Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação

Aplicação do Processamento Paralelo na Resolução de Problemas de Programação Linear de Grande Porte Utilizando o Princípio de Decomposição Dantzig-Wolfe

Alinson Aguiar Donato

Fortaleza – CE
Dezembro de 2002
Alinson Aguiar Donato

Aplicação do Processamento Paralelo na Resolução de Problemas de Programação Linear de Grande Porte Utilizando o Princípio de Decomposição Dantzig-Wolfe

Dissertação submetida à Coordenação do Mestrado em Ciência da Computação do Departmento de Computação da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre Em Ciência da Computação (M. Sc).

Área de Concentração: Fundamentos da Computação
Linha de Pesquisa: Matemática Computacional
Orientador: Prof. Mauro Cavalcante Pequeno, Dr.

Fortaleza
Universidade Federal do Ceará

Abstract

Large-scale Linear Programming Problems are the ones whose number of equations and/or incognito can reach thousands. Solving these kind of problems by general methods can lead to a slow and expensive process because of the amount of computational processing required by the dimension of the problem. Fortunately, these problems usually have some special structure which can be exploited in order to develop efficient computational procedures.

The Dantzig-Wolfe Decomposition Principle is most efficient when applied to large-scale linear programming problems whose coefficient matrices have angular structure, i. e., one or more independent blocks linked by coupling equations. Each independent block corresponds to a smaller problem (subproblem) that can be solved by different processor in a simultaneous (parallel) way.

This work aims to apply paralelization techniques to the Dantzig-Wolfe decomposition principle in order to solve large-scale linear programming problems in a reasonable time, obtaining a trade-off between the cost (number of processors) and the benefit (resolution time).

Resumo

Problemas de Programação Linear de Grande Porte são aqueles cujo número de variáveis e/ou de restrições pode chegar a alguns milhares. Encontrar a solução de problemas desta natureza, via métodos genéricos, pode ocasionar um processo lento e desgastante, em função do processamento computacional requerido devido à dimensão do problema. Felizmente, tais problemas geralmente possuem alguma estrutura especial que pode ser explorada a fim de se desenvolver procedimentos computacionais mais eficientes.

O Princípio de Decomposição Dantzig-Wolfe se aplica adequadamente a problemas de programação linear de grande porte que possuem matriz dos coeficientes com estrutura *bloco-angular*, ou seja, a matriz é formada por um ou mais blocos independentes interligados por equações de acoplamento. Cada bloco corresponde a um problema menor (subproblema) que pode ser executado por um processador diferente de forma simultânea (em paralelo).

O objetivo do nosso trabalho é aplicar técnicas de paralelismo ao Princípio de Decomposição Dantzig-Wolfe com o propósito de resolver numericamente problemas de programação linear de grande porte em um tempo satisfatório e obter, assim, uma razão custo/benefício – em relação ao número de processadores e tempo de resolução do problema – o mais próximo possível do ideal.

Aos meus queridos pais, *Maria Lucia Aguiar* e *Aloísio Donato de Lima* eu dedico.

Agradecimentos

À Deus, fonte de toda sabedoria do mundo.

À minha família (meu pai: Aloísio Donato de Lima, minha mãe: Maria Lucia Aguiar, meus irmãos: Erick Aguiar Donato e Egline Aguiar)

À minha amada Danielle pelo carinho, amor, compreensão, paciência e incentivo.

Aos meus avós, tios e tias, em especial, à tia Maria Inês, tia Toinha, tia Doralice e tia Zuila.

Aos grandes amigos Dennis Moreira Gomes, Jamille Kesia Gomes de Lima, Paulo Sacha Vasconcelos Mendes e Francisco Valdizar Forte.

Ao grande amigo João Alberto (In memorian).

Aos meus amigos e colegas de profissão Cris Amon Caminha da Rocha, Cecílio Felinto de Oliveira Neto e Alex Feitosa Oliveira.

À todos os colegas do Departamento de Computação da Universidade Federal do Ceará pela colaboração e apoio.

Aos professores do curso de Mestrado em Ciência da Computação, em especial: Manoel Campelo, Ricardo Correa, Riverson Rios e Creto Vidal.

Aos professores, Dr. Antonio Clécio Fontelles Thomaz - Universidade Federal do Ceará e Dr. João Marques de Carvalho - Universidade Federal da Paraíba, ambos membros da comissão examinadora, que se dispuseram a colaborar com suas avaliações.

Ao prof. Dr. Mauro Cavalcante Pequeno - Universidade Federal do Ceará, que me orientou durante todo o trabalho.

À FUNCAP - Fundação Cearense de Amparo a Pesquisa, pelo apoio financeiro.

Ao CENAPAD/NE - Centro Nacional de Processamento de Alto Desempenho no Nordeste, em especial ao Paulo Benício.

A todo o pessoal do LIA - Laboratório de Inteligência Artificial.

E a todos os amigos que fiz durante essa caminhada.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O presente capítulo apresenta o objetivo da dissertação, enfocando o problema a ser resolvido. A seção 1.1 apresenta a motivação do nosso trabalho. A seção 1.2 mostra a revisão bibliográfica e a seção 1.3 descreve o objetivo da dissertação. A seção 1.4 apresenta como a dissertação está organizada e a seção 1.5 mostra o resumo do capítulo.

1.1.Apresentação do Problema

Podemos definir um problema de otimização como um problema que envolve a escolha de valores para um conjunto de variáveis interrelacionadas, com o intuito de alcançar um determinado objetivo. Esse objetivo determina a qualidade dessa escolha e, em Programação Linear, é modelado como uma função linear das variáveis do problema. A maximização (ou minimização) dessa função (função *objetivo*) está sujeita a algumas restrições que limitam o conjunto de valores que as variáveis podem assumir. Esse conjunto de restrições é representado matematicamente como um sistema de (in)equações lineares (veja Figura 1).

A Programação Linear nos fornece vários meios para modelar e encontrar a solução de tais problemas. Porém, podemos encontrar algumas dificuldades se tentarmos usar métodos genéricos de resolução de problemas de Programação Linear (como por exemplo, o Simplex) para encontrar a solução de um problema que possua características particulares. Chamamos de Problemas de Programação Linear de Grande Porte problemas em que o número de variáveis e/ou de restrições pode chegar a alguns milhares. Encontrar a solução de um problema desta natureza, via métodos genéricos, pode ocasionar um processo lento e desgastante, em função do esforço computacional requerido devido à dimensão do problema em questão.

Minimizar $z(x)=cx.$

$$\text{Sujeito a} \quad Ax = b.$$
$$x \geq 0$$

Figura 1.1: Modelagem matemática de problemas de programação linear. c, x e b são vetores e A é uma matriz.

O Princípio de Decomposição Dantzig-Wolfe se aplica adequadamente a problemas de otimização de grande porte que possuem uma certa estrutura especial. A idéia principal é utilizar técnicas de divisão e conquista a fim de tornar o problema original menos complexo e mais fácil de resolver. De acordo com Luenberger (1984), problemas de programação linear de grande porte geralmente possuem alguma estrutura especial, e esta pode e deve ser explorada a fim de se desenvolver procedimentos computacionais mais eficientes. De fato, na maioria dos casos, problemas de grande porte possuem uma matriz dos coeficientes bastante esparsa, onde podemos identificar alguma estrutura particular formada a partir dos coeficientes não-nulos da matriz.

É isso o que acontece com o método em questão. Ele atinge o ápice de sua eficiência quando é aplicado a problemas cuja matriz dos coeficientes possui uma estrutura *bloco-angular*, ou seja, a matriz é formada por um ou mais blocos independentes interligados por equações de acoplamento. Cada bloco corresponde a um problema menor que pode ser executado por um processador diferente de forma independente, justificando assim a utilização de paralelismo neste método. A estratégia se baseia em decompor o problema a ser resolvido em subproblemas, que por sua vez, são resolvidos fazendo uso do Método Simplex ou de uma de suas extensões (por exemplo, o Simplex Revisado). Tais subproblemas, por serem independentes, podem ser distribuídos entre diversos processadores e serem resolvidos simultaneamente (em paralelo), tentando assim alcançar um maior desempenho na resolução do problema original.

1.2.O Estado da Arte

Existem, na literatura, muitos artigos tratando da implementação de métodos de decomposição para resolução de problemas de programação linear. Dentre estes métodos, os mais citados são o Princípio de Decomposição Dantzig-Wolfe e a Decomposição de Benders, dentre outras variações dos mesmos. A maioria destes

artigos não tem como propósito encontrar uma maneira eficiente de efetuar a resolução dos problemas, ou seja, a preocupação principal não é reduzir o tempo de execução dos algoritmos. Alguns, por exemplo, apenas apresentam uma comparação (em termos de número de iterações necessárias para encontrar a solução) entre o método de decomposição estudado e alguma variação do mesmo, como em (Ahn, 1989), onde é mostrado um método baseado no Princípio de Decomposição Dantzig-Wolfe que considera vários problemas mestres ao invés de somente um.

Outros artigos mostram também uma comparação entre métodos de decomposição e métodos genéricos de resolução de problemas de otimização, como, por exemplo, em (Schiefer, 1976), onde foi resolvido um problema de dimensão 907 x 1.265, com 32 blocos independentes, ou como em (Williams, 1974), onde foram resolvidos dois problemas, um de dimensão 358 x 804, e outro de dimensão 1.805 x 3.236. Estes dois artigos fizeram uma comparação entre o método de Decomposição Dantzig-Wolfe e o Simplex, e chegaram a uma conclusão interessante. Concluiu-se que o método de decomposição acima converge para a solução ótima tão rápido quanto os métodos genéricos, porém, com uma vantagem: o método converge rapidamente para a solução nas primeiras iterações e lentamente na fase de ajuste. Dessa forma, se não é estritamente necessário encontrar a melhor solução, e sim uma solução 'quase ótima', o método de Decomposição Dantzig-Wolfe é perfeitamente aplicável.

Nazareth (1984) apresenta um conjunto de problemas de programação linear que é usado para ilustrar as características numéricas do algoritmo de decomposição Dantzig-Wolfe. Apesar de tais problemas possuírem um número pequeno de variáveis e restrições, eles identificam dificuldades numéricas muito comuns na implementação do algoritmo.

Apesar dos artigos citados acima terem encontrado resultados bastante interessantes, nenhum dos mesmos teve como objetivo reduzir o tempo de execução dos métodos. Em (Pinto, 1988), porém, é analisada a viabilidade da utilização de processamento paralelo para a solução de problemas de otimização de grande porte. O problema estudado foi o de despacho com restrições de segurança e controle corretivo utilizado no planejamento da operação de sistemas elétricos de potência. A metodologia de solução baseou-se no

emprego da Decomposição de Benders. Porém, devido a algumas restrições de hardware, a versão paralela desta aplicação foi simulada em uma máquina seqüencial, em vez de ser implantada diretamente em uma máquina paralela. Os resultados da simulação e implementação mostraram uma aceleração significativa na resolução do problema, que foi obtida devido ao emprego do paralelismo.

O ramo da ciência da computação conhecido como processamento paralelo tem crescido bastante nos últimos anos. A potencialidade computacional fornecida pelos computadores paralelos possibilitaram a resolução de uma variedade de problemas computacionais de grande porte de maneira bem mais eficiente. A partir do que foi mencionado anteriormente, podemos perceber que o método de decomposição Dantzig-Wolfe possui um algoritmo que claramente sugere a utilização do paralelismo. O objetivo da dissertação envolve esses aspectos e será detalhado na seção a seguir.

1.3.O Objetivo da Dissertação

O objetivo do nosso trabalho é aplicar técnicas de paralelismo ao Princípio de Decomposição Dantzig-Wolfe para resolução de problemas de programação linear, com o propósito de resolver numericamente problemas de grande porte em um tempo satisfatório e obter, assim, uma razão custo/benefício – em relação ao número de processadores e tempo de resolução do problema – o mais próximo possível do ideal. A idéia é estudar e analisar o método (algoritmo) de decomposição Dantzig-Wolfe a fim de formular um algoritmo paralelo para o mesmo, e implementá-lo usando um software para programação paralela disponível. Com o algoritmo (e o programa) paralelo em mãos, o próximo passo é efetuar testes de desempenho a fim de comparar a eficiência do algoritmo paralelo em relação a um software mundialmente reconhecido e largamente usado na resolução de problemas de programação linear de grande porte. Tal software chama-se CPLEX.

O algoritmo paralelo será implementado utilizando a biblioteca de paralelismo PVM, que é uma ferramenta que, associada à linguagem C, permite a implementação do paralelismo através do mecanismo de *Message Passing* (troca de mensagens). Implementaremos também uma versão seqüencial do método de decomposição (em C) com o intuito de verificar qual o fator de aceleração (*speedup*) obtido com a inclusão do

paralelismo. Todas os programas e rotinas necessários para a obtenção dos resultados do trabalho serão implementados em C, devido à portabilidade fornecida por essa linguagem.

Os testes de execução serão efetuados em um *cluster* de estações de trabalho presentes no Laboratório de Inteligência Artificial (LIA) do Departamento de Computação da Universidade Federal do Ceará (UFC).

1.4.A Organização da Dissertação

A dissertação contém 5 Capítulos e 1 Apêndice. O primeiro capítulo é a introdução e o quinto apresenta as conclusões, considerações finais e propostas para trabalhos futuros. Os demais capítulos contêm o desenvolvimento da dissertação.

O Capítulo 2 apresenta a fundamentação teórica em que se baseia o Princípio de Decomposição Dantzig-Wolfe, além do algoritmo resultante que será usado para resolver problemas de programação linear de grande porte. Nesse capítulo serão apresentados os conceitos relacionados à Programação Linear que serão usados no decorrer da dissertação.

O Capítulo 3 apresenta vários conceitos relacionados à Computação Paralela, mostrando a motivação para o uso de processamento paralelo, bem como as principais ferramentas disponíveis para a implementação do paralelismo. Nesse capítulo será apresentada a versão paralela para o algoritmo de decomposição Dantzig-Wolfe que será usada nos testes de desempenho, além de detalhes da implementação.

O Capítulo 4 mostra uma descrição dos recursos de hardware e software utilizados na implementação e nos testes e apresenta os resultados obtidos nos testes de desempenho. Além disso, esse capítulo apresenta uma análise comparativa dos tempos de execução obtidos nos testes de execução.

O Apêndice A contém todas as tabelas com os resultados obtidos nos testes de desempenho dos algoritmos.

O Apêndice B mostra a listagem de todos os programas utilizados na dissertação, incluindo todas as implementações do algoritmo de decomposição (paralela e seqüencial) e o simulador de problemas.

1.5.Resumo

Problemas de otimização envolvem a escolha de valores para um conjunto de variáveis interrelacionadas, com o intuito de alcançar um determinado objetivo, que em Programação Linear é modelado como uma função linear das variáveis do problema. A Programação Linear fornece vários meios para modelar e encontrar a solução desses problemas, porém, usar métodos genéricos para encontrar a solução de um problema que possua características particulares pode ocasionar um processo lento e desgastante.

Problemas de Programação Linear de Grande Porte são aqueles em que o número de variáveis e/ou de restrições pode chegar a alguns milhares. Tais problemas geralmente possuem alguma estrutura especial que deve ser explorada a fim de se desenvolver procedimentos computacionais mais eficientes. O Princípio de Decomposição Dantzig-Wolfe se aplica adequadamente a problemas desta natureza. Ele atinge o ápice de sua eficiência quando aplicado a problemas cuja matriz dos coeficientes possui uma estrutura *bloco-angular*, ou seja, a matriz é formada por um ou mais blocos independentes interligados por equações de acoplamento. Cada bloco corresponde a um problema menor que pode ser executado por um processador diferente de forma independente, justificando assim a utilização de paralelismo.

O objetivo do nosso trabalho é aplicar técnicas de paralelismo ao Princípio de Decomposição Dantzig-Wolfe para resolução de problemas de programação linear, com o propósito de resolver numericamente problemas de grande porte em um tempo satisfatório e obter, assim, uma razão custo/benefício – em relação ao número de processadores e tempo de resolução do problema – o mais próximo possível do ideal.

Capítulo 2

O Princípio de Decomposição Dantzig-Wolfe

O presente capítulo apresentará os conceitos sobre Programação Linear e a fundamentação teórica necessárias para o entendimento do Princípio de Decomposição Dantzig-Wolfe. A seção 2.1 faz uma introdução do que é um problema de programação linear e caracteriza tais problemas de acordo com seu tamanho. A seção 2.2 apresenta o Simplex, um dos métodos mais tradicionais para resolução de problemas de programação linear. A seção 2.3 mostra uma variação do Simplex computacionalmente mais eficiente chamada de Simplex Revisado e a seção 2.4 apresenta as principais características dos problemas de programação linear de grande porte. Finalmente, a seção 2.5 apresenta o desenvolvimento do método de decomposição, bem como o algoritmo resultante, e a seção 2.6 apresenta o resumo do capítulo.

2.1.Introdução

Podemos definir um problema de otimização como um problema que envolve a escolha de valores para um conjunto de variáveis inter-relacionadas, com o intuito de alcançar um determinado objetivo (Minoux, 1986). Esses problemas se caracterizam pelo grande número de soluções que satisfazem as condições impostas. A escolha de uma determinada solução como a melhor de todas para um problema depende do objetivo que se quer atingir. Esse objetivo determina a qualidade dessa escolha. Em Programação Linear, é modelado como uma função linear das variáveis do problema (Ver Figura 2.1). A maximização (ou minimização) dessa função (função *objetivo*) está sujeita a algumas restrições que limitam o conjunto de valores que as variáveis podem assumir. Esse conjunto de restrições é representado matematicamente como um sistema de equações lineares (veja Figura 1). A solução que satisfaz o conjunto de restrições do

problema ($Ax = b$ e $x \geq 0$) e que maximiza (ou minimiza) a função objetivo ($c^T x$) é denominada *solução ótima*.

$$\begin{aligned} \text{Minimizar} \quad & z(x) = c^T x. \\ \text{Sujeito a} \quad & Ax = b. \\ & x \geq 0 \end{aligned}$$

Figura 2.1: Modelagem matemática de problemas de programação linear. c, x e b são vetores e A é uma matriz.

Existem inúmeras situações da vida real que podem ser modeladas como um problema de programação linear. Na grande maioria dos casos, resolver um problema desse tipo consiste em encontrar a melhor maneira de utilizar um conjunto limitado de recursos a fim de atingir uma certa meta, como, por exemplo, maximizar lucros ou minimizar custos (Gass, 1985). Um exemplo bastante comum é o caso em que uma empresa deseja determinar como aplicar seus recursos de uma forma tal que se obtenha um plano de produção ótimo, que não somente satisfaça a demanda do mercado, como também maximize os lucros da empresa. O conjunto de restrições do problema em questão é determinado pela limitação dos recursos da empresa e pelas necessidades do mercado. A meta da empresa, ou seja, maximizar seu ganho, determina a função objetivo. O exemplo acima é um caso típico que resulta num problema relativamente simples, porém, problemas muito mais complexos podem ser modelados e solucionados através de Programação Linear.

Uma medida óbvia para determinar a complexidade de um problema de programação linear é seu tamanho, que pode ser medido em termos do número de variáveis e restrições envolvidas. Segundo (Minoux, 1986), podemos dividir os problemas de otimização, de acordo com o seu tamanho, em três grupos distintos: Problemas de *Pequeno Porte*, Problemas de *Médio Porte* e Problemas de *Grande Porte*. Devido à diversidade das técnicas existentes para resolução de tais problemas, e também ao avanço da tecnologia computacional atual, não podemos citar com precisão parâmetros (como dimensão da matriz dos coeficientes, por exemplo) que permitam discernir a qual grupo pertence determinado problema. Porém, a grosso modo, podemos dizer que problemas de pequeno porte são aqueles que podem ser resolvidos à mão ou cuja resolução pode ser facilmente acompanhada passo a passo. Problemas de médio

porte são aqueles cuja resolução necessita do auxílio do computador, mas sem aplicação de qualquer técnica especial ou de equipamento com alto poder de processamento. Finalmente, problemas de grande porte caracterizam-se por precisarem de algoritmos sofisticados, que geralmente exploram alguma estrutura particular, e requerem alto esforço computacional na sua resolução. O estudo e a resolução de tais problemas consistem na identificação dessas estruturas especiais, que são importantíssimas para o desenvolvimento de técnicas e procedimentos computacionais eficientes, os quais devem explorar de alguma forma tal estrutura para resolver o problema.

Problemas de Grande Porte são o alvo dos algoritmos apresentados no nosso trabalho. A seguir faremos uma descrição de um dos métodos mais utilizados para resolução de problemas de programação linear, o método Simplex, e mais tarde voltamos a falar sobre problemas de grande porte e suas características.

2.2.O Método Simplex

O método Simplex, que foi criado por G. B. Dantzig por volta de 1947, é uma técnica utilizada para encontrar, algebricamente, a solução ótima de um problema de programação linear, desde que tal solução exista. Para uma total compreensão do método, é necessário, antes, enunciar os teoremas nos quais ele se baseia. As demonstrações dos teoremas e a fundamentação teórica completa do método podem ser encontradas em (Dantzig, 1963).

Consideremos o seguinte problema de programação linear:

$$\text{Minimizar} \quad z(x) = c^T x \tag{1}$$

$$\text{Sujeito a} \quad Ax = b \tag{2}$$

$$x \geq 0 \tag{3}$$

Dizemos que tal problema encontra-se na *forma padrão*, pois a função objetivo deve ser minimizada, todas as restrições são igualdades e todas as variáveis são positivas. Qualquer problema de programação linear que não esteja na forma padrão pode ser facilmente transformado em um problema equivalente na forma padrão. No problema (1-3) acima, A é chamada de *matriz dos coeficientes* e possui m linhas e n colunas, com m < n. O vetor de m componentes *b* é chamado *vetor de termos*

independentes, e o vetor de n componentes *c* contém os *coeficientes da função objetivo*. *x* é um vetor de n componentes que conterá a solução do problema. Geometricamente, *x* pode ser interpretado como um ponto dentro do conjunto de soluções do problema.

Teorema 1: "O conjunto de todas as soluções viáveis de um problema de programação linear formam um conjunto convexo C."

Chamamos de *solução viável* um ponto (vetor) *x* que satisfaz as restrições (2) e (3). Um conjunto $C \in R^m$ é dito convexo se, e somente se, o ponto (vetor) resultante de uma combinação linear convexa de pontos (vetores) de C for também pertencente a C. Em outras palavras, se $x = \alpha_1 x^1 + \alpha_2 x^2$, com $\alpha_1 + \alpha_2 = 1$ e $\alpha_1, \alpha_2 \geq 0$, e se $x^1, x^2 \in C$, então $x \in C$. A interpretação geométrica diz que qualquer ponto situado no segmento de reta que liga dois pontos de C também pertence a C. Sendo assim, temos que segundo o Teorema 1, qualquer combinação linear convexa de soluções viáveis de um problema de programação linear também é uma solução viável, ou seja, o conjunto $C = \{x \in R^n : Ax = b, x \geq 0\}$ é convexo.

Teorema 2: "Toda solução viável básica do sistema $Ax = b$ é um ponto extremo do conjunto das soluções viáveis, isto é, do conjunto convexo C do Teorema 1."

Se uma matriz A (m x n) possui m colunas linearmente independentes, a matriz quadrada B formada por essas colunas é chamada de *matriz básica* ou simplesmente *base*. Se uma matriz A possui uma base B, então o sistema $Ax = b$ tem uma solução *x* para qualquer vetor *b* pertencente a R^m. Uma *solução viável básica correspondente à base B* é um ponto *x* (viável) que pode ser particionado em m variáveis básicas (correspondentes às colunas de A que estão em B) e n – m variáveis não-básicas (que são iguais a zero e correspondem às colunas de A que não estão em B). Além disso, chamamos de *ponto extremo* (ou *vértice*) de C um ponto que não pode ser expresso como uma combinação linear convexa de quaisquer outros dois pontos distintos pertencentes a C.

Teorema 3: "Se o problema possui alguma solução ótima, pelo menos uma delas é um ponto extremo do conjunto convexo C. Além disso, se a função objetivo assume o

valor ótimo em mais de um ponto extremo, então ela obtém o mesmo valor para qualquer combinação linear convexa desses pontos extremos."

Pode-se dizer então que a solução ótima de um problema de programação linear, se existir, está em um dos pontos extremos do conjunto de soluções viáveis, e, segundo o Teorema 2, é uma solução viável básica. Isso garante que o número de iterações para achar a solução ótima é finito. O que o método Simplex faz é percorrer os pontos extremos do conjunto de soluções viáveis do problema em busca da solução (ponto) ótima.

Para que um problema seja resolvido pelo Simplex, é necessário que o mesmo esteja na forma padrão. Para que o Simplex inicie, é necessário conhecer pelo menos um dos pontos extremos (solução viável básica) do conjunto de soluções viáveis. Supondo-se disponível uma solução viável básica inicial para (2) e (3), o método verifica se essa solução é ótima. Se não for, é porque algum dos pontos extremos adjacentes ao ponto correspondente à solução dada fornece, para a função objetivo, um valor menor do que o atual. O método então calcula a solução correspondente ao ponto extremo adjacente que mais diminui o valor da função objetivo. O método verifica novamente se tal ponto é ótimo, e se não for, o processo recomeça. O método pára quando, estando num ponto extremo, todos os pontos extremos a ele adjacentes fornecem, para a função objetivo, valores maiores do que o atual. É nessa hora que é importante o fato do conjunto de soluções viáveis ser convexo.

Algebricamente esse processo é feito da seguinte forma. Um ponto extremo é uma solução viável básica que inclui todas as variáveis básicas anteriores, com exceção de apenas uma delas. Portanto, para encontrar a próxima solução viável básica (ponto extremo adjacente), é necessário primeiro escolher uma variável básica para deixar a base atual, ou seja, torná-la não-básica, e depois, escolher uma variável não-básica para substituir a variável que saiu da base.

Resumindo, o método Simplex compreende os seguintes passos:

i. Encontrar uma solução básica viável inicial.

ii. Verificar se a solução atual é ótima. Se for, pare. Caso contrário, prossiga.

iii. Determinar a variável não-básica que deve entrar na base.

iv. Determinar a variável básica que deve sair da base.

v. Identificar a nova solução básica viável e voltar ao passo ii.

O método é baseado no fato de que no mínimo n – m componentes (variáveis) de x são iguais a zero, se x é um ponto extremo do conjunto de soluções viáveis. Assim, x pode ser particionado em um subvetor x_B, que contém m componentes (variáveis básicas), e um subvetor x_N, que contém n – m componentes (variáveis não-básicas com valores nulos). Da mesma forma, podemos particionar a matriz dos coeficientes A como

$$A = [\ B \mid N\ ] \tag{4}$$

onde B contém as m colunas correspondentes às variáveis básicas (x_B), e N contém as n – m colunas restantes.

Em cada iteração do método, uma variável básica (uma componente de x_B) é transformada em variável não-básica, e vice-versa. Em outras palavras, x_B e x_N trocam componentes. Geometricamente, esse processo de troca corresponde a mover-se de um ponto extremo do conjunto de soluções viáveis para um dos pontos extremos adjacentes a ele. É necessário, portanto, identificar esse ponto extremo adjacente, ou seja, escolher qual componente de x_B deve entrar em x_N (isto é, qual variável possivelmente deixará de ser zero), e qual componente de x_N deve entrar em x_B (isto é, uma variável que será zerada). Na verdade, é preciso fazer apenas a primeira dessas escolhas, já que a segunda é alcançada por (2) e (3). Senão vejamos, aplicando o particionamento de (4) em (2), obtemos que $Bx_B + Nx_N = b$. Expressando x_B em termos de x_N, temos

$$x_B = B^{-1}\ (b - Nx_N) \tag{5}$$

Particionando o vetor de coeficientes da função objetivo c em c_B e c_N (coeficientes das variáveis básicas e das variáveis não-básicas, respectivamente), e a partir de (5), temos

$$c^T x = c_B{}^T x_B + c_N{}^T x_N = c_B{}^T B^{-1}\ (b - Nx_N) + c_N{}^T x_N = c_B{}^T B^{-1} b + [c_N{}^T - c_B{}^T B^{-1} N\ ]^T\ x_N$$

O vetor $d_N = c_N{}^T - c_B{}^T B^{-1} N$ é chamado de *vetor de custos reduzidos*. Se todos os componentes de x_B forem estritamente positivos e pelo menos um dos componentes

(por exemplo, o i-ésimo componente) de d_N for negativo, é possível decrescer o valor atual da função objetivo ($c^T x$) fazendo com que o componente i de x_N se torne positivo (ou seja, entre na base), devendo-se ajustar x_B para não afetar a viabilidade da solução. Como $x_B \geq 0$, existe um limite superior para $x_{N,i}$, o i-ésimo elemento de x_N.

A princípio, podemos escolher qualquer $x_{N,i}$ com $d_{N,i} < 0$ para entrar na base. Se $d_N \geq 0$, então x, a solução corrente, é ótima. Se mais de um das componentes de d_N são negativas, em geral escolhe-se a componente que levará à máxima redução da função objetivo, ou seja, aquela cujo valor em d_N for a mais negativa deve entrar na base. Apesar dessa escolha ser um tanto quanto óbvia, não se pode garantir que a adoção desse critério fará o método convergir mais rapidamente.

Nesse ponto, sabendo que os elementos restantes de x_N permanecem iguais a zero, usando (2) temos que

$$x_B = B^{-1} (b - N_i x_{N,i}) \tag{6}$$

onde N_i é a coluna de N correspondente a $x_{N,i}$. O ideal é escolher como novo valor de $x_{N,i}$ o maior valor que satisfaça $x_B \geq 0$. Para obter $x_{N,i}$ explicitamente, podemos obter, a partir de (6), que

$$x_{N,i} = \min \{(B^{-1}b)_j / (B^{-1}N_i)_j : (B^{-1}N_i)_j > 0\} \tag{7}$$

onde j é a posição da componente de x_B que se tornará não-básica (ou seja, igual a zero).

Dessa forma, escrevendo sob a forma algorítmica, podemos detalhar ainda mais os passos do método Simplex enumerados anteriormente:

i. **Encontrar uma solução básica viável inicial:** encontrar algum ponto extremo do conjunto de soluções viáveis (x) que contenha pelo menos n – m componentes iguais a zero (x_N). Se não se conhece uma base B inicial, pode-se facilmente faze-lo através da inclusão de variáveis artificiais (método Simplex de duas fases), que podem ser excluídas no decorrer do processo. Mais detalhes podem ser encontrados em (Dantzig, 1963).

ii. **Verificar se a solução atual é ótima. Se for, pare. Caso contrário, prossiga:** a partir dos custos reduzidos correspondentes às variáveis não-básicas (d_N), podemos determinar se a solução atual é ótima. Se $d_N \geq 0$, ou seja, d_N não possui nenhum elemento negativo, então a solução ótima foi encontrada e o método deve parar. Caso contrário, prossiga para o próximo passo.

iii. **Determinar a variável não-básica que deve entrar na base:** escolhe-se alguma componente $x_{N,i}$ de x_N tal que seu custo reduzido seja negativo, ou seja, $d_{N,i} < 0$. Em geral escolhe-se a componente cujo custo reduzido é o mais negativo.

iv. **Determinar a variável básica que deve sair da base:** escolhe-se alguma componente $x_{B,j}$ de x_B tal que

$$(B^{-1}b)_j / (B^{-1}N_i)_j = \min \{(B^{-1}b)_k / (B^{-1}N_i)_k : (B^{-1}N_i)_k > 0, k = 1, ..., m\}$$

v. **Identificar a nova solução básica viável e voltar ao passo ii:** obtém-se o novo ponto extremo removendo-se do ponto extremo anterior a componente de x_B que se tornará não-básica e inserindo a componente de x_N que se tornará básica.

Por questão de conveniência, costuma-se colocar os dados do problema original ($Ax = b \Rightarrow Bx_B + Nx_N = b$) em um quadro (*tableau*) como mostrado a seguir:

	x_B	x_N	
x_B	B	N	b
Z	c_B^T	c_N^T	Q(x)

O método Simplex alcança seu objetivo (solução do problema) executando uma seqüência de operações de pivoteamento no quadro acima até que se obtenha a seguinte configuração (quadro final):

	x_B	x_N	
x_B	I	$B^{-1} N$	$B^{-1}b$
Z	0	$c_N^T - c_B B^{-1} N$	$Q(x) - c_B B^{-1} b$

De acordo com o quadro acima, a solução ótima x, pode ser obtida a partir de $x_B = B^{-1}b$ e $x_N = 0$. A maior parte do custo computacional do método vem da necessidade de, a cada iteração, calcular os vetores $B^{-1}b$ e $B^{-1}N_i$, e também calcular explicitamente a inversa da base (B^{-1}). Como veremos na seção a seguir, é possível eliminar grande parte desse esforço computacional fazendo algumas modificações na versão original do Simplex, obtendo um método bem mais eficiente em termos computacionais.

2.3.O Simplex Revisado

Com o intuito de tornar o Simplex mais eficiente computacionalmente, uma versão revisada do mesmo foi desenvolvida. Em geral, é essa versão que é implementada nos computadores. Segundo Dantzig (1963), a idéia é simples: a cada iteração do Simplex, muitas das informações contidas no tableau não são usadas pelo processo de decisão do método. Enquanto cada iteração do Simplex requer que um novo tableau inteiro seja calculado e armazenado, pode-se observar que são necessários apenas os seguintes elementos para o funcionamento do método:

- O vetor de custos reduzidos das variáveis não-básicas (d_N): É necessário para selecionar a variável a entrar na base, o que é feito encontrando-se algum $d_{N,i} < 0$.
- A inversa da base (B^{-1}): Se existir algum $d_{N,i} < 0$, é necessário encontrar a nova coluna P_i a entrar na base, que pode ser obtida multiplicando a matriz inversa da base (B^{-1}) pela coluna N_i correspondente à variável não-básica que irá entrar na base ($P_i = B^{-1}N_i$).
- A solução viável básica corrente ($B^{-1}b$): pode ser obtido na última coluna do tableau e é usado, juntamente com a coluna P_i, para escolher qual a variável a sair da base.

É válido notar que, apenas uma coluna não-básica do tableau corrente (P_i) é necessária em cada iteração. Como é bastante comum termos bem mais colunas do que linhas em um problema de programação linear, podemos ter um ganho bastante significativo, em termos de tempo de processamento e de espaço de armazenamento, se ignorarmos as colunas P_j, para $j \neq i$. Um procedimento ainda mais eficiente seria gerar, a partir dos dados do problema original, primeiro, o vetor de custos reduzidos (d_N) e, depois, a coluna P_i a entrar na base. É exatamente isso o que o Simplex Revisado faz, sendo necessário, para isso, apenas guardar a inversa da base (B^{-1}) corrente a cada iteração, para que os valores citados sejam obtidos. É fácil notar que para obter a coluna P_i ($B^{-1}N_i$) e a solução viável básica ($B^{-1}b$), é necessário apenas conhecer a inversa da base corrente. No caso do vetor de custos reduzidos (d_N), temos que $d_N = c^T_N - c_B B^{-1} N$, sendo necessário portanto manter apenas o vetor $\pi = c^T_B B^{-1}$, já que os elementos restantes são todos constantes (os elementos de π são denominados *multiplicadores simplex associados à base B*, e é devido a isso que o Simplex Revisado também é conhecido como *Método Simplex usando Multiplicadores*). Dessa forma, o tableau que guarda as informações necessárias ao método a cada iteração possui m + 1 linhas e m + 1 colunas, e tem a seguinte forma:

	x_B	
x_B	B^{-1}	$B^{-1}b$
	π	

A seguir apresentamos o algoritmo do Simplex Revisado.

Simplex Revisado (Algoritmo): Suponha que, em alguma iteração k, a inversa da base corrente, B^{-1}, a solução básica associada a B, $x_B = B^{-1}b$, e os dados originais do problema (A, *b* e *c*) estão disponíveis. A iteração k procede como se segue:

1) A linha m + 1 do tableau tem a seguinte forma (π_1, π_2, ..., π_m). Calcule os custos reduzidos d_N a partir dos coeficientes da função objetivo e da matriz N

(matriz formada pelas colunas de A correspondentes às variáveis não-básicas).

2) Se $d_N \geq 0$, pare. A solução ótima foi encontrada e está disponível no tableau (coluna m + 1). Caso contrário, selecione, para se tornar básica, uma variável não-básica $x_{N,i}$ cujo custo reduzido $d_{N,i}$ é negativo.

3) Calcule a coluna P_i a entrar na base, multiplicando N_i (coluna correspondente a $x_{N,i}$) à esquerda pela inversa da base ($P_i = B^{-1}N_i$).

4) Determine a variável básica $x_{B,j}$ a sair da base onde j é tal que:

$$(B^{-1}b)_j / P_{i,j} = \min \{(B^{-1}b)_k / (P_{i,j})_k : (P_{i,j})_k > 0, k = 1, ..., m\}$$

5) Efetue operações de pivoteamento no tableau de forma a encontrar nova base B e novos multiplicadores simplex π. Volte para o passo 1).

2.4.Estrutura Bloco-Angular e Problemas de Grande Porte

Em 1961, G. B. Dantzig e P. Wolfe publicaram o artigo "The Decomposition Algorithm for Linear Programming" (Dantzig, 1961), que apresentava um algoritmo de decomposição para resolução de problemas de otimização, que mais tarde ficou conhecido como Princípio de Decomposição Dantzig-Wolfe. Tal algoritmo, que é baseado na versão revisada do método Simplex vista na seção anterior, fornece um procedimento que se torna muito eficiente quando aplicado a problemas de programação linear cuja matriz dos coeficientes tem uma estrutura bloco-angular. Uma matriz que possui essa estrutura é caracterizada por um ou mais blocos de restrições independentes, interligados por algumas restrições comuns, chamadas de restrições de acoplamento (Ver Figura 2.2).

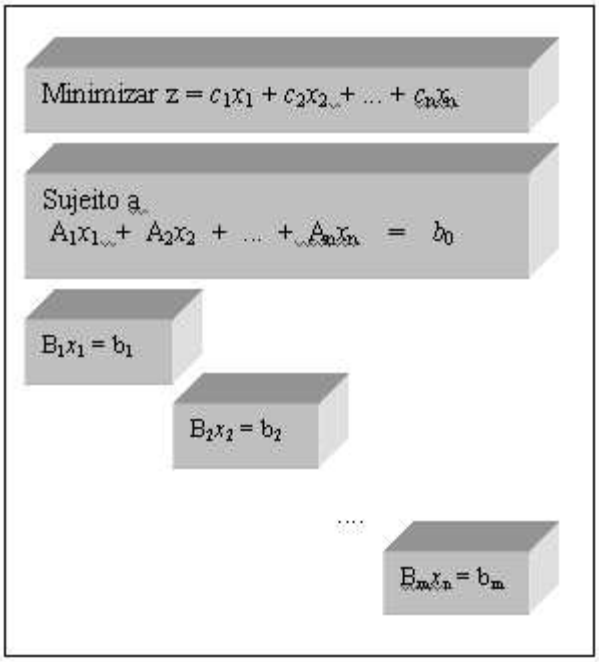


Figura 2.2: Estrutura Bloco-Angular.

A fim de apresentar uma situação típica que sugere o uso do princípio de decomposição Dantzig-Wolfe (estrutura bloco-angular), considere o caso de uma empresa que é dividida em departamentos e deseja minimizar o custo total de suas operações (função objetivo). Cada departamento possui necessidades internas que não interagem com as necessidades dos outros departamentos. A partir dessas necessidades obtém-se conjuntos (blocos) de restrições independentes que não interagem entre si. Além disso, existem recursos comuns que devem ser compartilhados entre os departamentos, os quais resultam em restrições que interligam todos os departamentos (restrições de acoplamento).

Entretanto, mesmo problemas que não possuem claramente a estrutura organizacional mostrada acima podem sugerir a aplicação do princípio de decomposição. Por exemplo, a resolução de problemas que envolvem a vida real é geralmente dificultada pelo tamanho dos mesmos, que na maioria dos casos são muito grandes. Em programação linear, o tamanho de um problema é determinado pelo seu número de variáveis e pelo seu número de restrições. Como já mencionamos anteriormente, o significado do termo "grande" depende tanto da eficiência dos

algoritmos que são usados para resolução do problema, como do equipamento computacional disponível. Hoje em dia, são considerados Problemas de Otimização de Grande Porte, aqueles cujo número de variáveis e restrições chegam a muitos milhares. Para resolver tais problemas de forma eficiente é necessário fazer-se uso de algoritmos especiais que devem ser obtidos a partir da exploração da estrutura do problema. Felizmente, problemas de grande porte quase sempre possuem alguma estrutura particular. Em programação linear, essa estrutura é identificada a partir dos coeficientes nulos e não-nulos da matriz de restrições do problema. Quando um problema é grande, o número de coeficientes não-nulos é muito pequeno em relação ao número total de coeficientes. Segundo Lasdon (1970), os coeficientes não-nulos geralmente encontram-se posicionados de uma forma ordenada, por exemplo, divididos em blocos ao longo da diagonal principal, com exceção de algumas poucas linhas ou colunas. Além disso, alguns problemas possuem um número muito grande de colunas ou um número muito grande de linhas, nunca os dois ao mesmo tempo. Dessa forma, facilmente nota-se que não é tão difícil encontrar em problemas de otimização de grande porte a estrutura bloco-angular citada anteriormente.

A seção a seguir descreve como o Princípio de Decomposição Dantzig-Wolfe foi desenvolvido e mostra porque é extremamente adequado aplicá-lo na resolução de problemas cuja matriz dos coeficientes possui estrutura bloco-angular.

2.5.O Princípio de Decomposição Dantzig-Wolfe

O algoritmo de decomposição Dantzig-Wolfe procura aproveitar a estrutura bloco-angular da matriz dos coeficientes de um problema de programação linear para reduzir a dimensão da base, criando um problema equivalente com grande redução no número de restrições (e, portanto, na dimensão da base). O método trabalha formando um "problema mestre", equivalente ao problema original, que possui um número de linhas um pouco maior do que o número de restrições de acoplamento do problema original (uma linha a mais para cada bloco independente), e um número de colunas que pode vir a ser muito grande (da ordem do número de pontos extremos dos conjuntos viáveis correspondentes aos blocos independentes). A característica mais relevante do método é um esquema de geração implícita de colunas não-básicas no problema mestre,

indispensável para tornar possível a resolução. O problema mestre é resolvido usando uma técnica chamada de "geração de colunas" (*colunm generation*), que consiste em resolver o problema sem considerar todas as colunas, gerando as mesmas à medida que elas vão sendo necessárias para o método Simplex. O algoritmo resultante trabalha com uma interação entre um conjunto independente de subproblemas e o problema mestre. Os subproblemas recebem do problema mestre um conjunto de parâmetros (multiplicadores simplex), utilizam esses parâmetros para calcular os coeficientes da função objetivo, encontram suas soluções e então as enviam de volta para o mestre. O problema mestre combina as soluções dos subproblemas recém-recebidas com as recebidas anteriormente e calcula novos multiplicadores, que são enviados novamente para os subproblemas. Essa interação prossegue até que um determinado critério de otimalidade seja atingido. Tal critério é baseado nos valores dos multiplicadores simplex e é testado sempre antes do problema mestre enviar novos multiplicadores para os subproblemas.

2.5.1. Fundamentação Teórica

O desenvolvimento do princípio de Decomposição Dantzig-Wolfe se baseia fundamentalmente em duas noções. A primeira noção é explicitada pelo Teorema 1 enunciado na seção 2, ou seja, "*um ponto pertence a um conjunto convexo limitado se, e somente se, ele pode ser obtido por uma combinação linear convexa dos pontos extremos desse conjunto*".

Em outras palavras, qualquer ponto x de um conjunto convexo limitado C pode ser obtido por uma combinação linear convexa dos pontos extremos x_i desse conjunto. Assim, as restrições próprias de um problema de programação linear podem ser substituídas pela enumeração de suas soluções extremas, havendo apenas que incluir restrições de convexidade nos multiplicadores utilizados. A prova desse teorema pode ser encontrada em (Lasdon, 1970). Na próxima seção mostraremos como o Teorema 1 foi usado no desenvolvimento do princípio de decomposição.

A segunda noção é o conceito de geração de colunas. Considere o seguinte problema de programação linear

$$\text{minimizar } z = \Sigma\, c_j x_j \quad (j = 1,..., n)$$
$$\text{sujeito a} \quad \Sigma\, a_j x_j = b \quad (j = 1,..., n)$$
$$x_j \geq 0 \quad (j = 1,..., n)$$

onde a_j e b são vetores com m componentes, m < n. Assuma que uma solução básica viável inicial x_B, associada a uma base B, está disponível, onde c_B são os coeficientes das variáveis básicas na função objetivo. Tal solução, se existir, pode ser encontrada usando a primeira fase do método Simplex. Os multiplicadores simplex associados a essa base são $\pi = c_B B^{-1}$ e são sempre disponibilizados pelo método Simplex. Para melhorar a solução básica viável, calculamos os custos reduzidos das variáveis não-básicas ($d_{N,j} = c_{N,j} - \pi N_j$). Se o menor dos custos reduzidos for negativo ($\min\{d_{N,j}\} = d_{N,s} < 0$), então a solução corrente pode ser melhorada se introduzirmos uma das variáveis com custo reduzido negativo (possivelmente $x_{N,s}$, a variável com menor custo reduzido) na base, através de operações de pivoteamento.

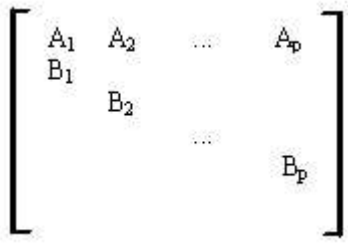
Se o número de colunas do problema for muito grande (de milhares a milhões de colunas), encontrar o valor do menor custo reduzido ($d_{N,s}$), calculando $d_{N,j}$ para cada coluna j, pode ser um processo bastante tedioso e ineficiente. Entretanto, existe uma variedade de técnicas que podem ser usadas para permitir que se calcule os custos reduzidos de apenas um subconjunto das colunas a_j. Essa abordagem é chamada de geração de colunas, pois esse pequeno subconjunto de colunas é gerado de acordo com a necessidade.

Por outro lado, problemas de programação linear podem ser grandes tanto no número de variáveis como também no número de restrições. Uma das formas de tratar esse problema é construir um problema equivalente com um número menor de restrições, porém com um número de variáveis bem maior, e então aplicar o método de geração de colunas para resolver o problema equivalente em questão. Esta é a base do princípio de Decomposição Dantzig-Wolfe para problemas de programação linear que será apresentado a seguir.

2.5.2. O Método de Decomposição Dantzig-Wolfe

2.5.2.1.Desenvolvimento do Princípio de Decomposição

Considere o problema de programação linear cuja matriz de restrições possui uma estrutura bloco-angular com p blocos, onde $p \geq 1$:



Observe que qualquer problema de programação linear pode ter essa forma, basta, para isso, considerar p = 1, o que equivale a particionar suas restrições em dois subconjuntos:

$$\text{minimizar } z = c^T x \tag{1}$$

sujeito a

$$A_1 x = b_1 \quad (m_1 \text{ restrições}) \tag{2}$$

$$A_2 x = b_2 \quad (m_2 \text{ restrições}) \tag{3}$$

$$x \geq 0 \tag{4}$$

A fim de simplificar a nossa discussão, assumiremos que o conjunto convexo $C_2 = \{x \mid A_2 x = b_2,\ x \geq 0\}$ é limitado. As modificações necessárias para que os resultados encontrados possam ser aplicados ao caso em que C_2 é ilimitado podem ser encontradas em (Dantzig, 1963) ou em (Lasdon, 1970). A partir da suposição acima e pelo Teorema 1, qualquer elemento de C_2 pode ser escrito como

$$x = \Sigma\, \lambda_j x^j \tag{6}$$

onde

$$\Sigma\, \lambda_j = 1, \quad \lambda_j \geq 0 \tag{7}$$

e x^j são os pontos extremos do poliedro C_2.

O problema original (1)-(4) pode ser visto como se segue: escolha, dentre todas as soluções de (3)-(4), aquelas que satisfazem (2) e minimizam z. Para isso, substitua (6) em (2) e obtenha

$$\Sigma\,(A_1 x^j)\lambda_j = b_1 \tag{8}$$

Substituindo (6) em (1), obtemos z em termos das variáveis λ_j:

$$z = \Sigma\,(cx^j)\lambda_j \tag{9}$$

Definindo

$$A_1 x^j = p_j \tag{10}$$

$$cx_j = f_j \tag{11}$$

obtemos o seguinte problema de programação linear em λ_j:

$$\text{minimizar } \Sigma f_j \lambda_j \tag{12}$$

$$\text{sujeito a} \quad \Sigma p_j \lambda_j = b_1 \tag{13}$$

$$\Sigma \lambda_j = 1 \tag{14}$$

$$\lambda_j \geq 0 \tag{15}$$

O problema acima, habitualmente designado por Problema Mestre (*master program*), é completamente equivalente ao problema original e possui as seguintes características:

- É definido em termos de λ_j: A resolução do problema mestre fornece os valores ótimos dos λ_j, podendo-se obter os correspondentes valores das variáveis originais (x) com recurso à expressão (6) e às soluções extremas dos subproblemas.
- Possui apenas $m_1 + 1$ linhas, que se comparando às $m_1 + m_2$ linhas do problema original, pode constituir uma redução drástica e um ganho significativo, se m_2 for grande o suficiente.
- Possui também tantas colunas quanto o conjunto convexo C_2 possui pontos extremos, um valor que pode ser incomportável no caso em que m_2 é grande.

Este último aspecto é definitivo, pois não é viável nem necessário considerar todas essas colunas para encontrar a solução do problema mestre. Se assim fosse, não se justificaria fazer uma transformação para um problema mais trabalhoso que o original. Ao invés disso, usamos uma técnica de geração de colunas, gerando colunas para entrar na base à medida que for necessário. Para ilustrar o processo, considere o custo reduzido (d_N) para a variável λ_j:

$$d_{N,j} = f_j - \pi [\, p_j^T \;\; 1 \,]^T \tag{16}$$

Particionamos π, o vetor de multiplicadores simplex, em $\pi = (\pi_1, \pi_0)$, onde π_1 corresponde às restrições (13) e o escalar π_0 à restrição (14). Então, usando as definições de f_j e p_j em (10) e (11), $d_{N,j}$ pode ser escrito como

$$d_{N,j} = (c - \pi_1 A_1)x^j - \pi_0 \tag{17}$$

O critério usual do Simplex para selecionar a variável λ_s a entrar na base é encontrar

$$\min \{d_{N,j}\} = d_{N,s} = (c - \pi_1 A_1)x^s - \pi_0 \tag{18}$$

É bom lembrar que x^j é um ponto extremo de C_2, e que $d_{N,j}$ é linear em x^j. Já que a solução ótima de um problema de programação linear cujo conjunto viável é limitado sempre ocorre em um ponto extremo desse conjunto, (18) é equivalente ao seguinte subproblema:

$$\text{minimizar } (c - \pi_1 A_1)x \tag{19}$$

$$\text{sujeito a} \qquad A_2 x = b_2, \qquad x \geq 0 \tag{20}$$

A fim de encontrar uma coluna para entrar na base do problema mestre, resolvemos este subproblema, obtendo uma solução x^s, que minimiza a função objetivo $(c - \pi_1 A_1)x$. Em outras palavras, se $(c - \pi_1 A_1)x^s - \pi_0 < 0$, a variável a entrar na base é λ_s, pois o custo reduzido correspondente ($d_{N,s}$) é o menor dentre todos os custos reduzidos e é menor do que zero ($\min\{d_{N,j}\} = (c - \pi_1 A_1)x^s - \pi_0 < 0$). Dessa forma, segundo o método Simplex, a coluna a entrar na base é

$$p_s = [\, (A_1 x^s)^T \;\; 1 \,]^T \tag{21}$$

que é formada segundo (10). A partir de (11), obtemos que o custo de x^s na função objetivo é

$$f_s = cx^s \tag{22}$$

A variável básica a sair da base é encontrada assim como no Simplex, ou seja, encontrando j tal que:

$$Y_j = (B^{-1}b)_j / (B^{-1}p_s)_j = \min \{(B^{-1}b)_k / (B^{-1}p_s)_k : (B^{-1}p_s)_k > 0,\ k = 1, ..., m\}$$

Claro que, no caso em que o menor custo reduzido Y_j for maior ou igual a zero, o problema mestre (e, portanto, o problema original) não é limitado, terminando o processo.

Tudo isso se torna particularmente atraente se p, o número de blocos independentes na estrutura bloco-angular, é maior do que um, ou seja, se o problema a ser resolvido tiver a forma

$$\text{minimizar } z = c_1x_1 + c_2x_2 + ... + c_px_p \tag{23}$$

$$\begin{aligned} \text{sujeito a} \quad & A_1x_1 + A_2x_2 + ... + A_px_p = b_0 \\ & B_1x_1 = b_1 \\ & B_2x_2 = b_2 \\ & \quad ... \quad ... \\ & B_px_p = b_p \end{aligned} \tag{24}$$

$$x_1 \geq 0,\ x_2 \geq 0,\ ...,\ x_p \geq 0, \qquad p \geq 1 \tag{25}$$

onde $A_i \in R^{m \times n_i}$, $B_i \in R^{m_i \times n_i}$.

Agora temos p subproblemas ao invés de um. A partir do Teorema 1 e assumindo que os poliedros convexos C_i's dados por $C_i = \{x_i \mid B_ix_i = b_i,\ x_i \geq 0\}$ (i = 1, ..., p) são todos limitados, chegamos ao problema mestre apresentado a seguir:

$$\text{minimizar } \Sigma f_{ij}\lambda_{ij} \tag{26}$$

$$\text{sujeito a} \quad \Sigma p_{ij}\lambda_{ij} = b_0 \tag{27}$$

$$\Sigma \lambda_{ij} = 1 \qquad (i = 1, ..., p) \tag{28}$$

$$\lambda_{ij} \geq 0 \tag{29}$$

onde

$$f_{ij} = c_ix^j_i \tag{30}$$

$$p_{ij} = A_ix^j_i \tag{31}$$

com os x^j_i's sendo os pontos extremos do poliedro C_i.

O problema (26)-(29) difere do problema (12)-(14) obtido anteriormente em dois aspectos fundamentais:

- O novo problema mestre excede em p (e não em 1) o número de restrições de acoplamento do problema original;
- Cada subproblema i é representado separadamente por um conjunto de variáveis λ_{ij}, como se segue:

$$B_i x_i = b_i, \quad x_i \geq 0 \tag{32}$$

$$x_i = \Sigma\, \lambda_{ij} x^j_i \tag{33}$$

Resolveremos (26)-(29) pelo método Simplex usando geração de colunas. Seja B uma base (de dimensões $(m_1 + p)$ x $(m_1 + p)$) e sejam $(\pi, \pi_{01}, \ldots, \pi_{0p})$ os multiplicadores simplex para esta base, onde π está associado às restrições (27) e π_{0i} às restrições (28). Dessa forma, obtemos o seguinte custo reduzido para λ_{ij}:

$$d_{N,ij} = (c_i - \pi A_i) x^j_i - \pi_{0i} \tag{34}$$

A fim de escolher uma variável a entrar na base, temos que calcular o menor dentre todos os custos reduzidos $d_{N,ij}$. Em princípio, ter-se-ia que dispor, neste ponto, de todas as colunas correspondentes a variáveis não-básicas, o que, como se disse, tornaria desinteressante a aplicação da decomposição. Entretanto, não precisamos conhecer todos os custos reduzidos $d_{N,ij}$, mas apenas encontrar $\min\{d_{N,ij}\}$, para um determinado i fixo. Isto corresponde a encontrar a solução extrema x^s_k que

minimiza o valor da expressão (34), o que é equivalente a resolver o i-ésimo subproblema que tem a expressão (34) como função objetivo:

$$\text{minimizar } (c_i - \pi_1 A_1)x_i \tag{35}$$

$$\text{sujeito a } \quad B_i x_i = b_i, \qquad x_i \geq 0, \tag{36}$$

Encontrando a solução de todos os p subproblemas e chamando de z^0_i, i = 1, ..., p, a solução do subproblema i, temos:

- Se $\min\{z^0_i - \pi_{0i}\} \geq 0$, todas as colunas correspondentes a variáveis não-básicas do problema mestre possuem custos reduzidos não-negativos, o que corresponde a dizer que a solução ótima foi encontrada. Pode–se então reconstruir a solução ótima do problema original, recorrendo à expressão (6) e às soluções extremas dos subproblemas que estão na base.

- Caso contrário, a coluna s a entrar na base é tal que $\min\{z^0_i - \pi_{0i}\} = z^0_s - \pi_{0s}$, onde x_s é solução ótima do subproblema s. Concluindo, temos então que a nova coluna a entrar na base é dada por

$$[\ (A_s x_s)^T \quad u_s^T\ ]^T \tag{37}$$

  onde u_s é um vetor de p componentes com um 1 na posição s e com zero nas demais.

Repare que, em todo o processo, não foi necessário gerar explicitamente mais do que p soluções extremas dos subproblemas (uma para cada). Uma vez formada a coluna a entrar na base, a seleção da variável a sair da base e as operações de pivoteamento fazem-se conforme o algoritmo original do Simplex. O processo continua com o cálculo do novo vetor de multiplicadores simplex.

2.5.2.2.O Algoritmo

A seguir, apresentaremos o algoritmo para resolver o problema (23)-(25) utilizando o princípio de decomposição descrito acima. Suponha que temos uma solução básica viável inicial para o problema mestre que está associada à base B e aos multiplicadores simplex (π_1, π_{0i}).

1) Usando os multiplicadores simplex π_1, gere novos coeficientes de função objetivo para cada um dos subproblemas (32)-(33) e os resolva, obtendo soluções ótimas $x_i(\pi_1)$ e valores ótimos para função objetivo z_i^0.

2) Considerando o critério usual do Simplex para escolha da variável a entrar na base, calcule min $\{d_{N,ij}\}$ (custo reduzido do problema mestre), que é o mínimo entre os mínimos custos reduzidos de cada subproblema. Se min $d_{N,ij}$ = min$\{z^0_i - \pi_{0i}\} = z^0_s - \pi_{0s} \geq 0$, a solução ótima do problema mestre foi encontrada e o procedimento deve parar. Pode-se calcular a solução ótima do problema original a partir da solução ótima do problema mestre e das soluções extremas dos subproblemas.

3) Se $z^0_s - \pi_{0s} < 0$, forme uma nova coluna para entrar na base multiplicando B^{-1} por p, onde $p = [(A_s x^s)^T \; u_s^T]^T$ e u_s é um vetor com todas as posições nulas exceto a posição s, que é igual a 1. O novo coeficiente da função objetivo é dado por $f_s = c_s x^s$.

4) Encontramos uma variável a sair da base usando o mesmo critério do Simplex. Se for verificado que o problema mestre é ilimitado o procedimento deve parar.

5) Tendo-se selecionado uma variável a sair da base, efetuamos as operações de pivoteamento, como no algoritmo Simplex normal, obtendo nova inversa da base e um novo vetor de multiplicadores simplex.

6) Retornamos ao passo 1 e repetimos o procedimento até que a solução ótima seja encontrada.

Se o problema mestre é não degenerado, o valor da função objetivo (z) decresce a cada iteração de uma quantidade não nula, e, portanto, as bases não são repetidas. Já que existe uma quantidade finita de possíveis bases, o princípio de decomposição

encontra o ótimo em um número finito de iterações. Note que a solução ótima do problema mestre não é necessariamente uma das soluções dos subproblemas, mas sim uma combinação linear convexa dessas soluções. É válido lembrar que, como nem sempre dispomos de uma solução básica viável inicial, é necessário modificar o algoritmo acima de modo a considerar o caso em que não exista tal solução. A construção dessa solução é semelhante à dos problemas comuns (sem nenhuma estrutura especial), recorrendo-se a variáveis de folga e variáveis artificiais.

2.5.3. Considerações sobre Eficiência do Algoritmo

Com relação à eficiência do algoritmo, é esperado que a parte mais pesada do processo descrito anteriormente seja a resolução dos subproblemas. No entanto, é possível diminuir bastante o tempo de execução, notando que, em muitos casos, a solução ótima de alguns subproblemas não muda (ou exige apenas mais uma iteração) entre duas iterações sucessivas do problema mestre. O uso de procedimentos de pós-otimização revela-se aqui bastante útil.

Uma outra hipótese consiste na alteração da estratégia de seleção das variáveis a entrar na base, escolhendo, não o custo reduzido mínimo, mas o primeiro custo reduzido negativo. Essa abordagem diminuiria o número médio de subproblemas a serem resolvidos em cada iteração do método, mas poderia forçar um aumento no número de iterações do problema mestre (ou seja, o método poderia convergir mais lentamente para a solução ótima), o que seria uma desvantagem dessa abordagem.

Finalmente, torna-se evidente que a aplicação do princípio de decomposição implica em grande cuidado na implementação computacional, dadas as dimensões envolvidas nos problemas reais em que este procedimento se torna interessante. A eficiência global, nomeadamente em relação aos tempos de execução, e a obtenção de soluções corretas e precisas, implicam normalmente na escrita de muitas linhas de código, que permitam aproveitar todas as particularidades favoráveis da estruturas dos problemas.

2.6.Resumo

O Princípio de Decomposição Dantzig-Wolfe é baseado na versão revisada do método Simplex e fornece um procedimento que se torna muito eficiente quando aplicado a problemas de programação linear de grande porte cuja matriz dos coeficientes tem uma estrutura bloco-angular. O método trabalha formando um problema equivalente (*problema mestre*) ao problema original, que possui um número de linhas um pouco maior do que o número de restrições de acoplamento do problema original e um número de colunas que pode vir a ser muito grande. O problema equivalente é resolvido usando técnicas de geração de colunas (*colunm geberation*), que consiste em resolver o problema sem considerar todas as colunas, gerando apenas as que são necessárias a cada iteração.

O algoritmo resultante trabalha com uma interação entre um conjunto independente de subproblemas e o problema mestre. Os subproblemas são obtidos a partir dos blocos independentes da matriz dos coeficientes. A interação prossegue até que um determinado critério de otimalidade seja atingido.

A aplicação do princípio de decomposição implica em grande cuidado na implementação computacional, dadas as dimensões envolvidas nos problemas reais em que este procedimento se torna interessante.

No próximo capítulo, iremos apresentar o conceito de paralelismo e explicar como este foi incorporado ao método de decomposição Dantzig-Wolfe, enfocando a metodologia utilizada.

Capítulo 3

O Paralelismo

Este capítulo mostrará os conceitos relacionados ao paralelismo, além de como este foi empregado na implementação do algoritmo de decomposição visto no capítulo anterior. A seção 3.1 mostra a motivação para a inclusão do paralelismo em sistemas de computação. A seção 3.2 mostra os principais paradigmas do processamento paralelo, apresentando conceitos e citando ferramentas de hardware e de software comumente usadas na implementação do paralelismo. A seção 3.3 apresenta o algoritmo paralelo formulado para o Princípio de Decomposição Dantzig-Wolfe, bem como alguns detalhes e considerações referentes à implementação. A seção 3.4 mostra o resumo do capítulo.

3.1.Introdução

Já se passaram cerca de 60 anos desde a introdução do primeiro computador eletrônico digital por John von Neumann. As cinco décadas passadas têm presenciado muitos avanços na tecnologia da computação que foram possibilitados pela disponibilidade de componentes eletrônicos mais baratos e bem mais rápidos. Porém, a grande maioria dos computadores atuais, suas arquiteturas e seu modo de operar, seguem mais ou menos o mesmo modelo conceitual formulado por von Neumann (Amorim, 1988). Tal abordagem baseia-se em uma unidade de controle (UC) que obtém uma instrução e seus parâmetros de uma unidade de memória (UM) e envia esses elementos para uma unidade de processamento (UP). Na unidade de processamento, a instrução é executada e o resultado produzido é enviado de volta para a unidade de memória. Em um computador que trabalha dessa forma, as instruções são executadas seqüencialmente, ou seja, uma após a outra.

Muitos esforços têm sido feitos no sentido de obter circuitos mais rápidos que permitam acelerar a velocidade de processamento dos computadores. Muitas técnicas foram desenvolvidas a fim de melhorar o modelo proposto por von Neumann, como, por exemplo, a *memória cache* e *pipelines* (que mais tarde levaria ao conceito de *supercomputador*) (Amorim, 1988). Porém, os avanços obtidos nestas pesquisas não vêm se mostrando muito satisfatórios, visto que as necessidades de processamento crescem de maneira bem mais rápida. Essa disparidade se deve ao fato evidente de que o crescimento da velocidade de processamento dos computadores seqüenciais (monoprocessados) é limitado pela rapidez de evolução da tecnologia dos componentes eletrônicos empregados para construir processadores. Uma saída nesse caso seria permitir que os computadores possuíssem não somente um, mas vários agentes processadores trabalhando em conjunto. Foi assim que surgiu a idéia de incluir paralelismo nos sistemas de computação, o que deu origem a um dos conceitos mais excitantes do ramo da Ciência da Computação, a *Computação Paralela*.

A Computação Paralela fornece uma abordagem diferente daquela proposta por von Neumann para a resolução de um problema através de um computador. A arquitetura de um *computador paralelo* consiste em uma coleção de unidades de processamento (processadores ou computadores) que trabalham cooperativamente no sentido de resolver um determinado problema de forma simultânea, com cada unidade de processamento trabalhando em uma parte do problema de uma forma independente. Esse novo conceito permite aumentar de forma apreciável a capacidade de processamento das máquinas, já que o tempo necessário para resolver um problema através do uso de um único processador tende a ser significantemente reduzido quando são usados múltiplos agentes (processadores) (Zomaya, 1996).

A pesquisa em torno da Computação Paralela vem sendo bastante motivada por uma variedade de fatores. Por exemplo, os avanços da tecnologia VLSI (very large scale integration) facilitaram a produção de componentes eletrônicos de alta velocidade e de preço bem mais acessível. Isso estimulou o uso da computação paralela para resolver problemas computacionalmente difíceis, e os resultados obtidos foram bem mais satisfatórios do que os obtidos com o uso de outras arquiteturas. Segundo Blelloch (1996), desde a década de 70, muitos pesquisadores têm se esforçado no sentido de

desenvolver e analisar algoritmos paralelos eficientes para resolver problemas que não possuem bons algoritmos seqüenciais.

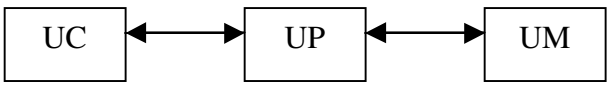
A área de Processamento Paralelo oferece um grande potencial no que diz respeito ao desempenho na solução de problemas complexos, e por outro lado traz dificuldades aos projetistas de software, face à necessidade de serem reprojetados quase todos os algoritmos. Talvez mais do que em qualquer área, é essencial uma abordagem em que tanto a teoria, quanto o hardware e o software do sistema sejam intimamente interligados. Além da própria Ciência da Computação, em que problemas de difícil tratamento ocorrem em Otimização Combinatória, algumas das áreas em que o *Processamento Paralelo* oferece possibilidades promissoras incluem as diversas Engenharias, Física, Meteorologia e Economia, entre outras.

3.2.Paradigmas do Processamento Paralelo

3.2.1. Classificação das Arquiteturas de Computadores

Computadores operam executando instruções em um conjunto de dados. Um fluxo de instruções informa ao computador o que ele deve fazer em cada passo. Em 1966, Flynn propôs o primeiro método para classificar arquiteturas de computadores (Flynn, 1966). O método se baseia nas possibilidades de combinação entre uma ou mais seqüências de instruções atuando sobre uma ou mais seqüências de dados, produzindo assim as seguintes quatro classes de computadores:

- **SISD** (**S**ingle **I**nstruction stream, **S**ingle **D**ata stream): Uma seqüência de instruções, uma seqüência de dados. Corresponde aos computadores seqüenciais convencionais de von Neumann, nos quais só existe uma única unidade de controle (UC) que decodifica seqüencialmente instruções que operam sobre um conjunto de dados. Um algoritmo que é executado por um computador SISD é chamado de *algoritmo seqüencial*.



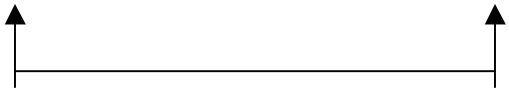


Figura 3.1: Arquitetura SISD (UC = Unidade de Controle, UP = Unidade de Processamento, UM = Unidade de Memória)

- **SIMD** (**S**ingle **I**nstruction stream, **M**ultiple **D**ata stream): Uma seqüência de instruções, múltiplas seqüências de dados. Corresponde aos processadores matriciais nos quais vários elementos processadores são conectados por uma única unidade de controle (UC). Esta decodifica seqüencialmente instruções e as transmite para todos os elementos processadores (UP's), que as executam em conjuntos diferentes de dados provenientes de fluxo de dados distintos. Por exemplo, para um computador SIMD com N processadores, cada processador executará a mesma instrução ao mesmo tempo, mas cada um em seu próprio conjunto de dados. Nesse caso os processadores trabalham de forma *síncrona*. Para se obter alto desempenho usando um computador SIMD é necessário reescrever algoritmos convencionais a fim de manipular dados simultaneamente enviando instruções para todos os processadores.

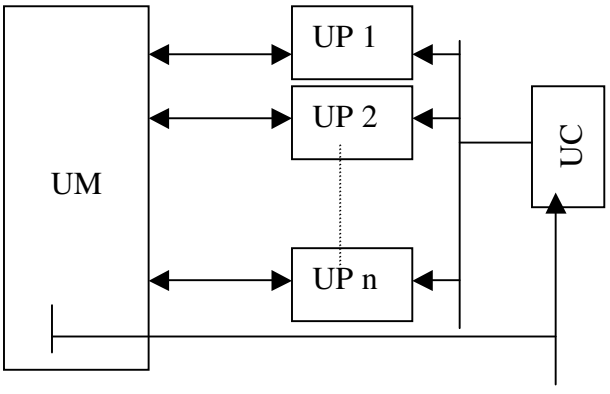


Figura 3.2: Arquitetura SIMD (UC = Unidade de Controle, UP = Unidade de Processamento, UM = Unidade de Memória)

- **MISD** (**M**ultiple **I**nstruction stream, **S**ingle **D**ata stream): Múltiplas seqüências de instruções, uma seqüência de dados. Cada processador tem sua própria unidade de controle e compartilha uma unidade de memória comum

onde os dados residem. O paralelismo é realizado permitindo que cada processador execute uma operação diferente no mesmo conjunto de dados ao mesmo tempo. Computadores pertencentes a essa classe não são muito comuns.

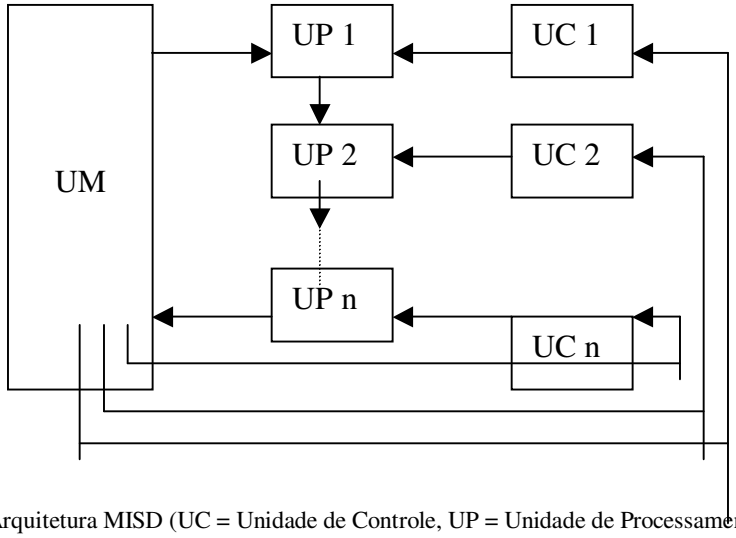


Figura 3: Arquitetura MISD (UC = Unidade de Controle, UP = Unidade de Processamento, UM = Unidade de Memória)

- **MIMD** (**M**ultiple **I**nstruction stream, **M**ultiple **D**ata stream): Múltiplas seqüências de instruções, múltiplas seqüências de dados. Computadores nos quais o paradigma de computação paralela melhor se aplica. Nesse caso, existem N processadores, N fluxos de instruções e N fluxo de dados. Os processadores são do mesmo tipo daqueles usados em uma máquina SISD, ou seja, cada processador tem sua própria unidade de controle e sua memória local. Tais processadores são considerados bem mais poderosos do que os usados por uma máquina SIMD.

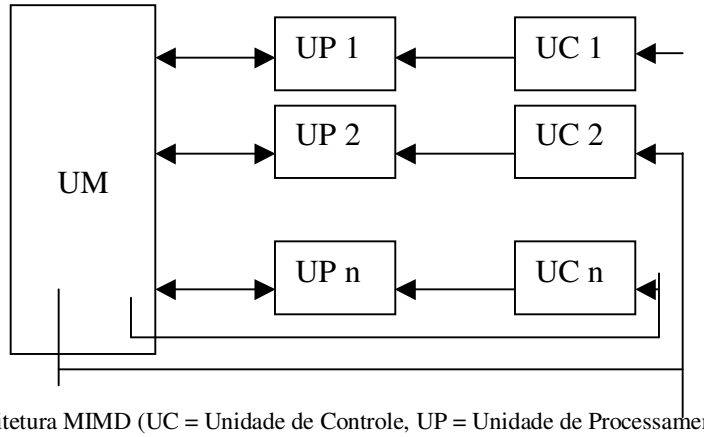


Figura 3.4: Arquitetura MIMD (UC = Unidade de Controle, UP = Unidade de Processamento, UM = Unidade de Memória)

Das quatro abordagens apresentadas acima, o modelo MIMD é a mais utilizada para construir computadores paralelos, seguido de perto pelo modelo SIMD e por último, pelo modelo MISD. Como complemento para a classificação de Flynn, uma outra classe, conhecida como SPMD (**S**ingle **P**rogram, **M**ultiple **D**ata), é usada para descrever casos em que programas que possuem o mesmo código devem ser executados em conjuntos de dados distintos, de forma *síncrona* (como em SIMD), ou de forma *assíncrona* (como um caso especial de MIMD).

Zomaya (1996) ainda divide os computadores com arquitetura paralela em duas categorias, de acordo com o modelo de acesso à memória: os que possuem *memória compartilhada* (*shared memory*) e os que possuem *memória distribuída* (*distributed memory*). No primeiro caso os processadores não possuem nenhuma memória local e compartilham uma memória comum. A memória é acessada por somente um processador de cada vez, o que reduz a velocidade de acesso à memória. Por outro lado, a comunicação entre os processadores pode ser feita através da escrita/leitura na memória, o que aumenta a velocidade de comunicação. A desvantagem é que são necessárias técnicas de sincronização para a leitura e gravação. No caso da memória distribuída, cada processador possui sua própria memória local que só pode ser acessada por ele. Isso aumenta a velocidade de acesso à memória, já que esse acesso será feito sem interferência. Uma outra vantagem é o fato de não existir limite para o número máximo de processadores. Nessa abordagem, a comunicação entre os

processadores é feita através de troca de mensagens (*message passing*), o que gera um *overhead* que reduz a velocidade de comunicação.

3.2.2. Conceitos Importantes

Quando dividimos um problema grande e complexo em vários problemas menores e mais simples que podem ser resolvidos simultaneamente por processadores distintos de forma independente, dizemos que tal problema é *paralelizável* (veja Figura 3.5). Podem existir dois tipos diferentes de paralelismo em um problema paralelizável. O *paralelismo de dados* ocorre quando cada subproblema obtido a partir da quebra do problema maior corresponde à mesma tarefa, ou seja, os processadores executam o mesmo conjunto de instruções sobre conjuntos de dados diferentes. Um outro caso é o do *paralelismo funcional*, que ocorre quando os subproblemas são diferentes entre si, ou seja, cada processador executa um conjunto de instruções diferente sobre um conjunto de dados que não é necessariamente o mesmo.

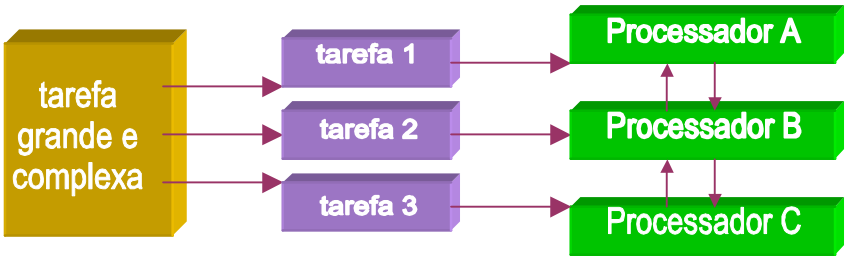


Figura 3.5: Processamento Paralelo.

Para encontrar a solução de um problema paralelizável deve-se desenvolver um *algoritmo paralelo*. Segundo Amorim (1988), um algoritmo paralelo corresponde a uma coleção de processos concorrentes executando de forma cooperativa para resolverem um dado problema. Os processos precisam interagir para sincronizarem suas atividades e para trocar informações. Dessa forma, podemos classificar os algoritmos paralelos em *síncronos* e *assíncronos*, de acordo com a forma de sincronização. Se, num algoritmo, determinados processos devem esperar outros processos completarem para continuarem executando, eles são denominados algoritmos síncronos. Neste caso o tempo de processamento será determinado pelo tempo que o processo mais lento leva para

executar. No caso dos algoritmos assíncronos, nenhum processo precisa esperar outro completar sua tarefa, resultando, teoricamente, em algoritmos mais rápidos, já que um processo pode continuar executando em vez de ficar esperando outros processos terminarem de executar alguma tarefa.

3.2.3. Custo e Cálculo de Desempenho

Arquiteturas de sistemas monoprocessados e de sistemas de computação paralela possuem muitas características comuns. Porém, existem vários aspectos relacionados a paralelização que não estão presentes em sistemas seqüenciais. É necessário conhecer e levar em consideração esses aspectos antes de avaliar o desempenho de algoritmos paralelos.

Uma dessas características é a *alocação de tarefas*, que consiste na quebra da carga total de trabalho em tarefas menores que são atribuídas a diferentes processadores, sempre levando em consideração as tarefas que são interdependentes e não podem ser executadas simultaneamente. Para atingir o mais alto grau de desempenho, é importante garantir que as tarefas estejam distribuídas de forma que o potencial máximo de cada processador esteja sendo utilizado. Esse processo é denominado *balanceamento de carga* (*load balancing*) e é considerado um problema extremamente difícil de resolver, pertencente a uma classe de problemas chamada NP-completo (*nondeterministic-polynomial-time-complete*). Um segundo aspecto bastante relevante, que já foi mencionado anteriormente, é o tempo de comunicação entre processadores. O problema da comunicação cresce à medida que o número de processadores aumenta e é altamente dependente da tecnologia empregada nas linhas de comunicação utilizadas para interligar os processadores.

Uma vez que um algoritmo paralelo tenha sido elaborado, é interessante que exista uma maneira de avaliá-lo e analisá-lo quanto a sua eficiência. Algumas das medidas usadas na avaliação de algoritmos paralelos serão apresentados a seguir.

3.2.3.1.Tempo de Execução

Acelerar a velocidade de processamento parece ser uma das razões fundamentais para o desenvolvimento de sistemas de computação paralela, e a única medida de

desempenho que é completa e de confiança é o tempo de execução (Zomaya, 1996), que é definido como *o tempo gasto pelo algoritmo para resolver um problema em um computador paralelo*, ou ainda *o tempo passado desde o momento em que o algoritmo inicia até seu término*.

Se nem todos os processadores de um computador paralelo começam e terminam sua execução simultaneamente, então o tempo de execução é igual ao tempo passado desde o momento em que o primeiro processador começou a executar até o momento em que o último processador conclui suas operações. Dependendo da aplicação, o tempo de execução pode ser medido em segundos (s) ou em milissegundos (ms).

O desempenho de um processador também pode ser medido em termos no número de instruções de máquina executadas por unidade de tempo. Uma das unidades mais comuns é o MIPS (millions of instructions per second). Outra unidade bastante comum é a MFLOPS (millions of floating-point operations per second). Essas medidas são especialmente úteis para supercomputadores e para aplicações científicas ou de engenharia.

3.2.3.2.Speedup

É bastante natural avaliar um algoritmo paralelo comparando-o com o melhor algoritmo seqüencial desenvolvido para o problema em questão. Portanto, o melhor indicador da qualidade de um algoritmo paralelo seria a relação entre o tempo de execução do algoritmo seqüencial e o tempo de execução do algoritmo paralelo. Essa medida é chamada de *speedup* ou *fator de aceleração*, e é definido como

$$S(N) = \frac{\text{Tempo de Execução do Algoritmo Seqüencial } (t_s)}{\text{Tempo de Execução do Algoritmo Paralelo } (t_p)} \qquad (1)$$

É óbvio que quanto maior o speedup (S(N)) melhor é o algoritmo paralelo. Teoricamente, o ideal seria atingir um speedup igual a N quando se usar N processadores para resolver um problema em paralelo. Entretanto, na prática, tal speedup é impossível de ser atingido por vários motivos:

a. É quase sempre impossível decompor um problema em N tarefas, com cada uma precisando, para executar, de 1/N do tempo necessário para executar o problema original seqüencialmente.

b. Na maioria dos casos, a estrutura do computador paralelo usado para resolver o problema impõe restrições que tornam o tempo de execução desejado inatingível (por exemplo, o *overhead* de sincronização).

Segundo a *Lei de Amdahl* (1988), o speedup é limitado pela quantidade de paralelismo embutida em um algoritmo, que pode ser caracterizada por um parâmetro *f*, a fração de processamento que deve ser feita seqüencialmente. Amdahl afirma que o speedup máximo de um sistema com N processadores ao executar um algoritmo é uma função de *f*, como mostrado abaixo

$$S(N) \leq \frac{1}{f + (1 - f) / N} \leq \frac{1}{f} \qquad (2)$$

Perceba que S(N) = 1 (ou seja, nenhum speedup) quando *f* = 1 (ou seja, todas as operações feitas seqüencialmente), S(N) = N (speedup ideal) quando f = 0 (todas as operações feitas em paralelo). É válido lembrar que, na maioria dos casos, o tempo de comunicação entre os processadores contribui substancialmente para aumentar o tempo de execução e esse percentual de tempo não é levado em consideração na fórmula de Amdahl. Geralmente, o maior desempenho é atingido quando há um balanceamento entre o paralelismo e o overhead de comunicação. A idéia é encontrar um ponto de equilíbrio onde a adição de mais processadores não fornece uma melhora de desempenho devido o tempo gasto com comunicação.

Segundo Kronsjö (1996), uma forma de medir a *eficiência* (ϕ) da resolução de um problema através de um algoritmo paralelo é calcular a razão entre o speedup (S(N)) e o número de processadores utilizados (N), como mostrado em (3)

$$\frac{S(N)}{\qquad} \qquad (3)$$

$$\phi = \frac{S(N)}{N} \qquad (3)$$

Assim, podemos definir eficiência como a fração de tempo que os processadores estão sendo utilizados para processamento. É fácil notar que $\phi \leq 1$, e o ideal seria obter 100% de eficiência, ou seja, $\phi = 1$. Porém, para isso acontecer, o algoritmo paralelo deveria ser tal que, nunca, nenhum processador poderia estar ocioso ou fazendo algum outro trabalho que não fosse resolver o problema. Tal situação é impossível de ocorrer.

Um outro aspecto importante relacionado à eficiência é a *escalabilidade*. Dizemos que quando um problema é escalável o suficiente, podemos sempre obter eficiência aumentando o número de processadores. Porém, assim como em computadores seqüenciais, o desempenho é limitado pelas leis da física (Zomaya, 1996). Em geral, esperamos os melhores resultados em speedup quando temos o número de processadores igual ou próximo ao número de tarefas a serem executadas em paralelo.

Finalmente, podemos introduzir o conceito de *custo* de um processamento, que pode ser obtido a partir do produto entre o tempo de execução e o número de processadores utilizados. O custo de uma execução seqüencial é simplesmente o tempo de execução (t_s). O custo de execução paralela é obtido multiplicando o tempo da execução paralela (t_p) pelo número de processadores utilizados (N). Dessa forma, a partir de (1) e (3), temos:

$$t_p \times N = \frac{t_s}{S(N)} \qquad \frac{t_s}{\phi}$$

O custo é considerado ótimo para um determinado algoritmo paralelo quando o custo da execução paralela é proporcional ao custo da execução seqüencial (Zomaya, 1996).

3.2.4. Software para Programação Paralela

O método mais conveniente de desenvolver programas para serem executadas em um computador paralelo seria o uso de um compilador que automaticamente

paralelizasse programas seqüenciais (*conversão automática*). Um exemplo é um compilador que tenta fazer paralelização automática de loops. Uma desvantagem desse método é a pouca habilidade dos compiladores em detectar o paralelismo eficientemente, o que depende da estrutura do programa. Até hoje, a paralelização automática de programas seqüenciais não tem resultado em programas paralelos eficientes (Jurczyk, 1996). Em geral, o próprio programador é responsável por mapear um algoritmo seqüencial em um paralelo (*conversão manual*). Segundo Perrott (1996), duas abordagens são usadas para implementar algoritmos paralelos:

- Definir uma nova linguagem de programação.
- Extender linguagens seqüenciais existentes, como C e FORTRAN, por exemplo, através da inclusão de bibliotecas de rotinas.

Qualquer que seja a abordagem utilizada, ela deve levar em consideração características como comunicação entre processadores e sincronização, e isso pode ser feito através do mecanismo de *message passing* (troca de mensagens) ou através de memória compartilhada distribuída. No nosso trabalho utilizamos a segunda abordagem usando o mecanismo de troca de mensagens fornecido pela biblioteca de rotinas PVM.

3.2.4.1.Parallel Virtual Machine (PVM)

O software que utilizamos para implementar o paralelismo chama-se PVM (*Parallel Virtual Machine*) e foi desenvolvido pelo *Oak Ridge National Laboratory* (ORNL) da Universidade de Tennessee. O PVM é uma das ferramentas mais populares para o desenvolvimento de programas paralelos e permite que uma coleção heterogênea de estações de trabalho e supercomputadores (máquinas de diferentes arquiteturas) interligados por uma rede de comunicação funcione como uma única máquina paralela de alto desempenho (*Máquina Virtual Paralela*) (Dongarra, 1994).

O PVM é um conjunto de integrado de ferramentas de software e bibliotecas que emulam uma plataforma de computação concorrente heterogênea, flexível, de propósito geral, de computadores interconectados de arquiteturas variadas (Oliveira, 1999). PVM possui um modelo de programação simples que é baseado em um conjunto de processos assíncronos que são executados tipicamente como programas individuais em cada sistema da rede. A comunicação entre esses processos ocorre por troca de mensagens

explícita. O PVM transparentemente manipula todo o roteamento de mensagens, conversão de dados, e escalonamento através da rede. O usuário escreve uma aplicação como um conjunto de tarefas cooperativas e cada tarefa acessa o PVM através de uma biblioteca de rotinas de interface padrão. Estas rotinas permitem determinar o início e o fim de tarefas na rede bem como controlar a comunicação e a sincronização entre elas.

O sistema PVM é composto de duas partes:

- PVM Daemon (*pvmd*): Reside em todos os computadores que compõem a máquina virtual e é o responsável pelo controle e roteamento das mensagens. Ele fornece um ponto de contato, autenticação, controle de processos e detecção de falhas.
- Biblioteca de Programação (libpvm): Uma biblioteca de rotinas contendo um repertório completo de primitivas que são necessárias para a cooperação entre tarefas de uma aplicação. Permite a uma tarefa interagir com o *pvmd* e com outras tarefas, e contém rotinas que podem ser chamadas pelo usuário para passagem de mensagens, distribuição de processos, coordenação de tarefas e modificação da máquina virtual. *Libpvm* é escrita em C e suporta aplicações em C e C++. Existem versões de bibliotecas PVM que dão suporte a outras linguagens (como FORTRAN e Java, por exemplo).

O PVM usa o paradigma de programação *Mestre/Escravo*. Um processo (*mestre*) coordena e controla a execução dos outros processos (*escravos*), e é responsável por criar os processos que vão trabalhar no problema, coordenar a entrada de dados iniciais para cada processo e coletar os resultados produzidos por cada um. A comunicação entre processos mestre e escravos ocorre nas situações a seguir:

- Processo mestre inicializa (*spawn*) processos escravos.
- Processo mestre transmite dados para os processos escravos realizarem a computação necessária.
- Processos escravos recebem dados do processo mestre.
- Processos escravos transmitem os resultados para o processo mestre.

- Processo mestre recebe resultados produzidos pelos processos escravos.

Em PVM, a transmissão dos dados é composta pela inicialização de um buffer onde serão colocados os dados, pelo empacotamento desses dados para envio (codificação da mensagem) e pelo envio dos dados pela rede. A recepção dos dados consiste na coleta dos dados que chegam pela rede e no desempacotamento desses dados (decodificação da mensagem). A biblioteca *libpvm* contém funções para empacotar todos os tipos de dados primitivos em uma mensagem, em um dos vários formatos de codificação.

Na grande maioria dos casos, a implementação de algoritmos paralelos em computadores paralelos usando PVM ocorre da seguinte forma. O programador decompõe o programa seqüencial em programas separados e cada um deles será escrito em uma linguagem de programação convencional (como C ou Fortran) e compilado para ser executado nas diferentes máquinas da rede. Cada programa corresponde a uma tarefa compondo a aplicação. O conjunto de máquinas (*máquina virtual*) que será utilizado para processamento deve ser definido antes do início da execução dos programas. Para isso, cria-se um arquivo (*hostfile)* com o nome de todas as máquinas disponíveis que será lido pelo PVM.

Para executar uma aplicação, um usuário geralmente inicia um processo (usualmente o mestre) manualmente em uma máquina. Este processo subseqüentemente inicia outros processos PVM (escravos), eventualmente resultando em um conjunto de processos que executam um processamento local e trocam mensagens entre si a fim de resolver o problema dado.

Para implementar o Princípio de Decomposição Dantzig-Wolfe, que foi descrito no capítulo 2, sob uma abordagem paralela, usamos o PVM e seguimos esse mesmo procedimento. O motivo pelo qual escolhemos o PVM como ferramenta de paralelização será apresentado posteriormente. Na próxima seção, apresentamos, com detalhes, todo o processo de paralelização do método de decomposição. Para obter mais informações sobre programação paralela e PVM consulte (DONGARRA 1994) ou então o site oficial do PVM na Internet.

3.3.Abordagem Paralela para o Princípio de Decomposição Dantzig-Wolfe

O objetivo principal do processamento paralelo é executar computações de maneira mais rápida e eficiente, usando vários processadores trabalhando de forma concorrente. Muitas aplicações, principalmente as que trabalham com uma grande quantidade de dados (como em problemas de otimização de grande porte), podem ser executadas mais eficientemente usando computação paralela (Jaja, 1992). Lembrando o que foi discutido no capítulo 2, o Princípio de Decomposição Dantzig-Wolfe é um método extremamente adequado para resolução de problemas de programação linear de grande porte, quando a matriz dos coeficientes do problema possui uma estrutura bloco-angular. O método de decomposição consiste na transformação do problema original em um problema equivalente chamado problema mestre (obtido a partir das restrições de acoplamento), que interage com um conjunto independente de subproblemas (obtidos a partir dos blocos independentes da matriz dos coeficientes) até que a solução ótima tenha sido encontrada. A idéia aqui é aliar o princípio de decomposição à computação paralela com o intuito de obter um algoritmo paralelo eficiente, que forneça um custo próximo do ótimo em relação aos algoritmos seqüenciais presentes nos softwares comerciais atuais (como o CPLEX, por exemplo).

Não é difícil perceber que o método de decomposição tem um algoritmo que é naturalmente paralelo, pois a criação de vários subproblemas independentes que não interagem entre si permite que cada um dos mesmos seja resolvido por um processador diferente. Como os subproblemas só interagem com o problema mestre, o algoritmo paralelo correspondente sugere claramente o uso do modelo de programação mestre/escravo, caracterizado pela presença de um processo principal (processo mestre) que controla e coordena a execução de um conjunto de processos subordinados (processos escravos).

Numa abordagem paralela, os processos escravos são responsáveis pela resolução dos subproblemas e um processo mestre é responsável por coordenar a resolução do problema mestre. A cada iteração, seguindo o método de Dantzig-Wolfe, os processos escravos recebem do problema mestre um conjunto de parâmetros (multiplicadores simplex), que são utilizados para calcular os coeficientes da função objetivo de seu subproblema. Depois, eles encontram suas soluções (usando o Simplex Revisado, por exemplo) e então as enviam para o processo mestre. Este é incumbido de coletar as

soluções dos subproblemas e aplicá-las no problema mestre, verificando se o critério de parada do método foi atingido, ou seja, se a solução ótima foi encontrada. Se a solução obtida não for ótima, o processo mestre calcula novos multiplicadores simplex e os enviaria para os processos escravos (Veja Figura 3.6).

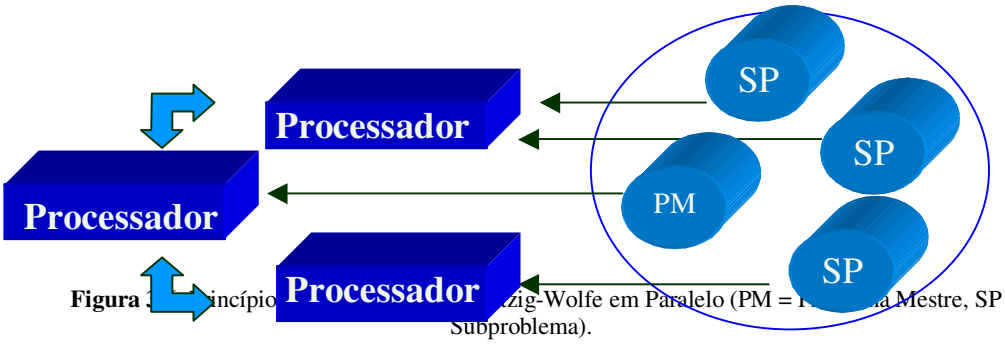


Figura [illegible] princípio [illegible]tzig-Wolfe em Paralelo (PM = [illegible]a Mestre, SP = Subproblema).

A partir das idéias apresentadas acima, formulamos um algoritmo paralelo para resolver um problema de programação linear usando o princípio de decomposição Dantzig-Wolfe. Tal algoritmo será descrito na seção a seguir.

3.3.1. O Algoritmo Paralelo

Em uma abordagem inicial, nosso principal propósito era desenvolver um algoritmo paralelo que resolvesse um problema de programação linear de grande porte usando o Princípio de Decomposição Dantzig-Wolfe, aproveitando a sua característica paralela. Nosso segundo objetivo seria otimizar o algoritmo resultante de forma a obter eficiência e um speedup interessante.

Considerando esses dois objetivos e o que foi discutido nas seções anteriores, formulamos o seguinte algoritmo paralelo para o Princípio de Decomposição Dantzig-Wolfe:

1. O *processo mestre* lê os dados do problema original e extrai desses dados as *restrições de acoplamento* (A), os subproblemas (B_i) e a *função objetivo* (c);

2. O *processo mestre* constrói o quadro inicial para o *Problema Mestre*, calcula uma solução básica viável inicial (associada a uma base B) e calcula o vetor de *multiplicadores simplex* correspondente (π);

3. O *processo mestre* envia para cada *processo escravo i* os coeficientes da *função objetivo* (c_i), os coeficientes das *restrições de acoplamento* (A_i), seu *subproblema* correspondente (B_i) e os *multiplicadores simplex* (π);

4. Cada *processo escravo i* recebe os dados enviados pelo *processo mestre* e calcula os coeficientes da *função objetivo* de seu *subproblema* ($c_i - \pi_1 A_i$);

5. Aplicando o Simplex Revisado, cada *processo escravo i* encontra a solução ótima (z_i^0) de seu *subproblema*, que corresponde a

$$\text{minimizar } (c_i - \pi_1 A_1) x_i$$
$$\text{sujeito a} \qquad B_i x_i = b_i, \qquad x_i \geq 0$$

6. Cada *processo escravo i* calcula o menor *custo reduzido* correspondente a seu *subproblema* ($z^0_i - \pi_{0i}$);

7. Cada *processo escravo i* envia para o *processo mestre* seu *custo reduzido* correspondente ($z_i^0 - \pi_{0i}$);

8. O *processo mestre* recebe dos *processos escravos* os menores *custos reduzidos* (d_N) e calcula o menor deles ($\min \{d_{N,ij}\} = \min\{z^0_i - \pi_{0i}\}$);

9. Se o menor dentre os menores *custos reduzidos* for maior ou igual a zero ($\min d_{N,ij} = z^0_s - \pi_{0s} \geq 0$), o procedimento deve parar. A solução ótima do *Problema Mestre* foi encontrada e pode ser calculada a partir da solução ótima do *Problema Mestre* e das soluções dos *subproblemas*. O *processo mestre* avisa aos processos escravos que o processamento deve parar;

10. Caso contrário (se $z^0_s - \pi_{0s} < 0$), o *processo mestre* gera a nova coluna a entrar na base multiplicando B^{-1} por p, onde $p = [(A_s x^s)^T \; u_s^T]^T$ e u_s é um vetor com todas as posições nulas exceto a posição s, que é igual a 1. O novo coeficiente da função objetivo é dado por $f_s = c_s x^s$.

11. O *processo mestre* determina a variável a sair da base (se existir), faz as operações de pivoteamento e obtém nova inversa da base e novo vetor de *multiplicadores simplex*;
12. O *processo mestre* envia os novos *multiplicadores simplex* para os *processos escravos*;
13. Volte para o passo 4.

Nas próximas seções iremos descrever como foi feita a implementação de cada um dos passos do algoritmo acima e mostrar abordagens alternativas de implementação. No capítulo 4 apresentaremos os resultados obtidos nos testes de desempenho do algoritmo e detalhes relacionados aos recursos computacionais utilizados.

3.3.2. A Implementação Passo a Passo

Na implementação do algoritmo da seção anterior, usamos a biblioteca de rotinas PVM para a linguagem C. Escolhemos essa combinação ferramenta/linguagem devido à portabilidade e flexibilidade fornecidas pela linguagem C, e devido a testes de desempenho e comparações efetuadas em trabalhos anteriores, quando verificamos que o PVM e a linguagem C obtêm um melhor desempenho do que outras combinações (como MPI e FORTRAN, por exemplo) quando aplicados na resolução de problemas que envolvem manipulação de vetores e matrizes (Pequeno, 2000).

No passo 1 do algoritmo, a leitura dos dados do problema original é executada. Para tanto implementamos um procedimento em C que lê, a partir de um arquivo de texto comum, um problema de programação linear cuja matriz dos coeficientes tem estrutura bloco-angular, supondo que os dados estão em um formato padrão utilizado pela maioria dos softwares comerciais da área (*LP file format*). Tal rotina é executada pelo processo mestre antes da criação dos processos escravos e os dados lidos são armazenados em variáveis (matrizes e vetores). Posteriormente, uma porção desses dados é enviada para os processos escravos. Uma abordagem diferente para esse passo seria permitir que os próprios processos escravos lessem os dados necessários para o processamento (seu subproblema, por exemplo), reduzindo o tempo gasto pelo processo mestre para ler o problema e eliminando a necessidade de envio dos dados para os

processos escravos. Entretanto, é preciso considerar o caso em que nem todos os módulos processadores da rede possuem memória secundária (como em máquinas *diskless*) e que os dados estão armazenados na memória secundária de apenas um (ou alguns) dos processadores (o qual deve executar o processo mestre). Nesse caso, o processo mestre seria obrigado a ler o problema original completo e decidir para quais processos escravos enviar os dados. A fim de generalizar e evitar essa complexidade optamos por usar a primeira abordagem, até porque o tempo gasto na leitura dos dados do problema não será considerado nos testes de desempenho.

Após a leitura dos dados, o processo mestre tem condições de formular o problema mestre e preparar o quadro inicial correspondente a uma solução viável básica associada a uma base B. Desenvolvemos uma função para realizar tal tarefa que põe o problema lido na forma padrão, incluindo as variáveis de folga necessárias e variáveis artificiais (método Simplex de duas fases) caso não obtenhamos uma matriz identidade (base inicial) a partir das colunas referentes às variáveis de folga. Tal função retorna um código que indica para o processo mestre se será necessário usar um método de duas fases ou não.

Depois de formar o quadro inicial, podemos obter, a partir dele, o vetor de multiplicadores simplex que deve ser enviado para os processos escravos. Isso ocorre no passo 3. Nesse passo, além dos multiplicadores simplex (π), devem ser enviados também os dados dos subproblemas (A_i, B_i e c_i). O envio dos dados é feito usando a primitiva básica para envio de dados fornecida pelo PVM, a *pvm_send*. Essa rotina envia para um processo PVM *receptor* uma mensagem que está armazenada em um *buffer* do processo PVM *emissor* e tem como parâmetro o identificador do processo (*TID*) que deverá receber a mensagem. A rotina *pvm_send* é assíncrona (ou sem bloqueio), ou seja, um processo que a chama continua o processamento mesmo antes da mensagem chegar no processo receptor. Nesse caso, a responsabilidade de garantir que as mensagens estão sendo enviadas corretamente é do programador. Um possível erro no envio da mensagem pode ser detectado através do valor retornado pela rotina *pvm_send*, que é negativo caso a mensagem não tenha sido enviada corretamente.

Os processos escravos recebem os dados enviados pelo processo mestre (passo 4) através de outra primitiva para troca de mensagens fornecida pelo PVM, a *pvm_recv*. Essa rotina é síncrona (ou com bloqueio), pois bloqueia o processo que a chamou até que uma mensagem destinada àquele processo seja recebida e armazenada em um *buffer*. Se a mensagem já chegou quando a rotina *pvm_recv* é chamada, o processo continua o processamento imediatamente. O sucesso do recebimento é verificado tal como na rotina *pvm_send*. Se a mensagem não tiver sido recebida corretamente o valor retornado pela rotina será negativo.

Nesse ponto, é iniciada a iteração principal do método. Nos passos 5 e 6 cada processo escravo calcula os coeficientes da função objetivo de seu subproblema e encontra sua solução, que será usada no cálculo do menor custo reduzido. A fim de resolver seus subproblemas, os processos escravos usam o Simplex Revisado, que implementamos como um procedimento. A implementação permite que tal procedimento possa ser substituído por outro sem maiores conseqüências, por exemplo, por uma rotina otimizada de algum software comercial disponível.

Lembramos que escolhemos o método Simplex Revisado para encontrar a solução dos subproblemas devido a maior eficiência do mesmo em termos computacionais, pois tal método se mostra bastante eficiente em aspectos como esforço computacional e espaço de memória requerido. Entretanto, também é possível utilizar outros métodos de resolução de problemas de programação linear como o Dual-Simplex ou o Primal-Dual para tal tarefa, caso a eficiência não seja um fator prioritário.

No passo seguinte (passo 7), cada processo escravo envia para o processo mestre uma mensagem contendo o menor custo reduzido correspondente usando a primitiva *pvm_send*. Para receber essas mensagens o processo mestre usa a primitiva *pvm*_recv, e depois calcula o mínimo dentre todos os custos reduzidos contidos nas mensagens (passo 8). Nesse momento o processo mestre verifica se o critério de otimalidade foi alcançado, isto é, se o menor custo reduzido é positivo (passo 9). Se for, a solução deve ser calculada e o processo mestre deve avisar aos processos escravos que o método chegou ao final. Esse aviso pode ser feito de duas formas diferentes. A primeira alternativa seria o processo mestre enviar uma mensagem (*pvm_send*) para os processos

escravos contendo um código que indicasse se o procedimento iria continuar. Quando um processo escravo recebesse (*pvm_recv*) uma mensagem com o código de "pare", terminaria sua execução, ou seja, os processos escravos permaneceriam resolvendo subproblemas (em um *loop* do tipo *while*, por exemplo) até receberem um aviso do processo mestre para parar. Outra solução seria permitir que o processo mestre "matasse" os processos escravos quando ele detectasse que o método deveria parar. Isso poderia ser feito utilizando uma primitiva fornecida pelo PVM para eliminar processos chamada *pvm_kill*. Essa rotina tem como parâmetro o identificador do processo (*TID*) que deverá ser eliminado. A fim de determinar qual a melhor entre as duas abordagens fizemos alguns testes de execução, os quais não mostraram diferença significativa nos tempos. Dessa forma, escolhemos a primeira abordagem.

Caso a solução ótima ainda não tenha sido encontrada, o processo mestre, seguindo o critério usual do Simplex, forma a nova coluna a entrar na base (passo 10), seleciona a variável a deixar a base e efetua as operações de pivoteamento necessárias (passo 11), encontrando o novo vetor de multiplicadores simplex que será enviado para os processos escravos (passo 12) para que eles repitam os passos 4, 5, 6 e 7. Essa iteração prossegue até que o processo mestre identifique uma solução ótima.

Para concluir, é bom lembrar que o método para quando o processo mestre encontra a solução ótima do problema mestre e não do problema original. É necessário também encontrar a solução ótima do problema original, que pode ser calculada a partir da solução do problema mestre e da solução dos subproblemas. Devido a isso, torna-se necessário guardar, a cada iteração, as soluções de cada um dos subproblemas, no processo mestre ou nos processos escravos. Como apenas a solução de um subproblema será aproveitada a cada iteração, e seria necessário identificar quais soluções um processo escravo deveria guardar e quais poderia descartar, a opção mais conveniente e adequada é manter as soluções dos subproblemas, que serão usadas para o cálculo da solução ótima do problema original, no processo mestre.

3.3.3. Considerações Finais sobre o Algoritmo Paralelo

Analisando o algoritmo da seção 3.1 com um pouco mais de profundidade, pode-se elaborar alguns questionamentos e, conseqüentemente, sugerir modificações. A

primeira pergunta seria porque não permitir que cada processo escravo gere sua própria coluna se ele tem todas as informações necessárias para isso. A resposta é simples. Como apenas uma única variável irá entrar na base, apenas uma única coluna deverá ser gerada. Mesmo que os processos escravos gerem suas colunas simultaneamente, poderia ocorrer de um deles atrasar, o que comprometeria o trabalho dos outros processos. Já que é o processo mestre quem decidirá qual coluna incorporar ao problema mestre, é justo e aconselhável que ele próprio fique encarregado de receber dos processos escravos os custos reduzidos e gerar somente a coluna correspondente ao menor dentre os menores custos reduzidos.

Outra possível indagação seria acerca do critério de escolha da coluna a entrar na base. No método Simplex, escolhe-se a variável cujo custo reduzido é o menor. Porém, isso não é obrigatório, é apenas uma forma de tentar acelerar a velocidade de convergência do método. Pode-se, por exemplo, escolher para entrar na base qualquer variável cujo custo reduzido seja estritamente negativo (< 0). Isso nos faz pensar qual seria a viabilidade de modificar o algoritmo acima a fim de que o mestre colocasse na base a coluna correspondente ao primeiro custo reduzido estritamente negativo que o mesmo receber. Assim, eliminaríamos a necessidade de esperar pelos resultados de todos os processos escravos para continuar a execução do método, e o tempo de resolução dos subproblemas seria determinado pelo tempo obtido pelo processo mais rápido, e não pelo mais lento. O ponto baixo dessa abordagem é a possibilidade do método convergir mais lentamente para a solução ótima, devido à escolha arbitrária de pontos extremos no conjunto de soluções viáveis.

É bom lembrar também que o algoritmo apresentado anteriormente admite que cada subproblema é suficientemente grande para ser resolvido de forma exclusiva pelo seu processador, ou seja, cada processador encontra a solução de somente um subproblema. Se os subproblemas são relativamente pequenos, pode ser que se torne computacionalmente inviável (devido ao tempo de comunicação, por exemplo) a atribuição de um único subproblema a cada processador. Dessa forma, talvez seja mais adequado permitir que cada processador seja responsável pela resolução de mais de um subproblema, dependendo do tamanho dos mesmos. Em outras palavras, para obter um desempenho próximo do ótimo, é necessário distribuir a carga de processamento

(subproblemas a serem resolvidos) de forma equilibrada. Note que esse *balanceamento de carga* se torna obrigatório no caso em que o número de processadores disponíveis é inferior ao número de subproblemas.

Sendo assim, se considerarmos que cada processador pode resolver mais de um subproblema, podemos voltar à questão referente ao critério de escolha de um custo reduzido negativo. Agora, a abordagem de escolher qualquer custo reduzido estritamente negativo ao invés do menor de todos se torna bem mais interessante, pois cada processador resolverá seus subproblemas seqüencialmente. Se logo na resolução do primeiro subproblema for obtido um custo reduzido negativo, elimina-se a necessidade de resolver os subproblemas restantes e o custo reduzido obtido é enviado imediatamente para o processo mestre. Isso pode reduzir consideravelmente o tempo de resolução dos subproblemas, se o número de subproblemas a serem resolvidos por cada processador for suficientemente grande.

Note que existe um motivo pelo qual estamos insistindo na discussão sobre a redução do tempo gasto pela resolução dos subproblemas. Segundo Nazareth (1984), a parte mais dispendiosa da resolução de um problema de programação linear pelo método de decomposição Dantzig-Wolfe é a resolução dos subproblemas, e esse fato foi comprovado nos testes de execução que realizamos cujos resultados serão apresentados no capítulo 4.

Para finalizar, vale salientar que as modificações sugeridas acima, apesar de bem intuitivas na teoria, podem não levar a algoritmos mais eficientes na prática, devido à existência de elementos externos tais como tempo de comunicação entre processos e carga da rede. Entretanto, testes de desempenho e comparações entre as abordagens apresentadas acima são bastante válidos e serão levados em consideração em trabalhos futuros.

3.4.Resumo

O objetivo principal do processamento paralelo é executar computações de maneira mais rápida e eficiente, usando vários processadores trabalhando de forma concorrente. Muitas aplicações, principalmente as que trabalham com uma grande quantidade de

dados (como em problemas de otimização de grande porte), podem ser executadas mais eficientemente usando computação paralela.

O método de decomposição Dantzig-Wolfe tem um algoritmo que é naturalmente paralelo que sugere o uso do modelo de programação mestre/escravo. Os processos escravos são responsáveis pela resolução dos subproblemas e um processo mestre é responsável por coordenar a resolução do problema mestre.

Aliando o princípio de decomposição à computação paralela pretendemos obter um algoritmo paralelo eficiente, que forneça um custo próximo do ótimo em relação aos algoritmos seqüenciais presentes nos softwares comerciais atuais.

No capítulo a seguir mostraremos os resultados obtidos nos testes de desempenho, bem como a metodologia empregada para determinar se resultados satisfatórios foram obtidos.

Capítulo 4

Resultados dos Testes de Desempenho e Considerações

Este capítulo apresentará os resultados obtidos nos testes de execução realizados com o algoritmo paralelo descrito no capítulo anterior. A seção 4.1 introduz as duas abordagens em que os testes de desempenho foram realizados. A seção 4.2 descreve os recursos computacionais de hardware e de software que foram utilizados na implementação e execução do algoritmo. A seção 4.3 mostra, em termos de tempos médios de execução, os resultados obtidos nos testes das duas abordagens apresentadas na seção 4.1. Na seção 4.4 temos o resumo do capítulo.

4.1.Introdução

Nesse capítulo serão apresentados os resultados obtidos nos testes de desempenho do algoritmo paralelo para o princípio de decomposição Dantzig-Wolfe. Esses testes foram executados de acordo com duas abordagens diferentes. A primeira delas faz uma comparação entre o algoritmo paralelo e um algoritmo seqüencial para o princípio de decomposição desenvolvido nos mesmos moldes do algoritmo paralelo. O objetivo dessa comparação é mostrar a viabilidade da aplicação do paralelismo ao princípio de decomposição, de forma a obtermos eficiência, ou seja, conseguirmos um speedup satisfatório e um custo próximo do ideal. Numa segunda abordagem fazemos uma comparação entre o algoritmo paralelo e um software comercial mundialmente reconhecido na resolução de problemas de otimização chamado CPLEX (2001). A idéia é mostrar que o princípio de decomposição aliado ao paralelismo, quando aplicado a problemas de programação linear de grande porte cuja matriz dos coeficientes possui estrutura bloco-angular, possui melhor desempenho do que um software que usa procedimentos que são altamente otimizados, mas não exploram as características particulares da estrutura do problema.

4.2.Recursos Computacionais Utilizados

4.2.1. Recursos de Hardware

Os programas foram executados em um *cluster* de quatro máquinas presentes no Laboratório de Inteligência Artificial (LIA) do Departamento de Computação da Universidade Federal do Ceará (UFC). Cada máquina do cluster possui a mesma configuração e é equipada com um processador Pentium 4 de 1.5Ghz, 512 MB de memória RAM e disco rígido de 40.0 GB. As máquinas são interligadas por uma rede Ethernet de 10 Mbps de velocidade.

4.2.2. Recursos de Software

Ambos os algoritmos (paralelo e seqüencial) para o princípio de decomposição Dantzig-Wolfe que foram usados nos testes da primeira abordagem foram implementados usando a linguagem de programação C. O paralelismo foi implementado através do mecanismo de troca de mensagens (*message passing*) fornecido pela biblioteca de rotinas da ferramenta PVM (Parallel Virtual Machine). Os motivos pelos quais escolhemos usar C e PVM na implementação dos algoritmos já foram discutidos em capítulo anterior.

Todos os programas foram implementados para a plataforma UNIX. Porém, como os recursos de software utilizados (C e PVM) apresentam a portabilidade como uma de suas características fundamentais, podemos facilmente fazer a migração dos programas para qualquer outra plataforma (como Windows por exemplo).

4.2.2.1.Gerador de Problemas

Para efetuar os testes de execução dos algoritmos, construímos um simulador, que gera problemas de programação linear cuja matriz dos coeficientes possui uma estrutura bloco-angular. Para manter a coerência, tal programa também foi implementado em C e gera um arquivo de texto contendo um problema de programação linear num formato padrão (*LP file format* - Ver Figura 4.1). Isso é importante para facilitar a leitura dos dados do problema pelos algoritmos (paralelo e seqüencial) e pelo CPLEX (cujas rotinas de leitura suportam tal formato). Para alimentar o simulador é necessário o fornecimento do número de subproblemas, do número de restrições de acoplamento e

do número de variáveis e de restrições de cada subproblema. Os coeficientes do problema são gerados aleatoriamente através de uma rotina para gerar números randômicos em C (*srand*).

```
Maximize
x1 + 2x2 + 3x3

subject to

-x1 + x2 + x3 <= 20
x1 - 3x2 + x3 <= 30
x1 <= 40

end
```

Figura 4.1: *LP File Format* (formato do arquivo gerado pelo programa gerador).

O código fonte do simulador pode ser encontrado comentado no Apêndice B.

4.2.2.2.CPLEX

CPLEX é uma ferramenta consagrada mundialmente usada para resolução de problemas de programação linear. CPLEX inclui um *otimizador interativo*, que é um programa executável que pode ler um problema interativamente ou a partir de arquivos em alguns formatos padrão (incluindo o *LP format*). O *otimizador* resolve o problema e mostra a solução interativamente ou em arquivos de texto. O pacote CPLEX fornece também uma *biblioteca de rotinas* em C que permite ao programador inserir rotinas de otimização do CPLEX em programas escritos em C, Visual Basic, Java e FORTRAN. Essa biblioteca encontra-se disponível para as plataformas UNIX e Windows.

Na segunda abordagem dos testes de execução, implementamos um programa em C usando as rotinas fornecidas pelo CPLEX para resolver os problemas gerados pelo simulador. O programa resultante consiste basicamente em chamadas a rotinas pertencentes à biblioteca *cplex.h* do CPLEX para criar, ler e resolver problemas de programação linear. Tal programa foi usado nos testes de desempenho a fim de comparar seu tempo de execução com o tempo obtido pelo algoritmo paralelo para o princípio de decomposição Dantzig-Wolfe.

4.2.2.3.Validação dos Resultados e da Precisão

Com o intuito de validar os resultados obtidos (soluções ótimas encontradas) pelos algoritmos na resolução dos problemas gerados, e também para avaliar a sua precisão, usamos o otimizador interativo do pacote CPLEX. As soluções ótimas foram obtidas nas duas abordagens com uma precisão de aproximadamente 10^{-3}.

4.2.2.4.Critérios Usados na Análise dos Resultados

O critério utilizado para avaliar o desempenho das implementações foi o tempo de execução obtido pelos programas para resolver os problemas de programação linear gerados. Esse tempo foi medido em segundos (s) e não leva em consideração o tempo gasto pelos programas para ler os problemas nem para imprimir os resultados. Consideramos apenas o tempo para tratamento inicial do problema e para resolução do mesmo.

A coleta de tempo foi feita através de uma função pertencente à biblioteca de rotinas *time.h* da linguagem C chamada *gettimeofday*. Essa função tem como parâmetro uma estrutura do tipo registro cujos campos correspondem a valores de tempo (tais como dia, hora, minuto e segundo). Quando a função é chamada, o registro é preenchido com valores referentes ao tempo atual. Nos algoritmos, fazemos uma chamada a *gettimeofday* antes de iniciar o processo de montagem do problema (t_{inicio}) e imediatamente depois da solução ter sido encontrada (t_{fim}). O tempo de execução do algoritmo é obtido subtraindo os valores de tempo calculados nas duas situações ($t_{fim} - t_{inicio}$).

Na seção a seguir, além dos tempos de execução obtidos, apresentaremos também a análise de desempenho dos algoritmos, em termos do speedup e da eficiência obtidos nas execuções.

4.3.Resultados dos Testes de Desempenho

4.3.1. Considerações Iniciais

Os testes foram realizados em problemas cujo número de variáveis e restrições (dimensão) variam entre 1000 e 5000. Por causa da limitação oferecida pelo hardware utilizado, isto é, o número máximo de processadores disponíveis é de 4 processadores,

resolvemos gerar problemas com no máximo 4 subproblemas de mesmo tamanho. Essa escolha foi feita devido a alguns testes preliminares que detectaram que quando o número de subproblemas aumenta, mas a dimensão da matriz dos coeficientes, o número de restrições de acoplamento e o número de processadores se mantêm os mesmos, o speedup médio obtido pelo algoritmo paralelo se mantém constante. Veja que, dessa forma, não teremos mais do que 4 processos escravos, o que garante que um processo não irá competir com outro pelo uso de um processador.

Outro detalhe importante é o número de restrições de acoplamento. Se esse número for muito grande em relação ao número de total de restrições do problema (mais da metade, por exemplo), a aplicação do método de decomposição se torna inviabilizada devido ao tamanho do problema mestre resultante, que pode ter quase o mesmo número de restrições do problema original (dependendo do número de subproblemas) e um número de variáveis bem maior. Além disso, no algoritmo paralelo, a carga de processamento no processo mestre se tornaria muito grande em relação à carga de processamento dos processos escravos, o que poderia inviabilizar também o uso do paralelismo (devido ao tempo de comunicação entre os processos). Sendo assim, nos testes de desempenho usamos um número de restrições de acoplamento variando entre 5% e 10% do número total de restrições do problema.

4.3.2. Abordagem 1

Na primeira abordagem, fazemos uma comparação entre o algoritmo paralelo e o algoritmo seqüencial para o princípio de decomposição Dantzig-Wolfe. O objetivo era mostrar, na prática, o quanto é adequado aplicar o paralelismo ao princípio de decomposição. Os testes foram feitos com o propósito de comparar as versões paralela e seqüencial do algoritmo de decomposição e de determinar qual o speedup obtido. Aproveitamos essa seção de testes também para mostrar como o método de decomposição se comporta bem melhor do que um método genérico de resolução de problemas de programação linear (Simplex) quando aplicado a um problema cuja matriz dos coeficientes possui estrutura bloco-angular. O Simplex usado nessa abordagem é o mesmo Simplex Revisado que os algoritmos paralelo e seqüencial do princípio de decomposição utilizaram para encontrar as soluções dos subproblemas.

Na tabela 4.1, mostramos os tempos médios obtidos nos testes de execução em um problema de dimensão 1000. Os tempos estão em segundos (s) e são mostrados de acordo com o número de restrições de acoplamento e com o número de processadores utilizados.

		Número de Restrições de Acoplamento		
		50	**70**	**100**
Simplex Revisado		178.49 s	179.25 s	171.16 s
Número de Processadores (DW)	**Seqüencial**	73.53 s	86.62 s	112.18 s
	2	46.79 s	54.96 s	72.31 s
	3	38.22 s	46.00 s	61.91 s
	4	28.70 s	33.25 s	46.76 s

Tabela 4.1: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 1000.

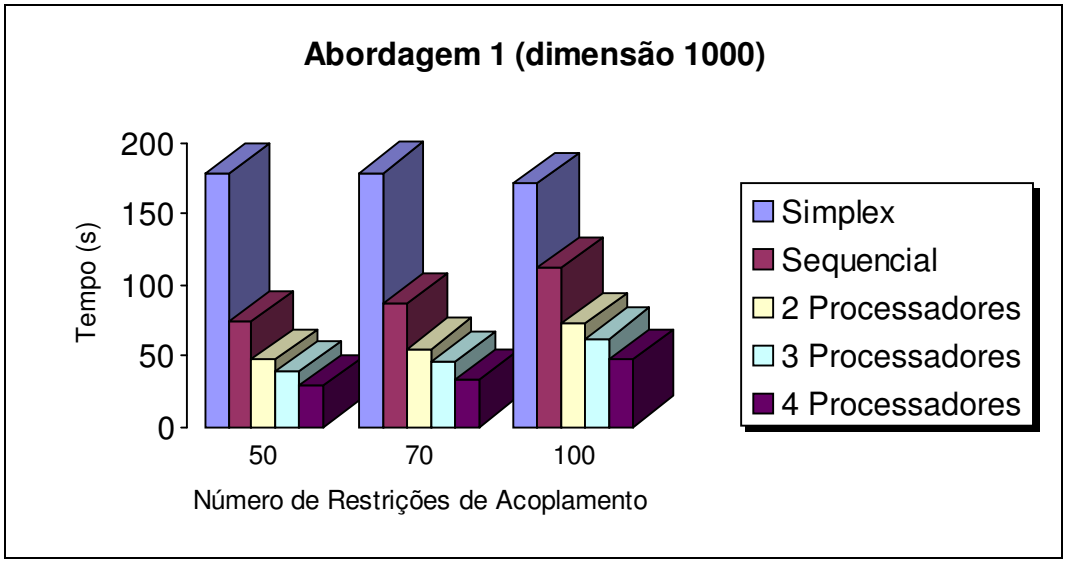


Gráfico 4.1: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 1000.

De acordo com os resultados mostrados na tabela 4.1, o algoritmo (seqüencial) que usa o princípio de decomposição consegue obter um ganho bastante significativo com relação ao Simplex Revisado quando aplicado ao problema com estrutura bloco-angular em questão. Podemos perceber que quanto menor o número de restrições de acoplamento, maior o ganho. No caso do problema com 50 restrições de acoplamento, o tempo de execução obtido pelo Simplex Revisado é aproximadamente 2.5 vezes maior do que o obtido pelo algoritmo de decomposição. Posteriormente poderemos verificar que isso ocorre também com o algoritmo paralelo, isto é, quanto menos restrições de acoplamento tivermos, maior será o ganho obtido com o paralelismo.

Ainda com relação aos resultados mostrados na tabela 4.1, não resta dúvida acerca da viabilidade de aplicação do paralelismo. Considerando que um problema de dimensão 1000 não é um problema muito grande, mesmo no caso em que foram usados somente 2 processadores (2 processos escravos em cada processador) e 100 restrições de acoplamento tivemos um speedup de 1.55, o que corresponde a 77.5 % de eficiência. Como vimos no capítulo anterior, o speedup ótimo, nesse caso, seria 2 e corresponderia a 100% de eficiência. Porém sabemos que os fatores externos (como tempo de comunicação) tornam esses parâmetros impossíveis de alcançar.

Já no caso em que foram usados 4 processadores (um processo escravo por processador) e 100 restrições de acoplamento, conseguimos um speedup próximo a 2.5, o que dá em torno de 60% de eficiência. Nesse caso, apesar de termos conseguido um tempo de execução bem inferior ao obtido com 2 processadores, a eficiência foi menor, o que significa dizer que a redução do tempo de execução não foi proporcional ao aumento do número de processadores.

Se observarmos os tempos obtidos quando usamos 50 restrições de acoplamento o fato se repete. Quando usamos 2 processadores obtivemos um speedup de 1.58 e 79% de eficiência, enquanto que com 4 processadores tivemos um speedup de 2.6 e 65% de eficiência. Essa redução de eficiência pode ter sido causada pelo tamanho dos subproblemas, os quais, por não serem muito grandes, talvez não justifiquem a adição de mais processadores aos 2 iniciais. Em outras palavras, o tempo de comunicação

adicionado ao tempo de execução total foi maior do que a redução de tempo obtida com a inclusão de mais paralelismo.

Para chegar a uma conclusão acerca da afirmação acima vamos investigar os resultados que foram obtidos nos testes com um problema maior. A tabela 4.2 mostra os tempos médios obtidos nos testes efetuados em problemas de dimensão 5000. Como podemos perceber, agora os tempos são bem maiores do que os obtidos com o problema de dimensão 1000. Assim como na tabela 1, os tempos estão sendo mostrados em função do número de restrições de acoplamento e do número de processadores utilizados.

		Número de Restrições de Acoplamento		
		250	**350**	**500**
Número de Processadores (DW)	**Seqüencial**	8556.34 s	10448.15 s	13787.08 s
	2	4987.74 s	5957.81 s	8118.26 s
	3	3808.24 s	4599.05 s	6433.94 s
	4	2475.81 s	2987.06 s	4014.14 s

Tabela 4.2: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 5000.

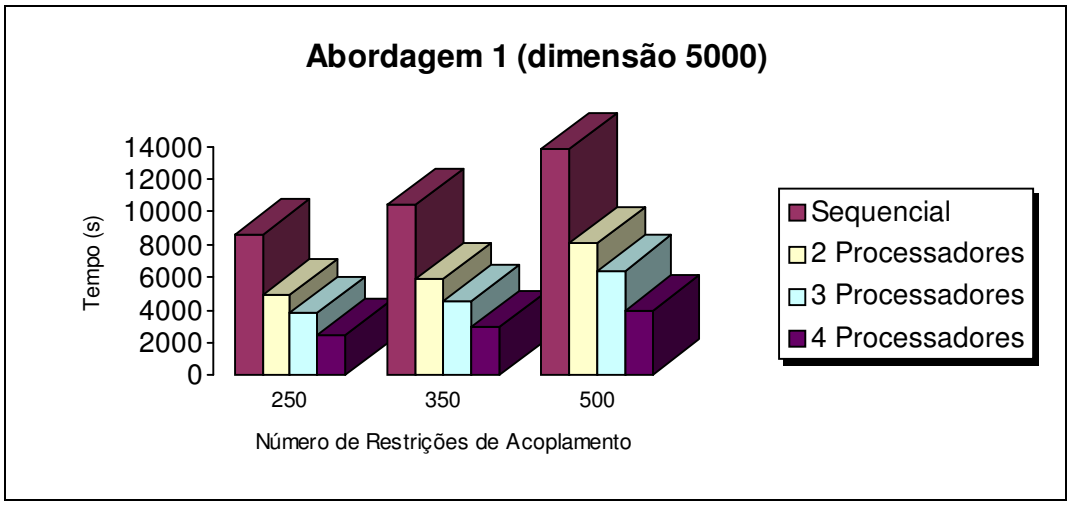


Tabela 4.2: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 5000.

De acordo com a tabela 4.2, no caso que temos 500 restrições de acoplamento, quando usamos 2 processadores, obtivemos um speedup de 1.7 e 85.5% de eficiência, enquanto que quando usamos 4 processadores, alcançamos um speedup de 3.43 e 85.8% de eficiência. Obtivemos um aumento de eficiência quando adicionamos mais 2 processadores, além do tempo de execução ter ficado bem inferior ao do algoritmo seqüencial. Isso ocorreu porque os subproblemas eram grandes o bastante para requerer um grau mais alto de paralelismo, ou seja, um aumento do número de processadores. O mesmo fato se repete nos testes com 250 restrições de acoplamento. O speedup obtido usando 2 processadores é de 1.72, o que corresponde a 86% de eficiência, enquanto que, usando 4 processadores, obtivemos um speedup de 3.46 e 86.5%.

Podemos observar que, tanto na tabela 4.1 quanto na tabela 4.2, conseguimos obter maior eficiência (speedup) quando o problema possui menos restrições de acoplamento. Isso se justifica pelo fato de que quanto menor o número de restrições de acoplamento, maior será a quantidade de dados (restrições) que poderá ser processada em paralelo.

No Apêndice A mostraremos os tempos médios de execução (e os speedups) referentes aos testes de desempenho efetuados sobre problemas cuja matriz dos coeficientes possuem dimensão variando entre 1200 e 4500. As tabelas seguem o

mesmo padrão das apresentadas nessa seção (em função do número de restrições de acoplamentos e do número de processadores utilizados).

4.3.3. Abordagem 2

Na segunda abordagem, fazemos uma comparação entre o algoritmo de decomposição paralelo e um software comercial mundialmente reconhecido na resolução de problemas de otimização chamado CPLEX. O objetivo é avaliar o desempenho da versão paralela do algoritmo de decomposição Dantzig-Wolfe comparando-a com uma rotina de otimização usada por um software comercial. Para realizar a comparação, usamos a função *CPX_lpopt* fornecida pelo pacote CPLEX. Essa função recebe como parâmetro um problema de programação de programação linear e retorna a solução ótima desse problema, caso ela exista.

Nessa seção de testes usamos sempre o número máximo de processadores disponíveis, já que o objetivo era obter um tempo de resolução inferior ao obtido pelo CPLEX. Na tabela 4.3 mostramos os tempos de execução obtidos nos testes com problemas cujo número de restrições de acoplamento correspondem a 5% do número total de restrições do problema.

		Método Utilizado		
		DW Sequencial	**DW Paralelo**	**CPLEX**
Dimensão da matriz dos Coeficientes	**1000**	74.72 s	25.91 s	6.32 s
	1400	235.90 s	95.16 s	18.80 s
	2000	604.18 s	209.83 s	54.73 s
	2500	1008.20 s	345.96 s	84.19 s
	3000	3137.37 s	1063.82 s	274.67 s
	3500	4862.94 s	1606.87 s	394.49 s
	4000	6361.82 s	2003.79 s	498.21 s
	4500	7607.67 s	2249.97 s	589.38 s
	5000	8561.38 s	2481.84 s	697.96 s

Tabela 4.3: Tempos médios de execução obtidos nos testes da abordagem 2 (DW em paralelo x CPLEX) para problema de dimensão entre 1000 e 5000.

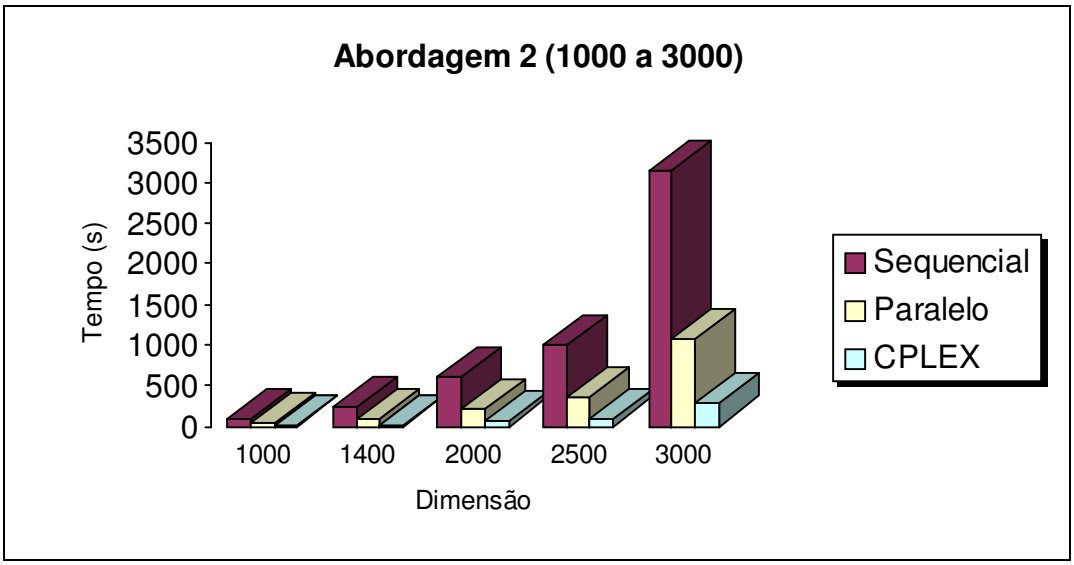


Gráfico 4.3a: Tempos médios de execução obtidos nos testes da abordagem 2 (DW em paralelo x CPLEX) para problema de dimensão entre 1000 e 3000.

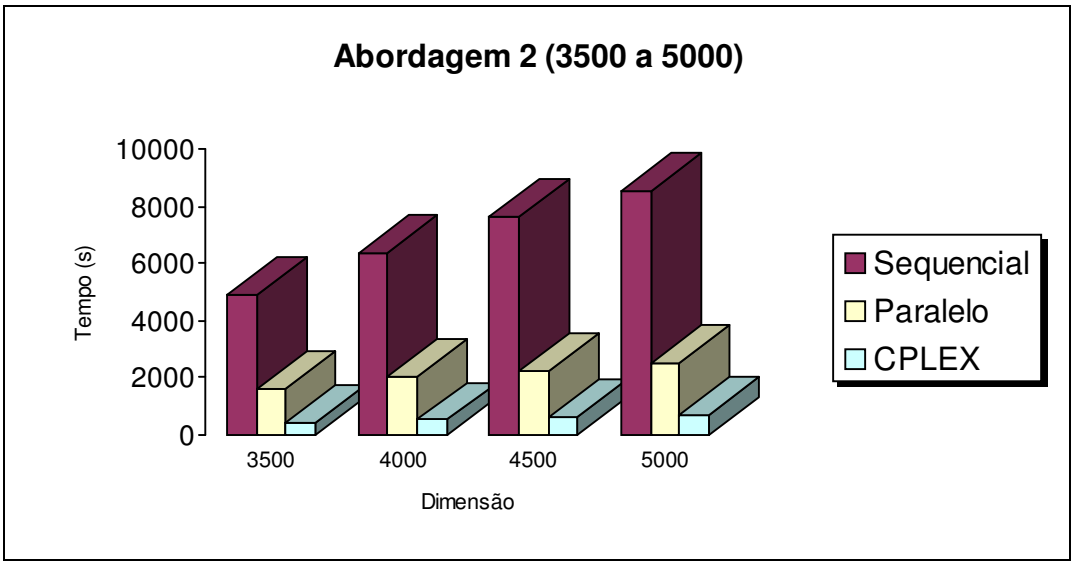


Gráfico 4.3b: Tempos médios de execução obtidos nos testes da abordagem 2 (DW em paralelo x CPLEX) para problema de dimensão entre 3500 e 5000.

Como podemos facilmente observar na tabela 4.3, os tempos obtidos quando usamos a rotina *CPX_lpopt* do CPLEX para resolver os problemas dados é em torno de 4 vezes menor do que os tempos obtidos pela versão paralela do algoritmo de decomposição, o que pode parecer um resultado surpreendente, para não dizer péssimo.

Entretanto, decidimos avaliar qual teria sido o motivo que causou uma diferença de eficiência tão grande entre os dois métodos.

Como já foi mencionado anteriormente, a parte do algoritmo de decomposição que demanda mais tempo é a resolução dos subproblemas. Seguindo esse fato, iniciamos nosso processo de busca medindo o tempo gasto pelos processos escravos para resolverem seus subproblemas. Resolvemos novamente o problema de dimensão 1000 e obtivemos os seguintes resultados:

Tempo médio para o DW Seqüencial resolver o problema	74.27 s
Tempo médio para o DW Paralelo resolver o problema	26.82 s
Tempo médio para o CPLEX resolver o problema	6.27 s
Tempo médio gasto pelo DW Paralelo para resolver os subproblemas	24.05 s
Tempo médio gasto pelo DW Paralelo em outros processamentos	2.77 s

Pelos resultados mostrados acima, vemos claramente que os tempos obtidos pela versão paralela foram altos devido à quantidade de tempo gasta pelos processos escravos para resolver os subproblemas (quase 90% do tempo total de processamento). Isso ocorreu porque os processos escravos encontram a solução dos subproblemas através de um método (Simplex Revisado) implementado por nós que não possui nenhuma técnica especial de otimização, ao contrário da função *CPX_lpopt* do CPLEX. Tal método causa um aumento bastante significativo no tempo de resolução dos subproblemas e conseqüentemente no tempo total de execução do algoritmo paralelo.

Dessa forma, uma maneira de reduzir o tempo de execução do algoritmo é encontrar um meio de reduzir o tempo gasto pelos processos escravos para resolver os subproblemas. Para isso, consideremos os seguintes resultados obtidos em problemas de dimensão 1000.

Tempo médio para o Simplex Revisado resolver o problema	175.68 s
Tempo médio para o CPLEX resolver o problema	6.48 s

Segundo os tempos mostrados acima, a função *CPX_lpopt* do CPLEX é cerca de 30 vezes mais rápida do que a versão revisada do Simplex usada pelos processos escravos. O ideal então seria permitir que os processos escravos resolvessem seus subproblemas através da função *CPX_lpopt*, o que possivelmente nos daria um tempo de execução bem mais satisfatório. Entretanto, devido a um problema financeiro não pudemos efetuar essa modificação. O CPLEX não é um software "livre" e está instalado somente em uma das estações usadas nos testes. Sendo assim, a única alternativa é modificar a versão seqüencial do algoritmo de decomposição e verificar quais mudanças ocorreram e que previsões podem ser feitas com relação ao algoritmo paralelo.

Sendo assim, substituímos (na versão seqüencial do algoritmo de decomposição) a chamada ao Simplex Revisado implementado por nós por uma chamada à função *CPX_lpopt* do CPLEX e refizemos os testes de execução. Os resultados obtidos são apresentados na tabela 4.4.

A tabela 4.4 mostra uma comparação entre os tempos obtidos usando a versão paralela do algoritmo de decomposição, usando o CPLEX e usando a nova versão seqüencial (que utiliza a função *CPX_lpopt* para resolver os subproblemas). Podemos perceber que ocorreu uma redução significativa nos tempos obtidos pela nova versão seqüencial do algoritmo de decomposição em relação à versão anterior (que usava o Simplex Revisado). Em alguns casos, os tempos obtidos pela nova versão seqüencial são melhores até do que os obtidos pela versão paralela. Isso ocorre nos problemas menores, com dimensão entre 1000 e 4000.

		Método Utilizado		
		DW Paralelo **(usando o Simplex Revisado para resolver os subproblemas)**	DW Seqüencial **(usando o *CPX_lpopt* para resolver os subproblemas)**	**CPLEX**
Dimensão da matriz dos Coeficientes	**1000**	25.91 s	23.97 s	6.31 s
	1400	95.15 s	74.88 s	18.80 s
	2000	209.83 s	200.98 s	54.73 s
	2500	345.95 s	329.19 s	84.19 s
	3000	1063.82 s	1104.18 s	274.67 s
	3500	1606.87 s	1503.16 s	394.49 s
	4000	2003.79 s	1995.51s	498.21 s
	4500	2249.97 s	2402.37s	589.38 s
	5000	2481.84 s	2747.59s	697.96 s

Tabela 4.4: Tempos médios de execução obtidos nos testes da abordagem 2 (DW Sequencial usando o CPLEX para resolver subproblemas x CPLEX) para problema de dimensão entre 1000 e 5000.

Apesar da redução obtida com a modificação, o desempenho do CPLEX com relação ao algoritmo de decomposição seqüencial ainda é melhor. Porém, como foi mencionado anteriormente, ficamos impossibilitados de efetuar os testes com uma versão paralela que utilizasse a função *CPX_lpopt* para resolver os subproblemas. Dessa

forma não temos como afirmar com certeza se o algoritmo paralelo possui melhor eficiência do que a implementação usando CPLEX.

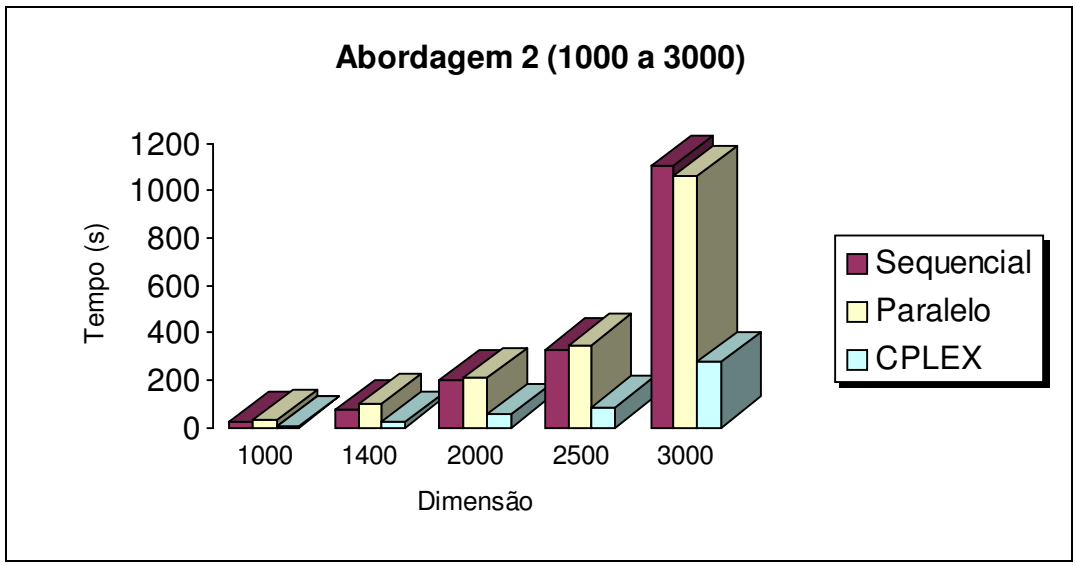


Gráfico 4.4a: Tempos médios de execução obtidos nos testes da abordagem 2 (DW Sequencial usando o CPLEX para resolver subproblemas x CPLEX) para problema de dimensão entre 1000 e 3000.

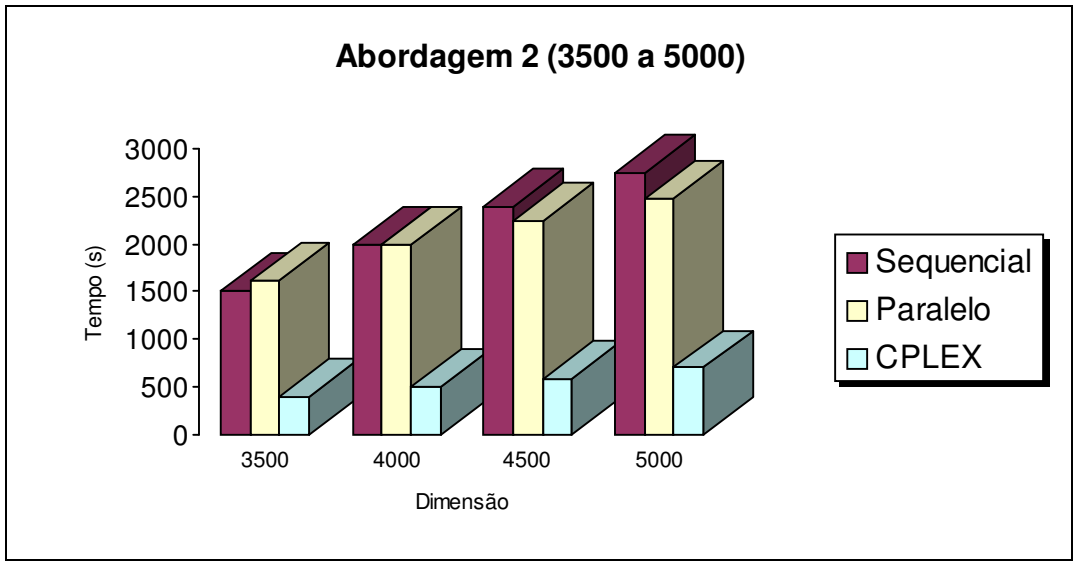


Gráfico 4.4b: Tempos médios de execução obtidos nos testes da abordagem 2 (DW Sequencial usando o CPLEX para resolver subproblemas x CPLEX) para problema de dimensão entre 3500 e 5000.

Entretanto, observando os resultados da tabela 4.4, podemos ficar bastante otimistas com relação à eficiência da nova versão do algoritmo paralelo. Se lembrarmos que, nos testes da abordagem 1, quando usamos 4 processadores, o speedup médio obtido foi superior a 3 e que em casos onde os problemas a resolver são muito grandes a diferença entre o tempo gasto pelo CPLEX e pelo algoritmo seqüencial cresce linearmente em função do tamanho do problema, podemos afirmar que se o problema a ser resolvido for grande o suficiente e se tivermos mais de 4 processadores disponíveis o algoritmo paralelo terá um tempo tão bom quanto ou melhor do que o obtido pelo CPLEX.

Para visualizar melhor, veja, na tabela 4.4, os resultados obtidos com os problemas de dimensão 1000 e 5000. No problema de tamanho 1000 o algoritmo seqüencial usando a função *CPX_lpopt* obteve tempo inferior ao obtido pelo algoritmo paralelo. Essa vantagem foi diminuindo e se invertendo com o aumento do tamanho dos problemas. No problema de tamanho 5000, o tempo obtido pelo algoritmo paralelo já é melhor. Essa diferença tende a crescer juntamente com o tamanho dos problemas, principalmente se o número de processadores disponíveis também aumentar. Como esperamos que haja uma redução bastante significativa no tempo de execução do algoritmo paralelo (como ocorreu na versão seqüencial), se substituirmos o Simplex Revisado pela função *CPX_lpopt* na resolução dos subproblemas, podemos ter previsões bem otimistas com relação à eficiência do algoritmo paralelo resultante.

4.4.Resumo

Os testes de desempenho foram executados de acordo com duas abordagens diferentes. A primeira delas faz uma comparação entre o algoritmo paralelo e um algoritmo seqüencial para o princípio de decomposição desenvolvido nos mesmos moldes do algoritmo paralelo. O objetivo dessa comparação é mostrar a viabilidade da aplicação do paralelismo ao princípio de decomposição. Numa segunda abordagem fazemos uma comparação entre o algoritmo paralelo e um software comercial mundialmente reconhecido na resolução de problemas de otimização chamado CPLEX. A idéia é mostrar que o princípio de decomposição aliado ao paralelismo, quando aplicado a problemas de programação linear de grande porte cuja matriz dos coeficientes possui estrutura bloco-angular, pode possuir melhor desempenho do que

um software que usa procedimentos que são altamente otimizados, mas não exploram as características particulares da estrutura do problema.

Na primeira abordagem conseguimos resultados bastante satisfatórios, enquanto que na segunda abordagem verificamos que deve haver uma preocupação especial com o tempo gasto na resolução dos subproblemas. No capítulo a seguir apresentaremos algumas considerações finais sobre os resultados obtidos nos testes de desempenho, além das conclusões obtidas no nosso trabalho.

Capítulo 5

Conclusões

Este capítulo mostra as principais considerações e conclusões acerca dos resultados obtidos nos testes de desempenho. A seção 5.1 faz um resumo do que foi visto nos capítulos anteriores. A seção 5.2 mostra quais as conclusões obtidas a partir dos resultados dos testes. A seção 5.3 apresenta sugestões para trabalhos futuros. A seção 5.4 mostra um resumo do capítulo.

5.1.Introdução

A presente dissertação tem como principal objetivo investigar a viabilidade da aplicação do processamento paralelo na resolução de problemas de programação linear de grande porte, utilizando um método baseado no Princípio de Decomposição Dantzig-Wolfe. Tal método utiliza técnicas de divisão e conquista para quebrar um problema originalmente grande em vários problemas menores, que podem ser resolvidos simultaneamente por processadores independentes. A principal motivação para o trabalho é o grande esforço computacional desprendido ao se resolver um problema de grande porte utilizando métodos de resolução convencionais.

O Princípio de Decomposição Dantzig-Wolfe atinge maior eficiência quando aplicado a problemas cuja matriz dos coeficientes possui estrutura bloco-angular. Essa estrutura é caracterizada pela presença de um ou mais blocos de restrições independentes, interligados por algumas restrições comuns, chamadas de restrições de acoplamento. O método trabalha formando um problema equivalente ao problema original chamado de "problema mestre". Tal problema possui um número de linhas um pouco maior do que o número de restrições de acoplamento do problema original (uma linha a mais para cada bloco independente), e um número de colunas que pode vir a ser muito grande (da ordem do número de pontos extremos dos conjuntos viáveis correspondentes aos blocos independentes), e é resolvido usando uma técnica chamada

de "geração de colunas", que consiste em resolver o problema sem considerar todas as colunas, gerando apenas as que vão sendo necessárias a cada iteração.

O algoritmo resultante trabalha com uma interação entre um conjunto independente de subproblemas e o problema mestre. Os subproblemas recebem do problema mestre um conjunto de parâmetros (multiplicadores simplex), e os utilizam para encontrar suas soluções. O problema mestre combina as soluções dos subproblemas com as anteriores e calcula novos multiplicadores. Essa interação prossegue até que a solução ótima seja encontrada.

Para atingir a nossa meta, realizamos uma análise comparativa sob duas abordagens distintas. Na primeira, implementamos o algoritmo de decomposição em duas versões, uma seqüencial e uma paralela, e efetuamos uma série de testes de execução com o intuito de determinar em que casos a inclusão do paralelismo fornece o maior aumento de eficiência, e encontrar uma razão custo/benefício próxima da ótima. Em uma segunda abordagem comparamos o algoritmo paralelo usado nos testes da primeira abordagem com o CPLEX, um software bastante utilizado mundialmente na resolução de problemas de otimização de grande porte. O propósito dessa abordagem é mostrar que o processamento paralelo aliado ao princípio de decomposição compõe uma ferramenta tão eficiente quanto um software altamente otimizado como o CPLEX.

Os algoritmos foram implementados e executados em um cluster de 4 estações de trabalho presentes do Laboratório de Inteligência Artificial (LIA) do Departamento de Computação da Universidade Federal do Ceará (DC/UFC). As estações de trabalho são equipadas com um processador Pentium 4 de 1.5Ghz, 512 MB de memória RAM e disco rígido de 40.0 GB e são interligadas por uma rede Ethernet, cuja taxa de transferência de dados ponto-a-ponto é de 10 Mbps.

Para implementar o paralelismo utilizamos a ferramenta para programação paralela PVM (Parallel Virtual Machine), que permite que uma coleção (possivelmente heterogênea) de computadores interligados por uma rede de comunicação trabalhem cooperativamente para resolver um problema computacional de grande porte. O PVM utiliza o modelo de programação Mestre/Escravo, onde um programa (mestre) é responsável por coordenar e controlar a execução de um conjunto de programas

(escravos), que resolvem o problema em questão paralelamente. Esse modelo de programação se adapta perfeitamente ao princípio de decomposição aqui estudado.

5.2.Considerações sobre os Testes de Desempenho

Os testes foram realizados em problemas gerados por um simulador. Esse programa produz arquivos de texto em um formato padrão contendo problemas de programação linear cuja matriz dos coeficientes possui estrutura bloco-angular. Os coeficientes não-nulos da matriz são gerados randomicamente. Realizamos testes em problemas com dimensão variando entre 1000 e 5000. O limite de 5000 restrições e variáveis foi determinado pela quantidade de memória disponível no hardware utilizado. Devido também a uma limitação do hardware (número de processadores), geramos problemas cuja matriz bloco-angular possuía 4 blocos independentes.

A implementação do algoritmo paralelo usando o modelo de programação Mestre/Escravo do PVM resultou em um processo mestre, responsável pela resolução do problema mestre, e quatro processos escravos, responsáveis por encontrar a soluções dos subproblemas. O processo mestre é responsável por criar os processos escravos, coletar os resultados produzidos por eles e determinar se o critério de parada do método foi atingido, ou seja, se a solução ótima do problema foi encontrada. Cada processo escravo resolve seu subproblema usando os multiplicadores simplex enviados pelo mestre, e, depois, envia a solução encontrada para o processo mestre.

Os tempos obtidos nos testes de desempenho foram apresentados em função da dimensão do problema, do número de restrições de acoplamento e do número de processadores utilizados (N). Nos testes da primeira abordagem verificamos que:

- O tempo de execução obtido pelo Simplex Revisado na resolução de problemas com estrutura bloco-angular é em torno de 2.5 vezes maior do que o obtido pelo algoritmo de decomposição seqüencial.
- Em todos os problemas testados, o algoritmo paralelo obteve maior speedup quando o problema possui menos restrições de acoplamento (5% do número total de restrições do problema).

- Em todos os problemas testados, o algoritmo paralelo obteve maior speedup, e conseqüentemente maior redução do tempo de execução, quando utilizamos o número máximo de processadores (quatro).
- Para problemas com dimensão variando de 1000 a 4000, obtivemos maior eficiência (razão custo/benefício) quando foram utilizados somente 2 processadores.
- Para problemas com dimensão 4500 e 5000, obtivemos maior eficiência (razão custo/benefício) quando foram utilizados 4 processadores.
- Obtivemos o melhor speedup (3.46) nos problemas de dimensão 5000, quando o número de restrições de acoplamento correspondia a 5% do numero total de restrições, e quando utilizamos 4 processadores.

Resumindo as considerações acima, em todos os problemas testados o algoritmo paralelo obteve tempo de execução inferior ao obtido pelo algoritmo seqüencial. Entretanto, nos casos dos problemas com dimensão entre 1000 e 4000, a adição de mais processadores não resultou em uma redução de tempo satisfatória. Já com problemas de dimensão 4500 e 5000, obtivemos uma razão custo/benefício muito boa (cerca de 85% de eficiência). Isso acontece porque a quantidade de tempo adicionada pela comunicação não é proporcional à redução de tempo obtida pelo paralelismo.

Na segunda abordagem, comparamos o algoritmo paralelo e o CPLEX, resolvendo os problemas da abordagem anterior que possuíam um número de restrições de acoplamento correspondente a 5% do total de restrições. Nos testes da segunda abordagem verificamos que:

- Quando o algoritmo paralelo utilizou o Simplex Revisado para resolver os subproblemas, o CPLEX conseguiu ser, em média, 4 vezes mais rápido do que o algoritmo paralelo, e até 12 vezes mais eficiente do que o algoritmo seqüencial. Esse fato ocorreu devido ao grande tempo gasto na resolução dos subproblemas, em torno de 90% do tempo de resolução total do algoritmo paralelo.

- O CPLEX foi cerca de 30 vezes mais rápido do que o Simplex Revisado na resolução de um problema de programação linear de dimensão 1000.
- O algoritmo seqüencial que usou o CPLEX para resolver os subproblemas foi, em média, tão eficiente quanto o algoritmo paralelo que usou o Simplex Revisado para o mesmo propósito.
- O desempenho do CPLEX em relação ao algoritmo seqüencial que usa o próprio CPLEX para resolver os subproblemas ainda é melhor. Os tempos obtidos pelo CPLEX foram em torno de 4 vezes mais eficiente.

Apesar da redução de tempo obtida pelo algoritmo seqüencial com a substituição do Simplex Revisado pelo CPLEX para resolver os subproblemas não ter fornecido um tempo de execução inferior, podemos ficar bastante otimistas com relação à eficiência de uma nova versão do algoritmo paralelo obtida através da mesma modificação. Entretanto, ficamos impossibilitados de efetuar os testes com tal versão paralela, pois o CPLEX não é um software "livre" e está instalado somente em uma das estações usadas nos testes de desempenho.

Nos testes da abordagem 1, quando usamos 4 processadores, o speedup médio obtido foi superior a 3, e, nos casos onde os problemas a resolver são muito grandes, a diferença entre o tempo gasto pelo CPLEX e pelo algoritmo seqüencial cresce linearmente em função do tamanho do problema. Dessa forma, podemos afirmar que se o problema a ser resolvido for grande o suficiente e se tivermos mais de 4 processadores disponíveis o algoritmo paralelo terá um tempo tão bom quanto ou melhor do que o obtido pelo CPLEX.

5.3.Trabalhos Futuros

Nosso próximo objetivo é implementar o algoritmo paralelo com as mesmas modificações efetuadas no algoritmo seqüencial, e analisar o comportamento do algoritmo resultante com relação ao CPLEX. A idéia é tentar verificar se o algoritmo paralelo consegue ser mais rápido do que o CPLEX e determinar em que casos obtemos maior eficiência.

Sugerimos a seguir outras propostas para trabalhos futuros:

- Implementar a versão paralela do algoritmo de decomposição Dantzig-Wolfe utilizando outras combinações ferramenta/linguagem (como, por exemplo, MPI e FORTRAN) e efetuar uma análise comparativa.
- Implementar análise de sensibilidade no algoritmo de decomposição Dantzig-Wolfe.
- Implementar a Decomposição de Benders, nas versões paralela e seqüencial, e compará-las, verificando a viabilidade da utilização do paralelismo.
- Aplicar o algoritmo de decomposição paralelo na resolução de um problema real.

5.4.Resumo

Na primeira abordagem de testes, em todos os problemas testados o algoritmo paralelo obteve tempo de execução inferior ao obtido pelo algoritmo seqüencial. Entretanto, nos casos dos problemas com dimensão entre 1000 e 4000, a adição de mais processadores não resultou em uma redução de tempo satisfatória. Já com problemas de dimensão 4500 e 5000, obtivemos uma razão custo/benefício muito boa (cerca de 85% de eficiência).

Nos testes da segunda abordagem, apesar do algoritmo seqüencial não ter fornecido um tempo de execução inferior ao do CPLEX, podemos ficar bastante otimistas com relação à eficiência do algoritmo paralelo correspondente. Entretanto, ficamos impossibilitados de efetuar os testes com a versão paralela, pois o CPLEX não está disponível em todas as estações usadas nos testes de desempenho.

Nosso próximo objetivo é implementar um algoritmo paralelo equivalente ao algoritmo seqüencial da segunda abordagem, realizar testes de desempenho, e analisar o comportamento do algoritmo em relação ao CPLEX.

Referências Bibliográficas

AHN, Byong-Hun; RHEE, Seung-Kyu. A cooperative variant of Dantzig-Wolfe decomposition method. **Journal of the Operations Research**, **Society of Japan**, v. 32, n. 4, p.462-474, Dec. 1989.

AMDAHL, G. M. Limits of expectation. **The international Journal of Supercomputer Applications**, v. 2, n. 1, p. 88-97, 1988.

AMORIM, Cláudio L.; BARBOSA, Valmir C.; FERNANDES, Edil S. T. Uma introdução à computação paralela e distribuída. **Livro da VI Escola de Computação**, Campinas, 1988.

BAI, Z.; DEMMEL, J.; GU, M. Inverse free parallel spectral divide and conquer algorithms for nonsymmetric eigenproblems. Department of Mathematics, University of Kentucky, 1994.

BAL, Henri E. A comparative study of five parallel programming languages. Amsterdam: Dept. of Mathematics and Computer Science, Vrije Universiteit, 1991.

BLELLOCH, Guy E. Programming parallel algorithms. 1996. Disponível em: <http://web.scandal.cs.cmu.edu/www/cacm.html>. Acesso em: 15 Jan. 2003.

CHEN, Zhi-Long; POWELL, Warren B. A column generation based decomposition algorithm for a parallel machine just-in-time scheduling problem. **European Journal of Operational Research**, v. 116, p. 220–232, 1999.

ILOG CPLEX 7.1 – User's Guide. Mar. 2001.

CULLER, Davis E. et al. Parallel programming in Split-C. Berkeley: University of California, 1993.

DANTZIG, George B.; WOLFE, P. The decomposition algorithm for linear programming. **Econometrica**, v. 29, n. 4, Oct. 1961.

DANTZIG, George B. Linear programming and extensions. New Jersey: Princeton University,1963.

DEMMEL, James W. et al. Parallel numerical linear algebra. Berkeley: Cambridge University Press, 1993.

DONGARRA, J. et al. PVM: Parallel Virtual Machine. MIT Press, 1994.

FLYNN, M. J. Very high-speed computing systems. **Proc. IEEE**, v. 54, p. 1901–1909, 1966.

FUNG, Y. F. et al. A parallel solution to linear systems. **Microprocessors and Microsystems**, v. 26, p. 39-44, 2002.

GASS, Saul I. Linear programming: methods and applications. New York: McGraw-Hill Book, 1985.

JAJA, Joseph F. An introduction to parallel algorithms. Massachusetts: Addison-Wesley, 1992.

JAJA, Joseph F. Fundamentals of parallel algorithms. In Parallel and Distributed Computing Handbook. New York: McGraw-Hill, 1996.

JURCZYK, Michael; SCHWEDERSKI, Thomas. SIMD-Processing: concepts and systems. In Parallel and Distributed Computing Handbook. New York: McGraw-Hill, 1996.

KIM, K.; NAZARETH, J. L. The decomposition principle and algorithms for linear algebra. Department of Pure and Applied Mathematics, Washington State University, Pullman, Washington, 1991.

KRONSJÖ, Lydia I. PRAM Models. In Parallel and Distributed Computing Handbook. New York: McGraw-Hill, 1996.

LASDON, Leon S. Optimization theory for large systems. New York: Macmillan Publishing, 1970.

LUENBERGER, David G. Linear and nonlinear programming. Massachusetts: Addison-Wesley, 1984.

MACULAN, Nelson; PEREIRA, Mário V. F. Programação linear. São Paulo: Atlas, 1980.

MINOUX, M. Mathematical programming: theory and algorithms. New York: Wiley-Interscience, 1986.

NAZARETH, Larry. Numerical behavior of LP algrithms based upon the decomposition principle. **Linear Algebra and Its Applications**, v. 57, p. 181-189, Elsevier Science, 1984.

OLIVEIRA, J. G. Resolução de sistemas lineares de grande porte utilizando processamento paralelo. Campina Grande, 1999.

PEQUENO, M. C.; Donato, A. A.; Carvalho, K. Using the parallel processing in solution of large scale linear equations systems for WAVES project. Proceedings of the III international WAVES Workshop in Freising-Weihenstephan, ISBN 3-00-006630-6, Monique-Alemanha, Mar. 2000.

PEQUENO, M. C.; Donato, A. A.; Fonteles, C. Algorithms in a parallel approach for applications of optimization in problems for large systems. German - Brazilian Workshop on Neotropical Ecosystemys - Achievements and Prospects of Cooperative Research, Hamburgo-Alemanha, Sep. 2000.

PERROTT, R. H. Parallel languages. In Parallel and Distributed Computing Handbook. New York: McGraw-Hill, 1996.

PINTO, Hermínio J.; TEIXEIRA, Mario J. Aplicação de processamento paralelo em otimização de grande porte. **Pesquisa Operacional**, v. 8, n. 2, Dez. 1988.

PUCCINI, Abelardo de L. Introdução à programação linear. Rio de Janeiro: Ao Livro Técnico S. A., 1972.

PVM Home Page. Disponível em: <http://www.epm.ornl.gov/pvm>. Acesso em: 15 Jan. 2003.

SCHIEFER, G. Solution of large regional-planning problems with the method of Dantzig-Wolfe. **Zeitschrift für Operations Research**, v. 20, p. B1-B16, 1976.

WENTGES, Paul. Weighted Dantzig-Wolfe decomposition for linear mixed-integer programming. **Int. Trans. Opl. Res.**, v. 4, n. 2, p. 151-162, 1997.

WILLIAMS, H. P.; REDWOOD, A. C. A structured programming model in the food industry. **Operational Research Quarterly**, v. 25, p. 517-527, 1974.

ZOMAYA, Albert Y. Parallel and distributed computing: the scene, the props, the players. In Parallel and Distributed Computing Handbook. New York: McGraw-Hill, 1996.

Apêndice A

Programas Utilizados nos Testes de Desempenho

A seguir mostraremos os tempos médios de execução referentes aos testes de desempenho efetuados sobre problemas cuja matriz dos coeficientes possuem dimensão variando entre 1200 e 4500. As tabelas seguem o mesmo padrão das apresentadas no capítulo 4 (em função do número de restrições de acoplamentos e do número de processadores utilizados).

		Número de Restrições de Acoplamento		
		50	**70**	**100**
Simplex Revisado		178.49 s	179.25 s	171.16 s
Número de Processadores (DW)	**Seqüencial**	73.53 s	86.62 s	112.18 s
	2	46.79 s	54.96 s	72.31 s
	3	38.22 s	46.00 s	61.91 s
	4	28.70 s	33.25 s	46.76 s

Tabela A.1a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 1000.

		Número de Restrições de Acoplamento		
		50	**70**	**100**
Número de Processadores	**2**	1.57	1.57	1.55
	3	1.92	1.88	1.81
	4	2.56	2.6	2.4

Tabela A.1b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 1000.

		Número de Restrições de Acoplamento		
		60	**84**	**120**
Número de Processadores (DW)	**Seqüencial**	153.35 s	170.52 s	222.46 s
	2	97.12 s	108.28 s	146.05 s
	3	80.53 s	89.38 s	118.36 s
	4	59.08 s	66.41 s	87.40 s

Tabela A.2a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 1200.

		Número de Restrições de Acoplamento		
		60	**84**	**120**
Número de Processadores (DW)	**2**	1.58	1.57	1.52
	3	1.9	1.89	1.87
	4	2.59	2.56	2.54

Tabela A.2b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 1200.

		Número de Restrições de Acoplamento		
		70	**98**	**140**
Número de Processadores (DW)	**Seqüencial**	234.60 s	268.45 s	352.59 s
	2	146.25 s	168.02 s	225.64 s
	3	124.71 s	143.71 s	192.91 s
	4	91.56 s	104.28 s	140.56 s

Tabela A.3a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 1400.

		Número de Restrições de Acoplamento		
		70	**98**	**140**
Número de Processadores (DW)	**2**	1.6	1.59	1.56
	3	1.88	1.86	1.82
	4	2.56	2.57	2.5

Tabela A.3b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 1400.

		Número de Restrições de Acoplamento		
		100	**140**	**200**
Número de Processadores (DW)	**Seqüencial**	600.92 s	721.72 s	923.88
	2	364.62 s	441.76 s	578.93
	3	305.71 s	367.91 s	471.96
	4	212.45 s	256.37 s	332.01 s

Tabela A.4a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 2000.

		Número de Restrições de Acoplamento		
		100	**140**	**200**
Número de Processadores (DW)	**2**	1.64	1.63	1.59
	3	1.96	1.96	1.95
	4	2.83	2.81	2.78

Tabela A.4b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 2000.

		Número de Restrições de Acoplamento		
		125	**175**	**250**
Número de Processadores (DW)	**Seqüencial**	1008.37 s	1197.81 s	1605.44 s
	2	603.59 s	734.35 s	1002.37 s
	3	496.63 s	613.52 s	849.14 s
	4	346.39 s	429.03 s	575.26 s

Tabela A.5a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 2500.

		Número de Restrições de Acoplamento		
		125	**175**	**250**
Número de Processadores (DW)	**2**	1.67	1.63	1.6
	3	2.03	1.95	1.89
	4	2.91	2.8	2.79

Tabela A.5b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 2500.

		Número de Restrições de Acoplamento		
		150	**210**	**300**
Número de Processadores (DW)	**Seqüencial**	3151.46 s	3742.67 s	4566.06 s
	2	1877.82 s	2276.29 s	2816.77 s
	3	1612.81 s	1849.10 s	2426.02 s
	4	1068.13 s	1292.88 s	1585.41 s

Tabela A.6a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 3000.

		Número de Restrições de Acoplamento		
		150	**210**	**300**
Número de Processadores (DW)	**2**	1.68	1.64	1.62
	3	2.05	2.02	1.88
	4	2.95	2.89	2.88

Tabela A.6b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 3000.

		Número de Restrições de Acoplamento		
		175	**245**	**350**
Número de Processadores (DW)	**Seqüencial**	4855.58 s	5977.71 s	7810.81 s
	2	2915.21 s	3536.68 s	4737.06 s
	3	2421.92 s	2952.32 s	3306.30 s
	4	1607.61 s	2001.27 s	2720.41 s

Tabela A.7a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 3500.

		Número de Restrições de Acoplamento		
		175	**245**	**350**
Número de Processadores (DW)	**2**	1.66	1.69	1.64
	3	2.0	2.02	1.96
	4	3.02	2.98	2.87

Tabela A.7b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 3500.

		Número de Restrições de Acoplamento		
		200	**300**	**400**
Número de Processadores (DW)	**Seqüencial**	6377.59 s	7652.49 s	8754.65 s
	2	3753.38 s	4552.79 s	5282.94 s
	3	3068.12 s	3720.19 s	4375.69 s
	4	2014.07 s	2480.06 s	2887.01 s

Tabela A.8a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 4000.

		Número de Restrições de Acoplamento		
		200	**300**	**400**
Número de Processadores (DW)	**2**	1.7	1.68	1.65
	3	2.08	2.05	2.0
	4	3.16	3.08	3.03

Tabela A.8b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 4000.

		Número de Restrições de Acoplamento		
		225	**315**	**450**
Número de Processadores (DW)	**Seqüencial**	7610.02 s	9257.59 s	11849.71 s
	2	4452.60 s	5484.88 s	6965.08 s
	3	3532.97 s	4219.50 s	5333.79 s
	4	2258.16 s	2788.25 s	3476.12 s

Tabela A.9a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 4500.

		Número de Restrições de Acoplamento		
		225	**315**	**450**
Número de Processadores (DW)	**2**	1.71	1.68	1.7
	3	2.15	2.19	2.22
	4	3.37	3.32	3.4

Tabela A.9b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 4500.

		Número de Restrições de Acoplamento		
		250	**350**	**500**
Número de Processadores (DW)	**Seqüencial**	8556.34 s	10448.15 s	13787.08 s
	2	4987.74 s	5957.81 s	8118.26 s
	3	3808.24 s	4599.05 s	6433.94 s
	4	2475.81 s	2987.06 s	4014.14 s

Tabela A.10a: Tempos médios de execução obtidos nos testes da abordagem 1 (algoritmo paralelo x algoritmo sequencial) para problema de dimensão 5000.

		Número de Restrições de Acoplamento		
		250	**350**	**500**
Número de Processadores (DW)	**Seqüencial**	8556.34 s	10448.15 s	13787.08 s
	2	1.71	1.7	1.69
	3	2.24	2.27	2.14
	4	3.46	3.49	3.43

Tabela A.10b: Speedups médios obtidos nos testes da abordagem 1 para problema de dimensão 5000.

		Método Utilizado		
		DW Sequencial	**DW Paralelo**	**CPLEX**
Dimensão da matriz dos Coeficientes	**1000**	74.72 s	25.91 s	6.32 s
	1400	235.90 s	95.16 s	18.80 s
	2000	604.18 s	209.83 s	54.73 s
	2500	1008.20 s	345.96 s	84.19 s
	3000	3137.37 s	1063.82 s	274.67 s
	3500	4862.94 s	1606.87 s	394.49 s
	4000	6361.82 s	2003.79 s	498.21 s
	4500	7607.67 s	2249.97 s	589.38 s
	5000	8561.38 s	2481.84 s	697.96 s

Tabela A.11: Tempos médios de execução obtidos nos testes da abordagem 2 (DW em paralelo x CPLEX) para problema de dimensão entre 1000 e 5000.

		Método Utilizado		
		DW Paralelo **(usando o Simplex Revisado para resolver os subproblemas)**	DW Seqüencial **(usando o *CPX_lpopt* para resolver os subproblemas)**	**CPLEX**
Dimensão da matriz dos Coeficientes	**1000**	25.91 s	23.97 s	6.31 s
	1400	95.15 s	74.88 s	18.80 s
	2000	209.83 s	200.98 s	54.73 s
	2500	345.95 s	329.19 s	84.19 s
	3000	1063.82 s	1104.18 s	274.67 s
	3500	1606.87 s	1503.16 s	394.49 s
	4000	2003.79 s	1995.51s	498.21 s
	4500	2249.97 s	2402.37s	589.38 s
	5000	2481.84 s	2747.59s	697.96 s

Tabela A.12: Tempos médios de execução obtidos nos testes da abordagem 2 (DW Sequencial usando o CPLEX para resolver subproblemas x CPLEX) para problema de dimensão entre 1000 e 5000.