comunicação apresentam um custos de execução mais elevados que as operações realizadas localmente por cada processo.

A questão que agora se levanta é encontrar um valor para o desvio s e, desta forma, para N_s que dê origem ao menor custo de execução do algoritmo. Para determinar o melhor valor de s os vários passos do algoritmo são reunidos numa só equação $C(s)$ que traduz o custo de execução do algoritmo para um determinado valor de s. Esta equação

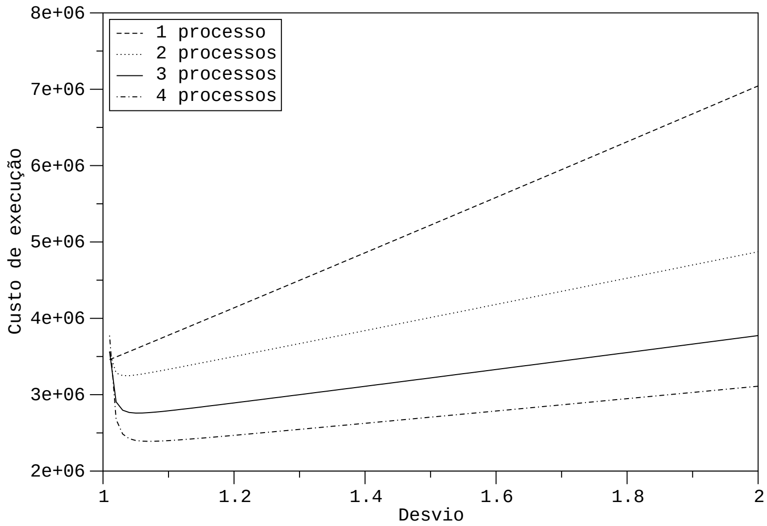


Figura 6.6: Variação do custo de execução do algoritmo de ordenação em função do *desvio*, estimado para um conjunto de treino com 100 000 exemplos.

apresenta termos em s, $1/s$, $1/s^2$ e $log_2(s)$. Para determinar o valor óptimo do desvio poderiam ser utilizados métodos numéricos para encontrar as raízes da derivada da equação $C(s)$. Contudo, e dado que $C(s)$ é já uma aproximação do custo de execução do algoritmo, é bastante mais simples calcular $C(s)$ para um intervalo de valores aceitáveis de s e encontrar o valor de s que dê origem ao menor valor de $C(s)$. Este valor pode então ser utilizado pelo algoritmo para determinar o número apropriado de amostras.

Utilizando a abordagem anterior para a procura do melhor valor de s, e para diferentes números de processos e dimensões dos dados, obteve-se, para um custo de execução do algoritmo próximo do óptimo, um valor de s aproximadamente igual a 1,03 (figura 6.6). Este resultado confirma-se com os resultados experimentais apresentados na tabela 6.2. Para a obtenção dos resultados, o algoritmo de ordenação foi executado com 2, 3 e 4 processos para os conjuntos de dados synthetic com 100 000, 200 000 e 400 000 exemplos. Também, neste conjunto de experiências o valor de s variou entre 1,01 e 1,05.

Para analisar o desempenho do algoritmo de ordenação foram medidos os tempos de execução para o conjunto de testes realizados com os três conjuntos de dados "synthetic" com 100 000, 200 000 e 400 000 exemplos. Com base nos tempos de execução obtidos foram determinados os speedups para o algoritmo a partir da equação 6.1, apresentados nos gráficos da figura 6.7. Na figura 6.8 representam-se os speedups estimados para o algoritmo determinados a partir da estimativa realizada do seu custo de execução. Como se verifica pelos gráficos de ambas as figuras, o comportamento do algoritmo corresponde ao estimado. Os baixos valores do speedup verificados devem-se principalmente aos custos associados às comunicações. Enquanto que o número de valores e, desta forma, a quantidade de processamento em cada processo decresce à medida que é utilizado um maior número de processos, os custos associados às

100K Exemplos		200K Exemplos		400K Exemplos	
2 Processos					
S	Sort	S	Sort	S	Sort
1,01	40925	1,01	194673	1,01	1176820
1,02	**40770**	1,02	**180589**	1,02	1432997
1,03	40922	1,03	186786	1,03	1250632
1,04	42180	1,04	187566	1,04	**919308**
1,05	41009	1,05	193660	1,05	1284413
3 Processos					
1,01	40529	1,01	199611	1,01	1097798
1,02	**40309**	1,02	**179969**	1,02	1571713
1,03	40460	1,03	188040	1,03	1001741
1,04	41818	1,04	191232	1,04	**921940**
1,05	40655	1,05	192622	1,05	1340864
4 Processos					
1,01	40538	1,01	196073	1,01	1231810
1,02	40251	1,02	**178116**	1,02	1318534
1,03	40505	1,03	188184	1,03	1048039
1,04	41070	1,04	189383	1,04	**937489**
1,05	**40055**	1,05	193779	1,05	1146133

Tabela 6.2: Tempos de execução obtidos para o algoritmo de ordenação em função do *desvio*.

comunicações crescem com o aumento do número de processos. Desta forma, verifica-se uma pequena diminuição no tempo de execução do algoritmo com o aumento do número de processos.

Contudo, são esperados melhores resultados para problemas de maior dimensão como se verifica pelos resultados apresentados na tabela 6.3. Os speedups foram estimados para a execução do algoritmo para um conjunto de exemplos com $1 * 10^6$ exemplos e um número de processos entre 5 e 30. Estes resultados são bastante próximos dos resultados obtidos para o algoritmo de ordenação apresentado em [DNS91].

Número de processos	Número de amostras	Speedup
5	3706	1
10	5293	1,52
15	6221	1,92
20	6879	2,20
25	7390	2,37
30	7807	2,46

Tabela 6.3: Speedups estimados para o algoritmo de ordenação.

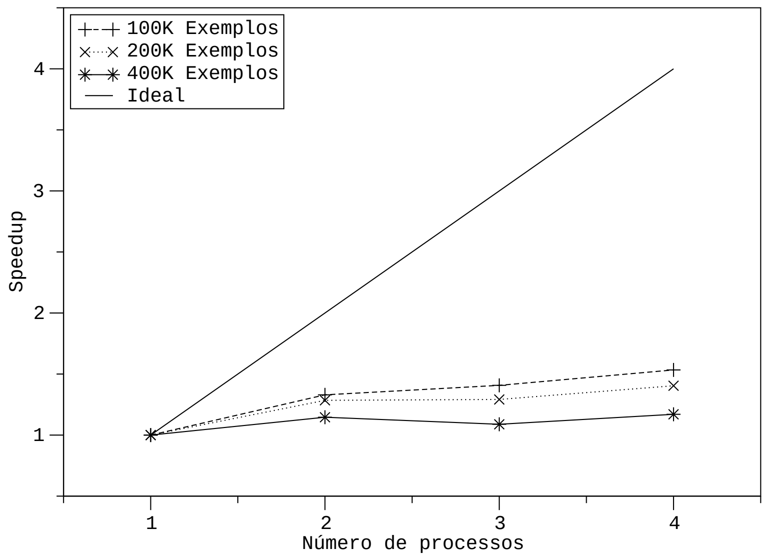


Figura 6.7: Speedups obtidos para o algoritmo de ordenação.

6.2.2 Crescimento da árvore de decisão

O processo de construção de um nó da árvore de decisão em paralelo pode ser resumido nos passos seguintes:

```
1. Constrói_Nó(N).
   1.1. Se todos os exemplos cobertos por N pertencem à mesma classe Ci:
        1.1.1. Retorna uma folha.
   1.2. Se não:
        1.2.1. Para cada atributo A:
               1.2.1.1. Calcula o ganho da divisão baseada em A.
        1.2.2. Encontra o atributo Ai que melhor divide os exemplos.
        1.2.3. Cria os nós filhos de N de acordo com a melhor divisão.
        1.2.4. Divide a lista do atributo Ai.
        1.2.4. Partilha por todos os processos da atribuição global
               dos exemplos.
        1.2.5. Divide as restantes listas de atributos.
        1.2.6. Para cada filho Ni de N:
               1.2.6.1. Encontra as distribuições globais para Ni.
        1.2.7. Retorna N.
```

Com base nos passos anteriores e considerando que o número de exemplos é bastante superior ao número de atributos e ao número de processos, o custo de execução para a construção de um nó da árvore de decisão pode ser estimado como:

- $N/p * C_1$ para o cálculo dos ganhos dos atributos contínuos e para a divisão das

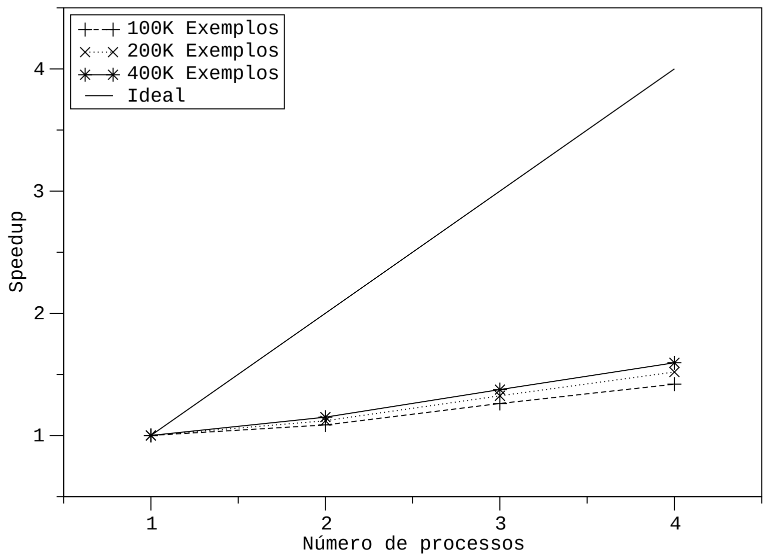


Figura 6.8: Speedups estimados para o algoritmo de ordenação.

várias listas de atributos, em que N representa o número de exemplos cobertos pelo nó e p o número de processos.

- $(p-1)/p * N * C_2$ para a reunião global das atribuições ordenadas pelo índice global dos exemplos.
- $N/p * log_2(N) * C_3$ para, na pior situação, procurar a atribuição dos exemplos.
- $log_2(p) * C_4$ para determinar as distribuições globais.

Considerando os passos anteriores obtém-se a seguinte equação para a estimativa do custo de construção de um único nó da árvore de decisão, onde as constantes C_x representam o custo de execução unitário associado cada passo:

$$T_p(n,p) = N/p * C_1 \; + \; (p-1)/p * N * C_2 \; + \; N/p * log_2(N) * C_3 \; + \; log_2(p) * C_4 \quad (6.4)$$

Para as estimativas de desempenho as constantes C_2 e C_4, associadas a custos de comunicações, foram substituídas pelo valor 4 e as restantes pelo valor 1.

Na figura 6.9 estão representados os tempos de execução da fase de crescimento da árvore de decisão do algoritmo em função do número de processos para cada um dos conjuntos de treino utilizados (100 000, 200 000 e 400 000 exemplos). Por simples inspecção dos gráficos é claramente notada uma descida do tempo de execução do algoritmo à medida que são utilizados mais processos na sua execução.

Na figura 6.10 estão representados os speedups calculados a partir dos resultados anteriores. Na figura 6.11 apresentam-se os speedups estimados para a fase de crescimento

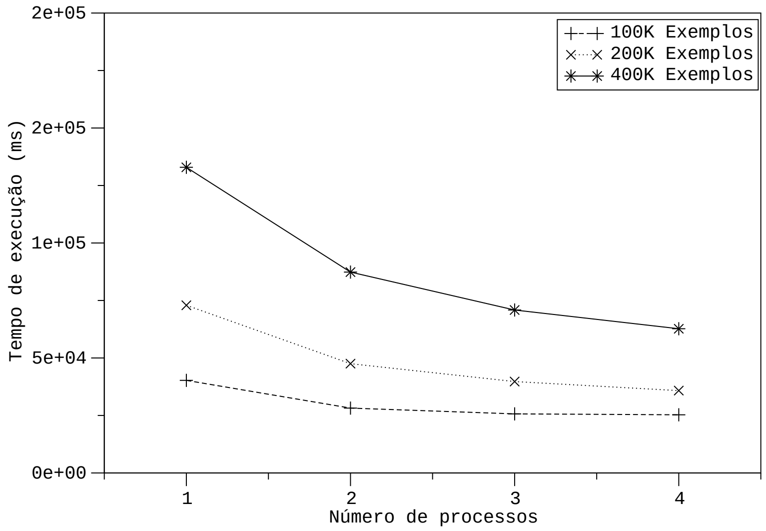


Figura 6.9: Tempos de execução obtidos para a fase de crescimento da árvore de decisão.

da árvore calculados a partir da equação 6.4. Os resultados experimentais são bastante próximos dos custos estimados em que os melhores speedups são obtidos para o maior conjuntos de exemplos. Contudo, com o aumento do número de processos e, desta forma, com uma diminuição do número de exemplos em cada processo, o overhead associado às comunicações e à partilha das atribuições dos exemplos aos nós provocam uma diminuição do speedup, como também é verificado nas curvas da figura 6.10.

A figura 6.12 mostra a percentagem do tempo associado ao overhead em relação ao tempo de execução utilizado na fase de crescimento da árvore. Para os conjuntos de dados com 100 000 e 200 000 exemplos é clara uma subida do overhead com o aumento do número de processos. Para estes dois casos o overhead deve-se principalmente às comunicações entre os processos que cresce com o aumento do número de processos. Para execução do algoritmo em que é utilizado o conjunto de dados com 400 000 exemplos verifica-se uma diminuição do overhead. Neste caso o overhead deve-se principalmente à partilha das atribuições dos exemplos aos nós em que parte do processamento é realizado localmente e depende do número de exemplos em cada processo. Como, com o aumento do número de processos o número de exemplos em cada processo diminui, o overhead associado à partilha das atribuições também diminui.

Na figura 6.13 estão representados os gráficos dos speedups obtidos com o algoritmo paralelo comparativamente à fase de construção da árvore de decisão do sistema C4.5. Os tempos de execução do algoritmo paralelo representam o tempo de execução da fase de ordenação dos atributos contínuos e da fase de construção da árvore de decisão permitindo, desta forma, uma melhor comparação dos dois algoritmos dado que a ordenação dos valores dos atributos contínuos é um passo essencial para a construção da árvore. Os resultados mostram que o algoritmo apresenta bons speedups e, como esperado, os melhores resultados são obtidos para os conjuntos de treino com 200 000

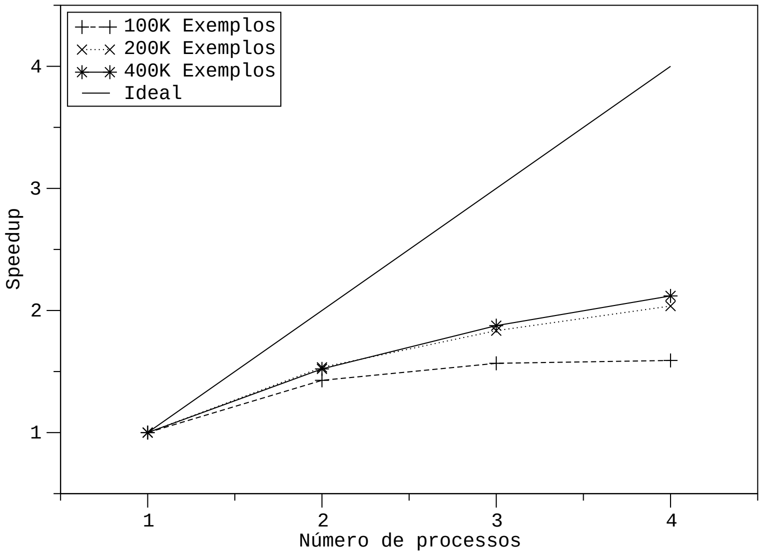


Figura 6.10: Speedups obtidos para a fase de crescimento da árvore de decisão.

e 400 000 exemplos. Ao contrário do esperado, foram obtidos melhores resultados para a execução do algoritmo com 200 000 exemplos que os obtidos com 400 000 exemplos. Esta situação deve-se à árvore construída a partir do conjunto de treino com 200 000 exemplos ser maior do que a obtida a partir do conjunto de dados com 400 000 exemplos. Logo, para o conjunto de treino com 200 000 exemplos é necessário um maior processamento na construção da árvore que, ao ser realizado em paralelo, dá origem a um maior ganho nos tempos de execução relativamente ao algoritmo do C4.5.

Com o objectivo de avaliar o comportamento do algoritmo com a variação do número de atributos contínuos, foram utilizados três conjuntos de dados "synthetic" com 200 000 exemplos cada. Cada exemplo no primeiro conjunto é descrito por 2 dos atributos contínuos, no segundo por 3 e no terceiro por 5. Em todos conjuntos foram também utilizados os três atributos discretos e os dois atributos contínuos utilizados pela *Função 2* para a classificação dos exemplos. Os speedups obtidos para a execução do algoritmo para cada um destes conjuntos estão representados nos gráficos da figura 6.14. Assim como esperado, os melhores resultados foram obtidos para o conjunto de dados com o maior número de atributos contínuos, dado que, para a avaliação dos atributos contínuos, é necessário um maior processamento.

Os resultados representados nos gráficos da figura 6.15 mostram a escalonabilidade calculada para a fase de crescimento da árvore de decisão considerando os tempos de execução em que, para um número constante de exemplos em cada processo de 50 000, 75 000 e 100 000 exemplos, o algoritmo foi executado com 1, 2, 3 e 4 processos. A partir dos resultados obtidos verifica-se que a fase de crescimento da árvore escalona bem. A subida dos valores da escalonabilidade à medida que o número de processos aumenta deve-se principalmente às comunicações entre os processos durante a construção da árvore. A comunicação que mais contribui para o overhead é a partilha das

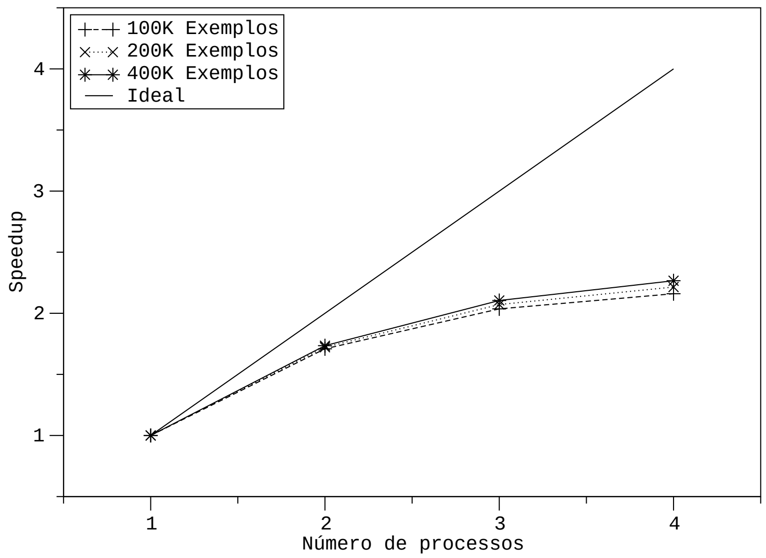


Figura 6.11: Speedups estimados para a fase de crescimento da árvore de decisão.

distribuições dos exemplos para determinar as distribuições globais.

6.2.3 Simplificação da árvore de decisão

A simplificação da árvore de decisão não é uma das fases onde o paralelismo é mais necessário. Contudo, esta fase foi também implementada em paralelo, seguindo a mesma abordagem utilizada na fase de crescimento da árvore, beneficiando da vantagem de dispor de p processos a processar $1/p$ dos exemplos cobertos por cada nó.

O custo de execução do processo de simplificação de um nó da árvore de decisão pode ser aproximadamente estimado por:

- $N/p*C_1$ para cada processo encontrar a distribuição local dos exemplos cobertos pelo nó em que N representa o número total de exemplos cobertos pelo nó e p o número de processos.
- $log_2(p) * C_2$ para encontrar as distribuições globais assumindo que é realizada uma comunicação colectiva de redução estruturada em árvore.
- $N/p * C_3$ para a divisão dos exemplos cobertos pelo nó para simplificar as sub-árvores com raiz nos nós filho do nó actual.

Considerando os custos anteriores, onde as constantes C_x representam o custo de execução unitário de cada passo, o custo estimado para a simplificação de um nó da árvore de decisão pode ser aproximado pela seguinte equação:

$$T_p(n,p) = N/p * C_1 \ + \ log_2(p) * C_2 \quad (6.5)$$

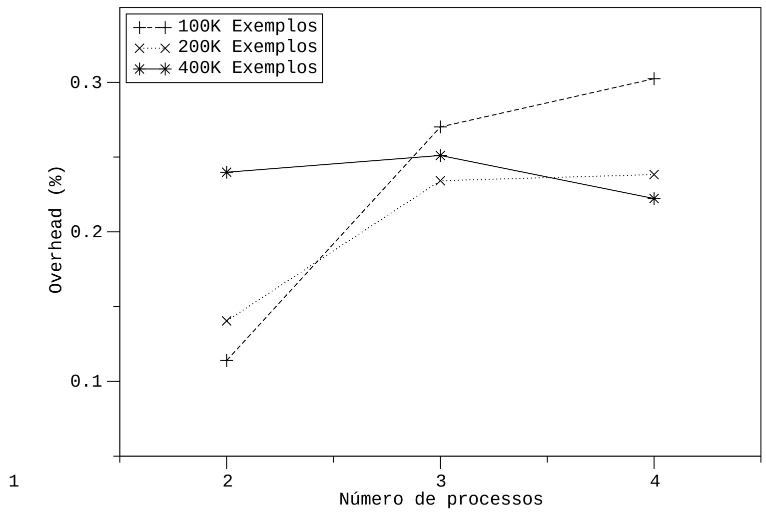


Figura 6.12: Percentagem de overhead na fase de crescimento da árvore de decisão.

Para as estimativas de desempenho da fase de simplificação a constante C_2, associada a custos de comunicações, foi substituída por 1000 e a constante C_1 por 1. A diferença elevada entre os valores das duas constantes deve-se à quantidade de processamento realizado localmente na fase de simplificação ser bastante menor do que nas restantes fases do algoritmo. Desta forma, as comunicações representam um maior peso no custo de execução.

É importante salientar que, na equação 6.5, a constante C_2 está associada ao custo das comunicações enquanto que a constante C_1 está associada ao custo de operações aritméticas e de atribuições de valores, realizadas durante o processo de simplificação ao serem percorridas as listas de valores dos atributos. Desta forma, espera-se que o valor da constante C_2 seja bastante maior que o valor de C_1 ($C_2 >> C_1$).

Na figura 6.16 estão representados os speedups obtidos com base nos tempos de execução da fase de simplificação da árvore de decisão do algoritmo em função do número de processos para cada um dos conjuntos de treino utilizados (100 000, 200 000 e 400 000 exemplos). Na figura 6.17 apresentam-se os speedups estimados para esta fase do algoritmo obtidos a partir da equação 6.5. Os resultados experimentais são bastante próximos dos custos estimados em que, como esperado, os melhores speedups são obtidos para o maior conjuntos de exemplos. A diminuição do speedup verificada para a execução do algoritmo com um maior número de processos deve-se, principalmente, ao overhead associado às comunicações para a partilha das distribuições dos exemplos no nó que cresce com o aumento do número de processos.

Na figura 6.18 são apresentados os resultados obtidos para a escalonabilidade da fase de simplificação da árvore de decisão considerando os tempos de execução em que, para um de número constante de exemplos em cada processo de 50 000, 75 000 e 100 000 exemplos, o algoritmo foi executado com 1, 2, 3 e 4 processos. A partir dos resultados obtidos verifica-se que a fase de crescimento da árvore escalona bem. A subida dos

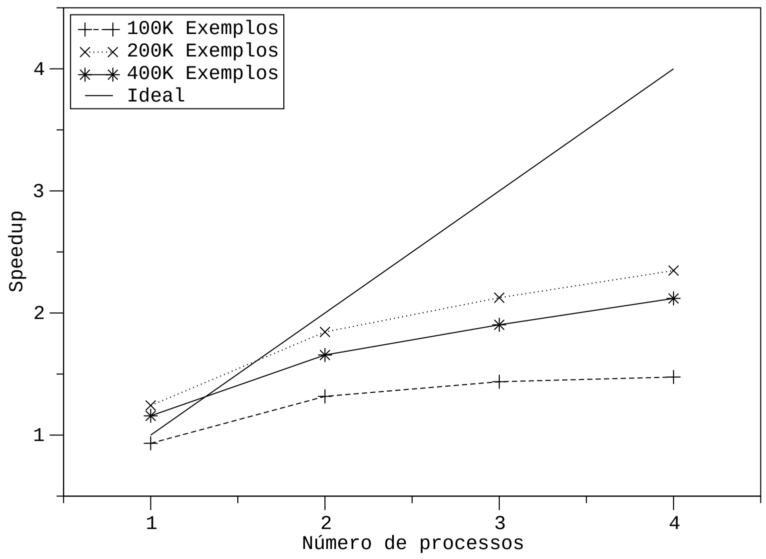


Figura 6.13: Speedups obtidos para a fase de crescimento da árvore de decisão em relação ao algoritmo do C4.5.

Número de processos	100K Exemplos	200K Exemplos	400K Exemplos
1	22726	40099	62125
2	14784	23674	35439
3	12867	18239	26061
4	12789	16391	22220
C4.5	4064	10468	16281

Tabela 6.4: Tempos de execução obtidos para a fase de simplificação do algoritmo paralelo e do sistema C4.5.

valores da escalonabilidade à medida que o número de processos aumenta deve-se principalmente às comunicações entre os processos para a partilha das distribuições.

Apesar dos bons resultados obtidos para os speedups, os tempos de execução da fase de simplificação são ainda bastante elevados quando comparados com os tempos do algoritmo do sistema C4.5. Os tempos de execução dos dois algoritmos para os conjuntos de dados "synthetic" com 100 000, 200 000 e 400 000 exemplos são apresentados na tabela 6.4. Os elevados tempos de execução observados, relativamente ao C4.5, devem-se principalmente ao processamento necessário para a utilização e manutenção das listas de valores dos atributos. Por exemplo, antes da simplificação dos filhos de um nó, os exemplos cobertos pelo nó são reorganizados. No sistema C4.5, a reorganização passa simplesmente por percorrer uma lista de apontadores para os exemplos e alterar a sua ordem. No algoritmo desenvolvido, para este passo é necessário percorrer cada uma das listas de atributos e reorganizar os seus valores. Contudo, a partir dos resultados obtidos e dos custos estimados para fase de simplificação, são previstos melhores resultados para conjuntos de dados maiores dos que os utilizados.

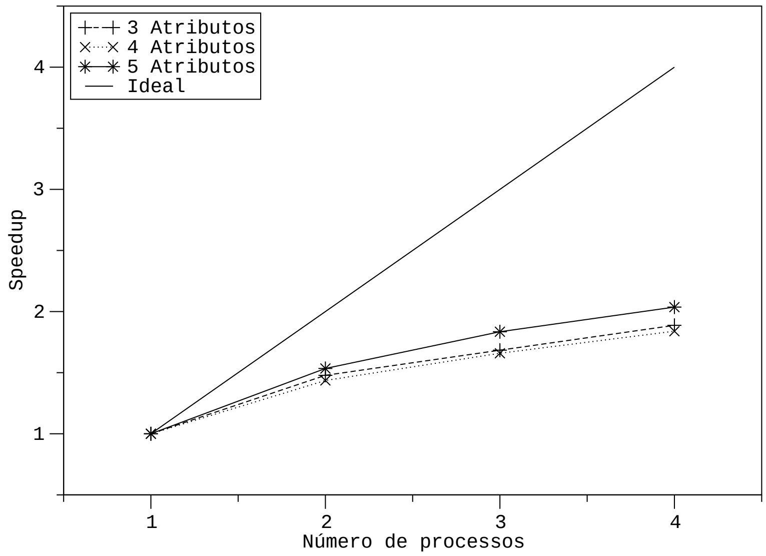


Figura 6.14: Speedups obtidos com a variação do número de atributos para a fase de crescimento da árvore de decisão.

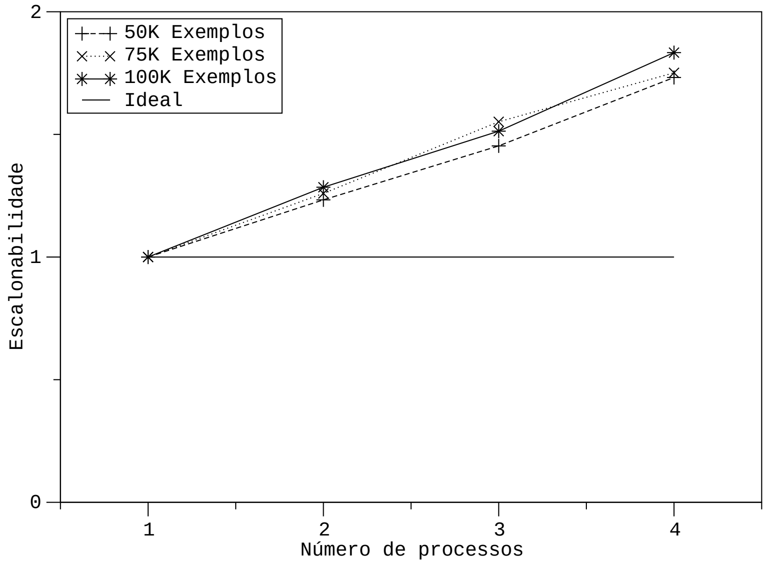


Figura 6.15: Escalonabilidade obtida para a fase de crescimento da árvore de decisão.

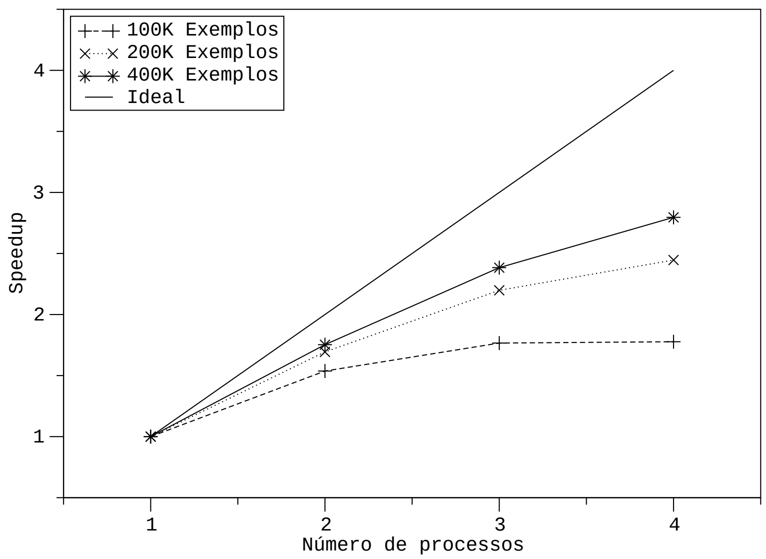


Figura 6.16: Speedups obtidos para a fase de simplificação da árvore de decisão.

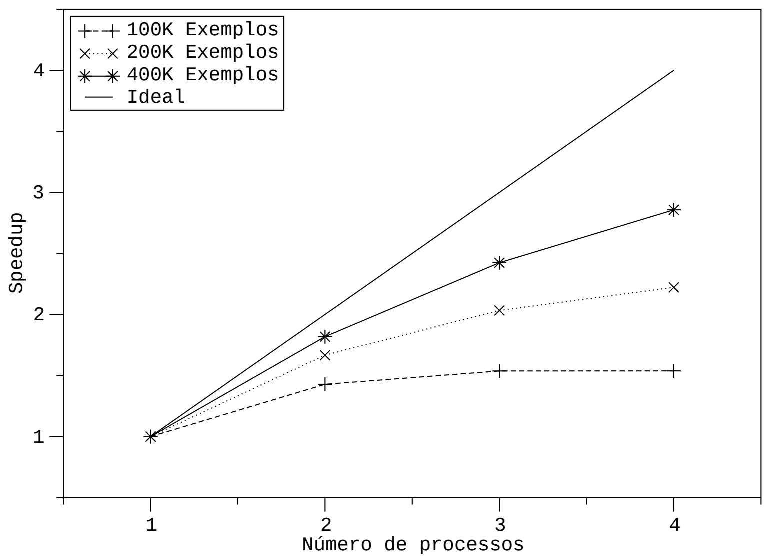


Figura 6.17: Speedups estimados para a fase de simplificação da árvore de decisão.

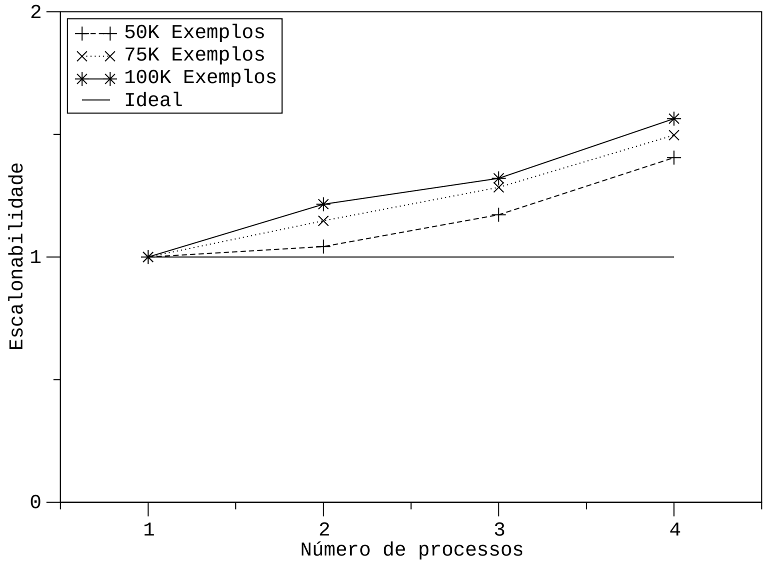


Figura 6.18: Escalonabilidade obtida para a fase de simplificação da árvore de decisão.

Capítulo 7

Conclusão

Neste trabalho foram apresentadas, de uma forma resumida, diversas abordagens para o desenvolvimento de algoritmos de construção de árvores de decisão em paralelo, agrupadas pelo tipo de abordagem utilizada: paralelismo de controlo, paralelismo nos dados e paralelismo híbrido. Da comparação entre as vantagens e desvantagens apresentadas pelas várias abordagens, conclui-se que o paralelismo nos dados com distribuição horizontal apresenta-se como a melhor abordagem para a construção de árvores de decisão em paralelo. Contudo, esta abordagem ainda apresenta alguns problemas, principalmente relacionados com o elevado volume de comunicações necessário para a construção de um nó da árvore. Foi também identificado que uma abordagem ao paralelismo híbrido é uma possível solução para o problema do overhead associado às comunicações das abordagens de paralelismo nos dados.

Considerando as diferentes abordagens, foi desenvolvido um novo algoritmo de construção de árvores de decisão em paralelo, onde são considerados os mesmos critérios de avaliação e as mesmas divisões do conjunto de exemplos que no algoritmo do sistema C4.5. As árvores de decisão obtidas são iguais às obtidas pelo C4.5, apresentando a mesma precisão na classificação.

Da implementação do novo algoritmo paralelo desenvolvido salientam-se ainda os seguintes pontos:

- Implementação de um novo algoritmo de construção de árvores de decisão em paralelo com distribuição horizontal dos dados.
- Ordenação dos valores dos atributos contínuos realizada uma única vez no início do processo de construção da árvore de decisão.
- Crescimento da árvore de decisão em "primeiro em largura".
- Capacidade de tratar valores desconhecidos nos atributos.
- Construção da árvore de decisão até uma profundidade pré determinada.

O desenvolvimento do algoritmo paralelo teve como resultado lateral o desenvolvimento de uma nova variante do algoritmo de indução de árvores de decisão do C4.5, do qual se salientam as seguintes características:

- Menores tempos de execução que os do algoritmo do sistema C4.5, em particular, para conjuntos de dados com um grande número de atributos contínuos.
- Ordenação dos valores dos atributos contínuos uma única vez no início do processo de construção da árvore de decisão.
- Pesquisa mais eficiente dos pontos de divisão para os nós da árvore que consideram testes sobre valores de atributos contínuos.
- Implementação de diferentes métodos de crescimento da árvore de decisão, nomeadamente, crescimento "primeiro em profundidade", "primeiro em largura" e "primeiro melhor".
- Capacidade de tratar valores desconhecidos nos atributos.
- Construção da árvore de decisão até uma profundidade pré determinada.

Os resultados experimentais apresentados no capítulo anterior mostram que o algoritmo desenvolvido apresenta um bom desempenho quer em relação ao speedup observado quer em relação à escalonabilidade. Desta forma, pode-se concluir que o algoritmo desenvolvido cumpre os objectivos propostos de uma nova implementação de um algoritmo de construção de árvores de decisão, escalonável e adequado para a classificação de grandes conjuntos de dados.

7.1 Trabalho futuro

Um dos principais factores que limitam o desempenho do algoritmo desenvolvido, já identificado durante a discussão dos resultados experimentais, é o overhead associado às comunicações durante a divisão dos exemplos na fase de crescimento da árvore de decisão.

Propõe-se que em desenvolvimentos futuros do algoritmo este seja alterado para a utilização de uma abordagem híbrida à construção da árvore de decisão em paralelo, aproveitando a abordagem paralela com distribuição horizontal dos dados já implementada. Seria apenas necessário implementar a abordagem ao paralelismo de controlo, a utilizar sempre que o custo associado às comunicações dominasse o tempo de processamento de construção dos nós da árvore. Com base nos resultados apresentados pelas várias implementações descritas no quarto capítulo, a abordagem ao paralelismo de controlo mais indicada será uma abordagem semelhante à utilizada no algoritmo apresentado em [SHKS99].

Existem ainda alguns desenvolvimentos complementares que podem ser implementados e que passam pela introdução de algumas funcionalidades oferecidas pelo algoritmo do sistema C4.5. Funcionalidades como, por exemplo, considerar as divisões dos exemplos por grupos de valores dos atributos discretos durante a construção de um nó, não foram implementadas no algoritmo desenvolvido pois não foram consideradas relevantes para esta fase do trabalho.

Referências

[AGS01] N. Amado, J. Gama, and F. Silva. Parallel implementation of decision tree learning algorithms. In *EKDB 2001*. Springer Verlag, 2001.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. In *Special Issue on Learning and Discovery in Knowledge-Based Databases*, number 5(6), pages 914–925. IEEE, 1993.

[Amd67] G. M. Amdahl. Validity of th single-processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings 30*. AFIPS Press, 1967.

[BF82] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*. Morgan Kaufmann, Los Altos, California, 1982.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth International Group., 1984.

[Cat91] Jason Catlett. *Megainduction: Machine learning on very large databases*. PhD thesis, University of Sydney, 1991.

[Cha97] J. Chattratichat. Large scale data mining challenges and responses, 1997.

[DGST97] John Darlington, Yi-Ke Gno, Janjao Sutiwaraphun, and Hing Wing To. Parallel induction algorithms for data mining. In Xiaohui Liu, Paul Cohen, and Michael Berthold, editors, *Intelligent Data Analysis, Proc. of the 2th Intern. Conference*. Springer Verlag, LNCS 1280, 1997.

[DH73] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. New York, Willey and Sons, 1973.

[DNS91] D. J. DeWitt, J. F. Naughton, and D. A. Schneigder. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. of the 1st Int'l Conf. on Parallel and Distributed Information Systems*, 1991.

[DP97] Pedro Domingos and Michael Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*, 29:103–129, 1997.

[EMS93] F. Esposito, D. Malerba, and G. Semeraro. Decision tree pruning as a search in the state space. In P. Brazdil, editor, *Machine Learning: ECML93*. LNAI 667, Springer Verlag, 1993.

[FL98] Alex A. Freitas and Simon H. Lavington. *Mining Very Large Databases with Parallel Processing.* Kluwer Academic Publishers, 1998.

[KR88] Richard M. Kurp and Vijkya Rumachundran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, 1988.

[Kuf97] R. Kufrin. Decision trees on parallel processors, 1997.

[LER92] T. G. Lewis and H. El-Rewini. *Introduction to Parallel Computing.* Prentice-Hall International, Inc., 1992.

[MST94] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[Pac97] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[PFK98] Foster Provost, Tom Fawcett, and Ron Kohavi. Building the case against accuracy estimation for comparing induction algorithms. In *Proceedings of the 15th International Conference -ICML'98*, 1998.

[Qui92] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, Inc., 1992.

[Rug00] Salvatore Ruggieri. Efficient c4.5. Technical report, Università di Pisa, 2000.

[SAM96] John C. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 544–555. Morgan Kaufmann, 3–6 1996.

[Sha87] Alen Shapiro. *Structured Induction in Expert Systems.* Turing Institute Press, 1987.

[SHKS99] A. Srivastava, E. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. *Data Mining and Knowledge Discovery*, 3, 1999.

[UBC96] P. Utgoff, N. Berkman, and J. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning, Vol. 42, No. 1*, 1996.

[Utg88] P. Utgoff. Percepton trees - a case study in hybrid concept representation. In *Proceedings of the Seventh National Conference on Artificial Intelligence*. Morgan Kaufmann, 1988.

[Wis77] Patrick Henry Wiston. *Artificial Intelligence*. Addison-Wesley, Reading, Mass., 1977.

[ZnRA99] Mohammed Juveed Zaki and Ching-Tien Ho nad Rakesh Agrawal. Scalable parallel classification for data mining on shared memory multiprocessors. In *ICDE*, 1999.