UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE SISTEMAS DE ENERGIA ELÉTRICA

SOLUÇÃO DE EQUAÇÕES DE REDES DE ENERGIA ELÉTRICA EM COMPUTADORES MULTIPROCESSADORES

Antonio Padilha Feltrin

orientador: André Luiz Morelato França

Tese apresentada à Faculdade de Engenharia Elétrica - UNICAMP,
como parte dos requisitos para obtenção do título de Doutor
em Engenharia Elétrica

Campinas - maio - 1991

À minha família.

AGRADECIMENTOS

- a André Luiz Morelato França pela orientação.
- a Artur Pestana Ribeiro Costa, José Henrique Zilberberg e Sálvio Calichio Sobrinho do DIT-CPqD-TELEBRÁS que viabilizaram o uso do computador PPP.
- a Marcelo Rodrigues que desenvolveu as bibliotecas para explicitar o paralelismo no PPP.
- ao Departamento de Sistemas do CEPEL-ELETROBRÁS, que forneceu a biblioteca de software básico para uso paralelo do PPP.
- ao DSEE-FEE-UNICAMP e ao DEE-FEIS-UNESP pelo apoio.

SUMÁRIO

Neste trabalho propõe-se uma metodologia para a decomposição da etapa de solução direta de um sistema de equações linear esparso $Ax = b$, obtendo-se um conjunto de tarefas independentes que podem ser processadas em paralelo usando máquinas multiprocessadores. Atualmente computadores multiprocessadores tendem a apresentar baixo custo e desempenho comparável aos supercomputadores. Esta nova tecnologia poderá ser incorporada proximamente aos centros de controle em tempo real de sistemas de energia elétrica.

A metodologia baseia-se na utilização da matriz de fatores inversos (W) particionada. O esquema de particionamento proposto consiste em dividir a matriz W de acordo com a profundidade dos nós na árvore dos caminhos de fatoração da matriz A. Neste esquema todas as informações necessárias para gerar as partições são obtidas diretamente da árvore de fatoração, sendo portanto de fácil implementação. Os elementos de todas as partições, exceto da última, podem ser obtidos diretamente da matriz L, sem necessidade de cálculos adicionais. O esquema de particionamento garante que novos elementos diferentes de zero ("fill-ins" adicionais) serão gerados apenas na última partição porque propositalmente esta última partição *quebra* relações de precedência.

As substituições "forward" e "backward" são realizadas por linhas e por partições, e a estratégia proposta de divisão de tarefas atribui a um processador todo o conjunto de operações (multiplicação/adição) de uma linha em uma mesma partição.

Na última partição pode-se agrupar as etapas "forward", diagonal e "backward", tornando a solução desta partição uma única etapa de multiplicação de uma matriz W_{up} (cheia) pelo vetor b. Isto é vantajoso em uma série de casos.

A metodologia proposta transfere a velocidade da solução para a capacidade de realizar operações de ponto flutuante de cada processador, ou seja, à medida que processadores mais poderosos forem sendo desenvolvidos mais rápida será a solução em paralelo do problema quase na mesma proporção.

Uma solução direta que use uma metodologia como a proposta neste trabalho, consegue uma redução nos tempos de comunicação entre processadores, como demonstra os resultados obtidos em uma implementação num computador multiprocessador com arquitetura de memória híbrida (local e global).

SUMMARY

This thesis describes a methodology for decomposing the repeat solution process of the equation $Ax = b$ into independent tasks to be done in parallel, based on the matrix inverse factors (W-matrix) with partitions. It is a matter of fact that low-cost multiprocessor computers are now available featuring supercomputer-like performances.

The partitioning scheme proposed in this thesis consists of breaking up the W-matrix according the depths of the factorization path tree. In this scheme, all the information needed to generate the partitions can be obtained straightforward from the network factorization path tree. The partitioning algorithm is simple and ease to implement. The elements of W_i matrices, except the last partition, can be obtained directly from L-matrix elements, not requiring extra work. The proposed scheme guarantees that additional fills will be only created in the last partition.

The forward and backward solutions are performed by rows, and the strategy proposed is to schedule on each processor the operations corresponding to a row of each partition. It should be kept in mind that the multiprocessor environments are equipped with powerful unit processors and then it seems a sound strategy to perform the mult-add elementary operations inside the hardware in order to exploit its computing efficiency. This strategy seeks to match the parallel algorithm to the parallel architecture. The precedent relations - that give rise to delays - are replaced by mult-add operations performed inside the processor node without external communication.

In the last partition, the forward, diagonal and backward solutions may be gathered, and so all the operations can be expressed as the product of matrix W_{lp} by the updated components of vector b.

The performance results show that the potential speedup of the solution time is essentially bounded by the floating point operation capability of each processor, denoting that the methodology is a suitable way to exploit the growing power of the computing technology.

Conteúdo

Capítulo 1

INTRODUÇÃO

A utilização de computadores multiprocessadores está-se tornando cada vez mais atrativa, devido ao seu baixo custo e características de desempenho cada vez mais próximas dos supercomputadores. Estão disponíveis, por exemplo, novas e poderosas máquinas multiprocessadoras com performance de 480 até 7600 Mflops, com 8-128 processadores interconectados em uma arquitetura de memória distribuída [22], ou com performance variando de 2 a 130 Gflops, com até 65000 processadores em uma arquitetura de memória partilhada [29]. Evidentemente, este desempenho de pico não é fácil de ser atingido, porque depende de como a decomposição do problema é realizada e implementada para explorar as principais características da arquitetura.

Por outro lado, os centros de controle em tempo real de sistemas de energia elétrica estão passando por significativas transformações, podendo-se mesmo dizer que está em gestação uma nova geração de centros de controle que deverão, cada vez mais, incorporar técnicas de inteligência artificial, programação matemática e processamento paralelo. Em particular, esses novos centros de controle deverão ser muito mais eficientes que os atuais quando o sistema estiver em estado de emergência ou restaurativo, e para isso é necessário grande velocidade de processamento.

Praticamente todos os problemas de operação e controle de sistemas de energia elétrica exigem repetidas soluções das equações elétricas da rede (problema do fluxo de potência) que correspondem a um conjunto de equações algébricas não-lineares. Exatamente esse é o problema tratado neste trabalho, ou seja, como resolver este sistema de equações em um computador multiprocessador.

Hoje em dia, existem métodos (Newton e Desacoplado Rápido) que são muito eficientes na solução do problema em computadores com uniprocessador, principalmente pela sua rapidez de convergência. Cada iteração consiste em resolver em x um sistema na forma $Ax = b$, onde a matriz A é altamente esparsa, através de fatoração LDU e aplicação de solução direta $LDUx = b$. Esta abordagem é a que será seguida neste trabalho, no intuito de aproveitar a facilidade de convergência em detrimento dos métodos

iterativos como Jacobi, Gauss-Seidel ou SOR ("successive over relaxation") — que permitem explorar mais facilmente o paralelismo inerente ao problema mas apresentam taxas de convergência bem menores.

Este trabalho concentra-se em apresentar e testar uma metodologia para resolver em paralelo a etapa de solução direta do problema. Embora a etapa de fatoração de A contenha paralelismo a ser explorado, seguiu-se aqui a idéia de tentar paralelizar a parte do problema que é repetidamente resolvida visando atingir maiores ganhos globais. A metodologia proposta aqui baseia-se na partição da matriz dos fatores inversos (W), buscando uma forma de decomposição e atribuição de tarefas que *quebrem* relações de precedência e reduzam a comunicação entre processadores. Esta metodologia é desvantajosa para resolver o problema em computadores sequenciais, porém apresenta-se promissora para máquinas multiprocessadoras. Em linhas gerais, a metodologia consegue decompor o problema em uma sucessão de grandes passos sequenciais — o menor número possível — de tal modo que dentro de cada passo existam inúmeras tarefas independentes que possam ser realizadas em paralelo com um mínimo de comunicação. A estratatégia geral de alocação de tarefas visa concentrar todos os processadores disponíveis para fazer cada passo o mais rapidamente possível. Antes de iniciar cada passo é necessário sincronizar os processadores.

Esta tese está dividida em 5 capítulos e 5 apêndices. No Capítulo 2 apresentam-se os principais conceitos e definições sobre processamento paralelo que serão utilizados nos próximos capítulos.

No Capítulo 3 descrevem-se os conceitos básicos envolvidos no problema da solução direta, são apresentas e discutidas algumas abordagens que têm sido propostas para resolver as equações das redes elétricas utilizando processamento paralelo, e finalmente propõe-se uma nova metodologia para resolver o problema.

No Capítulo 4 são apresentados os resultados de aplicação da metodologia proposta, analisando-se o desempenho da decomposição e da implementação em uma máquina paralela.

Finalmente no Capítulo 5 são apresentadas as conclusões e as propostas de trabalhos futuros derivados da nova metodologia.

Capítulo 2

PROCESSAMENTO PARALELO

2.1 Introdução

O objetivo primordial que se tem em mente ao utilizar processamento paralelo é obter desempenho, isto é, resolver um problema no menor tempo possível. Processamento paralelo só se torna vantajoso se for possível decompor o problema em um número suficiente de processos que serão computados simultaneamente e de forma coordenada. A obtenção de ganho real em desempenho através de processamento paralelo depende de três fatores: (a) a forma como a decomposição/coordenação é realizada, que deve levar a um conjunto de tarefas independentes que possam ser executadas em paralelo; (b) necessidade de comunicação entre processadores, já que um processador para realizar sua tarefa precisa trocar informações com outros processadores; e (c) necessidade de sincronização das tarefas, pois em geral estas precisam ser coordenadas, ou seja, não podem ser feitas em uma ordem qualquer. Desta forma neste capítulo são revisados alguns conceitos básicos visando entender melhor os problemas que possam surgir no processamento paralelo.

Fontes de ineficiência

Um sistema com multiprocessadores consegue desempenho máximo na solução de determinado problema quando: todos os seus N processadores estão envolvidos na execução de trabalho útil; nenhum processador fica parado e nenhum processador faz a mesma tarefa que outro. Assim, a solução sequencial teria seu tempo de execução dividido por N. Este desempenho máximo é muito raro de se obter, porque surgem problemas que tendem a atrasar a solução [33]. Estes problemas são as chamadas fontes de ineficiência ("overheads") .

O conceito de "overhead" pode ser associado aos três fatores anteriormente mencionados. Deste modo, associado ao fator (a), surge um "overhead" inerente ao algoritmo, que é um atraso devido às operações de controle e atribuição de tarefas; além disto,

nem sempre os melhores algoritmos para processamento sequencial são os melhores para processamento paralelo, pois, às vezes, é necessário criar algum trabalho adicional para obter maior grau de paralelismo. Relacionado ao fator (b), tem-se um "overhead" ligado ao tempo gasto em comunicação de dados entre processadores e, associado ao fator (c), surge um "overhead" devido ao tempo que processadores ficam parados esperando por outros que executam tarefas mais lentas. Portanto, estes acréscimos ao tempo de solução final paralela, que na solução sequencial não aparecem, é o que se define como sendo "overhead" de um processamento paralelo.

Os problemas com os atrasos na solução paralela tendem a crescer à medida que o número de processadores aumenta o que pode levar a uma limitação no tamanho das máquinas paralelas se eficientes esquemas de comunicação não estiverem presentes.

Granularidade

A busca pela solução paralela mais rápida para um determinado problema pode levar à divisão deste no máximo número possível de tarefas independentes. Porém o máximo grau de paralelismo, geralmente, leva também ao máximo de "overhead", visto que o tempo gasto em decomposição e a necessidade de comunicação/sincronização destas tarefas podem introduzir grande atraso na solução. Por outro lado uma decomposição do problema que procure reduzir muito a comunicação entre processadores pode sobrecarregar mais um processador que outro (desbalanceamento de carga). Assim a distribuição de carga tem que levar em conta o acréscimo de tempo que será introduzido pela comunicação/sincronização das tarefas. A relação entre o tempo de computação útil e o tempo de comunicação/sincronização para realizar uma tarefa define a *granularidade da solução*. A decomposição contém um *grão grosso* se a relação é grande ou um *grão fino* se a relação é pequena.

Ganho

Também relacionado com os três fatores acima tem-se o conceito de ganho ("speedup"), que é utilizado para medir o desempenho de algoritmos em máquinas paralelas. O ganho pode ser definido através da relação (t_s/t_p) entre o tempo de execução do algoritmo sequencial (t_s) e o tempo da execução do algoritmo paralelo (t_p). Deste modo, tem-se uma idéia de quantas vezes a solução paralela é mais rápida que a solução sequencial.

2.2 Computadores de Alto Desempenho

Os tipos de computadores de alto desempenho podem ser classificados em [21]:

- Computadores "Pipeline"
- Computadores "Array"
- Computadores Multiprocessadores

Computadores "Pipeline"

O processo de execução de uma instrução em um computador digital envolve basicamente quatro estágios:

1. Busca da instrução na memória principal (E1);
2. Decodificação da instrução e identificação da operação (E2);
3. Busca do operador (E3);
4. Execução da instrução decodificada (E4).

Em computadores não-"pipeline" para iniciar-se o primeiro estágio de uma instrução (I_n) é necessário que o quarto estágio da instrução anterior (I_{n-1}) tenha terminado. Já em computadores "pipeline" as instruções podem ser sobrepostas de maneira a serem executadas em estágios sucessivos. Pode-se assim iniciar a execução do estágio E1 para uma instrução (I_n), quando outra instrução (I_{n-1}) já esteja no estágio E2. Na Figura 2.1(a) são mostrados os quatros estágios arranjados em uma forma linear; na Figura 2.1(b) mostra-se um diagrama tempo *versus* estágios para um computador "pipeline" e na Figura 2.1(c) mostra-se como seria o mesmo diagrama para um computador não-"pipeline".

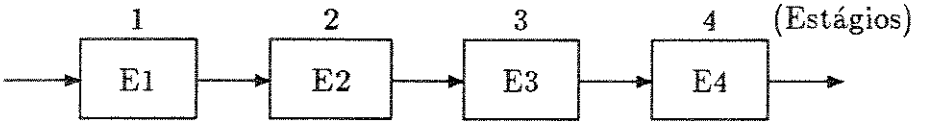


Figura 2.1(a) - Estrutura do processador "pipeline"

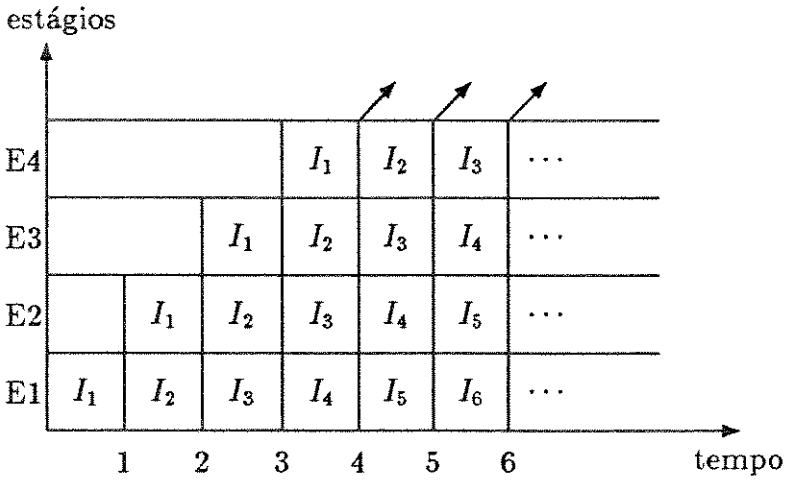


Figura 2.1(b) - Tempo *versus* estágios para um computador "pipeline"

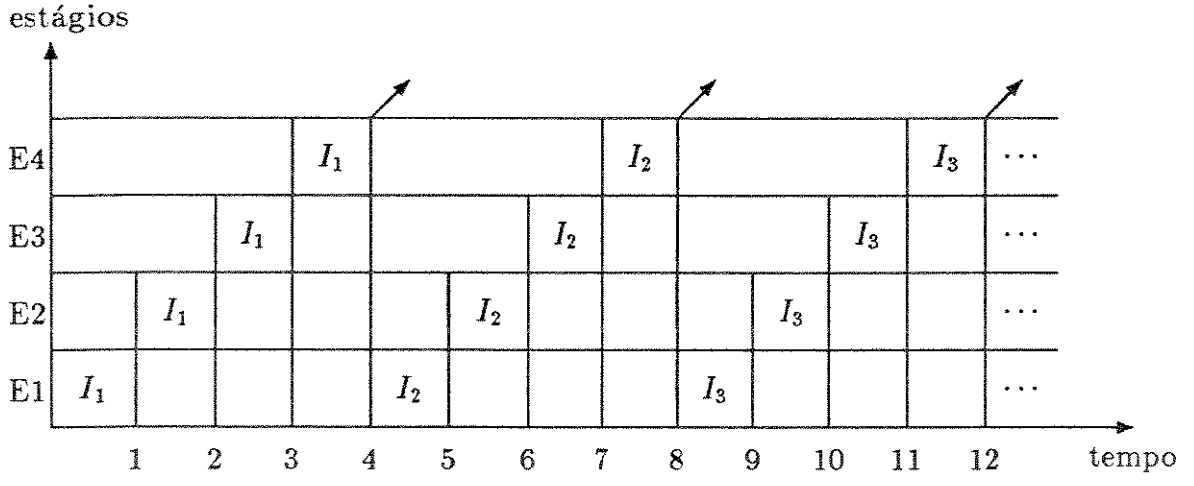


Figura 2.1(c) - Tempo *versus* estágios para um computador não-"pipeline"

Computadores "Array"

Uma estrutura básica dos processadores "array" é mostrada na Figura 2.2, onde p idênticos elementos processadores (EP), compostos de processadores e memórias, são capazes de executar a mesma operação em diferentes dados (dados locais). A unidade de controle comanda os processadores, que funcionam em modo síncrono ("lockstep"), ou seja, cada processador ou executa uma instrução difundida ("broadcast") pela unidade de controle ou ignora-a e se mantém ocioso.

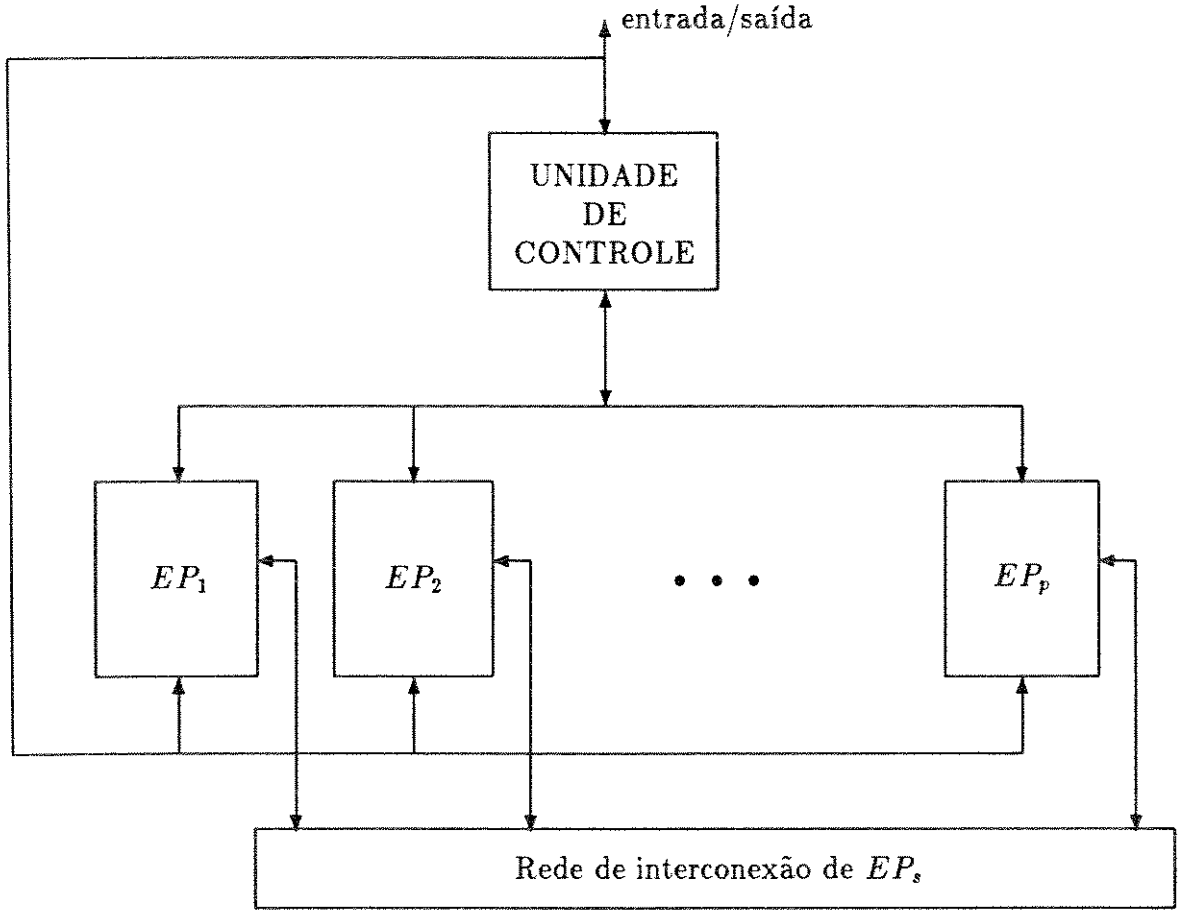


Figura 2.2 - Estrutura de um computador "array"

Computadores Multiprocessadores

São computadores com dois ou mais processadores independentes, cada qual capaz de executar seus próprios programas, com acesso a módulos de memórias, canais de comunicação, entrada/saída de dados e demais periféricos. As máquinas enquadradas nesta estrutura serão abordadas com maior detalhe nos itens seguintes deste capítulo, já que é o tipo de máquina que se pretende utilizar.

2.3 Arquitetura de Máquinas Paralelas

A arquitetura das máquinas paralelas tem influência direta no desempenho de algoritmos paralelos, já que este desempenho está ligado à troca de informações entre processadores. As arquiteturas usadas em máquinas paralelas possuem cada qual suas vantagens

e desvantagens, e dependendo disso obtém-se melhor ou pior desempenho na solução de um determinado problema.

Tentando esclarecer melhor este assunto, a referência [21] apresenta três esquemas para caracterizar as arquiteturas de computadores: (a) um esquema baseado na multiplicidade de dados e instruções; (b) outro baseado no processamento paralelo *versus* processamento serial; e (c) finalmente um esquema determinado pelo grau de paralelismo e "pipelining" em vários níveis do subsistema.

Estes esquemas podem ser usados para classificar as arquiteturas paralelas e com isto compreendê-las melhor. O caminho seguido aqui é o sugerido por [14], que consiste em levar em conta o esquema (a) acima mencionado e excluir arquiteturas que incorporem somente baixo nível de paralelismo (por exemplo, "pipelining").

O esquema (a) é conhecido como esquema de Flynn [16] e baseia-se nas possibilidades de combinação entre uma ou mais sequências de instruções atuando sobre uma ou mais sequências de dados, originando as seguintes classes:

- Única instrução, único dado/**SISD** (single instruction, single data). Corresponde aos computadores sequenciais convencionais nos quais as instruções são executadas sequencialmente sobre um conjunto de dados, como mostra a Figura 2.3(a).

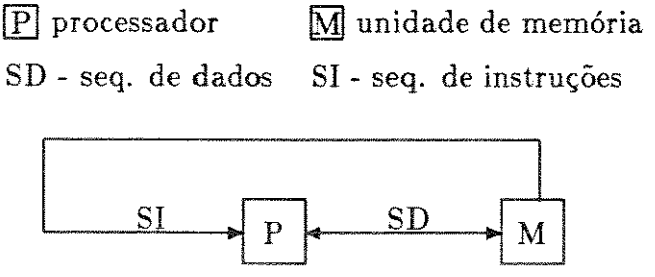


Figura 2.3(a) - Esquema SISD

- Única instrução, múltiplos dados/**SIMD** (single instruction, multiple data). Corresponde aos computadores "array", nos quais as instruções são executadas simultaneamente por várias unidades processadoras sobre os dados previamente carregados em suas memórias, como mostrado na Figura 2.3(b). As máquinas que exploram paralelismo maciço (connection machines) também se enquadram nesta categoria.

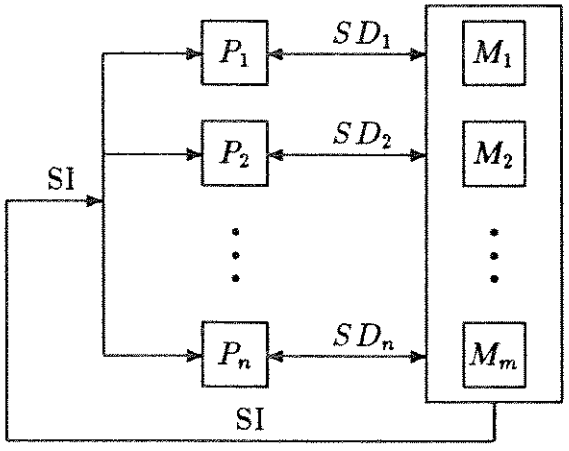


Figura 2.3(b) - Esquema SIMD

- Múltiplas instruções, único dado/**MISD** (multiple instruction, single data). Não existem computadores nesta categoria. Sua estrutura é mostrada na Figura 2.3(c).

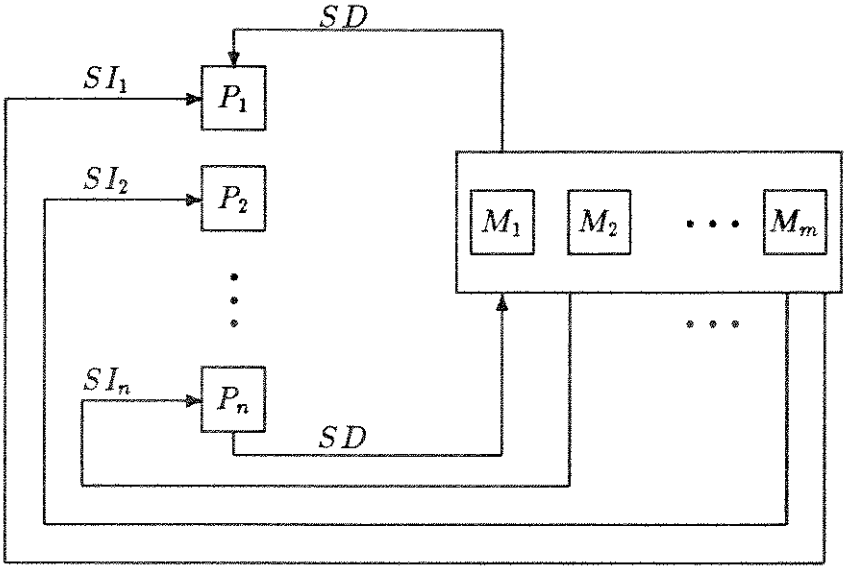


Figura 2.3(c) - Esquema MISD

- Múltiplas instruções, múltiplos dados/**MIMD** (multiple instruction, multiple data). Corresponde aos computadores multiprocessadores, onde cada unidade processadora recebe uma sequência de instruções diferente para executar sobre uma sequência de dados também diferente, como mostra-se na Figura 2.3(d).

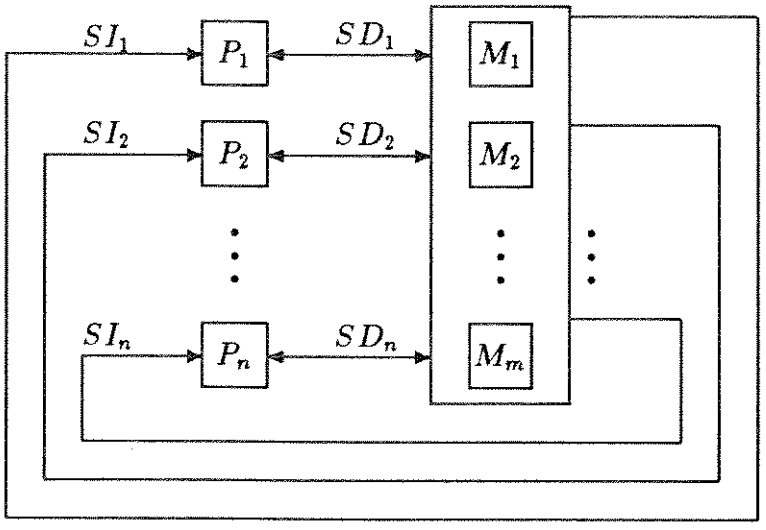


Figura 2.3(d) - Esquema MIMD

Dentro da classificação das arquiteturas paralelas propostas em [14], as arquiteturas que interessam diretamente à solução do problema deste trabalho são as classificadas como MIMD, e que por sua vez podem ser subdivididas em:

1. Memória distribuída;
2. Memória partilhada;
3. Memória híbrida.

2.4 A Arquitetura MIMD

Computadores MIMD realizam processamento paralelo através de processadores que operam de maneira autônoma mas interagem de acordo com diferentes formas de comunicação e acesso à memória.

Arquiteturas de memória distribuída

As arquiteturas de memória distribuída possuem nós de processamento constituídos de processador autônomo e sua memória local, sendo a conexão entre os nós feita via canais de comunicação, que são utilizados pelos nós para partilharem dados através de trocas de mensagens. Na Figura 2.4 mostra-se um esquema totalmente interconectado, onde cada processador possui uma conexão direta com todos os outros.

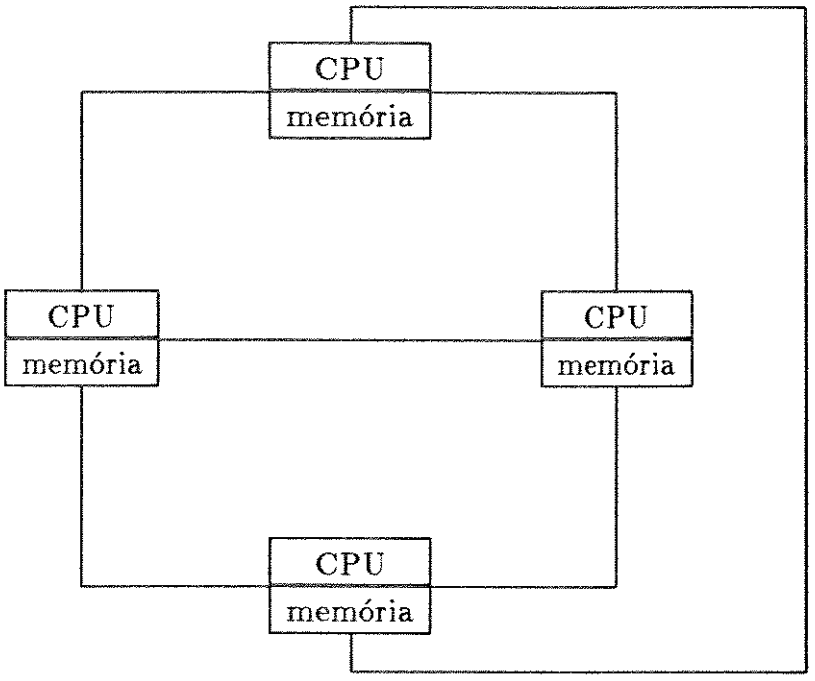


Figura 2.4 - Esquema de memória distribuída totalmente interconectado.

Neste tipo de arquitetura o ponto chave é a forma de interconexão dos processadores, uma vez que torna-se inviável a interconexão total quando há um grande número de processadores. Vários esquemas menos densos de conexão têm sido propostos para atenuar as dificuldades de comunicação, pois necessariamente um processador para acessar dados de outro precisa trocar informação através da rede. A figura 2.5 mostra os principais esquemas.

O desempenho de um algoritmo paralelo em uma máquina com memória distribuída depende de como a localização dos dados é casada com a sua utilização pelos processadores. Portanto, um algoritmo (incluindo a forma de decomposição/coordenação) feito para uma máquina pode ter um desempenho muito inferior em outra com configuração diferente.

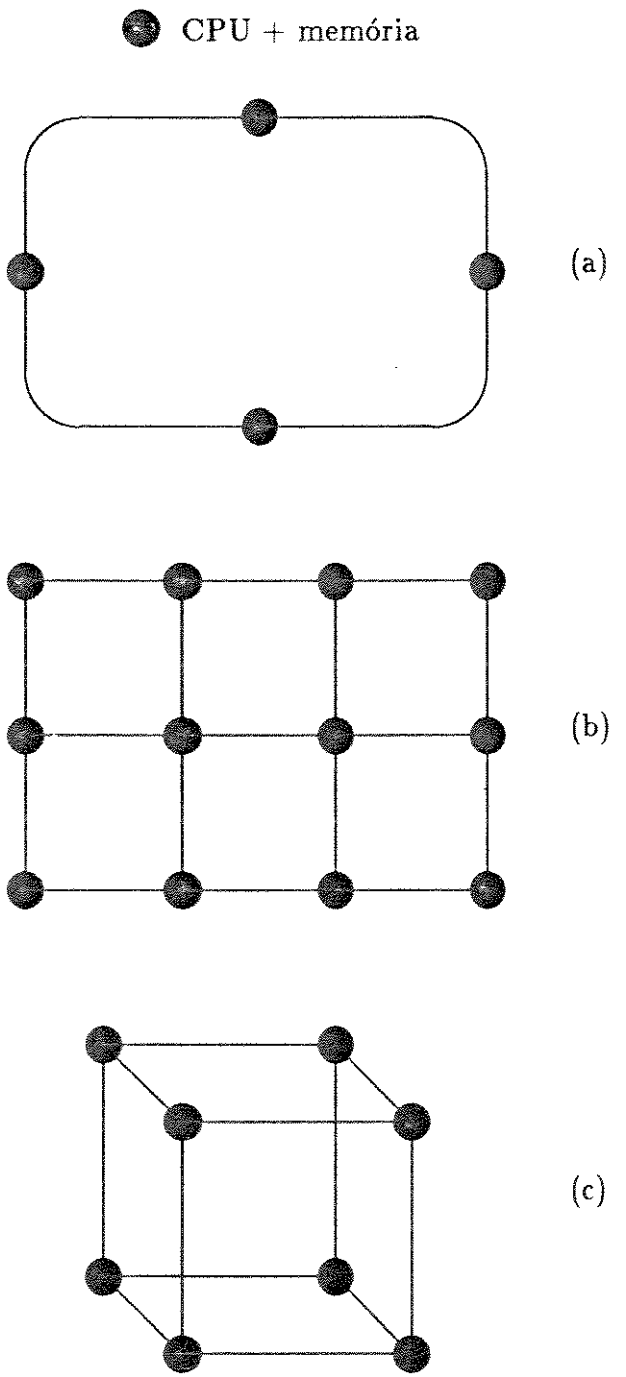


Figura 2.5 - Arquiteturas com memória distribuída.
(a) - Anel, (b) - Malha e (c) - Cubo

Arquiteturas de memória partilhada

Arquiteturas de memória partilhada efetuam a comunicação entre os processadores através do acesso que cada processador tem às memórias globais. Computadores de memória partilhada não apresentam os problemas das trocas de mensagens dos sistemas distribuídos, mas outros problemas surgem, tais como: congestionamento no acesso à memória devido ao tipo de barramento utilizado e sincronização do acesso aos dados, com necessidade de mecanismos de sincronização para evitar que uma posição da memória seja acessada por um processo antes que outro termine de atualizá-la. A figura 2.6 mostra alguns esquemas de interconexão entre processadores e memórias. Um simples barramento "time-sharing" não pode acomodar muitos processadores (de 4 a 20), e somente um de cada vez acessa o barramento. Uma conexão "crossbar" usa n^2 "crosspointer" para conectar n processadores com n memórias, e também possui um pequena capacidade de acomodação de processadores. Outra forma de interligação usa chaves multi-estágios, que apresentam uma importante característica que é a possibilidade de expansão para acomodar mais processadores e/ou memórias.

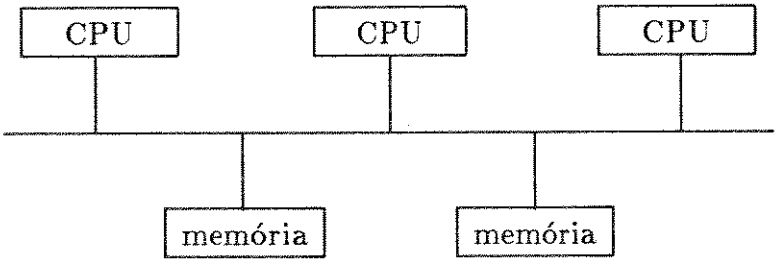


Figura 2.6(a) - Esquema memória partilhada com barramento comum.

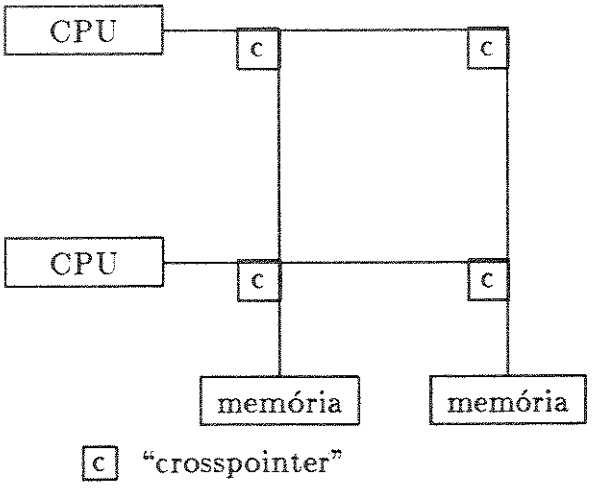


Figura 2.6(b) - Esquema memória partilhada/conexão "crossbar".

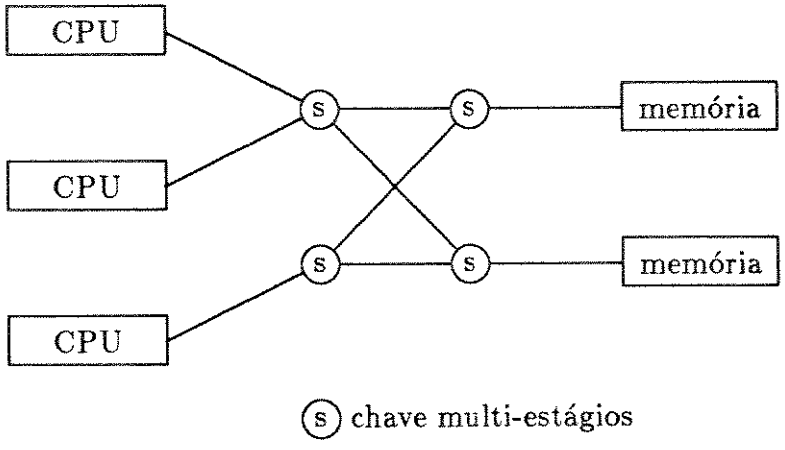


Figura 2.6(c) - Esquema memória partilhada/conexão com chave multi-estágios

Neste tipo de arquitetura um dos problemas está relacionado com a contenção (contention), que pode ocorrer de três formas [32]:

- Contenção na memória - ocorre quando diferentes processadores tentam acessar um módulo da memória ao mesmo tempo.
- Contenção na comunicação - ocorre quando diferentes processadores tentam utilizar a rede de comunicação para acessar módulos de memória.
- Tempo de latência (latency time) - é o tempo que os processadores gastam para acessar módulos de memória através das estruturas de interconexão.

As técnicas de redução da contenção são discutidas em [32] como sendo: redes sofisticadas de interconexão de processadores e memórias como a apresentada em [28]; utilização de memórias "cache"; organização dos módulos de memórias em local e global, gerando as arquiteturas híbridas; ou ainda, uma combinação destas.

Arquiteturas Híbridas

Nas arquiteturas híbridas são encontradas propriedades tanto dos sistemas de memória partilhada quanto dos sistemas de troca de mensagem. Como mostra a figura 2.7, os processadores têm certa quantidade de memória local e também compartilham outra parte de memória global. Se com esta arquitetura pode-se explorar as coisas boas de ambos os sistemas, por outro lado os problemas e desvantagens de ambos os sistemas também podem aparecer juntos. Isto depende de como os algoritmos paralelos são construídos e implementados.

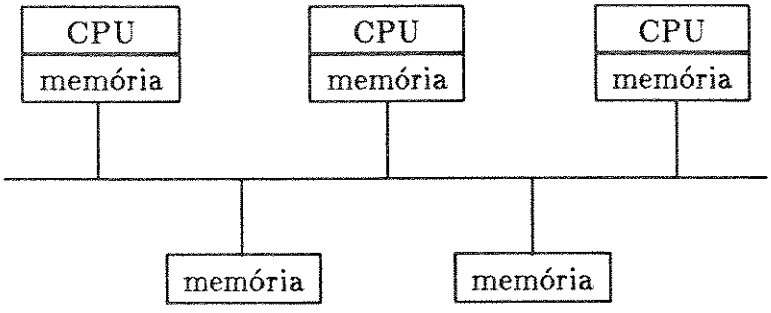


Figura 2.7 - Esquema de uma arquitetura híbrida com barramento comum

O modo de organização dos dados é o ponto principal para obter desempenho de um algoritmo a ser implementado em um sistema híbrido. Se o problema permitir, pode-se tratar um sistema híbrido como se fosse um sistema de memória partilhada. Neste caso, o algoritmo pode ter um bom desempenho se o acesso aos dados fora da memória local for eficiente, isto é, não causar congestionamento do barramento. Muitos destes dados, desde que não sejam de variáveis partilhadas, poderiam estar na memória local do respectivo processador, e assim evitar a comunicação dos processadores com as memórias partilhadas.

2.5 Ferramentas de Software

A programação de máquinas paralelas para a solução de problemas aplicados só pode ser feita convenientemente através de ferramentas de programação capazes de explorar o paralelismo. Esta tem sido uma das maiores dificuldades encontradas na utilização prática de computadores paralelos em computação científica [27]. Uma forma de tratar a questão consiste em declarar *explicitamente* o paralelismo, ou seja, o programador controla onde, como e o que será processado em paralelo. Outra forma, mais sofisticada e complexa, consiste em utilizar um compilador capaz de enxergar *automaticamente* o paralelismo e distribuir o trabalho pelos processadores, mas com isto pode-se não explorar todo paralelismo, obtendo-se pequeno ganho.

Três mecanismos têm sido usados para declarar *explicitamente* o paralelismo:

- incorporação de características de paralelismo no projeto das linguagens - isto maximiza o potencial para detecção automática de erros, porém existe uma dificuldade muito grande em se projetar linguagens genericamente aplicáveis e de fácil utilização;

- extensão de linguagens - as linguagens sequenciais podem ser estendidas para incorporar paralelismo através da adição de primitivas de compilação (ou pré-compilação), que facilitam a paralelização. A principal desvantagem é a dificuldade para construir o paralelismo de forma clara e lógica e,

- bibliotecas de rotinas de paralelismo - oferecem a vantagem da independência da linguagem. As desvantagens ficam por conta da limitação da aplicação ao tipo de máquina para a qual foi construída. Esta será a forma utilizada neste trabalho.

Pensando em paralelismo *explícito*, é fundamental ao programar-se uma máquina paralela ter em mente qual o tipo de modelo computacional que se pretende usar, ou seja, qual será a "imagem" da arquitetura da máquina vista pelo programador. Por exemplo, uma máquina de memória partilhada pode ser programada assumindo-se um modelo computacional de troca de mensagem ou um modelo de memória partilhada propriamente dita. Em geral, o que determina o tipo de modelo computacional é a forma de comunicação entre os processos (aqui entendidos como trechos de código) executados em cada processador. Outra preocupação importante ao definir-se o modelo computacional consiste em buscar o melhor casamento possível entre algoritmos e arquitetura.

Modelos de Troca de Mensagens

Modelos de troca de mensagens necessitam basicamente de duas novas funções, *send* e *receive* [23]. *Send* é usado para enviar mensagens contendo dados de um processador para outros, sendo que existem dois tipos de *send*: um que envia a mensagem e imediatamente depois continua o processamento normal (chamado *send* não-bloqueante) e outro (chamado *send* bloqueante), que interrompe o processamento até que a mensagem seja recebida.

Receive é usado para ler uma mensagem enviada por outro processador. Poderá também ser bloqueante ou não-bloqueante, sendo que ambos são necessários. Se um processo necessita de uma parte específica dos dados de outro processador, deve-se utilizar o *receive* bloqueante. Então o processo deve ficar interrompido até a chegada dos dados. Este tipo de *receive* pode levar a "deadlock", situação na qual dois processadores esperam informações um do outro. *Receive* não-bloqueante tem dois tipos de utilização, o uso mais comum é para uma recepção global, em que o processador recebe mensagens em diversas portas; outro tipo de uso é o processador ficar testando a porta de entrada, se não há mensagem, uma tarefa é executada, se há, outra tarefa é feita.

Modelo de Memória Partilhada

As extensões de linguagens necessárias em um modelo de memória partilhada, são bem maiores do que num modelo troca de mensagens. Primeiro é necessário distinguir quais dados são privados de cada processador e quais deles são globais (do conhecimento

de todos). Depois é preciso sincronização, porque parte dos dados são partilhados e poderá haver acesso fora de sequência.

Existem dois estilos de programação comumente usados para dividir o trabalho para cada processador [23]. No estilo *Fork-Join*, um processo é dividido em subprocessos (etapa *Fork*), os subprocessos são executados em diferentes processadores e aqueles que terminarem as tarefas primeiro esperam até que todos terminem (etapa *Join*). No estilo SPMD (single program, multiple data), cada processador executa o mesmo programa porém códigos diferentes dependendo do processador ou dos dados da memória partilhada. Ambos estilos precisam de códigos para acesso restrito a uma parte do programa, pois somente depois que um processador executar estes códigos é que está garantida a devida atualização dos dados que são solicitados por todos os processadores. Assim define-se uma *região crítica* como sendo uma parte do processo que contém códigos comuns a todos os processadores, porém, para serem executados por um único processador de cada vez. Uma *região crítica* é geralmente usada para operações com variáveis globais. Outra definição complementar é a de *região serial*, que é parte do processo que contém os códigos executados por um único processador, sendo os resultados passados para todos os processadores.

Em modelos de troca de mensagens, um *receive* bloqueante sincroniza; no estilo *Fork-Join*, o *Join* tem esta finalidade. No estilo SPMD outras construções são necessárias, por exemplo, pode-se usar uma *Barreira* que é um ponto do programa onde todos os processadores esperam pela chegada do último. Como uma *Barreira* é uma linha específica do programa onde um processador espera por todos, pode ocorrer de um processador nunca passar pela barreira levando a uma situação de "deadlock".

Capítulo 3

Multiprocessamento de Equações de Redes Elétricas

3.1 Introdução

Resolver um conjunto de equações algébricas lineares e esparsas é um dos problemas que frequentemente aparece na análise de sistemas de energia elétrica. Normalmente busca-se explorar as características de esparsidade destas equações, uma vez que apresentam grau de esparsidade maior que 95%.

Nos últimos anos vários trabalhos foram propostos com a finalidade de resolver este problema em computadores paralelos. Neste tipo de problema existem duas partes principais: a fatoração da matriz e a solução direta. A fatoração da matriz é uma etapa praticamente sequencial e normalmente realizada apenas uma vez durante todo processo de resolução, enquanto a solução direta é resolvida muitas vezes em cada estudo. Nos estudos de estabilidade transitória, por exemplo, esta etapa é repetida centenas ou mesmo milhares de vezes até o processamento final. Deste modo somente a obtenção da solução direta através de processamento paralelo será abordada neste trabalho.

3.2 Conceitos Básicos

As equações algébricas que descrevem as redes elétricas podem ser expressas, em geral, na forma $I = YV$: onde I é o vetor de correntes complexas; Y é a matriz admitância complexa esparsa e V é o vetor de tensões complexas. Dependendo do problema específico a ser resolvido (fluxo de potência, estabilidade transitória, transitórios eletromagnéticos, etc.) outras formas aparecem, mas todas mantém a estrutura da matriz Y. Desta forma é conveniente generalizar as equações da seguinte maneira:

$$Ax = b \tag{3.1}$$

onde A é uma matriz esparsa, simétrica pelo menos em estrutura, definida positiva e de ordem m, o vetor x é a solução desejada e b é um vetor conhecido e não esparso.

O método geral de solução da equação (3.1) que tem sido usado em computadores com uniprocessador é como segue.

A matriz A é fatorada por eliminação de Gauss, aplicando-se sucessivamente matrizes de transformação elementares [24], como as da seguinte expressão:

$$(T_U^q \cdots T_U^2 T_U^1 T_D^m \cdots T_D^2 T_D^1 T_L^q \cdots T_L^2 T_L^1) A = I \tag{3.2}$$

onde T_U^i é uma matriz identidade de ordem m, exceto em uma posição do triângulo superior, que contém um elemento que zera a respectiva posição de A; T_D^i é uma matriz identidade de ordem m, exceto na posição i da diagonal, que contém um elemento que torna a respectiva posição de A igual a um; T_L^i é uma matriz identidade de ordem m, exceto em uma posição do triângulo inferior, que contém um elemento que zera a respectiva posição de A e I é uma matriz identidade de ordem m.

Desta forma tem-se:

$$A^{-1} = T_U^q \cdots T_U^2 T_U^1 T_D^m \cdots T_D^2 T_D^1 T_L^q \cdots T_L^2 T_L^1 \tag{3.3}$$

Definindo-se,

$$L^i = (T_L^i)^{-1} \tag{3.4}$$
$$D^i = (T_D^i)^{-1} \tag{3.5}$$
$$U^i = (T_U^i)^{-1} \tag{3.6}$$

pode-se expressar A como:

$$A = L^1 L^2 \cdots L^q D^1 D^2 \cdots D^r U^1 U^2 \cdots U^q \tag{3.7}$$

Agrupando as transformações elementares, pode-se definir:

$$L = L^1 L^2 \cdots L^q \tag{3.8}$$
$$D = D^1 D^2 \cdots D^r \tag{3.9}$$
$$U = U^1 U^2 \cdots U^q \tag{3.10}$$

onde L é uma matriz triangular inferior com diagonal unitária, D é uma matriz diagonal e U é uma matriz triangular superior com diagonal unitária. Nestas matrizes não

surge nenhum novo elemento não-zero diferente daqueles das respectivas matrizes de transformação e portanto a esparsidade é preservada. Os elementos não-zero de L e U fora da diagonal são chamados de fatores triangulares.

Assim diz-se que A foi decomposta em fatores L, D e U:

$$A = LDU \tag{3.11}$$

Antes da fatoração é conveniente realizar uma ordenação nas linhas/colunas da matriz A, geralmente para minimizar o aparecimento de novos elementos diferentes de zero ("fill-ins"). Essa ordenação é equivalente a reordenar os nós da rede elétrica. A ordenação pode ter outras finalidades como será visto mais adiante.

Frequentemente a fatoração de A é implementada através da técnica de bifatoração [40] que permite explorar melhor as características da estrutura simétrica de L e U, facilitando a programação e armazenamento dos fatores triangulares. Assim, a matriz L é armazenada compactamente por coluna, de forma que a estrutura de uma coluna do triângulo inferior (L) contenha exatamente a mesma informação de estrutura da respectiva linha do triângulo superior (U), como ilustra a Figura 3.1, onde destaca-se a coluna i de L e a linha i de U.

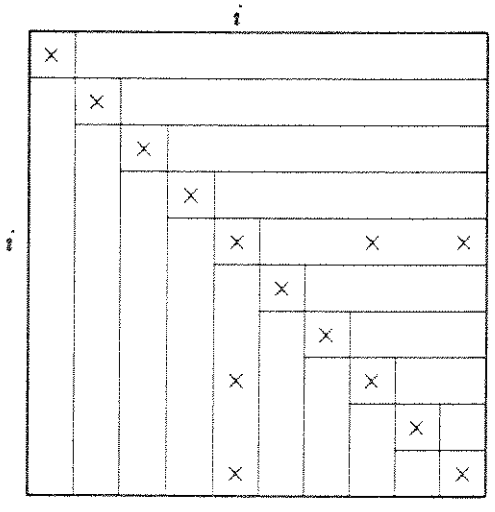


Figura 3.1 - Forma tradicional de armazenamento dos fatores triangulares.

Por facilidade, mas sem perda de generalidade, supõe-se daqui por diante, que a matriz A é simétrica e portanto $U = L^t$. Assim:

$$A = LDL^t \tag{3.12}$$

Após a fatoração de A, a solução de $Ax = b$ pode ser obtida em três passos:

1. $Lz = b$ (conhecida como substituição "forward");
2. $Dy = z$ (divisão de z pela diagonal);
3. $L^t x = y$ (conhecida como substituição "backward").

Na realidade as substituições são operadas uma a uma usando-se os fatores que são obtidos diretamente de L, D, L^t e que vão sendo aplicadas às respectivas posições do vetor b.

Pode-se operar também usando-se matrizes de fatores inversos (matriz W) da seguinte forma [15]:

$$A^{-1} = W^t D^{-1} W \tag{3.13}$$

onde $W = L^{-1}$, ou seja:

$$W = T_L^q \cdots T_L^2 T_L^1 \tag{3.14}$$

$$x = W^t D^{-1} W b \tag{3.15}$$

A equação (3.15) pode também ser resolvida em três passos:

1. $z = Wb$
2. $y = D^{-1} z$
3. $x = W^t y$

A diferença em relação ao procedimento anterior é que agora são usadas operações matriciais.

A matriz W pode ser calculada através da multiplicação da direita para a esquerda da equação (3.14), porém surgirão novos elementos não-zero em W ("fill-ins" adicionais), indicando que esta matriz é menos esparsa que L.

Sob o ponto de vista de uniprocessamento, a solução através das substituições "forward" / "backward" oferece as seguintes vantagens:

- não requer o cálculo da matriz de fatores inversos (W) .
- L é mais esparsa que W, o que implica em menos operações (multiplicações e adições) para obter a solução.

Se o objetivo é usar multiprocessamento então a abordagem através da matriz W é vantajosa, como será mostrado neste trabalho.

Às vezes pode ser conveniente agrupar algumas transformações elementares vizinhas como, por exemplo, as transformações correspondentes a uma coluna ou uma linha da matriz A, dependendo de como a eliminação estiver sendo feita. Neste caso pode-se expressar L como:

$$L = L_1 L_2 \cdots L_n \tag{3.16}$$

onde L_i é uma matriz identidade exceto na i-ésima coluna que contém a coluna i de L, e que corresponde ao agrupamento de todas as transformações elementares necessárias para processar a coluna i de A.

Consequentemente pode-se expressar W como:

$$W = L^{-1} = W_n \cdots W_2 W_1 \tag{3.17}$$

onde $W_i = L_i^{-1}$ e portanto tem os mesmos elementos fora da diagonal que L_i, apenas com os sinais trocados. Deve ser notado que L_i e W_i correspondem ao processamento de um nó da rede elétrica durante a eliminação de Gauss.

Desse modo a equação (3.15) fica:

$$x = W_1^t W_2^t \cdots W_n^t D^{-1} W_n \cdots W_2 W_1 b \tag{3.18}$$

Particionamento da Matriz W

Particionar uma matriz W (ou a matriz L se for o caso) significa agrupar W_i ou W_i^t adjacentes de tal modo a realizar as correspondentes operações em bloco. Cada bloco de colunas (ou linhas) é chamado de uma partição da matriz.

Desse modo:

$$x = W_{1p}^t W_{2p}^t \cdots W_{np}^t D^{-1} W_{np} \cdots W_{2p} W_{1p} b \tag{3.19}$$

onde W_{1p} W_{2p} e W_{np} representam as operações correspondentes à partição 1, partição 2 e partição n, respectivamente.

3.3 Solução Convencional Usando Substituições

Neste item serão apresentadas as operações necessárias para realizar as etapas "forward" e "backward" da solução direta, considerando que a matriz A tenha sido fatorada por um processo de bifatoração. Assume-se que a substituição "forward" é feita por coluna e a substituição "backward" é feita por linha. Suponha uma rede com nb nós, uma matriz L com um fator triangular $l_{j,k}$ na linha j e coluna k, e uma matriz D onde $d_{k,k}$ é o elemento da posição k da diagonal. Então a etapa "forward" consiste em:

Algoritmo F1

1. Faça $z = b$ e $k = 0$;
2. Faça $k = k + 1$;
   (a) Se $k = nb$ fim;
   (b) Caso contrário, segue;
3. Verifique se existem elementos diferentes de zero na coluna k;
   (a) Se não existem, vá para o passo 2;
   (b) Caso contrário, localize a linha j, faça $z_j = z_j - l_{j,k} z_k$ e volte ao passo 3.

As operações de divisão pela diagonal podem ser feitas juntamente com a "forward" ou com a "backward", mas por facilidade elas foram aqui colocadas em um outro algoritmo como segue:

Algoritmo D1

1. Faça $k = 0$;
2. Faça $k = k + 1$;
   (a) Se $k > nb$ fim.
   (b) Caso contrário, faça $y_k = z_k / d_{k,k}$ e volte ao passo 2;

E finalmente a "backward" pode ser feita como:

Algoritmo B1

1. Faça $x = y$ e $j = nb$;
2. Faça $j = j - 1$;
   (a) Se $j = 0$ fim;
   (b) Caso contrário, segue;
3. Verifique se existem elementos diferentes de zero na linha j;
   (a) Se não existem, vá para o passo 2;
   (b) Caso contrário, localize a coluna k, faça $x_j = x_j - l_{j,k} x_k$ e volte ao passo 3.

Deve ser notado que as operações envolvidas na "forward" e "backward" são operações de multiplicação e de adição. Observe-se também que existe uma certa ordem para realizar estas operações, a fim de garantir que a solução final seja correta. Com o objetivo de esclarecer a natureza destas relações, o que é fundamental para processamento paralelo, considere-se a rede e a matriz L da Figura 3.2 como um exemplo.

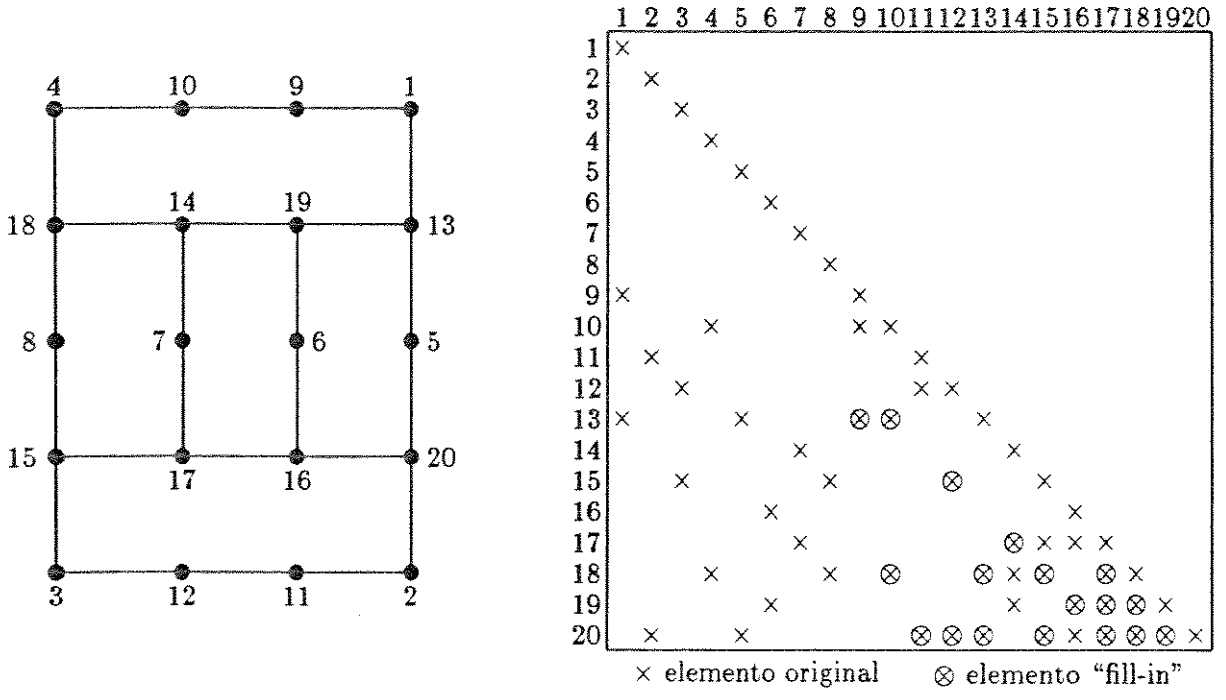


Figura 3.2 - Rede 20 nós [34] e estrutura da matriz L

As operações elementares para fazer a substituição "forward" são mostradas na Tabela 3.1.

Tabela 3.1 - Operações na substituição "forward".

Tarefa	Operação	Tarefa	Operação	Tarefa	Operação
# 1	$z_9 = z_9 - l_{9,1}z_1$	# 2	$z_{13} = z_{13} - l_{13,1}z_1$	# 3	$z_{11} = z_{11} - l_{11,2}z_2$
# 4	$z_{20} = z_{20} - l_{20,2}z_2$	# 5	$z_{12} = z_{12} - l_{12,3}z_3$	# 6	$z_{15} = z_{15} - l_{15,3}z_3$
# 7	$z_{10} = z_{10} - l_{10,4}z_4$	# 8	$z_{18} = z_{18} - l_{18,4}z_4$	# 9	$z_{13} = z_{13} - l_{13,5}z_5$
# 10	$z_{20} = z_{20} - l_{20,5}z_5$	# 11	$z_{16} = z_{16} - l_{16,6}z_6$	# 12	$z_{19} = z_{19} - l_{19,6}z_6$
# 13	$z_{14} = z_{14} - l_{14,7}z_7$	# 14	$z_{17} = z_{17} - l_{17,7}z_7$	# 15	$z_{15} = z_{15} - l_{15,8}z_8$
# 16	$z_{18} = z_{18} - l_{18,8}z_8$	# 17	$z_{10} = z_{10} - l_{10,9}z_9$	# 18	$z_{13} = z_{13} - l_{13,9}z_9$
# 19	$z_{13} = z_{13} - l_{13,10}z_{10}$	# 20	$z_{18} = z_{18} - l_{18,10}z_{10}$	# 21	$z_{12} = z_{12} - l_{12,11}z_{11}$
# 22	$z_{20} = z_{20} - l_{20,11}z_{11}$	# 23	$z_{15} = z_{15} - l_{15,12}z_{12}$	# 24	$z_{20} = z_{20} - l_{20,12}z_{12}$
# 25	$z_{18} = z_{18} - l_{18,13}z_{13}$	# 26	$z_{19} = z_{19} - l_{19,13}z_{13}$	# 27	$z_{20} = z_{20} - l_{20,13}z_{13}$
# 28	$z_{17} = z_{17} - l_{17,14}z_{14}$	# 29	$z_{18} = z_{18} - l_{18,14}z_{14}$	# 30	$z_{19} = z_{19} - l_{19,14}z_{14}$
# 31	$z_{17} = z_{17} - l_{17,15}z_{15}$	# 32	$z_{18} = z_{18} - l_{18,15}z_{15}$	# 33	$z_{20} = z_{20} - l_{20,15}z_{15}$
# 34	$z_{17} = z_{17} - l_{17,16}z_{16}$	# 35	$z_{19} = z_{19} - l_{19,16}z_{16}$	# 36	$z_{20} = z_{20} - l_{20,16}z_{16}$
# 37	$z_{18} = z_{18} - l_{18,17}z_{17}$	# 38	$z_{19} = z_{19} - l_{19,17}z_{17}$	# 39	$z_{20} = z_{20} - l_{20,17}z_{17}$
# 40	$z_{19} = z_{19} - l_{19,18}z_{18}$	# 41	$z_{20} = z_{20} - l_{20,18}z_{18}$	# 42	$z_{20} = z_{20} - l_{20,19}z_{19}$

Esta rede exemplo será utilizada ao longo deste trabalho com o objetivo de ilustrar diferentes idéias.

3.4 Trabalhos Anteriores

Neste item serão apresentadas e discutidas algumas abordagens e técnicas que têm sido propostas para resolver as equações de redes elétricas utilizando processamento paralelo. Será enfocada a característica fundamental das abordagens, ou seja, a forma como o problema é decomposto em tarefas menores que serão coordenadamente executadas em paralelo.

Uma linha de pesquisa consiste em basear a decomposição do problema na própria topologia da rede elétrica, dividindo-a em subredes menores (diacóptica) e resolvendo primeiramente as partes para depois combinar e corrigir todas as soluções [20]. Seguindo esta linha, cita-se por exemplo, a referência [13] que propõe uma solução paralela para cálculo de estabilidade transitória e a referência [2] para transitórios eletromagnéticos. Dividindo-se a rede em várias subredes pode-se atribuir uma ou mais redes a cada processador, que terá a tarefa de calcular a solução deste subsistema, comunicando-se com os demais processadores apenas no final para coordenar a solução. A divisão da rede geralmente leva a subredes de tamanhos diferentes e isto pode causar grande desbalanceamento de carga entre os processadores e tornar o esquema ineficiente. A dificuldade inerente a esta abordagem é encontrar critérios gerais para dividir a rede e de que maneira realizar a coordenação sem apresentar muito "overhead".

Quando a divisão da rede proporciona mais subredes do que os processadores existentes na máquina, pode-se atribuir mais de uma subrede a um mesmo processador; já quando a divisão fornece um número de subredes menor do que o número de processadores, pode-se pensar em explorar também o potencial de paralelismo na solução de cada subrede, o que consiste em resolver um sistema do tipo $Ax = b$. A referência [2] nota que no próprio algoritmo de solução direta, para processamento sequencial, existe um potencial de paralelismo. Na etapa "forward" deste algoritmo, o laço externo corresponde a cada uma das colunas da matriz e o laço interno busca elementos não-zero em cada uma das linhas da coluna selecionada, como pode-se observar no algoritmo F1. Este laço interno pode ser feito em paralelo, mas como a grande maioria das colunas possuem poucas linhas com elementos diferentes de zero (2 a 4) então tem-se poucas tarefas a fazer em paralelo, tornando este algoritmo pouco eficiente para máquinas com vários processadores.

Outra linha de pesquisa baseia a decomposição na estrutura da matriz A. Uma técnica discutida em [5] é dividir a matriz em blocos de tarefas e então atribuir um bloco para cada processador disponível. Uma divisão em blocos possível, pode ser vista

na Figura 3.3, considerando a rede exemplo nota-se que os blocos não são totalmente independentes; por exemplo a tarefa #17 (Tabela 3.1) localizada no bloco 5 não pode ser realizada sem que a tarefa #1 do bloco 2 tenha sido completada. Estas relações de precedência tornam a divisão de A por blocos de pouca utilidade.

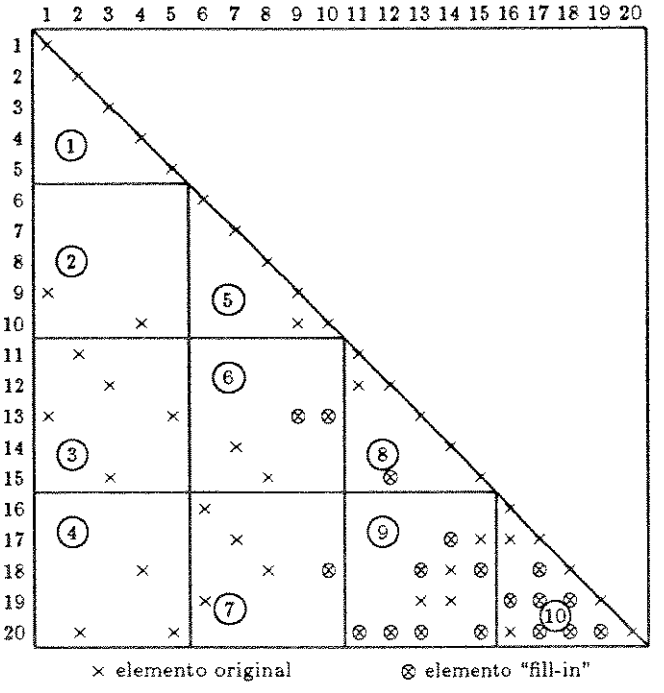


Figura 3.3 - Matriz L dividida em blocos.

Outro tipo de abordagem consiste em decompor a solução de $Ax = b$ em operações aritméticas elementares e então explorar o paralelismo inerente a elas. Vários trabalhos têm sido propostos nesta linha [5], [9], [36], [15], para obtenção da solução direta. Também são apresentadas soluções para o problema da fatoração de matrizes [6], [8], que possui alguma semelhança com o problema da solução direta.

Antes de discutir este tipo de abordagem deve-se analisar as operações aritméticas elementares e suas relações em detalhe. A solução direta compreende essencialmente uma sequência ordenada de tarefas de atualizações dos componentes do vetor b, onde cada uma delas é formada por operações aritméticas de multiplicação e adição. As relações entre cada uma destas tarefas podem ser observadas na Tabela 3.1, que mostra as tarefas para a "forward"da matriz L da Figura 3.2. Observando as primeiras 16 tarefas vê-se que #2 e #9 não podem ser processadas ao mesmo tempo por dois processadores. A ordem em que serão executadas não importa, mas uma deve ser feita antes da outra, pois ambas atualizam a posição 13 do vetor z. Problemas semelhantes ocorrem com #4 e #10, #6 e #15, #8 e #16. Pode-se dizer que entre estas tarefas não existe uma relação de precedência, mas sim uma relação de *não-simultaneidade*.

Observando-se agora a tarefa #17 nota-se que para realizá-la é necessário que z_9 esteja atualizado, ou seja, que #1 tenha sido executada. Então a tarefa #1 deve ser feita antes da #17, indicando uma efetiva relação de precedência. Estas relações de precedência podem ser extraídas de um grafo dos caminhos de fatoração, como o da Figura 3.4 obtido para a rede-exemplo de 20 nós (ver Apêndice B para a obtenção dos caminhos).

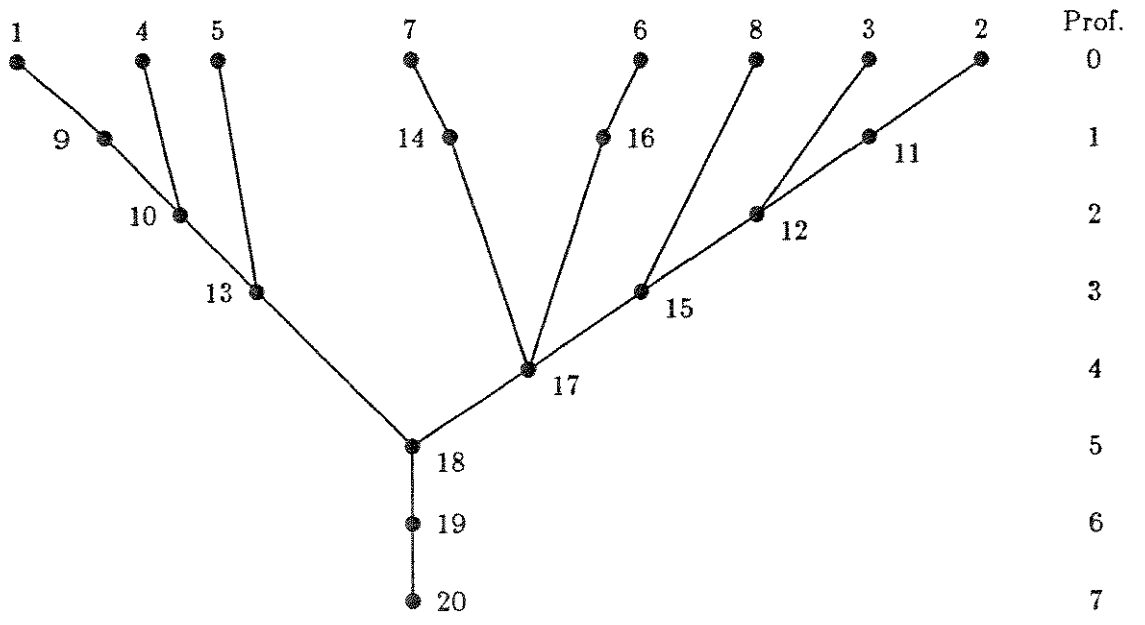


Figura 3.4 - Grafo dos caminhos de fatoração com indicação da profundidade

Na relação de tarefas da "forward" observa-se que para obter z_9 é necessário apenas z_1, e para obter z_{10} é necessário ter z_4 e z_9 atualizados. Isto equivale a observar o grafo e ver que para calcular z_{10} necessita-se somente da informação dos nós que antecedem o nó 10 nos caminhos de fatoração.

Construindo-se um grafo destas tarefas observam-se de forma mais clara suas relações. Isto é sugerido na referência [9] para a solução direta, considerando apenas as relações de precedência. As referências [6] e [8], que propõem algoritmos de decomposição e atribuição de tarefas para a fatoração de matrizes, constroem grafos tratando relação de *não-simultaneidade* como relação de precedência. Um grafo para as tarefas da Tabela 3.1 é mostrado na Figura 3.5, onde as setas indicam a relação de precedência e a ordem em que devem ser realizadas as operações elementares.

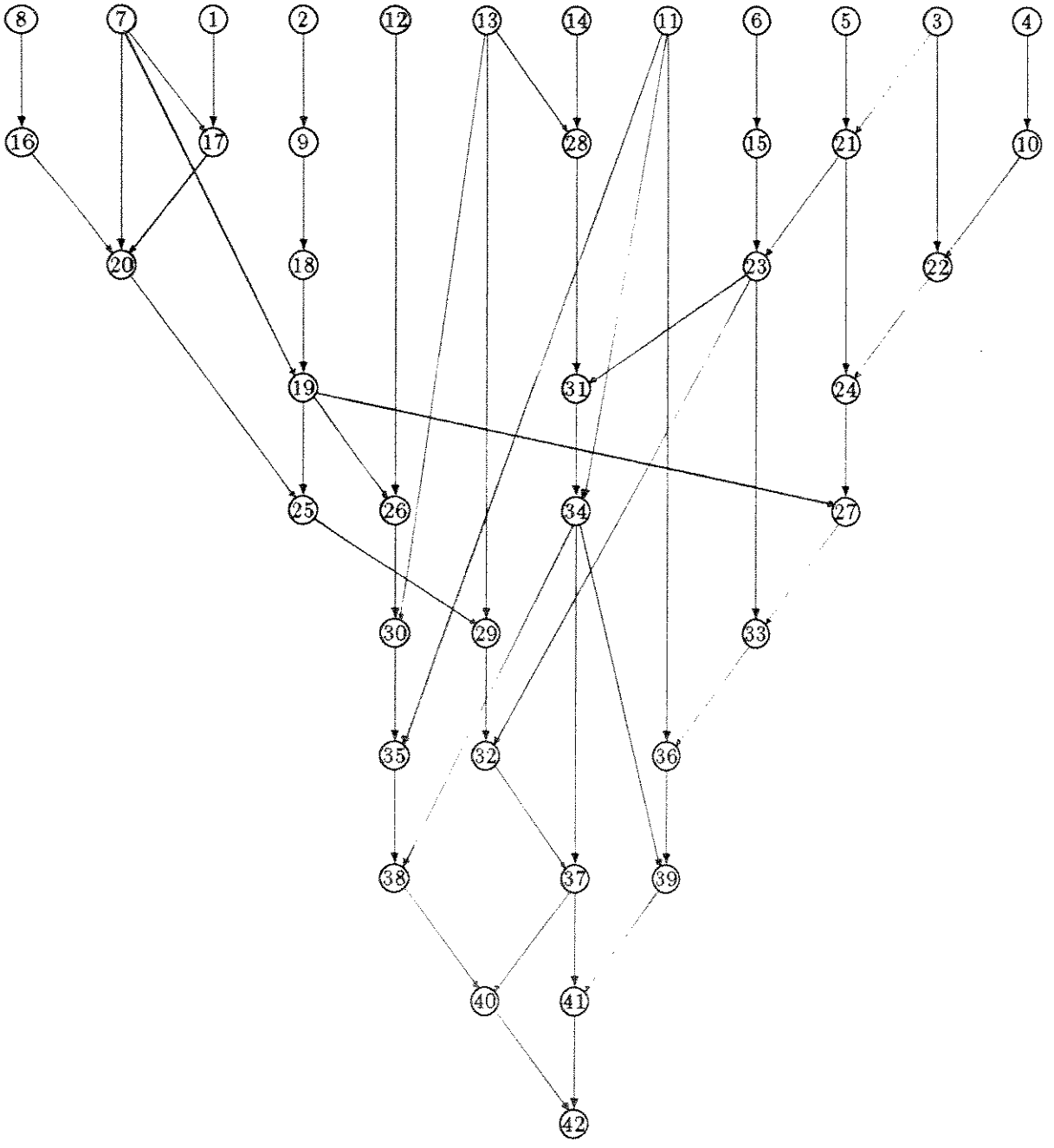


Figura 3.5 - Grafo das relações entre tarefas da rede 20 nós.

Neste grafo observa-se que no início existem muitas tarefas independentes, mas no

final as tarefas se tornam praticamente sequenciais, ou seja, a estrutura das tarefas possui a forma de um triângulo com a base em cima. Esta forma de estrutura poderá comprometer o desempenho de uma solução que use a decomposição em operações elementares atribuidas a cada processador, pois rapidamente muitos processadores ficarão parados sem ter tarefas para realizar. A referência [6] buscou alargar a base deste triângulo, através de uma ordenação da matriz que leve em conta a profundidade do nó no caminho de fatoração (ordenação ML-MD mostrada no Apêndice C), relaxando a minimização de "fill-ins" (ordenação MD). Uma ordenação tipo ML-MD gera trabalho extra que são os "fill-ins", mas proporcionam uma árvore de fatoração mais apropriada para processamento através de multiprocessadores, embora a estrutura em forma de triângulo continue. Outra dificuldade - talvez a principal que surge neste tipo de abordagem - é que para fazer uma tarefa (uma multiplicação mais uma adição) é necessário fazer um teste para verificar as relações de precedência, tentando descobrir se a tarefa pode ou não ser executada, e após a execução devem-se atualizar as relações de precedência para que as tarefas seguintes (abaixo no grafo) possam ser feitas. Isto representa trabalho extra e "overhead" de comunicação que podem prejudicar o desempenho, pois o algoritmo faz muitos testes e atualizações para pequenas tarefas (explorou-se o máximo de paralelismo mas o grão é muito *fino*).

Pensando em evitar os inconvenientes das relações de precedência e dos testes com estas relações, a referência [1] propõe uma solução usando o grafo dos caminhos de fatoração e os conceitos de "fast-forward" e "fast-backward" (veja Apêndice B) para montar um algoritmo de solução paralela. A base deste algoritmo são os caminhos de fatoração (CF) dos nós iniciais do grafo. O primeiro caminho é obtido partindo-se de um nó inicial aleatório e percorrendo-se o grafo até o nó final. Para os demais caminhos parte-se do nó inicial até encontrar um nó que já pertença a um caminho previamente percorrido. Para ilustrar este método, considere-se a rede da Figura 3.2 e o respectivo grafo da Figura 3.4:

$$
\begin{aligned}
CF_2 &= \{2, 11, 12, 15, 17, 18, 19, 20\} \\
CF_3 &= \{3\} \\
CF_8 &= \{8\} \\
CF_6 &= \{6, 16\} \\
CF_7 &= \{7, 14\} \\
CF_5 &= \{5, 13\} \\
CF_4 &= \{4, 10\} \\
CF_1 &= \{1, 9\}
\end{aligned}
$$

Partindo-se de CF_2, por exemplo, monta-se um vetor z_2 de dimensão n e igual a zero exceto nas posições correspondentes a CF_2 que são feitas iguais aos valores de b. Isto se

repete para cada um dos diferentes caminhos de fatoração. Desta forma cada processador recebe a tarefa de fazer a "fast-forward" de um caminho de fatoração, devendo haver comunicação para atualização do elemento do vetor correspondente ao nó da interseção de dois CFs. Neste exemplo são necessários 8 processadores: $P_1, P_2, P_3, \ldots, P_8$ obtêm solução de $z_2, z_3, z_8, z_6, z_7, z_5, z_4, z_1$ respectivamente. Finalmente estes vetores são somados para obter a solução:

$$z = z_2 + z_3 + z_8 + z_6 + z_7 + z_5 + z_4 + z_1 \tag{3.20}$$

As tarefas atribuídas a cada processador dependem do número de colunas a serem processadas, que são indicadas pelo caminho de fatoração, e do número de não-zeros em cada uma destas colunas. Assim neste algoritmo a distribuição de cargas não é uniforme (P1 ficou com a maior carga), mas isto pode ser melhorado buscando-se o nó inicial que tenha menor caminho de fatoração. Desta forma escolhe-se primeiro o nó 5, em seguida busca-se novamente o menor caminho, e encontra-se o nó 4. Seguindo esta lógica até o final obtém-se:

$$\begin{aligned}
CF_5 &= \{5, 13, 18, 19, 20\} \\
CF_4 &= \{4, 10\} \\
CF_1 &= \{1, 9\} \\
CF_6 &= \{6, 16\} \\
CF_7 &= \{7, 14\} \\
CF_8 &= \{8, 15\} \\
CF_3 &= \{3, 12\} \\
CF_2 &= \{2, 11\}
\end{aligned}$$

A ordenação da matriz pode influenciar bastante este algoritmo. Uma ordenação tipo MD procura minimizar "fill-ins" (menos elementos por coluna), mas gera grandes caminhos de fatoração. Uma ordenação ML-MD, que proporciona caminhos de fatoração menores, poderá não trazer grandes vantagens, uma vez que vai gerar mais "fill-ins" criando assim não-zeros adicionais por coluna. Uma ordenação conveniente para este algoritmo deve considerar a restrição de minimizar "fill-ins" e buscar CFs menores, como a ordenação MD-MNP (ver Apêndice C). Mesmo com todos estes refinamentos ainda permanece o forte desbalanceamento de tarefas, pois sempre vai haver um caminho muito maior que os outros. Uma exigência extra deste algoritmo é a resolução da equação (3.20) que pode introduzir um grande "overhead" de algoritmo. A referência [39] propõe uma solução usando caminhos de fatoração e explorando a melhor ordem de fazer as operações elementares, para uso com máquinas a fluxo de dados. Com

isto consegue-se maior ganho na decomposição do problema aumentando um pouco a comunicação.

Outra linha de pesquisa propõe particionamento de matriz a fim de explorar o paralelismo [15], [38]. Em [38], considera-se que os fatores triangulares LDU de A não são os únicos que permitem calcular a solução direta, é proposto achar fatores adicionais de A como mostrado abaixo:

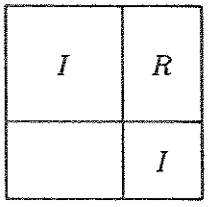
$$\begin{bmatrix} I & \\ K & I \end{bmatrix} \begin{bmatrix} L_{11} & \\ & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & \\ & U_{22} \end{bmatrix} \begin{bmatrix} I & R \\ & I \end{bmatrix} x = b$$

Nesta solução I é matriz identidade, L e U são matrizes triangulares inferior e superior, respectivamente, enquanto K e R são matrizes retangulares cujas dimensões dependem do particionamento. Este sistema pode ser resolvido de maneira similar às convencionais substituições "forward"/"backward":

$$\begin{bmatrix} I & \\ K & I \end{bmatrix} z_1 = b \qquad \text{(etapa 1)}$$

$$\begin{bmatrix} L_{11} & \\ & L_{22} \end{bmatrix} z_2 = z_1 \qquad \text{(etapa 2)}$$

$$\begin{bmatrix} U_{11} & \\ & U_{22} \end{bmatrix} z_3 = z_2 \qquad \text{(etapa 3)}$$



$z_4 = z_3$ (etapa 4)

Considerando-se cada etapa desta solução individualmente, pode-se observar o paralelismo existente. Na etapa 1 a parte superior do vetor z_1 é feita igual à parte superior de b, e a parte inferior é calculada através de substituição sucessiva das linhas de K. Esta substituição pode ser feita em paralelo, não existindo, desta forma, relações de precedência para obtenção de z_1. Na etapa 2 a parte superior e a inferior podem ser resolvidas independentemente uma da outra através de uma convencional "forward", sendo que a parte superior pode ser resolvida simultaneamente com a etapa 1, enquanto que a parte inferior precisa esperar pela solução completa da etapa 1. A etapa 3 é resolvida de forma similar a 2 e a etapa 4 similar a 1. Nesta metodologia pode-se variar o número de partições, e ainda propõe-se o uso de uma rede de PERT, para fazer a atribuição das tarefas paralelas. Os atrasos resultantes de uma implementação desta metodologia são citados em [37] (embora não se defina qual a arquitetura da máquina e o número de processadores), como sendo: "overhead" de algoritmo 1% a 25%, de comunicação 1% a 36% e de sincronização ("idle time") 1% a 92%. Nota-se que o "overhead" de comunicação não é tão limitante, e que o "overhead" de sincronização pode estabilizar o ganho.

A referência [15] propõe uma solução usando a matriz W (que contém os fatores inversos), transformando a equação (3.16) em:

1. $z = Wb$ - "forward" usando fatores inversos;
2. $y = D^{-1}z$ - divisão de z pela diagonal; e
3. $x = W^t y$ - "backward" usando fatores inversos.

A principal idéia desta abordagem é tentar eliminar as relações de precedência das operações elementares durante o processo de substituição. Nota-se que na "forward" usando W, todas as operações de multiplicação podem ser feitas independentemente uma das outras, mas infelizmente as operações de adição não podem ser feitas simultaneamente. Comparando esta solução com as soluções que usam as operações elementares como tarefa, percebe-se que as operações de multiplicação estão relacionadas com as relações de precedência e as operações de adição com as relações de *não-simultaneidade*. Desta forma usar a inversa da matriz L significa *quebrar* todas as relações de precedência,

permanecendo porém as relações de *não-simultaneidade*. O custo da *quebra* das relações de precedência é um trabalho extra, pois a matriz W não é tão esparsa quanto a matriz L. Estes novos elementos não-zeros são "fill-ins" adicionais que, sendo numerosos, podem contrabalançar o ganho obtido com a realização das multiplicações em paralelo [37]. A solução direta de $Ax = b$ pode ser obtida por:

$$x = W_1^t W_2^t \cdots W_n^t D^{-1} W_n \cdots W_2 W_1 b \tag{3.21}$$

onde W_i e W_i^t correspondem a todas as transformações elementares da coluna i na etapa "forward" e da linha i na "backward"respectivamente. Esta equação poderia ser processada em $2n$ etapas seriais (sem incluir a diagonal), entretanto é conveniente agrupar algumas matrizes W_i vizinhas de forma a ter poucas etapas sequenciais e ainda diminuir o número de "fill-ins" adicionais da matriz W. Este procedimento corresponde a particionar a matriz W em blocos. Por exemplo, dividindo-se W e W^t em três partições cada, tem-se:

$$x = W_a^t W_b^t W_c^t D^{-1} W_c W_b W_a b \tag{3.22}$$

A solução pode então ser obtida em 6 etapas seriais (não considerando a diagonal) partindo-se da direita para a esquerda, com todas as multiplicações dentro de uma mesma partição podendo ser feitas em paralelo [15], pois as posições do vetor b utilizadas para as multiplicações não são alteradas até o fim da partição. Entretanto, nem todas operações de adição dentro da mesma partição podem ser feitas em paralelo, porque atualizam a mesma posição do vetor b (elas não requerem relações de precedência, mas devem verificar relações de *não simultaneidade*).

Uma das preocupações ao particionar a matriz W é não gerar muitos "fill-ins" adicionais, o que pode comprometer o desempenho da solução. Neste sentido, a referência [4] propõe três esquemas de particionamento da matriz W que permitem controlar o aparecimento de "fill-ins" adicionais. Estes esquemas são baseados na profundidade dos nós no grafo de fatoração. Se os nós de mesma profundidade são ordenados em sequência, então ao agrupá-los numa partição p, não surgirão "fill-ins" adicionais nesta partição. Isto pode ser observado na matriz da Figura 3.2, onde ao se agrupar os nós 1 e 2 não é gerado nenhum novo elemento, pois $W_2 \times W_1$ gera o mesmo número de elementos não-zero que $L_1 \times L_2$. Porém, ao agrupar 9 e 10 nota-se que $W_{10} \times W_9$ gera cinco elementos não-zero abaixo da diagonal e $L_9 \times L_{10}$ gera quatro, surgindo portanto um "fill-in" adicional.

Os algoritmos de particionamento descritos em [4] têm como características:

- Algoritmo PA1 - nenhum "fill-in" adicional é permitido e nenhum cálculo extra é feito para obtenção das partições W. Faz-se o particionamento agrupando nós de uma mesma profundidade.

- Algoritmo PA2 - nenhum "fill-in" adicional é permitido e são necessárias algumas multiplicações entre W_i vizinhas para obter as partições W. Este algoritmo usa o mesmo critério de PA1, mais uma função para verificar se as posições onde surgiriam potenciais "fill-ins" já estão preeenchidas.
- Algoritmo PA3 - uma porcentagem de "fill-ins" adicionais é permitida por partição, sendo necessário multiplicações entre W_i para obter as partições W. Usa todos os critérios de PA2, e conta o número de "fill-ins" adicionais comparando-o com uma porcentagem pré-determinada.

Os dois primeiros algoritmos têm como vantagem o fato que não criam trabalho extra, mas geram muitas partições, significando muitos passos sequenciais e a maioria deles com pouco trabalho a ser realizado. O algoritmo PA3 pode gerar poucas partições se uma grande porcentagem de trabalho extra for permitida; mesmo assim algumas partições podem ser muito pequenas.

Uma implementação do uso de W particionada é utilizada em [17] para processamento vetorial, onde em cada partição explora-se o processamento vetorial para fazer as multiplicações, e em seguida, as adições são feitas com processamento escalar. Um algoritmo semelhante para processamento com multiprocessadores não traz muitos ganhos devidos principalmente ao "overhead" de algoritmo, que surge da necessidade de processar as adições depois das multiplicações. A referência [17] também apresenta um esquema de particionamento, buscando não gerar nenhum "fill-in" adicional. Este esquema é semelhante ao PA2, mas com um critério bastante simples para verificar se ao agrupar nós de profundidades diferentes não surgirão novos elementos. O critério consiste em verificar o grau de cada nó depois de uma ordenação MD. Percorrem-se os nós sequência de fatoração. Quando encontra-se um nó de grau n, e logo em seguida surge outro com grau $n-1$, iguala-se a profundidade deste outro nó à do anterior. Em seguida, faz-se uma reordenação dos nós no sentido da menor para a maior profundidade e usa-se em seguida o algoritmo PA1.

3.5 Metodologia Proposta

A metodologia proposta neste trabalho também usa a idéia de particionar W baseando-se na árvore dos caminhos de fatoração, mas o faz de uma maneira completamente diferente e com outros objetivos.

3.5.1 Esquema de Particionamento

O esquema de particionamento proposto neste trabalho consiste em particionar a matriz levando em consideração a profundidade dos nós, e busca *quebrar* relações de precedecência das pequenas partições (últimas), da seguinte forma:

- cada partição é associada a uma profundidade da árvore de fatoração, ou seja, a partição p corresponde a todos os nós de profundidade p;
- o particionamento prossegue até ser atingida uma profundidade escolhida (chamada *profundidade de corte*) e então todos os nós remanescentes são reunidos em uma única partição (chamada de *última partição*).

ALGORITMO DE PARTICIONAMENTO

1. Faça $i = 0$ e $p = 1$;
2. Defina a profundidade de corte P_{corte}
3. Agrupe os nós da profundidade i na partição p;
4. Faça $i = i + 1$ e $p = p + 1$
5. Verifique se i é diferente de P_{corte}
   (a) Se for volte ao passo 3;
   (b) Caso contrário agrupe todos os nós restantes em uma única partição W_{np}, e fim.

A escolha da profundidade de corte do passo 2, é heurística e não é crítica, pois o número de folhas por profundidade decresce rapidamente. Isto pode ser observado nas Tabelas 3.2, 3.3, 3.4 e 3.5, que considera diferentes critérios de ordenação. Os números que aparecem entre parenteses indicam a quantidade de elementos "fill-ins" gerados na matriz L pela respectiva ordenação.

Tabela 3.2 - Distribuição dos nós para várias ordenações para o sistema 118. (com 179 elementos originais não-zero na matriz L)

Prof.	MD (86)	MDML (86)	MDMNP (86)	MLMD (122)	MNPMD (127)
0	48	52	52	56	56
1	25	27	26	27	27
2	11	11	12	13	12
3	6	8	8	8	7
4	6	5	5	4	4
5	5	3	3	2	4
6	3	3	3	1	2
7	3	2	2	1	1
8	3	2	2	1	1
9	2	2	2	1	1
10	1	1	1	1	1
11	1	1	1	1	1
12	1	1	1	1	1
13	1			1	
14	1				
15	1				
16					
prof. max.	15	12	12	13	12

Tabela 3.3 - Distribuição dos nós para várias ordenações para o sistema 320. (com 470 elementos originais não-zero na matriz L)

Prof.	MD (417)	MDML (411)	MDMNP (411)	MLMD (578)	MNPMD (588)
0	140	153	153	161	161
1	51	57	56	65	62
2	30	32	33	30	29
3	18	16	17	17	18
4	17	11	11	10	11
5	9	9	8	7	7
6	9	5	5	6	5
7	7	3	3	4	4
8	5	3	3	2	3
9	2	2	2	2	2
10	2	2	2	2	2
11	2	2	2	1	1
12	2	2	2	1	1
13	2	2	2	1	1
14	2	2	2	1	1
15	2	2	2	1	1
16	2	2	2	1	1
17	1	2	2	1	1
18	1	2	1	1	1
19	1	1	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮
prof. max.	34	29	30	24	26

Tabela 3.4 - Distribuição dos nós para várias ordenações para o sistema 725. (com 935 elementos originais não-zero na matriz L)

Prof.	MD (562)	MDML (582)	MDMNP (573)	MLMD (879)	MNPMD (943)
0	339	361	361	402	402
1	135	148	147	152	146
2	71	72	72	68	64
3	43	34	36	31	31
4	31	26	25	19	20
5	19	15	15	11	11
6	11	10	10	7	8
7	9	8	8	6	6
8	9	7	8	4	4
9	8	6	6	3	4
10	6	4	4	2	3
11	5	3	3	2	3
12	4	2	3	2	3
13	3	2	2	2	3
14	2	2	2	1	2
15	2	2	2	1	1
16	2	2	2	1	1
17	2	2	2	1	1
18	2	2	2	1	1
19	2	2	2	1	1
20	2	2	2	1	1
21	2	2	2	1	1
22	2	2	1	1	1
23	2	1	1	1	1
24	1	1	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮
prof. max.	36	32	31	27	29

Tabela 3.5 - Distribuição dos nós para várias ordenações para o sistema 1729. (com 2154 elementos originais não-zero na matriz L)

Prof.	MD (1201)	MDML (1199)	MDMNP (1211)	MLMD (1732)	MNPMD (1905)
0	807	875	875	956	956
1	347	367	364	380	370
2	184	167	168	172	161
3	107	97	98	78	83
4	68	60	57	46	47
5	45	37	37	25	28
6	34	25	26	16	20
7	24	17	20	10	11
8	18	12	11	7	7
9	15	8	8	4	5
10	9	6	7	4	4
11	8	5	6	3	4
12	7	5	5	3	3
13	5	3	3	3	3
14	3	3	3	3	2
15	3	2	3	2	2
16	3	2	2	2	2
17	3	2	2	2	2
18	2	2	2	1	1
19	2	2	2	1	1
20	2	2	2	1	1
21	2	2	2	1	1
22	2	2	2	1	1
23	2	2	2	1	1
24	2	2	2	1	1
25	2	2	2	1	1
26	2	2	2	1	1
27	1	2	2	1	1
28	1	2	1	1	1
29	1	1	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮
prof. max.	47	42	41	30	38

Neste esquema é garantido que o aparecimento de "fill-ins" adicionais só poderá ocorrer na última partição, pois como já visto anteriormente reunir nós de uma mesma

profundidade em uma partição não gera nenhum novo elemento; já agrupar nós de diferentes profundidades pode acarretar novos elementos diferentes de zero.

No particionamento proposto o esquema de ordenação tem influência na medida que algumas técnicas fornecem árvores com menores profundidades máximas, o que significa mais nós por partição (principalmente nas primeiras). Estas árvores geram mais tarefas independentes e devem ser usadas sempre que possível. Algumas técnicas de ordenação não minimizam o aparecimento de "fill-ins", por exemplo, as ordenações que utilizam como primeiro critério de escolha a profundidade do nó (tipo ML-MD). As ordenações que usam como primeiro critério o grau do nó (MD, MD-MNP, MD-ML, etc) minimizam "fill-ins", mas geram menos nós nas primeiras profundidades. Para uma descrição detalhada dos principais esquemas de ordenação utilizados, veja o Apêndice C.

A influência de vários esquemas de ordenação na metodologia será discutida mais adiante na apresentação dos resultados e desempenho de algoritmos paralelos.

Ordenando a rede da Figura 3.2 através do esquema ML-MD, obtém-se o grafo de fatoração da Figura 3.6.

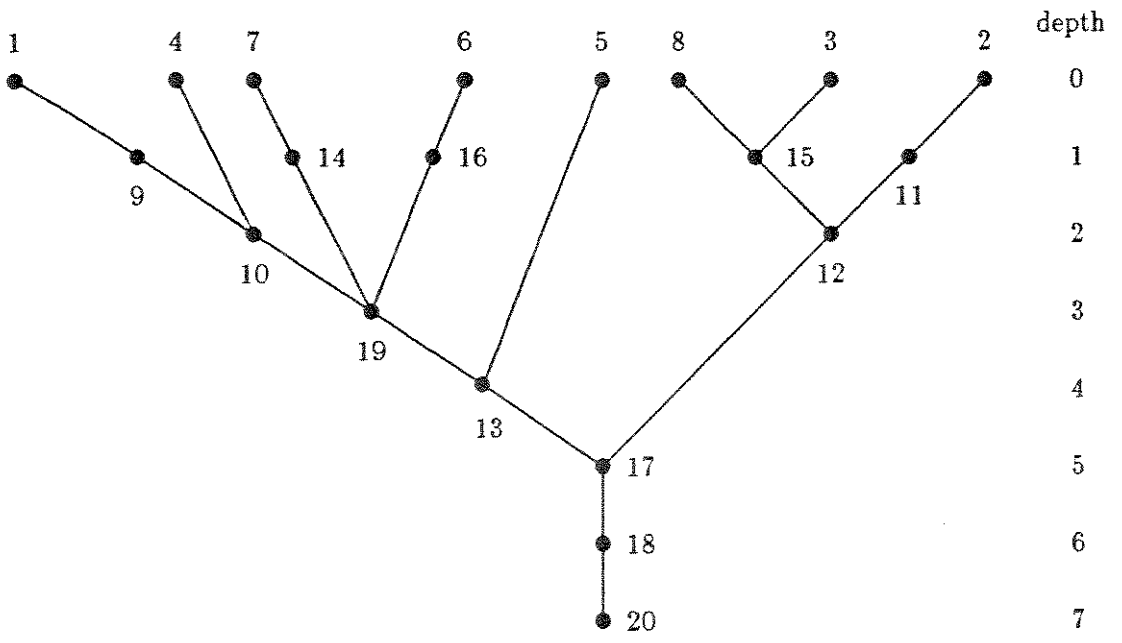


Figura 3.6 - Grafo de fatoração depois da ordenação ML-MD

A Figura 3.7 mostra a estrutura da matriz W para a rede 20 nós ordenada pelo esquema ML-MD, com 3 partições obtidas através do esquema de particionamento proposto.

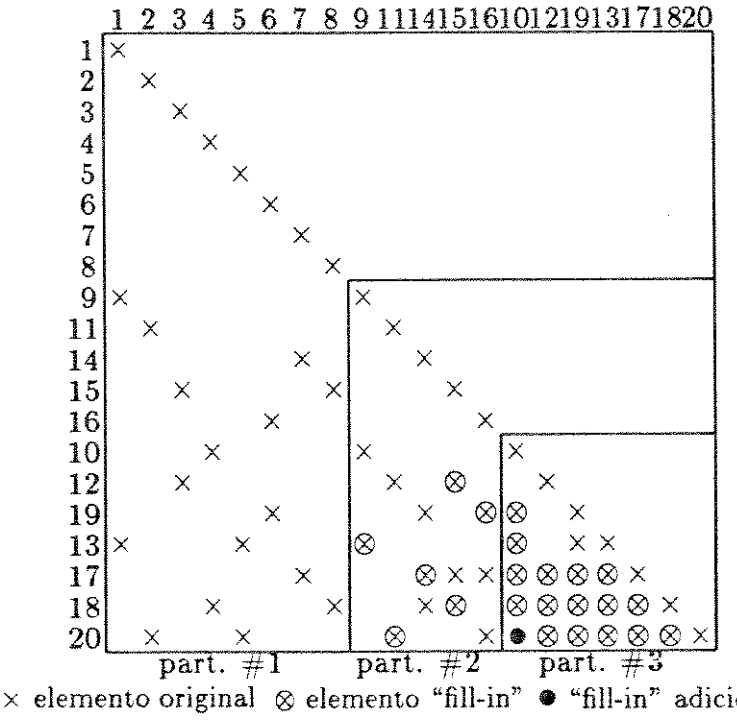


Figura 3.7 - Matriz W particionada

A última partição W_{np} pode ser calculada através de multiplicações das matrizes W_i vizinhas, correspondentes aos nós finais da árvore de fatoração (ver Apêndice A). Como esperado nas duas primeiras partições não surgiu nenhum novo elemento não-zero, enquanto que na última apareceu apenas um "fill-in" adicional.

Deve ser notado que todas as informações necessárias para o particionamento podem ser obtidas da árvore de fatoração da rede, e que somente a última partição requer cálculos extras para sua obtenção.

3.5.2 Tarefas Independentes e Granularidade

Um algoritmo paralelo que use particionamento da matriz W e tenha como grão de cada tarefa as operações elementares, necessita checar relações de *não simultaneidade* para atualizar as respectivas posições do vetor b. Isto implica que dentro de uma mesma partição, as tarefas não são totalmente independentes.

Verificando as características das operações de atualização do vetor b e a forma como tradicionalmente são feitas a "forward" e a "backward", nota-se que a "forward" é processada por coluna atualizando os elementos de b correspondentes às linhas dos elementos da matriz W; portanto, quando as colunas de uma mesma partição vão ser processadas paralelamente pode haver conflito na atualização do vetor b. A "backward" é processada por linha e as operações dos elementos de W^t sobre o vetor b atualizam

a mesma posição de b, surgindo conflito se os elementos de uma mesma linha forem operados por processadores diferentes (o grão da tarefa continua sendo a operação elementar). Porém não surgirá nenhum problema se em uma mesma partição, cada linha for atribuída a um processador como uma tarefa indivisível. Isto faz com que todas as tarefas sejam totalmente independentes dentro de cada partição durante a "backward".

Pensando em fazer todo o processo de solução direta com tarefas independentes dentro de cada partição, este trabalho propõe que a forma tradicional de se fazer a "forward" seja alterada. Aqui a "forward" será realizada por linha, da mesma forma que a "backward". Com isso é possivel alterar o grão de cada tarefa, que antes era uma operação elementar, para o conjunto de todas as operações elementares de uma mesma linha, dentro da mesma partição.

Realizando-se a solução direta por linhas e por partições, com todas as operações de uma linha na partição sendo uma tarefa indivisível, obtém-se uma solução onde todas as tarefas são independentes dentro de uma mesma partição. Assim a "forward" paralela pode ser obtida em alguns passos seriais, ou mesmo em um único passo, se o número de "fill-ins" adicionais não comprometer o desempenho.

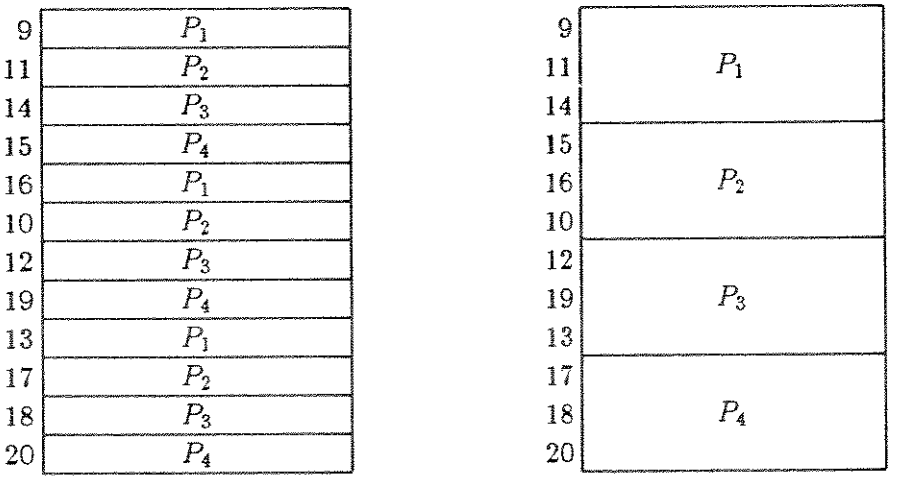
A Tabela 3.6 descreve as tarefas na "forward" para a primeira partição da matriz W da Figura 3.7, mostrando a independência das tarefas dentro de uma partição. As 12 tarefas da Tabela 3.6 correspondem às 16 primeiras tarefas da "forward" convencional da Tabela 3.1.

Tabela 3.6 - Operações na substituição "forward" por linha para a primeira partição de W da Figura 3.7.

Tarefa	Operação
# 1	$z_9 = z_9 + w_{9,1}z_1$
# 2	$z_{11} = z_{11} + w_{11,2}z_2$
# 3	$z_{14} = z_{14} + w_{14,7}z_7$
# 4	$z_{15} = z_{15} + w_{15,3}z_3 + w_{15,8}z_8$
# 5	$z_{16} = z_{16} + w_{16,6}z_6$
# 6	$z_{10} = z_{10} + w_{10,4}z_4$
# 7	$z_{12} = z_{12} + w_{12,3}z_3$
# 8	$z_{19} = z_{19} + w_{19,6}z_6$
# 9	$z_{13} = z_{13} + w_{13,1}z_1 + w_{13,5}z_5$
# 10	$z_{17} = z_{17} + w_{17,7}z_7$
# 11	$z_{18} = z_{18} + w_{18,4}z_4 + w_{18,8}z_8$
# 12	$z_{20} = z_{20} + w_{20,2}z_2 + w_{20,5}z_5$

A atribuição de tarefas para diversos processadores pode ser feita de várias maneiras,

e em função disto pode-se ter diferentes tamanhos para as tarefas, o que leva a diferentes granularidade do processamento. Quando o número disponível de processadores para fazer uma partição da "forward" é maior ou igual ao número de linhas a serem processadas na partição então o tamanho de cada tarefa corresponde às operações de cada linha e desse modo tem-se uma granularidade mais fixa. Os tamanhos de cada tarefa não serão todos iguais, pois o número de elementos por linha na "forward" varia de um a quatro para o sistema IEEE-118, e de um a dez para o sistema Sul-sudeste brasileiro 1729, conforme indicam as Talelas 4.1 e 4.4. Quando o número de processadores é menor que o número de linhas (fato que geralmente acontece), o tamanho de cada tarefa passa a ser igual ao numero de operações das linhas que são atribuídas a cada processador, e então o balanceamento de carga passa a depender não somente do número de elementos em cada linha, mas também de quais linhas são atribuídas aos processadores. Neste caso tem-se mais flexibilidade para balancear a carga, dependendo da sofisticação que se queira dar ao algoritmo de divisão de tarefas. A referência [25] propõe algumas formas de divisão das linhas pelos processadores, aqui serão analisados dois esquemas: um esquema de linhas alternadas (semelhante à distribuição de cartas no jogo de baralho) e um esquema por blocos, como mostra a Figura 3.9, onde supõe-se quatro processadores.



Esquema de linhas alternadas

Esquema por blocos.

Figura 3.9 - Esquemas de atribuição de tarefas.

Observando-se os "mapas topológicos" (figuras de matrizes indicando os elementos diferentes de zero) de várias matrizes mostradas em [15] e [4], nota-se que as últimas linhas da matriz tendem a apresentar mais elementos não-nulos, significando tarefas maiores. Isto indica que um esquema por blocos pode levar a um desbalanceamento significativo. O esquema por linhas alternadas tende a ser mais balanceado, já que, como

o próprio nome diz, as linhas são atribuídas alternadamente. Esta distribuição oferece facilidades também em termos de programação, uma vez que atribuindo-se um número (npro) a cada processador $(1, 2, 3, \ldots, n)$ e informando-se o número de processadores presentes na solução ($ntot = n$), cada processador localiza facilmente a primeira linha e dá um passo igual a $nproc$ para localizar a próxima linha, e assim por diante.

Um algoritmo para a "forward" por linha com particionamento de W em np partições, e para um computador com $ntot$ processadores é o seguinte:

ALGORITMO F2

1. Faça $npro$ = número do processador;
2. Faça $ini = npro - ntot$, $p = 0$ e $i = 0$;
3. Faça $p = p + 1$ e $nell$ = número de nós da partição p;
   (a) Se $p > np$ fim;
   (b) Caso contrário segue;
4. Faça $ini = ini + nell$;
5. Faça $i = i + ini + ntot$;
   (a) Se $i > nb$ vá para 6;
   (b) Caso contrário segue;
6. Verifique se existem não zero na linha i na partição p;
   (a) Se não existem, vá para 4;
   (b) Caso contrário localize a coluna k e faça $z_i = z_i + w_{i,k} b_k$ e volte ao passo 5;
7. Faça $b = z$ (ponto de sincronização) e volte ao passo 3;

O número máximo de processadores que pode ser utilizado na "forward" é igual ao número de linhas de W a serem processadas na primeira partição. Este número é a diferença entre o número total de nós e o número de nós da profundidade zero.

Um algoritmo, nas mesmas condições, para a "backward" é o seguinte:

ALGORITMO B2

1. Faça $npro$ = número do processador;
2. Faça $p = np + 1$, $i = nb - npro + ntot$ e $ifim = nb$;

3. Faça $p = p - 1$ e $nell$ = número de nós da partição p;

   (a) Se $p < 1$ fim;

   (b) Caso contrário segue;

4. Faça $ifim = ifim - nell$;

5. Faça $i = i - ntot$;

   (a) Se $i \leq ifim$ vá para 6;

   (b) Caso contrário segue;

6. Verifique se existem não-zero na linha i;

   (a) Se não existem, vá para 4;

   (b) Caso contrário faça $x_i = x_i + w_{i,k}y_k$, e volte ao passo 5;

7. Faça $y = x$ (ponto de sincronização) e volte ao passo 2;

O número máximo de processadores que pode ser utilizado na "backward" é igual ao número de linhas de W^t a serem processadas na primeira partição. Este número é exatamente o número de nós da profundidade zero. Então o número suficiente de processadores para a solução direta, é o maior valor entre o máximo requerido pela "forward" e o máximo requerido pela "backward".

Quando se faz a "forward" e a "backward" por linhas, é necessário um duplo conjunto de apontadores para indicar linhas e colunas das matrizes triangulares superior e inferior, conforma mostrada no Apêndice D.

Neste tipo de solução, usando-se poucas partições, as últimas linhas da última partição apresentam maiores números de elementos por linha, e em particular a última linha que é sempre cheia. Assim propõe-se mudar a forma de resolver a última partição agrupando-se, para esta partição, as etapas "forward", diagonal e "backward" em uma única etapa de processamento.

3.5.3 Última partição

Quando se faz o particionamento da matriz W através do algoritmo proposto, todas as partições contêm nós de uma mesma profundidade, exceto a última que contém nós de diversas profundades porque buscou-se propositadamente *quebrar* relações de precedência. Esta eliminação das relações de precedência tem como contrapartida a necessidade de calcular os fatores inversos, podendo aparecer "fill-ins" adicionais que, em princípio, podem levar a uma solução mais demorada. Será mostrado mais adiante

que o empacotamento dos nós na última partição não atrasa a solução.

Supondo que as duas primeiras partições da matriz W da Figura 3.7 já tenham sido processadas, pode-se, por facilidade de representação, apresentar a última partição W_{3p} como a matriz da Figura 3.10, onde mostra-se também um grafo de fatoração dos nós referentes a esta partição.

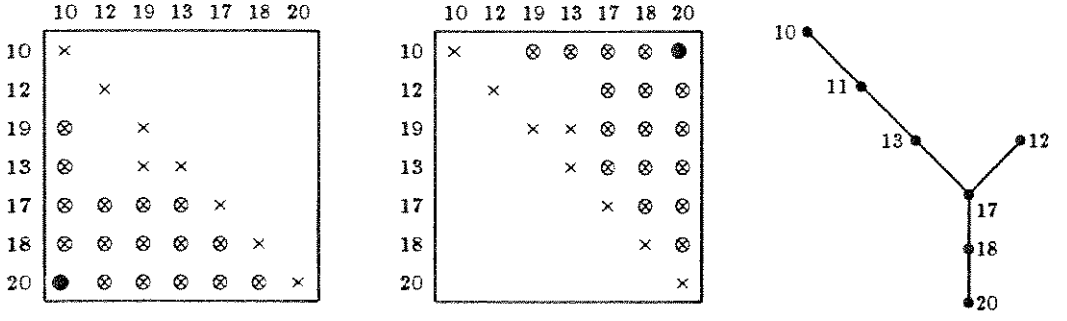


Figura 3.10 - (a) Partição W_{3p}, (b) Partição W^t_{3p}
(c) Grafo de fatoração para os nós da partição 3

Uma coluna k desta matriz W_{np} (L^{-1}_{np}) contém elementos não-zero nas linhas correspondentes aos nós sucessores ao nó k no grafo de fatoração. A linha k contém elementos não-zero nas colunas correspondentes aos nós predecessores ao nó k. Esta informação indica que a última linha de uma última partição W de uma rede elétrica qualquer não ilhada será sempre cheia, pois o último nó do grafo possui todos os demais como predecessores. Isto significa que ao realizar-se a etapa "forward" em paralelo tendo uma linha como tarefa independente, a maior tarefa na última partição é obviamente a última linha.

Assim, no uso da rede exemplo para se fazer a etapa "forward", usando um número de processadores igual ao número de linhas de W_{np}, o tempo de solução mínimo é o tempo necessário para processar a linha 20. Para fazer a "backward" este tempo mínimo é o da linha 10 de W^t_{np}.

Contando este tempo em operações (multiplicações/adições), apenas para a "forward" + "backward", tem-se:

$$t_{wn} = 6 + 5 = 11 \qquad \text{operações}$$

Sabendo que cada tarefa corresponde a todas as operações de uma linha, e que a maior linha do triângulo inferior é cheia, pode-se então alterar a maneira de resolver a última partição, fazendo-se:

$$W_{up} = W_{np}^t D^{-1} W_{np} \tag{3.23}$$

A matriz W_{up} contém os valores de D^{-1} nas posições referentes aos nós das $(n-1)$ partições, e exatamente os elementos da inversa da última partição (A_n^{-1}), onde aparecem novos elementos não-zero (a matriz é cheia). Desta forma pode-se resolver esta partição em vez de fazer a etapa de divisão pela diagonal, ou seja:

$$y = W_{up} z \tag{3.24}$$

Usando-se agora um número suficiente de processadores para resolver esta etapa, o tempo mínimo de solução é:

$$t_{wup} = 7 \qquad \text{operações}$$

Portanto, apesar dos novos elementos $(A^{-1} fill-ins)$, este tempo é inferior ao tempo t_{wn} que não inclui a etapa da diagonal. Este tempo $t_{wup} = 7$ é também inferior ao menor tempo que seria necessário para fazer em paralelo a "forward" e a "backward" usando as matrizes L_{np} e L_{np}^t, ou seja, a soma do número de elementos diferentes de zero da linha mais cheia de L_{np} mais a soma dos não-zero da linha mais cheia de L_{np}^t, esta soma é:

$$t_{ln} = 9 \qquad \text{operações}$$

Então a solução usando W_{up} (apesar dos $W fill-ins$ e dos $A^{-1} fill-ins$) torna-se mais eficiente, e com a vantagem extra de que todas as tarefas podem ser executadas em paralelo sem necessidade de testes de relações de precedência, e em apenas um passo sequencial.

Usando a idéia explicada acima, o processo de solução fica:

$$z = W_{(n-1)p} \cdots W_{2p} W_{1p} b \tag{3.25}$$
$$y = W_{up} z \tag{3.26}$$
$$x = W_{1p}^t W_{2p}^t \cdots W_{(n-1)p}^t y \tag{3.27}$$

Pode-se então construir algoritmos semelhantes a **F2** e **B2** para a solução que utilize a resolução da última partição em um passo único via W_{up}. No Apêndice D mostram-se, com detalhes, como é feito o armazenamento e os diagramas de blocos para as soluções sequencial e paralela, usando ou não W_{up}.

3.5.4 Obtenção do Número Ótimo de Partições

O particionamento proposto anteriormente é simples, porém a forma de escolha da *profundidade de corte* é heurística, ou seja, quando o número de nós com mesma profundidade começa a ficar pequeno deve-se colocar os nós restantes na última partição. A questão abordada neste item é: dado um sistema e um determinado número de processadores, qual é o número ótimo de partições?

Entende-se por número ótimo de partições aquele que leva a uma solução com menor número de operações de ponto flutuante, não levando-se em conta os possíveis atrasos de paradas em pontos de sincronização.

Para achar o número ótimo de partições é necessário descobrir a maior tarefa em cada partição, contando o número de operações elementares que cada processador faz. Assim o esquema de particionamento torna-se (lembrando que última partição é cheia):

1. Faça $p = 0$. Agrupe os nós da profundidade p em uma partição p, faça nu_p igual ao número de operações elementares da maior tarefa da última partição (que reune os nós restantes);
2. Faça $p = p + 1$ e agrupe os nós desta profundidade em uma partição. Calcule o número de operações elementares (nof) da maior tarefa da "forward", o número de operações elementares (nob) da maior tarefa da "backward" e faça nu_{p+1} igual ao número de operações da maior tarefa da última partição (que agora reune os nós que ainda não pertencem a nenhuma partição);
3. Faça $nu = nof + nu_{p+1} + nob$;
4. Verifique se nu é menor que nu_p
   (a) Se for, faça $nu_p = nu_{p+1}$ e volte ao passo 2;
   (b) Caso contrário agrupe os nós da profundidade p com os restantes em uma única partição W_up, e fim.

3.5.5 Síntese da Metodologia

A metodologia apresentada neste trabalho é baseada em uma solução utilizando particionamento da matriz de fatores inversos W, sendo agora realizada em passos sequenciais de acordo com o número de partições. O esquema de particionamento é feito com base na profundidade dos nós da árvore de fatoração, sendo que no ínicio do particionamento todos os nós de uma mesma profundidade são agrupados em uma mesma partição. Quando as partições passam a possuir poucos nós, agrupam-se todos os nós restantes em uma última partição, *quebrando-se* assim, as relações de precedência entre estes nós.

As etapas "forward" e "backward" são feitas por linhas e por partições, atribuindo-se a um processador todas as operações de multiplicação-adição de uma linha dentro da mesma partição. Conseguindo-se desta forma que todas as tarefas fiquem independentes dentro de uma partição. Esta estratégia pode ser aplicada à todas as partições, no entanto para a última partição pode ser mais vantajoso agrupas as etapas "forward", diagonal e "backward" em uma única, obtendo-se a matriz W_{up}. Este procedimento é analisado no capítulo seguinte.

Capítulo 4

Análise de Desempenho

4.1 Introdução

Neste capítulo busca-se inicialmente testar a metodologia de decomposição do problema, fazendo-se para isto simulações do algoritmo paralelo em um computador uniprocessador. Nestas simulações utiliza-se o número de operações multiplicação-adição como parâmetro de medida, não levando-se em conta as operações com os elementos da diagonal, que são operações que podem ser realizados simultaneamente. Esta análise da decomposição permitirá discutir a influência das ordenações e da forma de resolver a última partição na metodologia proposta.

Em seguida são apresentados os resultados de uma implementação da metodologia proposta numa máquina paralela de arquitetura híbrida. Aqui serão considerados todos os passos da solução direta, inclusive a etapa das operações com a diagonal. Usou-se nas simulações e implementação quatro diferentes sistemas de energia elétrica, cujas características são as seguintes:

- IEEE 118 nós - com 179 elementos originais diferentes de zero fora da diagonal na matriz L.
- Sul-Sudeste 320 nós - com 470 elementos originais diferentes de zero fora da diagonal na matriz L.
- Sul-Sudeste 725 nós - com 935 elementos originais diferentes de zero fora da diagonal na matriz L.
- Sul-Sudeste 1729 nós - com 2154 elementos originais diferentes de zero fora da diagonal na matriz L.

4.2 Análise da Decomposição do Problema

A metodologia proposta obtém a solução em poucos passos sequenciais, através de um particionamento da matriz W baseado nos nós de mesma profundidade na árvore de

fatoração. Esta forma de particionamento poderá gerar "fill-ins" adicionais somente na última partição, o que significa trabalho adicional. A primeira vista estes "fill-ins" podem atrasar a solução, mas deve-se lembrar que o seu aparececimento é causado pela *quebra* de relações de precedência na última partição. Deste modo a primeira análise a ser feita é ver a influência deste trabalho extra na solução paralela, para dimensionar até quanto é vantajosa a quebra de relações de precedência. Posteriormente são analisadas as influências do tipo de ordenação e do tratamento da última partição.

4.2.1 Desempenho *versus* "fill-ins" adicionais

A Tabela 4.1 mostra o número máximo de operações multiplicação-adição em cada passo sequencial em função do número de partições, assumindo que se tenha um número suficiente de processadores. Número suficiente de processadores, neste caso, significa dispor de processadores em quantidade suficiente para atribuir cada linha da partição a um processador. Os números entre parênteses indicam os "fill-ins" adicionais no triângulo inferior da última partição.

Tabela 4.1 - Sistema IEEE 118 (ordenação ML-MD).

Passos seriais	Partições					
	2 (1704)	3 (471)	4 (147)	5 (34)	6 (7)	7 (0)
1 (W_1)	4	4	4	4	4	4
2 (W_2)	–	4	4	4	4	4
3 (W_3)	–	–	4	4	4	4
4 (W_4)	–	–	–	3	3	3
5 (W_5)	–	–	–	–	2	2
6 (W_6)	–	–	–	–	–	2
7 (W_{up})	62	35	22	14	10	8
8 (W_6^t)	–	–	–	–	–	8
9 (W_5^t)	–	–	–	–	9	9
10 (W_4^t)	–	–	–	6	6	6
11 (W_3^t)	–	–	6	6	6	6
12 (W_2^t)	–	3	3	3	3	3
13 (W_1^t)	3	3	3	3	3	3
total	69	49	46	47	54	62

Observando-se a Tabela 4.1 nota-se que a existência de "fill-ins" adicionais não atrapalham o desempenho global da solução proporcionando melhores resultados que particionamentos que não produzem nenhum "fill-in" adicional. Vê-se, por exemplo, que a solução com 7 partições não gera nenhum "fill-in" adicional, mas no entanto

a solução requer 62 operações de multiplicação-adição, enquanto uma solução com 4 partições, tendo 147 "fill-ins" adicionais, requer apenas 46 operações. Estes resultados foram obtidos supondo-se que as etapas "forward", diagonal e "backward" correspondente à última partição são realizadas em um único passo ($W_{up} = W_n^t D^{-1} W_n$). Portanto o número ótimo de partições, citado no item 3.5.4, para este caso é quatro.

Comportamento semelhante é obtido na simulação dos demais sistemas conforme mostram as Tabelas 4.2, 4.3 e 4.4.

Tabela 4.2 - Sistema Sul-Sudeste 320 (ordenação ML-MD).

Passos seriais	Partições					
	3 (3809)	4 (1573)	5 (717)	6 (364)	7 (183)	8 (64)
1 (W_1)	5	5	5	5	5	5
2 (W_2)	5	5	5	5	5	5
3 (W_3)	–	4	4	4	4	4
4 (W_4)	–	–	3	3	3	3
5 (W_5)	–	–	–	3	3	3
6 (W_6)	–	–	–	–	3	3
7 (W_7)	–	–	–	–	–	3
8 (W_{up})	94	64	47	37	30	24
9 (W_7^t)	–	–	–	–	–	10
10 (W_6^t)	–	–	–	–	12	12
11 (W_5^t)	–	–	–	11	11	11
12 (W_4^t)	–	–	10	10	10	10
13 (W_3^t)	–	9	9	9	9	9
14 (W_2^t)	8	8	8	8	8	8
15 (W_1^t)	7	7	7	7	7	7
total	119	102	98	102	110	117

Tabela 4.3 - Sistema Sul-Sudeste 735 (ordenação ML-MD).

Passos seriais	Partições					
	4 (4598)	5 (2028)	6 (946)	7 (504)	8 (295)	9 (164)
1 (W_1)	9	9	9	9	9	9
2 (W_2)	5	5	5	5	5	5
3 (W_3)	4	4	4	4	4	4
4 (W_4)	–	4	4	4	4	4
5 (W_5)	–	–	4	4	4	4
6 (W_6)	–	–	–	3	3	3
7 (W_7)	–	–	–	–	3	3
8 (W_8)	–	–	–	–	–	3
9 (W_{up})	103	72	53	42	35	29
10 (W_8^t)	–	–	–	–	–	19
11 (W_7^t)	–	–	–	–	14	14
12 (W_6^t)	–	–	–	–	13	13
13 (W_5^t)	–	–	10	10	10	10
14 (W_4^t)	–	10	10	10	10	10
15 (W_3^t)	10	10	10	10	10	10
16 (W_2^t)	11	11	11	11	11	11
17 (W_1^t)	8	8	8	8	8	8
total	151	134	129	134	145	160

Tabela 4.4 - Sistema Sul-sudeste 1729 (ordenação ML-MD).

Passos seriais	Partições					
	4 (23088)	5 (9171)	6 (3858)	7 (1886)	8 (981)	9 (570)
1 (W_1)	9	9	9	9	9	9
2 (W_2)	7	7	7	7	7	7
3 (W_3)	5	5	5	5	5	5
4 (W_4)	–	4	4	4	4	4
5 (W_5)	–	–	4	4	4	4
6 (W_6)	–	–	–	4	4	4
7 (W_7)	–	–	–	–	4	4
8 (W_8)	–	–	–	–	–	3
9 (W_{up})	221	143	97	72	56	46
10 (W_8^t)	–	–	–	–	–	17
11 (W_7^t)	–	–	–	–	15	15
12 (W_6^t)	–	–	–	10	10	10
13 (W_5^t)	–	–	10	10	10	10
14 (W_4^t)	–	9	9	9	9	9
15 (W_3^t)	7	7	7	7	7	7
16 (W_2^t)	7	7	7	7	7	7
17 (W_1^t)	6	6	6	6	6	6
total	262	197	165	154	157	167

Como se vê existe um número ótimo de partições para cada rede a fim de realizar a solução mais rápida. Este número não corresponde ao número de partições que gera menos "fill-ins", pois partições com poucos nós entre as primeiras (que são partições grandes) e a última podem levar a soluções mais lentas. O número ótimo de partições pode ser obtido da tabela do número de nós por profundidade, levando-se em conta que a última partição será uma matriz cheia.

4.2.2 Ganho *versus* Ordenação

Para uma análise do "speedup" da métodologia em função do número de processadores, definem-se os seguintes parâmetros:

$$G = \frac{t_s}{t_p} \tag{4.1}$$

$$E = \frac{G}{ntot} \tag{4.2}$$

onde G é o ganho e significa quanto a solução paralela é mais rápida que a sequencial; t_s é o total de operações multiplicação-adição na solução sequencial; t_p é a soma total das operações multiplicação-adição realizadas na solução paralela pelo processador que é faz a tarefa maior; E é a eficiência e $ntot$ é o número total de processadores utilizados no processo. Para efeito de comparação a solução sequencial usada é aquela que reconhecidamente apresenta melhor desempenho para um processador (considerando b e x vetores não-esparsos), ou seja, usa-se ordenação MD (que minimiza "fill-ins") e fatores triangulares obtidos diretamente das matrizes L, D e U (não utiliza-se matriz W).

Neste tipo de análise não são levados em conta os tempos de comunicação/sincronismo entre processadores para cada tipo de máquina (memória partilhada, troca de mensagens ou híbrida). A comunicação/sincronismo envolvida em cada uma delas é discutida no Apêndice D, que mostra uma forma de fazer esta programação paralela. O ganho em termos do número de operações dá uma idéia do máximo "speedup" teórico que a metodologia permite.

Os resultados apresentados a seguir foram obtidos fazendo-se a "forward", diagonal e "backward" para a 'ultima partição em um único passo, ou seja, calculou-se a matriz W_{up}.

Ordenação ML-MD (calculando-se W_up)

As Tabelas 4.5, 4.6, 4.7 e 4.8 mostram o ganho G e a eficiência E para os quatro sistemas testes, com o particionamento que fornece o menor t_p (número de partições obtido de tabelas como as do ítém 4.2.1) em função do número de processadores. Nestas tabelas Suf significa um número suficiente de processadores e entre parênteses aparece qual deve ser este valor para cada sistema. Usou-se a ordenação ML-MD.

Tabela 4.5 - Ganho no sistema 118 ($t_s = 530$)

Processadores	Partições	t_p	G	E
Suf (62)	4	46	11,52	0,186
64	4	46	11,52	0,180
32	4	51	10,39	0,325
16	5	65	8,15	0,509
8	7	109	4,86	0,607
4	7	182	2,91	0,727

Tabela 4.6 - Ganho no sistema 320 ($t_s = 1774$)

Processadores	Partições	t_p	G	E
Suf (161)	5	98	18,10	0,114
64	5	108	16,42	0,256
32	7	141	12,58	0,393
16	7	215	8,25	0,515
8	8	337	5,20	0,650
4	9	592	2,99	0,747

Tabela 4.7 - Ganho no sistema 725 ($t_s = 2994$)

Processadores	Partições	t_p	G	E
Suf (402)	6	129	23,21	0,058
64	6	155	19,32	0,302
32	7	245	12,22	0,382
16	9	347	8,63	0,539
8	11	590	5,07	0,637
4	12	1048	2,86	0,715

Tabela 4.8 - Ganho no sistema 1729 ($t_s = 6710$)

Processadores	Partições	t_p	G	E
Suf (956)	7	154	43,57	0,045
64	8	187	27,16	0,424
32	8	402	16,69	0,521
16	9	658	10,20	0,637
8	12	1137	5,90	0,737
4	13	2110	3,18	0,795

O ganho obtido com muitos processadores é maior para as redes maiores porque o número de tarefas independentes aumenta significativamente com o tamanho da rede elétrica. Porém deve-se dispor do número de processadores necessários para explorar este paralelismo.

Das Tabelas 4.5 a 4.8 nota-se que o melhor número de partições para muitos processadores não é o melhor número para poucos processadores, e este comportamento pode

estar ligado não somente aos "fill-ins" adicionais da última partição, mas também aos critérios de ordenação e à forma de resolver a última patição.

Ordenação MD-MNP (calculando-se W_up)

Os testes feitos até aqui levaram em conta a ordenação ML-MD, que fornece mais nós nas primeiras partições sem preocupação com a minimização de "fill-ins" na matriz L. Agora passa-se a analisar a influência que diferentes ordenações possam ter no ganho. Observando as Tabela 3.2 a 3.5 nota-se que as ordenações que utilizam como primeiro critério de escolha o grau de cada nó minimizam o aparecimento de "fill-ins" na matriz L, com pequenas variações entre elas. Compare por exemplo os "fill-ins" das ordenações MD e MD-MNP. Já do ponto de vista da profundidade da árvore de fatoração, a ordenação MD-MNP oferece vantagens (mais nós por profundidade significa mais tarefas independentes por partição). Desta forma optou-se por apresentar resultados com esta ordenação. Nas Tabelas 4.9, 4.10, 4.11 e 4.12 são mostrados os ganhos e as eficiências com a ordenação MD-MNP calculando-se W_{up}.

Tabela 4.9 - Ganho no sistema 118 ($t_s = 530$)
(ordenação MD-MNP calculando-se W_{up})

Processadores	Partições	t_p	G	E
Suf (66)	5	46	11,52	0,174
64	5	46	11,52	0,180
32	5	50	10,60	0,331
16	6	63	8,41	0,556
8	9	98	5,41	0,676
4	9	161	3,29	0,822

Tabela 4.10 - Ganho no sistema 320 ($t_s = 1774$)
(ordenação MD-MNP calculando-se W_{up})

Processadores	Partições	t_p	G	E
Suf (167)	5	98	18,10	0,108
64	6	106	16,73	0,261
32	10	144	12,32	0,385
16	10	213	8,33	0,521
8	12	355	4,99	0,624
4	18	551	3,22	0,805

Tabela 4.11 - Ganho no sistema 725 ($t_s = 2994$)
(ordenação MD-MNP calculando-se W_{up})

Processadores	Partições	t_p	G	E
Suf (364)	7	129	23,21	0,064
64	8	159	18,83	0,294
32	11	243	12,32	0,385
16	13	323	9,27	0,579
8	16	521	5,74	0,717
4	20	885	3,38	0,845

Tabela 4.12 - Ganho no sistema 1729 ($t_s = 6710$)
(ordenação MD-MNP calculando-se W_{up})

Processadores	Partições	t_p	G	E
Suf (875)	10	169	39,70	0,045
64	12	255	26,31	0,411
32	12	401	16,73	0,523
16	14	639	10,50	0,656
8	24	1057	6,35	0,794
4	26	1872	3,58	0,895

Destas Tabelas pode-se observar que os "fill-ins" gerados pela ordenação ML-MD na matriz L, atrapalham o desempenho quando o número de processadores utilizados não é grande. Passa-se agora a analisar a influência da forma de resolver a última partição.

4.2.3 Influência do Tratamento da Última Partição

Ordenação ML-MD (sem usar W_{up})

Relembrando que o ganho mostrado nas Tabelas 4.5 a 4.12, foi obtido resolvendo-se a "forward" + diagonal + "backward" da última partição em um único passo ($W_{up} = W_n^t D^{-1} W_n$), propõe-se agora, a fim de verificar o desempenho desta técnica, resolver o problema sem obter W_{up}, ou seja, faz-se a "forward" e a "backward" normais para a última partição como é feito para as demais. Assim obtém-se as Tabelas 4.13, 4.14, 4.15 e 4.16 para os diferentes sistemas de energia elétrica, ordenados com a técnica ML-MD.

Tabela 4.13 - Ganho no sistema 118 ($t_s = 530$)
(sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (62)	4	55	9,64	0,155
64	4	55	9,64	0,151
32	4	60	8,83	0,276
16	5	73	7,26	0,454
8	6	108	4,91	0,614
4	6	181	2,93	0,732

Tabela 4.14 - Ganho no sistema 320 ($t_s = 1774$)
(sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (161)	5	117	15,16	0,094
64	5	127	13,97	0,0218
32	6	155	11,44	0,357
16	8	219	8,10	0,506
8	8	343	5,17	0,646
4	9	596	2,98	0,745

Tabela 4.15 - Ganho no sistema 725 ($t_s = 2994$)
(sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (402)	6	160	18,71	0,046
64	6	176	17,01	0,266
32	7	233	12,85	0,401
16	8	344	8,70	0,544
8	9	584	5,13	0,641
4	11	1048	2,86	0,715

Tabela 4.16 - Ganho no sistema 1729 ($t_s = 6710$)
(sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (956)	7	177	37,91	0,040
64	8	269	24,94	0,390
32	8	392	17,12	0,535
16	10	630	10,65	0,666
8	10	1120	5,99	0,749
4	13	2089	3,02	0,755

Ordenação MD-MNP (sem usar W_{up})

Nas Tabelas 4.17, 4.18, 4.29 e 4.20 são mostrados os ganhos com a ordenação MD-MNP resolvendo-se a última partição normalmente.

Tabela 4.17 - Ganho no sistema 118 ($t_s = 530$)
(ordenação MD-MNP sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (66)	6	52	10,19	0,154
64	6	52	10,19	0,159
32	6	56	9,46	0,297
16	6	69	7,68	0,480
8	8	99	5,35	0,669
4	8	159	3,33	0,832

Tabela 4.18 - Ganho no sistema 320 ($t_s = 1774$)
(ordenação MD-MNP sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (167)	6	121	14,66	0,088
64	6	130	13,65	0,213
32	9	159	11.16	0,349
16	9	216	8,21	0,513
8	13	339	5,23	0,654
4	16	549	3,23	0,807

Tabela 4.19 - Ganho no sistema 725 ($t_s = 2994$)
(ordenação MD-MNP sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (364)	8	152	19,70	0,054
64	8	182	16,45	0,257
32	11	229	13,07	0,408
16	12	325	9,21	0,288
8	15	516	5,80	0,725
4	18	878	3,41	0,852

Tabela 4.20 - Ganho no sistema 1729 ($t_s = 6710$)
(ordenação MD-MNP sem calcular W_{up})

Processadores	Partições	t_p	G	E
Suf (875)	10	200	33,55	0,038
64	11	282	23,79	0,372
32	12	401	16,73	0,523
16	17	624	10,75	0,672
8	23	10656	6,30	0,787
4	25	1869	3,59	0,897

Comparando os resultados das tabelas com e sem usar a matriz W_{up}, observa-se que para muitos processadores (mais que 32) fazer agrupamento da última traz melhores ganhos, já para poucos processadores há um ligeiro aumento no ganho das soluções sem calcular W_{up}. Isto ocorre devido ao aparecimento de mais elementos diferentes de zero quando resolve-se a última partição em um único passo (W_{up} é cheia).

4.2.4 Resumo dos Melhores Desempenhos

Para facilitar a análise comparativa dos resultados das Tabelas 4.6 a 4.20, extraiu-se os maiores ganhos para cada número de processadores montando-se uma nova tabela com os melhores desempenhos para cada sistema. As Tabelas 4.21 a 4.24 mostram os maiores ganhos e as melhores estratégias de ordenação e de tratamento da última partição.

Tabela 4.21 - Melhores desempenhos para o sistema 118

Processadores	Ordenação	W_{up}	G	E
Suf	ML-MD	com	11,52	0,186
	MD-MNP	com	11,52	0,174
64	ML-MD	com	11,52	0,180
	MD-MNP	com	11,52	0,180
32	MD-MNP	com	10,60	0,331
16	MD-MNP	com	8,41	0,526
8	MD-MNP	com	5,41	0,676
4	MD-MNP	sem	3,33	0,832

Tabela 4.22 - Melhores desempenhos para o sistema 320

Processadores	Ordenação	W_{up}	G	E
Suf	ML-MD	com	18,10	0,114
	MD-MNP	com	18,10	0,108
64	MD-MNP	com	16,73	0,261
32	ML-MD	com	12,58	0,393
16	MD-MNP	com	8,33	0,521
8	MD-MNP	sem	5,23	0,669
4	MD-MNP	sem	3,23	0,832

Tabela 4.23 - Melhores desempenhos para o sistema 725

Processadores	Ordenação	W_{up}	G	E
Suf	ML-MD	com	23,21	0,058
	MD-MNP	com	23,21	0,064
64	ML-MD	com	19,32	0,302
32	ML-MD	sem	13,07	0,401
16	MD-MNP	com	9,27	0,579
8	MD-MNP	sem	5,80	0,725
4	MD-MNP	sem	3,41	0,852

Tabela 4.24 - Melhores desempenhos para o sistema 1729

Processadores	Ordenação	W_{up}	G	E
Suf	ML-MD	com	43,57	0,045
64	ML-MD	com	27,17	0,424
32	ML-MD	sem	17,12	0,535
16	MD-MNP	sem	10,75	0,672
8	MD-MNP	com	6,35	0,794
4	MD-MNP	sem	3,59	0,897

Das tabelas observa-se que quando estão disponíveis muitos processadores (64 ou mais), tanto os "fill-ins" gerados na matriz L quanto os "fill-ins" adicionais gerados na última partição da matriz W não atrapalham o desempenho global, pois os melhores ganhos, no geral, foram obtidos com a ordenação ML-MD utilizando W_{up}. Para poucos processadores (8 ou menos), os "fill-ins" de L e os "fill-ins" adicionais atrapalharam o desempenho da solução, obtendo-se melhores resultados com a ordenação MD-MNP sem utilizar W_{up}. Já quando se tem entre 8 e 64 processadores o máximo ganho depende do tamanho do sistema e da forma de tratar W_{up}. A ordenação MD-MNP obtém melhores resultados para a maioria dos casos.

A Figura 4.1 ilustra a evolução do máximo ganho com o tamanho do sistema e com o número de processadores.

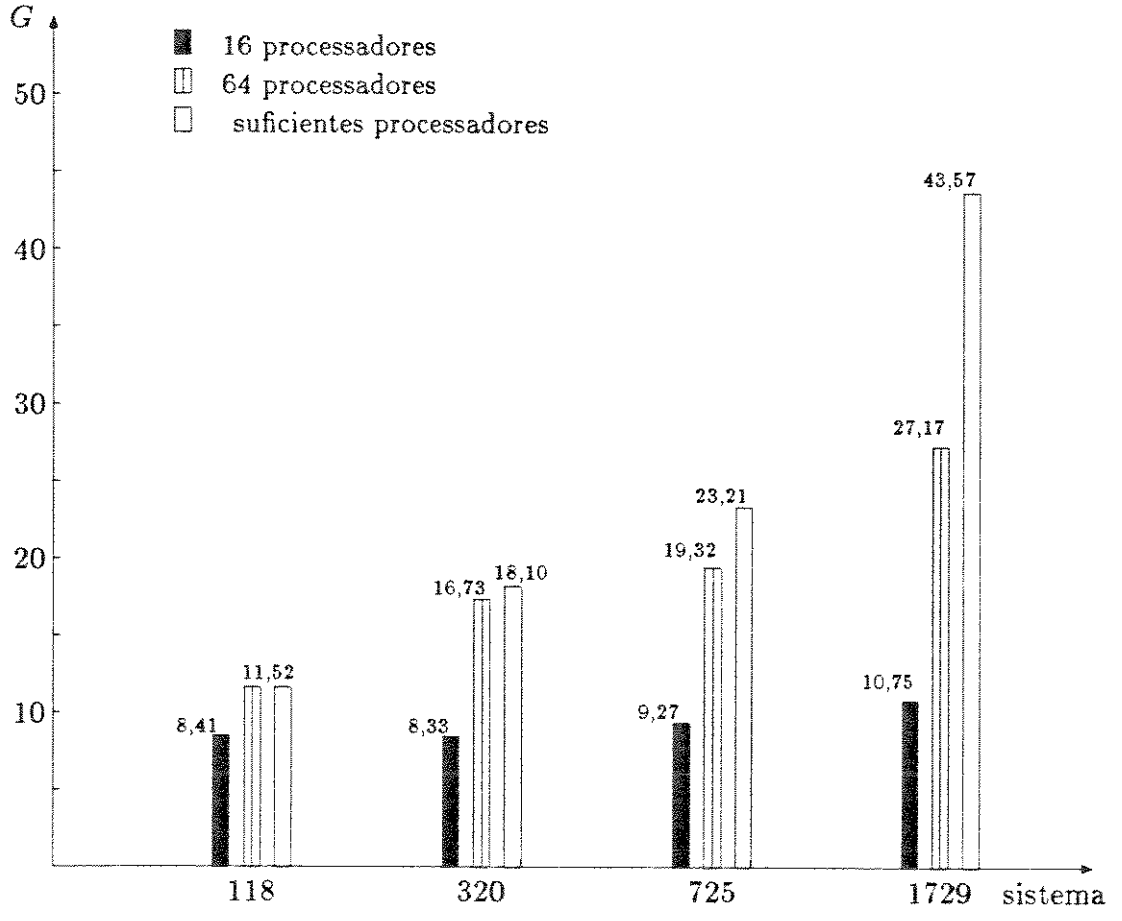


Figura 4.1 - Ganho X tamanho do sistema

Os resultados simulados permitem concluir que o fato de se usar W_{up} reduz o tempo da solução paralela de forma significativa quando o número de processadores disponíveis tende a se aproximar do número suficiente de processadores. Se o número de processadores é relativamente pequeno a vantagem de usar W_{up} tende a desaparecer. Isto é esperado pois o número de operações multiplicação-adição que caberá ao último processador a terminar suas tarefas poderá ser maior que no caso de processamento de W_n^t, D^{-1} e W_n separadamente. Deve porém ser lembrado que a utilização de W_{up} diminui o número de passos sequenciais.

4.2.5 Comparação com Outros Trabalhos

Os melhores resultados com a decomposição, o particionamento e a metodologia de fazer a última partição proposta neste trabalho foram: 11,52; 18,10; 23,21 e 43,57 para

os sistemas 118; 320; 725 e 1729 respectivamente. Simulando-se o paralelismo com um número suficiente de processadores, fazendo-se o particionamento apresentado em [17], e não calculando-se W_{up}, consegue-se os seguintes ganhos para os mesmos sistemas respectivamente: 7,46 (12 partições); 8,96 (21 partições); 12,96 (24 partições) e 22,59 (30 partições).

A seguir compara-se os ganhos da decomposição desta metodologia com ganhos teóricos mostrados em outros artigos, ressaltando-se que a comparação correta seria com resultados de implementações em máquinas paralelas e que apenas a referência [36] mostra resultados deste tipo de forma clara. Então a análise que se faz é apenas em termos de ganhos teóricos e possíveis dificuldades em termos de comunicação entre processadores. Assim, por exemplo, a metodologia proposta em [1] que buscar evitar atrasos com comunicação, mostra ganhos teóricos de: 3,8 e 7,6 para o mesmo sistema 118 e para um de 590 nós respectivamente, apenas para a etapa "forward". Uma metodologia que explora o paralelismo das operações elementares (portanto tem-se necessidade de muita comunicação) é proposta em [36], mostrando ganhos teóricos máximos de aproximadamente 40 para uma rede de 470 nós e de 90 para uma rede de 1557 nós. Esta referência mostra que para 8 processadores o ganho teórico aproxima-se de 8 para as ambas as redes, no entanto o ganho prático fica torno de 4,5, utilizando uma máquina paralela semelhante à apresentada no próximo item. A metodologia apresentada em [39], que em termos de atribuição de tarefas pode ser dita intermediária entre as propostas de [1] e [36], mostra ganhos teóricos de: 9,91; 36,9 e 63,6 para os sistemas 118; 590 e 1760 respectivamente.

4.3 Implementação em uma Máquina Paralela

4.3.1 Estrutura do computador utilizado

O computador multiprocessador utilizado neste trabalho é o Processador Preferencial Paralelo (PPP) desenvolvido no Centro de Pesquisas e Desenvolvimento da Telebrás, que é descrito em [10]. Sua arquitetura básica é mostrada na Figura 4.2.

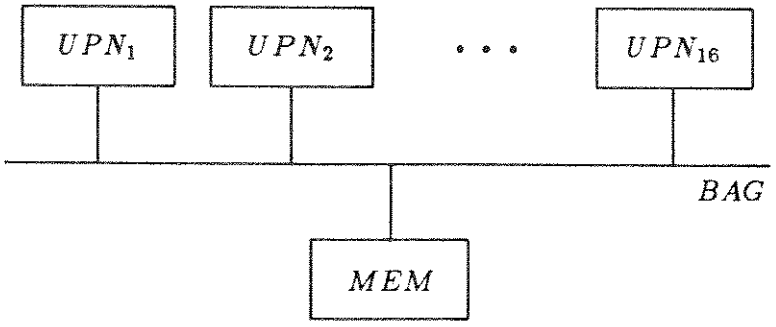
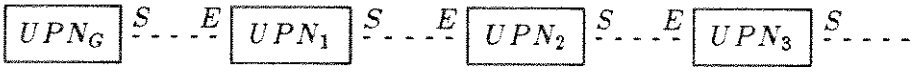


Figura 4.2 - Arquitetura do computador PPP

A versão do PPP utilizada é composta de 9 placas de processamento (UPN) baseadas no processador Intel iAPX 286, com memória RAM interna de 512 Kbytes. Todos os processadores compartilham uma placa de memória comum (MEM) de 128 Kbytes, podendo ser expandida até 2048 Kbytes, acessada através de um barramento global (BAG) com capacidade de escoamento de 10 Mbytes/s e com um esquema "daisy-chain" de controle de acessos concorrentes. O esquema "daisy-chain" é um tipo de conexão em que um sinal é serialmente distribuído para várias unidades, sendo que uma unidade qualquer da cadeia pode propagar o sinal para a seguinte se e somente se não for utilizá-lo. Este esquema é ilustrado na Figura 4.3.



UPN_G - unidade que gera o sinal
$UPN_1, UPN_2, \ldots$ - unidades que recebem e propagam o sinal
S - saída
E - entrada

Figura 4.3 - Esquema "daisy-chain"

Todas as placas processadoras podem executar programas usando tanto dados locais, armazenados na memória local, quanto dados globais, armazenados na memória partilhada e acessíveis a todos os processadores. O PPP apresenta, portanto, características de arquitetura híbrida, ou seja, memória comum mais memória distribuída.

A linguagem computacional utilizada neste trabalho foi o FORTRAN 77 convencional rodando sob um sistema operacional MS-DOS. Como o FORTRAN 77 não é uma linguagem para processamento paralelo, foi necessária a implementação de ferramentas básicas para acesso à memória partilhada e para mecanismos de sincronização.

Estas ferramentas ampliam a biblioteca do Fortran e foram escritas em Assembly [30], possuindo a seguinte forma:

- **GLOBAL (var)** - define a variável *var* (pode ser um vetor) como variável global, portanto será armazenada na memória partilhada; as variáveis globais são o meio de comunicação entre os processadores.
- **BARREI (ntot)** - define a existência de uma barreira, sendo que os processadores só executam o linha seguinte do programa quando todos os *ntot* processadores estiverem na barreira; esta função permite sincronizar os processadores.

4.3.2 Resultados no PPP

Para verificar o desempenho do método em um computador paralelo considerou-se a solução direta de um sistema $I = YV$, onde Y é a matriz adimitância complexa esparsa, I é um vetor complexo não-esparso das injeções de correntes e V é o vetor das tensões, cuja obtenção completa é desejada. Todas as variáveis do problema são armazenadas nas memórias locais dos processadores, exceto o vetor solução V que é declarado como global e armazenado na memória partilhada. A solução foi realizada para quatro sistemas de energia elétrica, usando diferentes números de processadores. Os algoritmos paralelos são descritos em detalhe no Apêndice D. As tabelas seguintes mostram o desempenho quando se calculam 50 soluções repetidas para cada sistema. Em todos os casos foi utilizada a ordenação ML-MD e são apresentados resultados sem usar W_{up} e usando W_{up}. Nas tabelas np significa número de particões utilizadas; t_s é o tempo em segundos usando a solução sequencial proposta em [40] com ordenação MD; t_m é o tempo em segundos da solução sequencial usando o mesmo método, ordenação e particionamento da solução paralela; t_p é o tempo em segundos da solução paralela; e $G = t_s/t_p$ é o ganho.

Caso sem usar W_{up}

As Tabelas 4.25, 4.26, 4.27 e 4.28, mostram o desempenho quando resolve-se a última partição como as demais.

Tabela 4.25 - Ganho em 50 soluções repetidas no sistema 118
($np = 6$, $t_s = 21,99$ e $t_m = 25,82$)

	Processadores							
	2	3	4	5	6	7	8	9
t_p	13,52	9,56	7,53	6,32	5,71	5,11	4,61	4,40
G	1,63	2,30	2,92	3,48	3,85	4,30	4,76	5,00

Tabela 4.26 - Ganho em 50 soluções repetidas no sistema 320
($np = 8$, $t_s = 71,10$ e $t_m = 87,64$)

	Processadores							
	2	3	4	5	6	7	8	9
t_p	44,83	31,32	24,12	19,95	17,42	15,49	13,85	12,86
G	1,56	2,27	2,95	3,56	4,08	4,59	5,13	5,53

Tabela 4.27 - Ganho em 50 soluções repetidas no sistema 725
($np = 9$, $t_s = 124,07$ e $t_m = 157,86$)

	Processadores							
	2	3	4	5	6	7	8	9
t_p	80,60	55,12	43,02	35,12	29,83	26,81	23,85	21,703
G	1,54	2,25	2,88	3,53	4,16	4,63	5,20	5,72

Tabela 4.28 - Ganho em 50 soluções repetidas no sistema 1729
($np = 10$, $t_s = 282,58$ e $t_m = 340,66$)

	Processadores							
	2	3	4	5	6	7	8	9
t_p	172,53	117,47	88,52	73,30	67,09	53,08	47,03	42,91
G	1,64	2,41	3,19	3,85	4,21	5,32	6,01	6,58

Caso com W_{up}

As Tabelas 4.29, 4.30, 4.31 e 4.32, mostram o desempenho alterando-se a forma de resolver a última partição, isto é, agora calcula-se a matriz W_{up} e portanto, resolve-se a "forward", a diagonal e a "backward" para esta matriz em um único passo.

Tabela 4.29 - Ganho no sistema 118, calculando-se W_{up}
$(np = 7,\ t_s = 21,99$ e $t_m = 25,71)$

	Processadores							
	2	3	4	5	6	7	8	9
t_p	13,46	9,61	7,53	6,43	5,88	5,33	4,61	4,34
G	1,63	2,29	2,92	3,42	3,74	4,13	4,76	5,07

Tabela 4.30 - Ganho no sistema 320, calculando-se W_{up}
$(np = 8,\ t_s = 71,10$ e $t_m = 89,72)$

	Processadores							
	2	3	4	5	6	7	8	9
t_p	45,82	31,48	24,17	20,00	17,09	15,77	13,52	12,75
G	1,55	2,26	2,94	3,56	4,16	4,51	5,23	5,58

Tabela 4.31 - Ganho no sistema 725, calculando-se W_{up}
$(np = 11,\ t_s = 124,07$ e $t_m = 154,04)$

	Processadores							
	2	3	4	5	6	7	8	9
t_p	79,23	54,61	46,64	35,11	29,78	27,25	24,12	22,14
G	1,57	2,27	2,66	3,53	4,17	4,55	5,14	5,60

Tabela 4.32 - Ganho no sistema 1729, calculando-se W_{up}
$(np = 12,\ t_s = 282,58$ e $t_m = 342,53)$

	Processadores							
	2	3	4	5	6	7	8	9
t_p	173,43	118,35	88,85	74,01	62,69	54,12	47,47	43,52
G	1,63	2,39	3,18	3,82	4,51	5,22	5,95	6,49

Estes resultados indicam o mesmo já concluído no item de decomposição do problema, ou seja, que para poucos processadores não se consegue melhorar o ganho com a solução usando W_{up}. A melhoria que observa-se em alguns casos ocorre quando a última partição fica balanceada com o mesmo número de linhas por processador. Os resultados práticos ratificam o fato que usar W_{up} só é vantajoso quando um processador não tem muito mais tarefas que outros.

4.3.3 Desempenho da Metodologia

Comparando-se os ganhos obtidos no PPP com os ganhos teóricos obtidos no item 4.2.2, observa-se que os ganhos práticos são maiores que os teóricos, em alguns casos. Isto acontece porque a comparação não está sendo feita nas mesmas condições. Os ganhos teóricos do item 4.2.2 foram obtidos considerando-se somente as operações de ponto flutuante de elementos fora da diagonal.

A fim de analisar corretamente o ganho no PPP é necessário calcular-se um ganho teórico que leve em conta todas as operações na solução direta. Para isto observa-se que existem operações com elementos da diagonal e fora da diagonal. As operações com elementos da diagonal correspondem a multiplicar D^{-1} pelo vetor b e portanto cada elemento da diagonal exige uma operação de multiplicação. Para os elementos fora da diagonal (etapas "forward" e "backward") é necessária uma operação de multiplicação mais uma de adição para cada elemento da matriz. Então existem apenas duas diferentes operações na solução direta, e fazer uma multiplicação complexa corresponde a aproximadamente fazer 2,5 adições complexas (resultado observado em simulações com processador 286). Desta forma pode-se atribuir um peso 5 para cada multiplicação e um peso 2 para cada adição, ao fazer-se uma estimativa do ganho teórico, de maneira similar ao que foi realizado no item 4.2 referente à decomposição do problema. Assim, por exemplo, o parâmetro t_s para o sistema IEEE-118 que contém 530 elementos fora da diagonal depois da ordenação MD, torna-se:

$$t_s = (530 + 118) \times 5 + 530 \times 2$$

O parâmetro t_p é calculado da mesma maneira, e o ganho teórico $G_t = t_s/t_p$, para os quatro sistemas simulados, é mostrado nas Tabelas 4.33, 4.34, 4.35 e 4.36, onde a solução paralela é feita sem calcular a W_{up} e depois de uma ordenação ML-MD. Nas tabelas M indica o número de operações com elementos da diagonal e MA é o total de operações com elementos fora da diagonal para diferentes números de processadores. Nestas tabelas também é mostrada a eficiência $E = G/ntot$ (G é o ganho usando $ntot$ processadores no PPP) e a eficiência de implementação $E_i = G/G_t$.

As Figuras 4.4, 4.5, 4.6 e 4.7 mostram a comparação dos ganhos obtidos no PPP e os teóricos.

Tabela 4.33 - Ganhos e eficiências no sistema 118 ($t_s = 4300$)
(solução em 6 partições)

Processadores	M	MA	t_p	G_t	G	E	E_i
8	15	108	831	5,17	4,76	0,59	0,92
6	20	136	1052	4,09	3,85	0,64	0,94
4	30	181	1417	3,03	2,92	0,73	0,96
2	59	322	2549	1,69	1,62	0,81	0,96

Tabela 4.34 - Ganhos e eficiências no sistema 320 ($t_s = 14018$)
(solução em 8 partições)

Processadores	M	MA	t_p	G_t	G	E	E_i
8	40	343	2601	5,39	5,13	0,64	0,95
6	54	432	3294	4,26	4,08	0,68	0,96
4	80	602	4614	3,04	2,95	0,74	0,97
2	160	1117	8619	1,63	1,56	0,78	0,96

Tabela 4.35 - Ganhos e eficiências no sistema 725 ($t_s = 24583$)
(solução em 9 partições)

Processadores	M	MA	t_p	G_t	G	E	E_i
8	91	584	4543	5,41	5,20	0,65	0,96
6	121	732	5729	4,29	4,16	0,69	0,97
4	182	1060	8330	2,95	2,88	0,72	0,97
2	363	1963	15556	1,58	1,54	0,77	0,97

Tabela 4.36 - Ganhos e eficiências no sistema 1729 ($t_s = 55615$)
(solução em 10 partições)

Processadores	M	MA	t_p	G_t	G	E	E_i
8	217	1120	8925	6,23	6.01	0,75	0,96
6	289	1485	11840	4,70	4,21	0,70	0,89
4	433	2114	16963	3,28	3,19	0,80	0,97
2	865	4101	33032	1,68	1,64	0,82	0,98

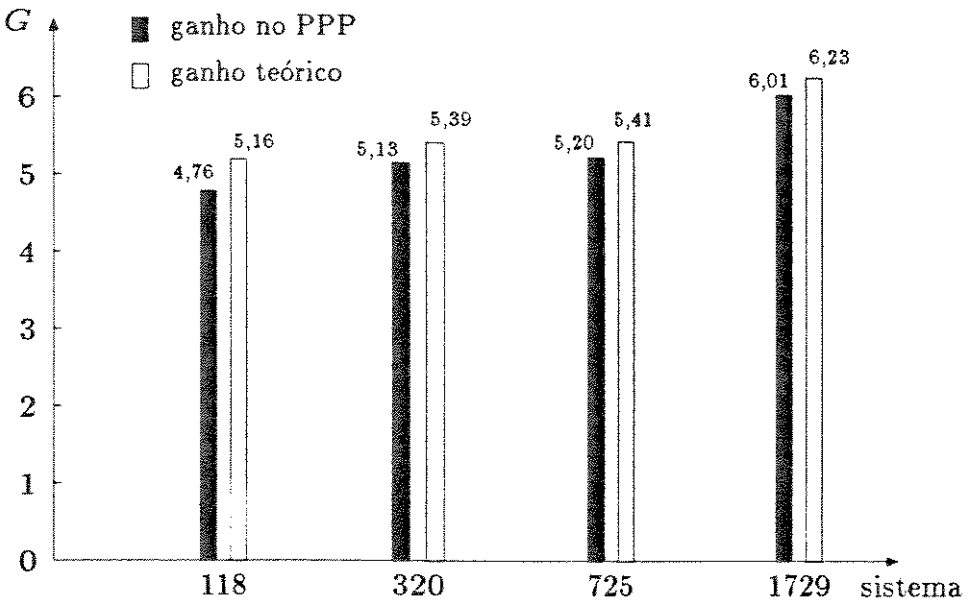


Figura 4.4 - Ganho utilizando 8 processadores

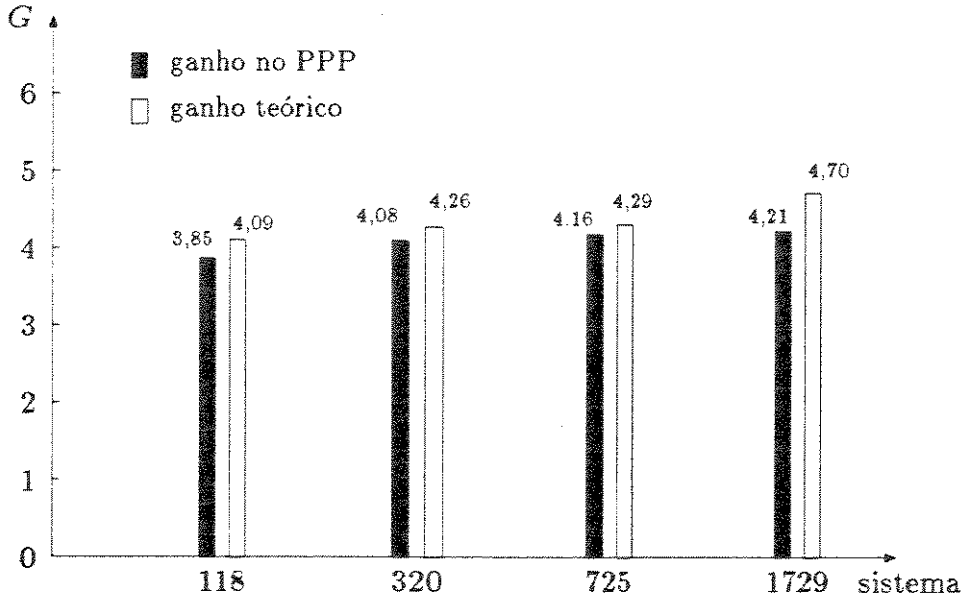


Figura 4.5 - Ganho utilizando 6 processadores

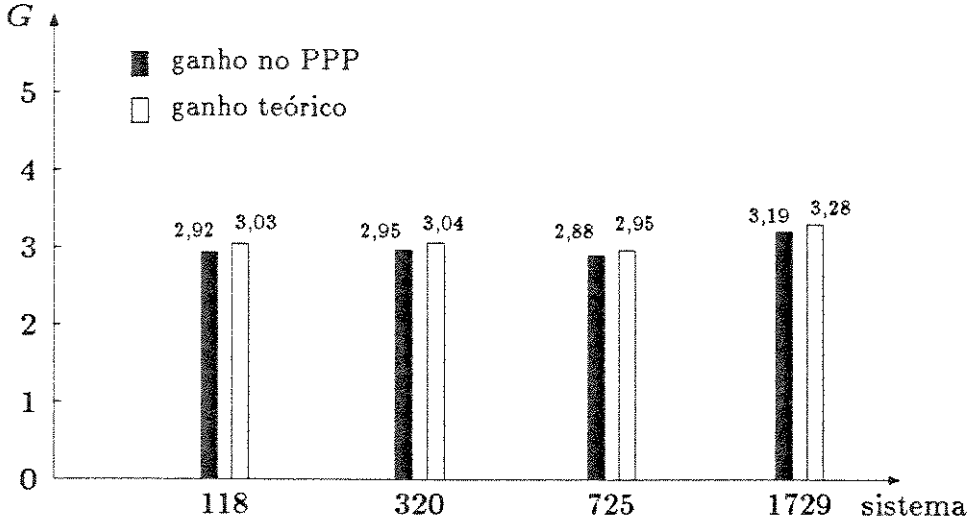


Figura 4.6 - Ganho utilizando 4 processadores

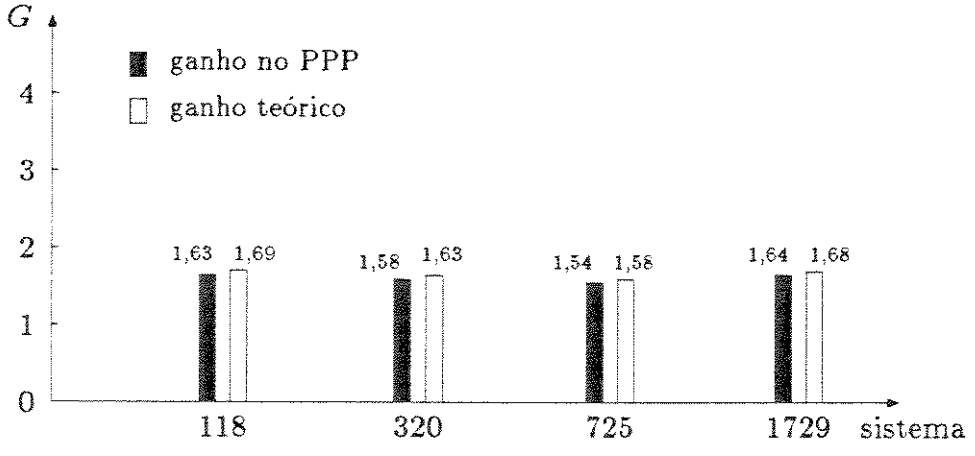


Figura 4.7 - Ganho utilizando 2 processadores

Estes resultados mostram que os ganhos obtidos na implementação são muito próximos aos ganhos teóricos, evidenciando que não existe atraso significativo de comunicação. Os diversos "overheads" serão discutidos no item seguinte.

Análise dos "Overheads"

No algoritmo implementado podem aparecer três tipos de atrasos na solução:

- "overhead" de algoritmo - que surge devido à mudança na forma de fazer a "forward" e principalmente pelos novos elementos diferentes de zero que aparecem ("fill-ins" em L e "fill-ins" adicionais).
- "overhead" de sincronização ("idle time") - devido a processadores que ficam parados esperando por outros, e é atribuído à formação de pequenas partições e/ou ao desbalanceamento de carga provocado pela atribuição de todas as operações de uma linha a um processador (*quebra de relações de não-simultaneidade*).
- "overhead" de comunicação - que surge devido à necessidade de busca e atualização de dados na memória partilhada; nesta implementação o principal atraso nesta categoria é a contenção.

O atraso geral da solução pode ser obtido através da relação G/G_{ot}, onde G é o ganho obtido com o PPP e G_{ot} é o ganho ótimo calculado através da divisão de t_s pelo número de processadores. O "overhead" de algoritmo pode ser obtido da relação t_m/t_s enquanto que o "overhead" de comunicação pode ser estimado através da relação G_t/G. O "overhead" de sincronização é obtido subtraindo os "overheads" de algoritmo e comunicação do atraso geral. As porcentagens, em relação à solução sequencial, de cada um destes tipos atrasos são mostradas nas Tabelas 4.37, 4.38, 4.39 e 4.40, para os quatro sistemas testados. Nestas tabelas o símbolo $\star$ significa valor muito próximo de zero.

Tabela 4.37 - "Overheads" no sistema 118
(17,4 % de "overhead" de algoritmo)

"overheads"	Processadores			
	2	4	6	8
geral	22,96	36,97	55,80	67,71
comunicação	3,68	3,77	6,23	8,61
sincronização	2,56	15,8	32,17	41,70

Tabela 4.38 - "Overheads" no sistema 320
(23,26 % de "overhead" de algoritmo)

"overheads"	Processadores			
	2	4	6	8
geral	26,10	35,70	47,00	55,84
comunicação	3,16	3,05	4,41	5,06
sincronização	$\star$	9,39	19,33	27,52

Tabela 4.39 - "Overheads" no sistema 725
(27,23 % de "overhead" de algoritmo)

"overheads"	Processadores			
	2	4	6	8
geral	29,90	38,70	44,28	53,78
comunicação	2,60	2,43	3,12	4,04
sincronização	*	9,04	13,93	22,51

Tabela 4.40 - "Overheads" no sistema 1729
(20,55 % de "overhead" de algoritmo)

"overheads"	Processadores			
	2	4	6	8
geral	22,11	25,30	42,45	33,14
comunicação	2,44	2,82	11,63	3,66
sincronização	*	1,93	10,27	8,93

Estes resultados indicam que o "overhead" de comunicação cresce muito pouco com o aumento do número de processadores. Este tipo de "overhead" é o que tem limitado a implementação de solução direta com métodos que procuram explorar o paralelismo das operações elementares. Isto leva a crer que a comunicação, na metologia proposta e usando arquiteturas híbridas, não limitará o numero de processadores. Por outro lado o "overhead" de sincronização limitará o numero de processadores, uma vez que chega-se a um ponto onde muitos processadores ficam sem ter o que fazer, devido ao tamanho das partições. Um indicativo disto é o valor deste "overhead" para os sistemas 118 e 1729 para 8 processadores. Já o "overhead" de algoritmo não varia significativamente para os diferentes sistemas, ficando na faixa dos 20 %. Finalmente pode-se dizer que estes "overheads" tendem a diminuir com o aumento do tamanho dos sistemas, pois consegue-se cada vez maiores partições, que significam mais tarefas independentes.

Capítulo 5

Conclusões

Neste trabalho apresentou-se uma forma de resolver em paralelo um conjunto de equações, que normalmente exige métodos iterativos para resolvê-lo. Investiu-se na solução direta de um conjunto de equações lineares esparsas, que é uma etapa do método de Newton (ou de seus derivados desacoplados) que normalmente deve ser resolvida muitas vezes na simulação de um problema. A metodologia proposta neste trabalho é baseada numa matriz de fatores inversos W, fazendo-se a solução por partições, cuja idéia foi primeiramente proposta por [15], sendo que este trabalho acrescentou três contribuições:

- A forma de fazer a "forward" é por linhas e por partições, podendo-se então atribuir a um processador todas as operações correspondentes a uma linha dentro de uma partição, como uma tarefa indivisível. A "backward" é feita por linha e por partições, sendo que todas operações de uma linha constitui uma tarefa indivisível. Esta forma de atribuição de tarefas não explora diretamente o máximo de paralelismo do problema, mas a redução em "overhead" de comunicação produz vantagens altamente compensadoras;

- O particionamento da matriz é feito com base na profundidade dos nós na árvore dos caminhos de fatoração da rede, sendo que todos os nós com uma mesma profundidade são agrupados em uma mesma partição. No ínicio do particionamento não há preocupação em aumentar o número de nós, pois as primeiras partições já possuem um grande número de tarefas independentes. Quando as partições passam a se tornar pequenas, agrupam-se todos os nós restantes em uma única partição. Desta maneira apenas esta última partição exige cálculos extras na sua obtenção, sendo que estes são feitos uma vez no ínicio e só serão refeitos quando houver mudanças na topologia da rede.

- Para a resolução da última partição utilizam-se duas estratégias, uma que faz as etapas "forward"+diagonal+"backward" usando os fatores inversos, e outra que agrupa estas três etapas em uma única, utilizando uma matriz cheia W_{up}. Isto traz vantagens em termos de "speedup" quando se dispõe de um número razoável de processadores.

Para resolver a solução direta em paralelo como descrito na tese, algum trabalho extra é necessário em relação à solução sequencial convencional. A primeira modificação é em relação à ordenação dos nós da rede elétrica. Para a metodologia proposta as ordenações mais vantajosas são aquelas que geram caminhos de fatoração com menores profundidades, destacando-se a ML-MD e MD-MNP, sendo que os ganhos obtidos com estas ordenações são praticamente iguais, e que a ordenação ML-MD oferece uma vantagem extra que é ordenar os nós na sequência das profundidades. Os tempos de processamento para fazer estas duas ordenações são praticamente iguais ao tempo de fazer a MD, como já mostrado em [19], sendo que a implementações de ambas é simples como descrito no Apêndice C. Deve-se lembrar que a ordenação é feita apenas uma vez em todo processo de solução.

O particionamento das primeiras $n-1$ partições é bastante rápido por não requerer nenhum cálculo para os fatores inversos. Estes cálculos são necessários apenas na última partição, sendo que eles também podem ser executados em paralelo [8], o que não foi considerado neste trabalho. Como a ordenação, o particionamento é feito apenas uma vez, sendo que o cálculo dos fatores inversos W só devem ser refeitos se houver mudança na topologia da rede.

A implementação em um protótipo de computador paralelo mostrou que é possivel obter ganhos significativos, e tornar a etapa de solução direta mais rápida. Do ponto de vista dos atrasos inseridos no problema, mostra-se que o "overhead" de comunicação - que tem invalidado muitas proposições (por exemplo, a referência [36] conseguiu um ganho em torno de 4,5 com 8 processadores utilizando usando uma máquina semelhante ao PPP em uma rede de 1557 nós) - é pequeno na máquina utilizada, embora os canais de comunicação não apresentem as características desejadas para evitar contenção no barramento. A implementação da metodologia em máquinas com arquiteturas diferentes (memória distribuída, por exemplo) e com mais processadores devem ser objetivos de investigação futura.

A realização mais rápida possível da etapa de solução direta traz vantagens em uma série de problemas de simulação de circuitos elétricos, podendo também tornar possível a implementação de novas funções em tempo real nos centros de controle de sistemas de energia elétrica, aumentando a segurança destes sistemas, como por exemplo o cálculo da estabilidade transitória. Um caminho indicativo disto já é apresentado em [11], onde explora-se o paralelismo no espaço e no tempo. Paralelismo no espaço significa resolver em paralelo o sistema de equações para um mesmo instante de tempo. Paralelismo no tempo consiste em calcular em paralelo grandezas em instantes de tempo diferentes, durante a simulação dinâmica, sendo que uma boa proposição para explorar este paralelismo é sugerida em [3].

A referência [11] também faz uma comparação entre o método de Newton VDHN (esquema simultâneo implícito deixando a matriz jacobiana constante durante alguns

passos de integração descrito em [26]) e o SOR-Newton que é um algoritmo com facilidades de paralelização, mas que mantém uma convergência razoalvelmente rápida [31]. Explorou-se o paralelismo no tempo para o VDHN e o paralelismo no tempo e no espaço para o SOR, mostrando-se ganhos em máquinas de arquiteturas diferentes. Os resultados indicam maiores ganhos com o SOR na máquina de memória partilhada e melhor desempenho do VDHN na máquina de memória distribuída.

A implementação de um programa de estabilidade transitória que explore o paralelismo no espaço e no tempo é um dos objetivos futuros de sequência deste trabalho. Também pretende-se investigar o desempenho da metodologia em computadores vetoriais, uma vez que as multiplicações podem ser vetorizadas nas $n-1$ primeiras partições [17], e que a última partição com W_{up}, como proposto neste trabalho, pode ser totalmente vetorizada.

No Apêndice E mostra-se uma cópia do "paper", parte desta tese, que será apresentado no IEEE 1991 Summer Power Meeting em San Diego USA, de 28 de julho a 1 de agosto de 1991.

Apêndice A

Formação das Matrizes L e W

A.1 Formação da matriz L

A formação da matriz L já foi apresentada na equação (3.8) e corresponde a fazer uma sequência de multiplicações de matrizes, que possuem a diagonal unitária e apenas um elemento diferente de zero fora da diagonal, ou seja, $L = L^1 L^2 \cdots L^q$. As matrizes L^r, $r = 1, 2, \ldots, q$ são matrizes elementares correspoendentes a operações efetuadas no triângulo inferior da matriz A dos coeficientes [24].

Sejam L^j e L^h duas destas matrizes elementares, mostradas na Figura A1.

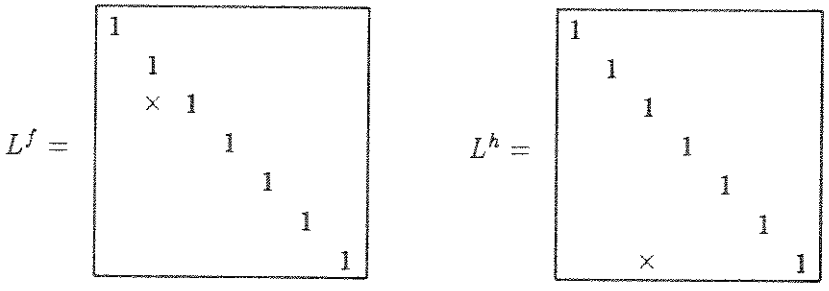


Figura A1 - Matrizes elementares L^j e L^h

Considere-se ainda que no processo de eliminação a matriz L^j *zerou* um coeficiente de A antes que L^h. Desta forma, na obtenção da matriz L deve-se realizar o produto $L^j L^h$ necessariamente nesta ordem. Quando este produto é efetuado nota-se uma importante relação mostrada na Figura A2.

$$L^f L^h = L^f + L^h - I_n = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & \times & 1 & & \\ & & & 1 & \\ & & & & 1 \\ & & & & & 1 \\ & & \times & & & & 1 \end{bmatrix}$$

Figura A2 - Produto de L^f por L^h

Isto indica que o produto $L^f L^h$ é dado pela superposição das matrizes L^f e L^h, sendo também, válido para o produto das matrizes L^r, $r = 1, 2, \cdots, q$, como a expressão seguinte:

$$L^1 L^2 \cdots L^q = I_n + \sum_{r=1}^{q} (L^q - I_n) = L \tag{A.1}$$

onde I_n é a matriz identidade de ordem n.

A matriz L preserva, parcialmente, o grau de esparsidade do triângulo inferior da matriz A. Ela contém todos os elementos não-zero de A mais alguns novos elementos ("fill-ins") gerados no processo de eliminação de Gauss.

A.2 Formação da matriz W

A matriz W é a inversa da matriz L, sendo obtida da seguinte forma:

$$W = L^{-1} = (L^q)^{-1} \cdots (L^2)^{-1} (L^1)^{-1} \tag{A.2}$$

As matrizes $(L^r)^{-1}$, $r = 1, 2, \ldots, q$ são idênticas às matrizes L^r a menos do sinal do elemento não-zero fora de diagonal. Note-se que para obtenção da matriz W os produtos das matrizes $(L^r)^{-1}$ são feitos na ordem inversa do que foi feito para obtenção de L.

Sejam as matrizes $(L^f)^{-1}$ e $(L^h)^{-1}$, inversas das L^f e L^h anteriormente consideradas, respectivamente. O resultado do produto entre elas, com a finalidade de obtenção da matriz W, é mostrado na Figura A3. Deve-se notar que a ordem é necessariamente a mostrada na figura.

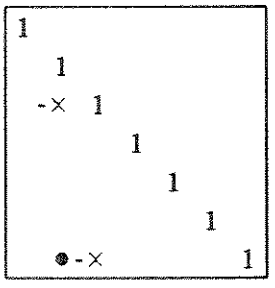
$(L^h)^{-1}(L^f)^{-1} =$ 

Figura A3 - Produto de $(L^h)^{-1}$ por $(L^f)^{-1}$

Deste resultado nota-se que: $(L^h)^{-1}(L^f)^{-1} \neq (L^h)^{-1} + (L^f)^{-1} - I_n$ e que novos elementos ("fill-ins" adicionais) podem surgir. O número de elementos não-zero tanto na matriz L, como na matriz W, depende da sequência de eliminação dos nós da rede elétrica (veja esquemas de ordenação dos nós no Apêndice C). Usando a ordenação MD, para quatro diferentes sistemas de potência, mostra-se na Tabela A1 o número de não-zeros fora da diagonal nas matrizes A (apenas o triângulo inferior), L e W, e a porcentagem de não-zeros em cada uma delas em relação ao triângulo inferior (sem contar a diagonal) de A^{-1} (cheia).

Tabela A1 - Quantidade de não-zero nas matrizes A, L e W.

	118		320		725		1729	
	não-zeros	%	não-zeros	%	não-zeros	%	não-zeros	%
A	179	2,59	470	0,92	935	0,18	2154	0,07
L	265	3,84	887	1,74	1497	0,28	3355	0,11
W	1132	16,40	6745	13,21	17243	6,65	54182	1,81

Apêndice B

Caminhos de Fatoração

B.1 Considerações Gerais

Métodos que exploram as características de esparsidade de matrizes foram introduzidos na década de 60 [35], tendo sido usados com sucesso na resolução de problemas de redes de energia elétrica. Mais recentemente, técnicas de vetores esparsos foram introduzidas propiciando melhores soluções para um grande número de problemas [34].

Considere-se um sistema de equações lineares do tipo:

$$Ax = b \tag{B.1}$$

onde A é simétrica, esparsa e definida positiva podendo ser fatorada como LDL^t.

O método de vetores esparsos foi idealizado tendo em vista resolver dois tipos de problemas:

- Solução de (B.1) quando o vetor b possui somente poucos elementos diferentes de zero, ou quando um pequeno número de elementos do vetor x são necessários calcular (problema de vetor esparso propriamente dito).
- Soluções repetidas de (B.1) com necessidade de se modificar parte da matriz A (problema de refatoração parcial).

A idéia básica para implementação da técnica de vetor esparso consiste em achar um caminho de fatoração para matriz A. Para isto, considere-se a rede, extraída de [34], mostrada na Figura B1.

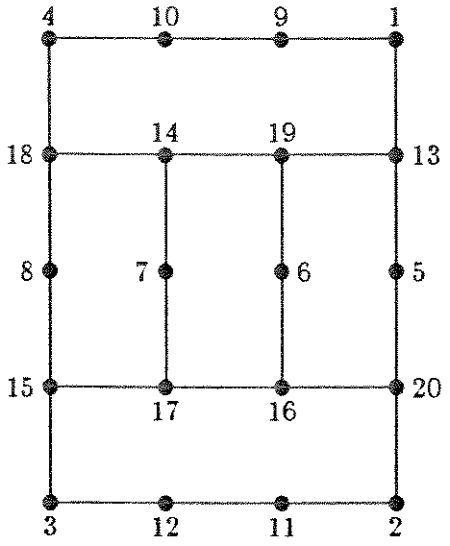


Figura B1 - Rede exemplo com 20 nós

A estrutura da matriz L correspondente a esta rede é mostrada na Figura B2, sendo a ordenação arbitrariamente selecionada.

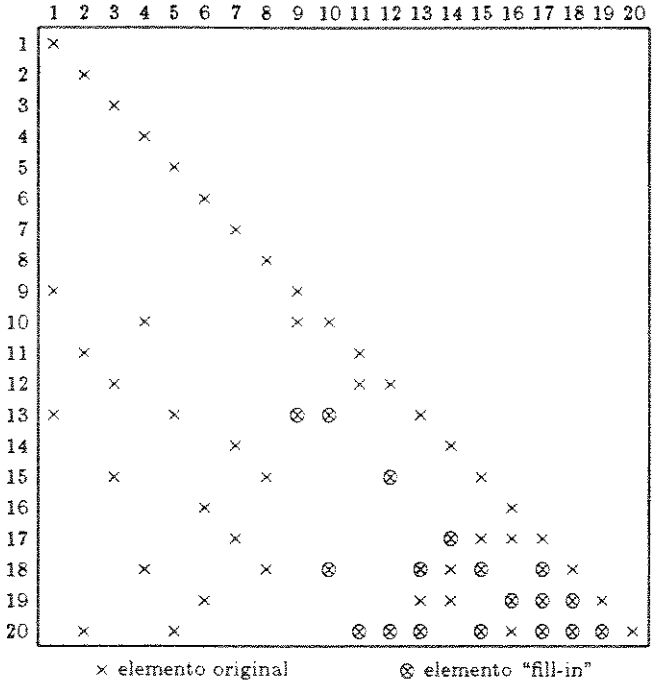


Figura B2 - Estrutura da matriz L

B.2 Obtenção dos Caminhos de Fatoração

A fim de facilitar a aplicação do método de vetores esparsos foi introduzido o conceito de "singleton" [34]. Um "singleton" é um vetor com somente um elemento não-zero (na posição k). Um caminho de fatoração para um "singleton" é uma lista ordenada das colunas de L, definida como segue:

1. Comece a lista com a coluna k de L;
2. Pegue a primeira linha que possua elemento diferente de zero na coluna k de L, faça k igual ao número da linha e inclua-o na lista;
3. Se k é a última coluna de L, pare. Caso contrário, volte ao passo 2.

Se o vetor b é um "singleton" então somente as colunas de L de seu caminho de fatoração são necessárias durante o processo de substituição "forward". Analogamente, se somente a k-ésima entrada de x é procurada, somente as linhas de L^t deste caminho são necessárias durante o processo de substituição "backward". Neste caso o caminho é percorrido na ordem reversa. Estes processos são chamados de "Fast-Forward" (FF) e "Fast-Backward" (FB), respectivamente [34].

Fazendo uma analogia entre fatoração e eliminação "forward", fica claro que somente as colunas envolvidas no apropriado caminho serão atualizadas em L quando a k-ésima coluna de A é modificada (Refatoração Parcial [12]).

Um caminho de fatoração associado a um vetor pode ser formado pela união de todos os possíveis "singletons", sendo que esta união pode ser representada em um grafo dos caminhos. A profundidade ("depth" ou length") de cada nó em um caminho de fatoração é o máximo número de nós em um caminho que antecede o nó no grafo. A Figura B3 mostra o grafo dos caminhos de fatoração para a matriz L da Figura B2.

Os caminhos de fatoração também podem ser visualizados diretamente na matriz L. A Figura B4 é uma reprodução da matriz L, onde as setas indicam a obtenção do caminho de fatoração para o nó 2.

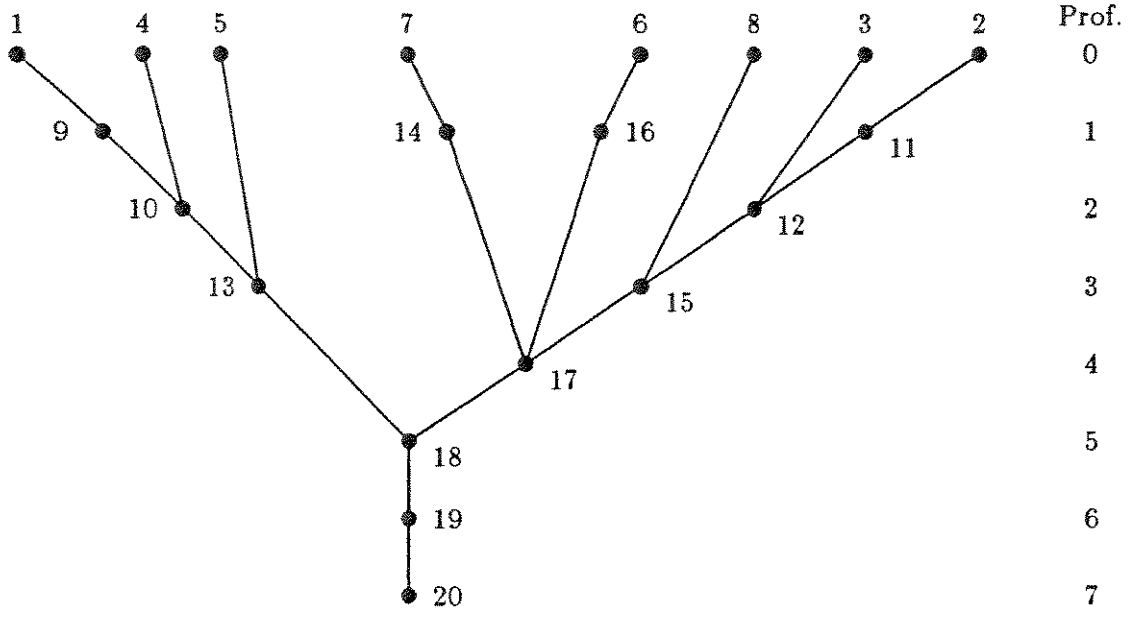


Figura B3 - Grafo dos caminhos de fatoração com indicação da profundidade

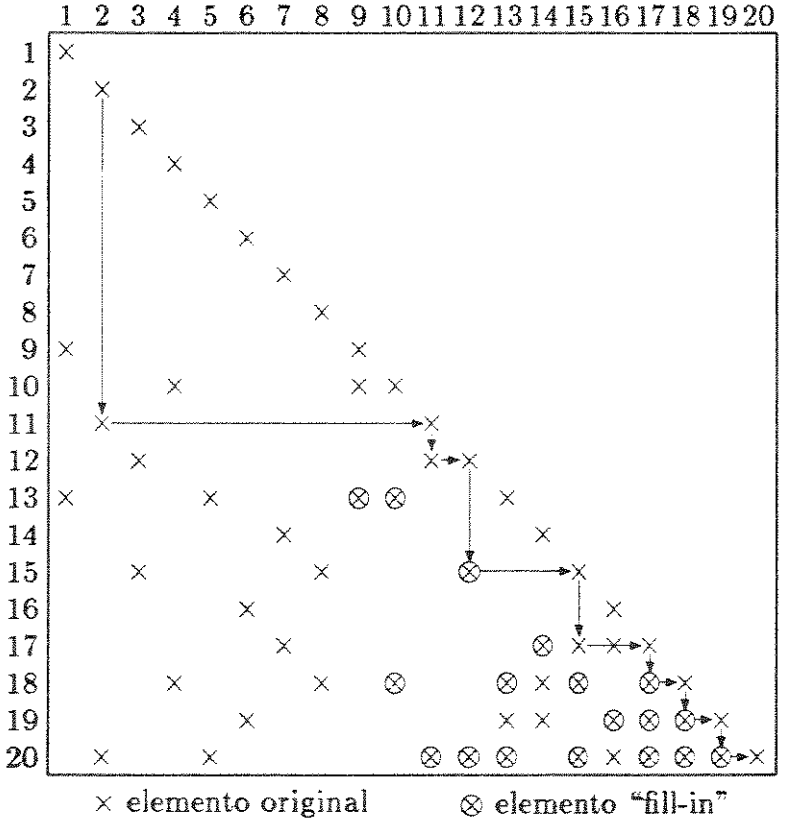


Figura B4 - Estrutura da matriz Lcom caminho de fatoração para o nó 2.

Partindo-se do elemento (2,2) da matriz L procura-se o primeiro elemento não-nulo da coluna 2, que é o elemento (11,2). Obtém-se em seguida o elemento (11,11) e procura-se o primeiro elemento da coluna 11, que é o elemento (12,11), e assim sucessivamente. Finalmente, obtém-se o caminho de fatoração para o nó 2 através dos elementos da diagonal que aparecem na análise, resultando no grafo da Figura B5.

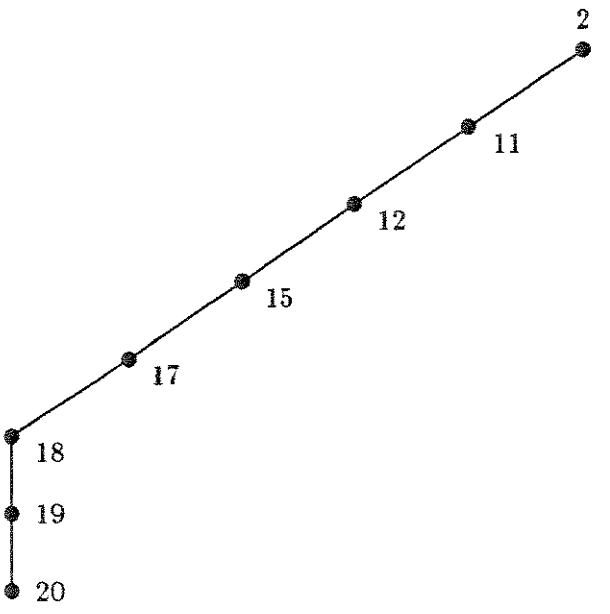


Figura B5 - Caminho de fatoração para o nó 2

O grafo da Figura B5 é um subgrafo da Figura B4.

A utilização dos caminho de fatoração é fundamental para o método de vetores esparsos e é através dela que se obtém uma economia de cálculo para solução do sistema (B.1) e/ou para a refatoração parcial de L.

B.3 Relação entre o Grafo dos Caminhos de Fatoração e a Matriz L^{-1}

Os nós que precedem um determinado nó em um grafo de fatoração podem ser obtidos a partir da matriz L^{-1}, ou seja, os nós predecessores do nó n são os nós correspondentes aos números das colunas com elementos não-zero na linha n de L^{-1}. Estas relações podem ser importantes quando, por exemplo, buscam-se ordenações que visem obtenção de grafos menos profundos, como discutido no Apêndice C. A Figura B6 mostra a matriz inversa da matriz L da Figura B2, a qual juntamente com o grafo Figura B3 permite

verificar a relação enunciada acima.

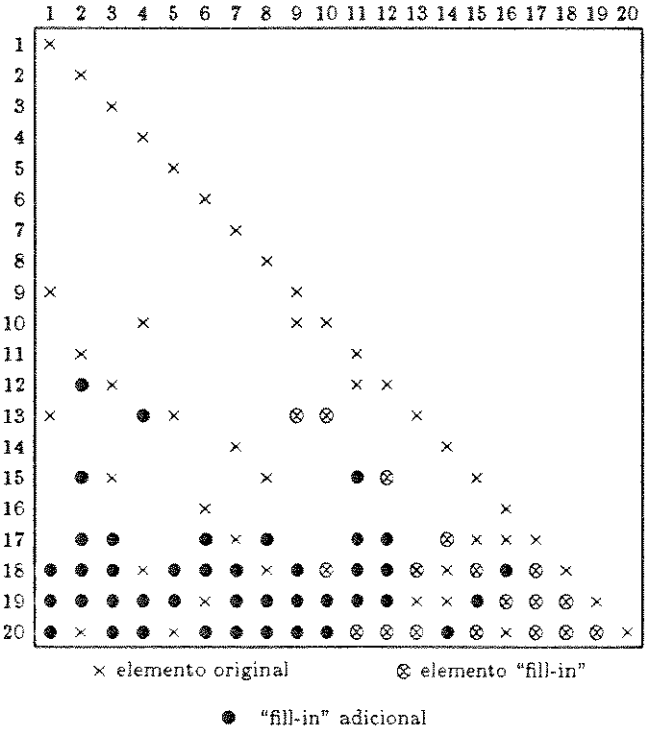


Figura B6 - Estrutura da matriz L^{-1}

Apêndice C

Ordenação dos Nós

C.1 Considerações Gerais

Ordenar nós de uma rede elétrica corresponde a ordenar as colunas/linhas da matriz resultante da modelagem. Os critérios de ordenação de colunas/linhas de matrizes esparsas de redes elétricas foram inicialmente propostos visando diminuir o aparecimento de novos elementos diferentes de zero durante a fatoração da matriz (elementos "fill-ins"). A referência [35], considerada clásssica, propõe os três seguintes esquemas de ordenação:

1 - **Mínimo-grau estático:** os nós são ordenados de acordo com o menor grau (número de nós vizinhos com ligação) da configuração original da rede;

2 - **Mínimo-grau dinâmico:** aplica-se o esquema 1 para primeiro nó a ser ordenado. Após a ordenação de um nó altera-se a configuração da rede supondo sua eliminação, atualiza-se o grau dos nós vizinhos, e volta-se a aplicar o esquema 1;

3- **Mínimo número de "fill-ins":** em cada estágio de ordenação simula-se a eliminação de todos os nós, sendo que aquele que gerar menor número de "fill-ins" é escolhido.

O esquema 2 tem sido o preferido por sua simplicidade e pelo reduzido número de "fill-ins" gerados, não muito difierente do esquema 3 que exige muito mais processamento. Já o esquema 1 gera muito mais "fill-ins", sendo por isso pouco util.

Mais recentemente outros critérios de ordenação têm sido propostos buscando adequar-se ao método de vetores esparsos [18] [7] e [19], ou particionamento de matrizes [15]. Neste apêndice apresenta-se um algoritmo para o esquema 2 e para os mais recentes esquemas. Utiliza-se a rede da Figura C1, extraída da referência [7], que é um exemplo muito particular mas permite mostrar claramente as diferenças das ordenações. Deve-se ressaltar que a numeração da rede pode levar a grafos de fatoração muito semelhantes para as diferentes ordenações mas, em geral, os sistemas reais possuem uma numeração

que leva a grafos diferentes.

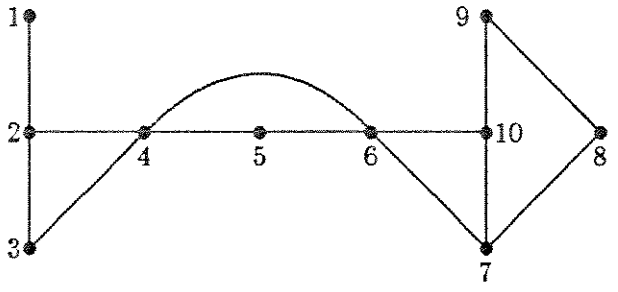


Figura C1 - Rede 10 nós

C.2 Ordenação Mínimo Grau (*Minimum Degree*, MD)

Esta ordenação é também conhecida como esquema 2 de Tinney [35] e baseia-se no grau de cada nó do sistema (número de nós vizinhos com ligação), atualizado em cada passo de fatoração da matriz.

É necessário definir os seguintes vetores:

NSEQ(i) - posição do nó i no final da ordenação. Inicializado em zero, significando que o nó i ainda não foi eliminado.

NGRAU(i) - grau do nó i. Inicializado com valor inteiro igual ao grau do nó na rede original.

Considerando-se uma rede com nb nós, tem-se:

ALGORITMO MD

1. Faça k=1;
2. Escolha um nó com nseq(i)=0 e com ngrau(i) mínimo. Faça nseq(i)=k;
3. Se $k = nb$, fim;
4. Para cada nó j vizinho de i com nseq(j)=0, faça ngrau(j)=ngrau(j)-1;
5. Para cada par de nós (m,n) vizinhos de i, mas não vizinhos um do outro, com nseq(m)=nseq(n)=0, crie uma nova ligação entre m e n.
   Faça ngrau(m)=ngrau(m)+1 e ngrau(n)=ngrau(n)+1;

6. Faça k=k+1 e volte ao passo 2.

Na Figura C2 mostra-se a matriz L resultante desta ordenação e o grafo dos caminhos de fatoração com indicação de profundidade, para a rede da Figura C1.

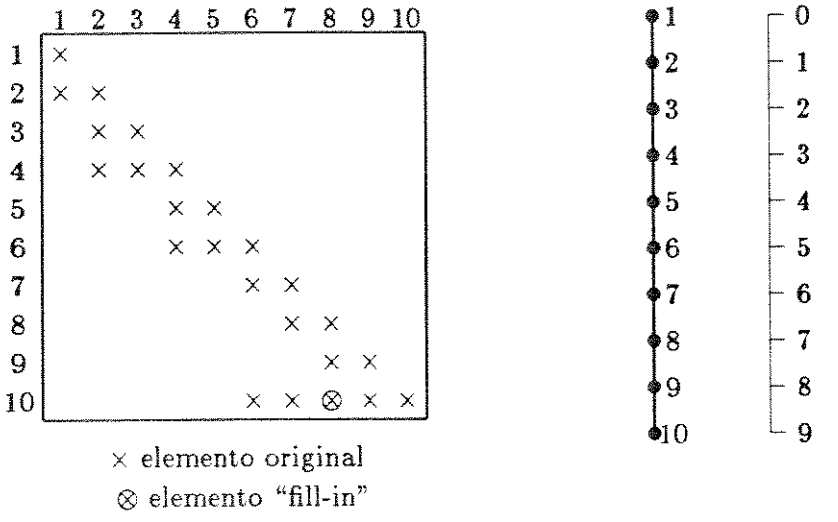


Figura C2 - Estrutura da matriz L e grafo de fatoração depois da ordenação MD

C.3 Ordenação Mínima Profundidade (*Minimum Length*, ML)

Esta ordenação leva em consideração a profundidade ("length" ou "depth") de cada nó no grafo de fatoração como critério de escolha, sendo que foi proposta por [7] associada à ordenação MD. Aqui mostra-se inicialmente a ordenação ML isolada por motivo de facilidade de apresentação, pois esta ordenação não apresenta nenhuma caracteristica vantajosa quando aplicada à redes reais.

A ordenação ML é muito simples de ser implementada e a profundidade de cada nó muito fácil de se atualizar durante a ordenação. No final desta ordenação os nós de uma mesma profundidade aparecem numerados consecutivamente, o que é um detalhe que pode ser util. Para mostrar este algoritmo é necessário definir o seguinte vetor:

NPROF(i) - vetor que contém a informação da profundidade do nó i em um caminho de fatoração. Inicialize com zero.

ALGORITMO ML

1. Faça k=1;
2. Escolha um nó com nseq(i)=0 e com nprof(i) mínimo. Faça nseq(i)=k;
3. Se $k = nb$, fim;
4. Para no cada j vizinho de i com nseq(j)=0 e nprof(j) $\leq$ nprof(i), faça nprof(j)=nprof(i)+1;
5. Para cada par de nós (m,n) vizinhos de i, mas não vizinhos um do outro, com nseq(m)=nseq(n)=0, crie uma nova ligação entre m e n.
6. Faça k=k+1 e volte ao passo 2.

Na Figura C3 mostra-se a matriz L resultante desta ordenação e o grafo dos caminhos de fatoração com indicação de profundidade para a rede da Figura C1.

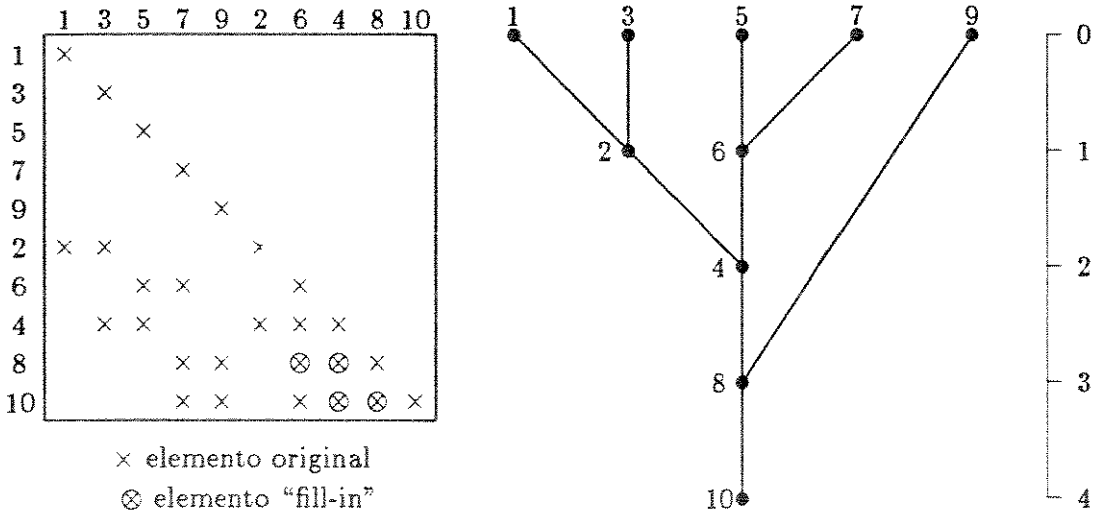


Figura C3 - Estrutura da matriz L e grafo de fatoração depois da ordenação ML

No algoritmo MD nota-se que no passo 2 aparece um grande número de nós com o mesmo grau, e o desempate é feito arbitrariamente (por exemplo, escolhendo-se o primeiro da lista). Esta forma de desempate não possui grande influência no número de "fill-ins", mas influencia bastante o grafo dos caminhos de fatoração. Desta forma têm sido propostas associações das ordenações MD e ML buscando melhores desempenhos.

C.3.1 Ordenação Mínimo Grau, Mínima Profundidade (MD-ML)

Neste algoritmo aplica-se o critério da mínima profundidade no grafo de fatoração, para escolher o nó quando há empate no grau (passo 2 do algoritmo MD) [7]. Assim tem-se:

ALGORITMO MD-ML

1. Faça k=1;
2. Escolha um nó com nseq(i)=0 e com ngrau(i) mínimo. Caso mais de um nó satisfizer esta condição, opte pelo nó com nprof(i) menor. Faça nseq(i)=k;
3. Se $k = nb$, fim;
4. Para cada nó j vizinho de i com nseq(j)=0, faça ngrau(j)=ngrau(j)-1, e se nprof(i) $\geq$ nprof(j), então faça nprof(j)=nprof(i)+1;
5. Para cada par de nós (m,n) vizinhos de i, mas não vizinhos um do outro, com nseq(m)=nseq(n)=0, crie uma nova ligação entre m e n.
   Faça ngrau(m)=ngrau(m)+1 e ngrau(n)=ngrau(n)+1;
6. Faça k=k+1 e volte ao passo 2.

Na Figura C4 mostra-se a matriz L resultante desta ordenação e o grafo dos caminhos de fatoração com indicação de profundidade para a rede da Figura C1.

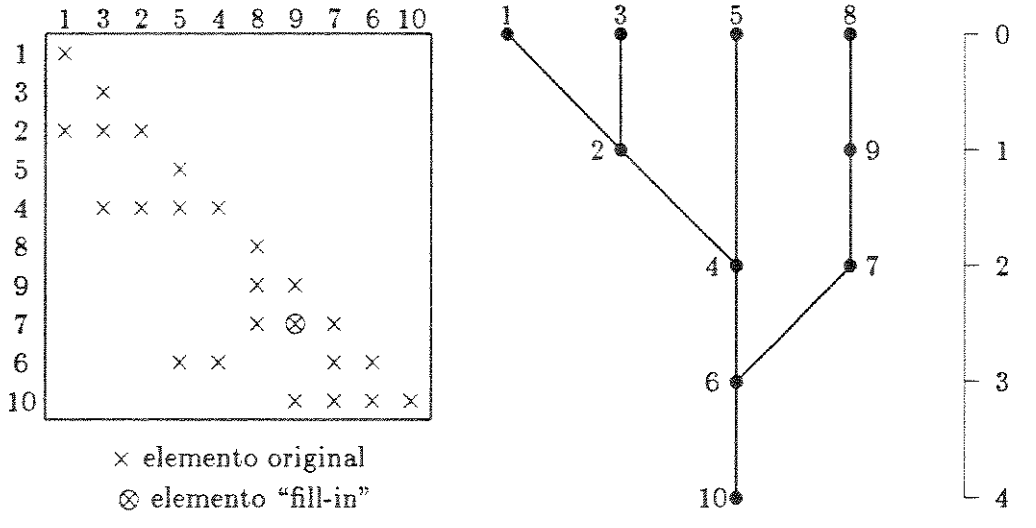


Figura C4 - Estrutura da matriz L e grafo de fatoração depois da ordenação MDML

C.3.2 Ordenação Mínima Profundidade, Mínimo Grau (ML-MD)

Os critérios adotados no passo 2 do algoritmo MD-ML são agora invertidos, ou seja:

ALGORITMO ML-MD

1. Faça k=1;
2. Escolha um nó com nseq(i)=0 e com nprof(i) mínimo. Caso mais de um nó satisfizer esta condição, opte pelo nó com ngrau(i) menor. Faça nseq(i)=k;
3. Se $k = nb$, fim;
4. Para cada nó j vizinho de i com nseq(j)=0, faça ngrau(j)=ngrau(j)-1, e se nprof(i) $\geq$ nprof(j), então faça nprof(j)=nprof(i)+1;
5. Para cada par de nós (m,n) vizinhos de i, mas não vizinhos um do outro, com nseq(m)=nseq(n)=0, crie uma nova ligação entre m e n.
   Faça ngrau(m)=ngrau(m)+1 e ngrau(n)=ngrau(n)+1;
6. Faça k=k+1 e volte ao passo 2.

Na Figura C5 mostra-se a matriz L resultante desta ordenação e o grafo dos caminhos de fatoração com indicação de profundidade para a rede da Figura C1.

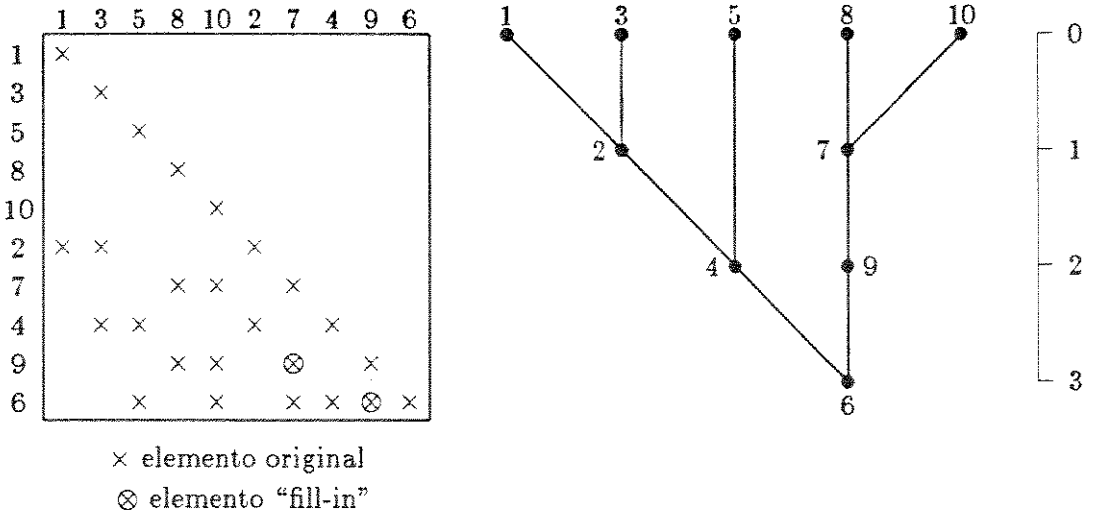


Figura C5 - Estrutura da matriz L e grafo de fatoração depois da ordenação MLMD

Deve ser observado que esta ordenação também fornece os nós de uma mesma profundidade numerados consecutivamentes, sendo que eles são ordenados da menor para a maior profundidade.

C.4 Ordenação Mínimo Número de Predecessores (MNP)

Observando-se a relação existente entre o grafo de fatoração e a matriz L^{-1} (já mostrada no Apêndice B), nota-se que reduzir "fill-ins" em L^{-1} é fazer com que cada nó tenha menos predecessores num grafo de fatoração, conseguindo-se portanto grafos mais apropriados para uso com método de vetores esparsos. Porém, perde-se o controle dos "fill-ins" em L, e isto pode comprometer a ordenação como ocorre com a ML. Esta ordenação foi proposta originalmente associada com a MD [19], visando melhorar a esparsidade de L^{-1} sem comprometer a esparsidade de L.

A fim de implementar esta ordenação, definem-se os seguintes vetores:

NPRE(i) - número de nós em um caminho de fatoração que precede o nó i. Inicializar com *um* (para incluir o próprio nó).

NFRO(i) - variável que assume o valor *zero* se o nó é fronteira (vizinho superior no caminho de fatoração). Inicializar com *um*.

A importância da definição de nós fronteira é para atualizar o conjunto de predecessores dos nós vizinhos que não permanecerão no mesmo caminho de fatoração, considerando a rede preenchida (que é a rede original mais as novas ligações que aparecem devido a eliminação de nós).

ALGORITMO MNP

1. Faça k=1;
2. Escolha um nó com nseq(i)=0 e com npre(i) mínimo. Faça nseq(i)=k e nfro(i)=0;
3. Se k=N, fim. Senão segue;
4. Para cada nó j vizinho de i:
   (a) Se nseq(j)=0, faça npre(j)=npre(j)+npre(i) e ngrau(j)=ngrau(j)-1;
   (b) Se nfro(j)=0, faça nfro(j)=1 e para cada m vizinho de j, da rede preenchida, com nseq(m)=0, faça npre(m)=npre(m)-npre(j);
5. Para cada par de nós (m,n) vizinhos de i, mas não vizinhos um do outro, com nseq(m)=nseq(n)=0, crie uma nova ligação entre m e n.
   Faça ngrau(m)=ngrau(m)+1 e ngrau(n)=ngrau(n)+1;
6. Faça k=k+1 e volte ao passo 2;

Na Figura C6 mostra-se a matriz L resultante desta ordenação e o grafo de fatoração com indicação de profundidade, para a mesma rede exemplo. Deve-se notar que os nós de uma mesma profundidade são numerados consecutivamente, como nas ordenações onde o primeiro critério de escolha é a profundidade.

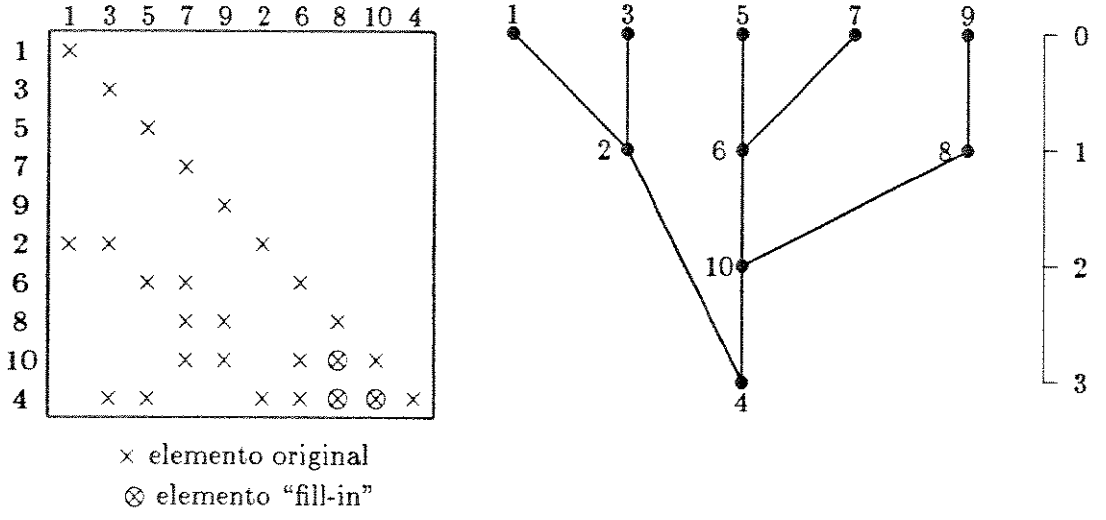


Figura C6 - Estrutura da matriz L e grafo de fatoração depois da MNP

C.4.1 Ordenação Mínimo Grau, Mínimo Número de Predecessores (MD-MNP)

Esta ordenação foi desenvolvida visando melhorar a esparsidade de L^{-1} sem comprometer a esparsidade de L, tornando-se portanto apropriada para explorar a esparsidade de vetores [19]. Utiliza-se a MD com um critério de desempate no passo 2. Este critério leva em consideração o conjunto de nós que precede aqueles em análise; isto é equivalente a localizar o número mínimo de elementos diferentes de zero em L^{-1}, conforme discutido no apêndice B.

ALGORITMO MD-MNP

1. Faça k=1;
2. Escolha um nó com nseq(i)=0 e com ngrau(i) mínimo. Caso mais de um nó satisfizer esta condição, opte pelo nó com npre(i) mínimo. Faça nseq(i)=k e nfro(i)=0;
3. Se k=nb, fim. Senão segue;
4. Para cada nó j vizinho de i:
   (a) Se nseq(j)=0, faça npre(j)=npre(j)+npre(i) e ngrau(j)=ngrau(j)-1;
   (b) Se nfro(j)=0, faça nfro(j)=1 e para cada nó m vizinho de j com nseq(m)=0, faça npre(m)=npre(m)-npre(j);
5. Para cada par de nós (m,n) vizinhos de i, mas não vizinhos um do outro, com nseq(m)=nseq(n)=0, crie uma nova ligação entre m e n.
   Faça ngrau(m)=ngrau(m)+1 e ngrau(n)=ngrau(n)+1;
6. Faça k=k+1 e volte ao passo 2;

Na Figura C7 mostra-se a matriz L resultante desta ordenação e o grafo de fatoração com indicação de profundidade, para a mesma rede exemplo.

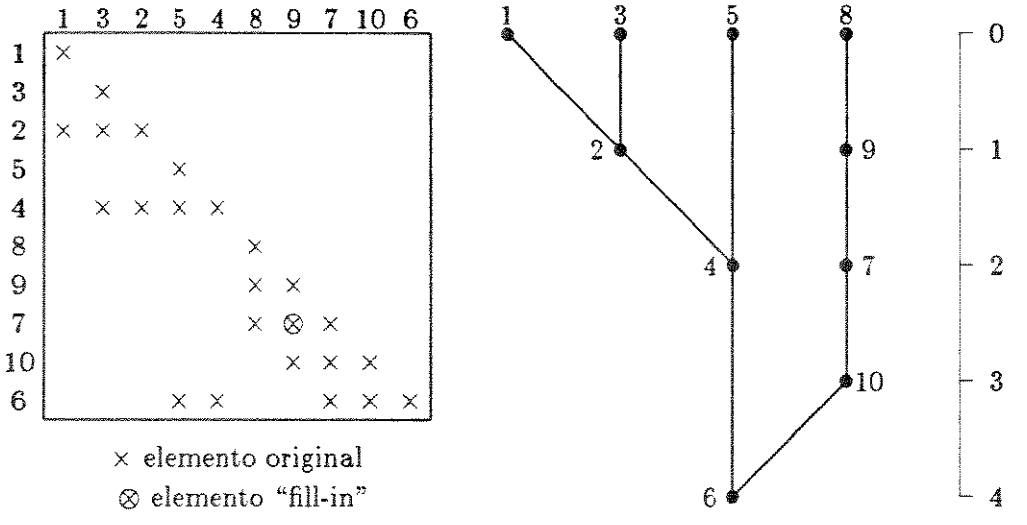


Figura C7 - Estrutura da matriz L e grafo de fatoração depois da MD-MNP

C.4.2 Ordenação Mínimo Número de Predecessores, Mínimo Grau (MNP-MD)

Neste algoritmo aplica-se o critério do mínimo número de predecessores para escolher um nó e quando houver empate usa-se um segundo critério que é o grau. Assim tem-se:

ALGORITMO MNP-MD

1. Faça k=1;
2. Escolha um nó com nseq(i)=0 e com npre(i) mínimo. Caso mais de um nó satisfizer esta condição, opte pelo nó com ngrau(i) mínimo. Faça nseq(i)=k e nfro(i)=0;
3. Se k=N, fim. Senão segue;
4. Para cada nó j vizinho de i:
   (a) Se nseq(j)=0, faça npre(j)=npre(j)+npre(i) e ngrau(j)=ngrau(j)-1;
   (b) Se nfro(j)=0, faça nfro(j)=1 e para cada nó m vizinho de j com nseq(m)=0, faça npre(m)=npre(m)-npre(j);
5. Para cada par de nós (m,n) vizinhos de i, mas não vizinhos um do outro, com nseq(m)=nseq(n)=0, crie uma nova ligação entre m e n.
   Faça ngrau(m)=ngrau(m)+1 e ngrau(n)=ngrau(n)+1;
6. Faça k=k+1 e volte ao passo 2;

Na Figura C8 mostra-se a matriz L resultante desta ordenação e o grafo de fatoração com indicação de profundidade, para a mesma rede exemplo.

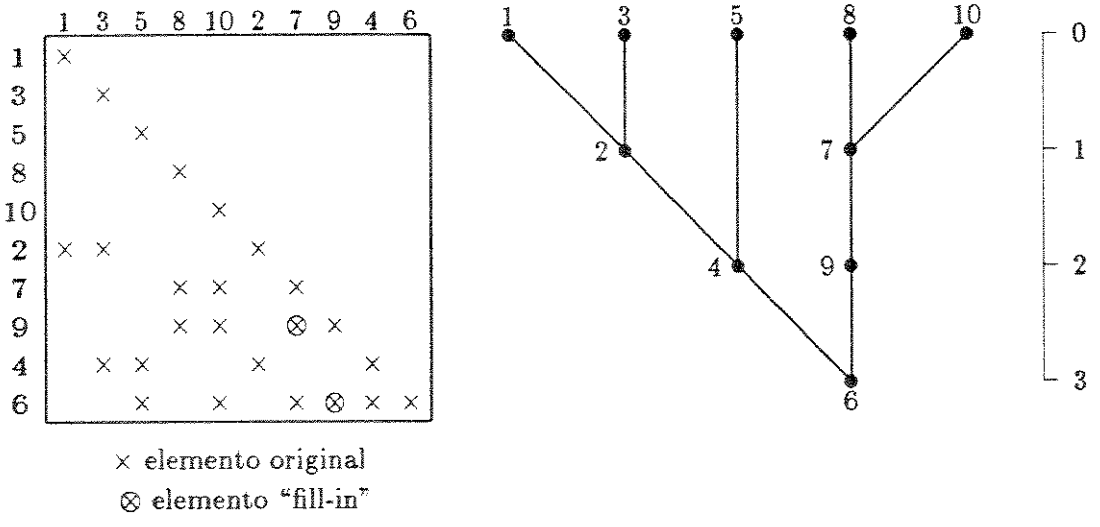


Figura C8 - Estrutura da matriz L e grafo de fatoração depois da ordenação MNP-MD

C.5 Uso das Ordenações

Quando o objetivo é apenas minimizar o número de "fill-ins" destaca-se a ordenação MD, por sua simplicidade e desempenho. A ordenação MD-MNP tem-se mostrado a mais eficiente para aplicação com o método de vetores esparsos, tanto para "fast-forward" e "fast-backward" como para refatoração parcial [19]. Neste tipo de aplicação a ordenação MD-ML também possui bom desempnho. Quando se necessita dos nós numerados consecutivamente na ordem das suas profundidades na árvore de fatoração, associado com grafos menos profundos e podendo-se relaxar um pouco a condição de minimizar "fill-ins", destaca-se a ordenação ML-MD.

As demais ordenações têm sido de pouca utilidade, sendo que as ordenações ML e MNP isoladas por não possuirem nenhum critério que busque minimizar "fill-ins' (nem ao menos como critério de desempate) acabam tendo desempenhos comprometidos tanto em termos de "fill-ins" gerados quanto em termos de profundidade dos grafos dos caminhos de fatorarão. Por exemplo, aplicando-se a ordenação ML ao sistema IEEE 118 são gerados 363 "fill-ins" e obtém-se um grafo com profundidade máxima igual a 20, já a ordenação MNP gera 211 "fill-ins" e um grafo de profundidade máxima igual a 14. Estes desempenhos são bem inferiores aos de outras ordenações mostradas na Tabela 3.2 do Capítulo 3.

Apêndice D

Implementação da Solução Direta

D.1 Considerações gerais

Neste apêndice mostra-se primeiramente a forma de armazenar compactamente as matrizes e a forma de solução para computadores sequenciais. Isto é feito com objetivo de mostrar mais claramente as mudanças necessárias para computadores com multiprocessadores. Em seguida, apresentam-se os algoritmos paralelos, usando W_{up} ou não, que rodam em cada processador.

D.2 Solução sequencial

Armazenamento compacto

O armazenamento compacto de uma matriz $A = LDU$ simétrica, esparsa e de dimensão $nb \times nb$ é feito guardando apenas os elementos diferentes de zero correspondentes a apenas um dos triângulos (matriz L ou U). Considere-se, por exemplo, somente L. Neste caso os elementos diferentes de zero em uma coluna de L terão seus simétricos na correspondente linha de U. Isto permite que, com apenas um conjunto de apontadores, obtenham-se os elementos de L e U, tendo-se armazenado apenas uma delas.

Considere-se que os elementos diferentes de zero da matriz L são armazenados em um vetor CE de comprimento max, onde max deve ser tal que permita, no mínimo, armazenar todos os elementos originais mais os elementos "fill-ins". No vetor CE são armazenados apenas os elementos diferentes de zero fora da diagonal, pois os elementos da diagonal são armazenados no vetor DE, que deve ter dimensão igual a nb.

A fim de localicar corretamente os fatores triangulares no vetor CE é necessário agora um conjunto de vetores apontadores que indiquem a que linha e coluna cada elemento de CE pertence. Assim, seguindo o proposto por [40], definem-se os seguintes

apontadores:

LCOL(m) - vetor de dimensão nb, que contém na posição k a informação de onde se encontra o primeiro elemento não-zero da coluna k de L, no vetor CE — por exemplo, l=lcol(k), x=CE(l) significa que x é o primeiro elemento da coluna k. Quando for encontrado um zero em LCOL significa que aquela é a última coluna da sequência de fatoração, e portanto só possui elemento na diagonal.

ITAG(m) - vetor de dimensão max, que indica o índice da linha do elemento de CE — seguindo o exemplo, i=itag(l) significa que o elemento x da coluna k pertence a linha i.

LNXT(m) - vetor de dimensão max, que indica em que posição de CE encontra-se o próximo não zero da coluna k, se houver — se ln=lnxt(l) for igual a zero significa que x é o último elemento da coluna k; se diferente de zero indica que o próximo elemento da coluna k está em CE(ln).

Considere-se agora a matriz L da Figura D1, originada da rede da Figura C1 depois de uma ordenação ML, onde se deve notar que o vetor NSEQ, definido no Apêndice C, contém exatamente a ordem de fatoração das linhas/colunas.

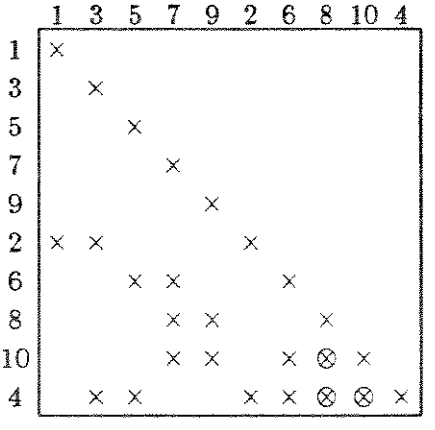


Figura D1 - Estrutura da matriz L

Os apontadores para esta matriz exemplo são mostrados na Tabela D1, considerando que os elementos da diagonal já estão armazenados em outro vetor DE.

Tabela D1 - Vetores apontadores e elementos da matriz L da Figura D1.

i	NSEQ(i)	LCOL(i)	ITAG(i)	LNXT(i)	CE(i)
1	1	1	2	0	×
2	3	2	4	0	×
3	5	3	2	4	×
4	7	0	4	0	×
5	9	5	6	6	×
6	2	7	4	0	×
7	6	9	10	8	×
8	8	12	4	0	×
9	10	14	6	10	×
10	4	16	8	11	×
11	–	–	10	0	×
12	–	–	10	13	⊗
13	–	–	4	0	⊗
14	–	–	8	15	×
15	–	–	10	0	×
16	–	–	4	0	⊗
17	–	–	–	–	–

Diagrama de blocos para "forward" e "backward"

Da forma como os fatores triangulares foram armazenados a "forward" deve ser feita por coluna e a "backward" por linha. A solução é implementada desta forma denomina-se "forward" convencional e "backward" convencional. Os diagramas de blocos da "forward" convencional e da "backward" convencional, são mostrados nas figuras D2 e D3, respectivamente. Esta é a forma frequentemente utilizada em computadores sequenciais. Para implementação de métodos de vetores esparsos esta forma de armazenamento é particularmente importante, pois permite associar cada coluna de L ou linha de U ao correspondente nó da rede elétrica.

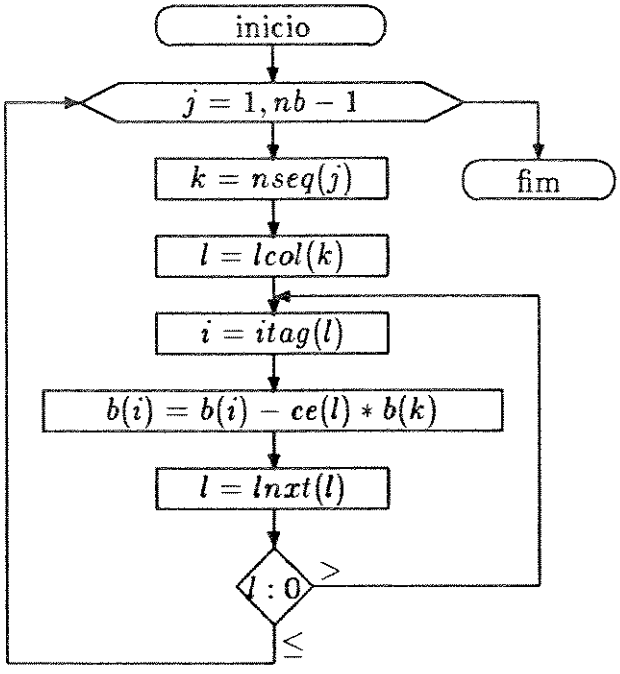


Figura D2 - Diagrama da substituição "forward"

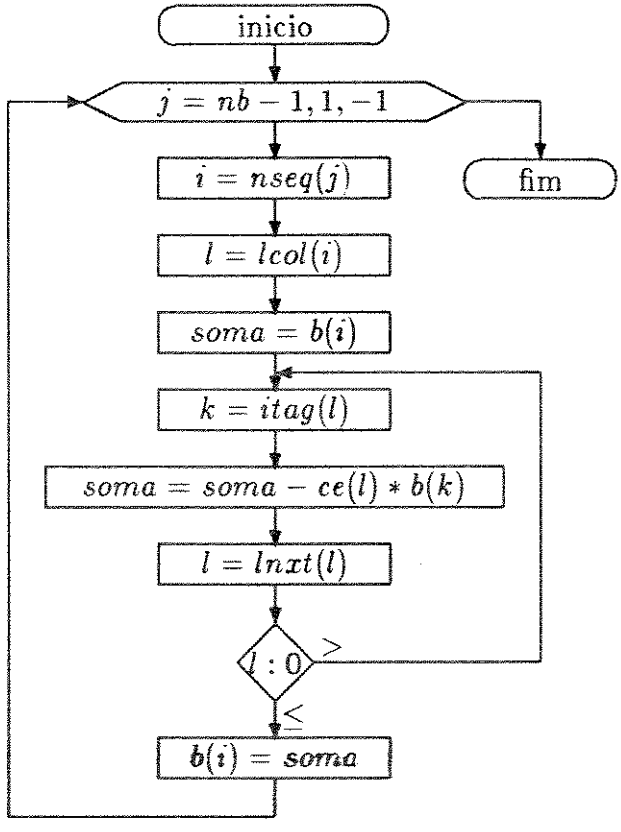


Figura D3 - Diagrama da substituição "backward"

D.3 Solução paralela

Importante

Nos diagramas dos algoritmos paralelos, o comando *barreira* significa que para prosseguir é necessário uma atualização dos valores do vetor solução. Assim quando utiliza-se um modelo computacional de troca de mensagens, o processador ao encontrar a *barreira* deve difundir os valores do vetor solução calculados por ele, para os demais processadores ("broadcast"), e esperar até que receba os valores calculados por todos os demais processadores. Já quando usa-se um modelo computacional de memória partilhada (ou híbrida, sendo o vetor solução uma variável global), o processador ao encontrar a *barreira* deve interromper o processamento até que chegue um comando de prosseguimento, que é emitido quando todos os processadores se acham na *barreira*.

D.3.1 Solução paralela sem usar W_{up}

Armazenamento compacto

Uma solução paralela como a proposta neste trabalho — que consiste em fazer a "forward" e a "backward" por linha — exige alterações na estrutura dos apontadores da matriz esparsa armazenada compactamente. Agora é necessário fazer a busca primeiro na linha da matriz L, e depois obter informações da coluna do elemento. Portanto, deve-se criar um novo conjunto de apontadores para informar as linhas e colunas dos elementos da matriz triangular inferior L. Como a "backward" já era processada por linha, a estrutura apresentada anteriormente serve perfeitamente.

Como a solução paralela deve ser processada por partições, é necessário ainda indicar a que partição cada elemento pertence. Isto pode ser obtido diretamente do vetor NPROF, também definido no Apêndice C, porém este procedimento não é muito eficiente em termos de programação. Desta forma, propõe-se que a matriz L seja armazenada por linha e por partições e para isto definem-se os novos seguintes apontadores:

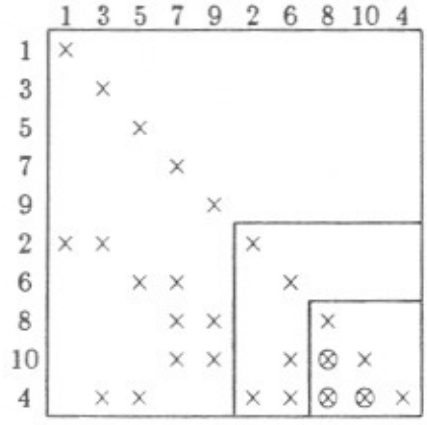
LROW(m,n) - matriz com *nb* linhas e *np* (número de partições) colunas, com funções idênticas ao vetor LCOL — por exemplo, l=LROW(k,n) indica que x=CE(l) é o primeiro elemento da linha *k* de L da *n-ésima* partição. Ao contrário de LCOL, LROW poderá conter um elemento igual a zero e isto indica que na partição não existe nenhum elemento não-zero na linha.

ITAGN(m) - vetor de dimensão *max*, com funções idênticas a ITAG, indicando agora a qual coluna o não-zero pertence.

LNXTN(m) - vetor de dimensão *max*, com funções idênticas a LNXT, porém um

elemento igual a zero em LNXTN agora indica que o elemento de CE é o último da linha na respectiva partição.

A Figura D4 mostra a matriz L da Figura D1, com três partições. Os novos apontadores são mostrados na Tabela D2.



× elemento original

⊗ elemento "fill-in"

Figura D4 - Estrutura da matriz L

Tabela D2 - Novos apontadores para armazenar L por linha e por partição.

i	NSEQ(i)	LROW(i,1)	LROW(i,2)	LROW(i,3)	ITAGN(i)	LNXT(i)	CE(i)
1	1	0	0	0	1	3	×
2	3	1	0	0	2	8	×
3	5	0	0	0	3	0	×
4	7	4	2	0	3	7	×
5	9	0	0	0	4	0	⊗
6	2	6	0	0	5	10	×
7	6	0	0	0	5	0	×
8	8	11	0	0	6	0	×
9	4	0	0	0	6	13	×
10	10	12	9	5	7	0	×
11	–	–	–	–	7	14	×
12	–	–	–	–	7	15	×
13	–	–	–	–	8	0	⊗
14	–	–	–	–	9	0	×
15	–	–	–	–	9	0	×
16	–	–	–	–	–	–	–
17	–	–	–	–	–	–	–

Diagramas de blocos para "forward" paralela e "backward" paralela

Agora a "forward" e a "backward" são feita por linha e por partição, sendo necessário definir um vetor auxiliar — NELL(np) - de dimensão igual ao número de partições — para indicar quantos nós existem em cada partição. Ademais as seguintes informações para o processamento paralelo são necessárias:

- ntot - número total de processadores disponíveis no momento;
- npro - número do processador (1,2,3,4,...)

Observe que na solução "forward" as primeiras $n-1$ partições fazem atualização no próprio vetor b, já para a última partição é necessário o uso de outro vetor (z) para processar paralelamente a solução. Desta forma, parte dos resultados da "forward" aparecem no vetor b e a parte dos nós da última partição aparecem no vetor z, como mostra a Figura D5. Assim, a divisão pela diagonal é feita colocando todos os resultados atualizados no vetor z, como mostra a Figura D6. Na "backward" paralela surge o mesmo problema da última partição como na "forward", sendo que agora utiliza-se o vetor x para saída da solução final. A Figura D7 mostra que no processamento da última partição usam-se valores de z para atualizar o vetor x e nas demais partições trabalha-se com o próprio vetor x.

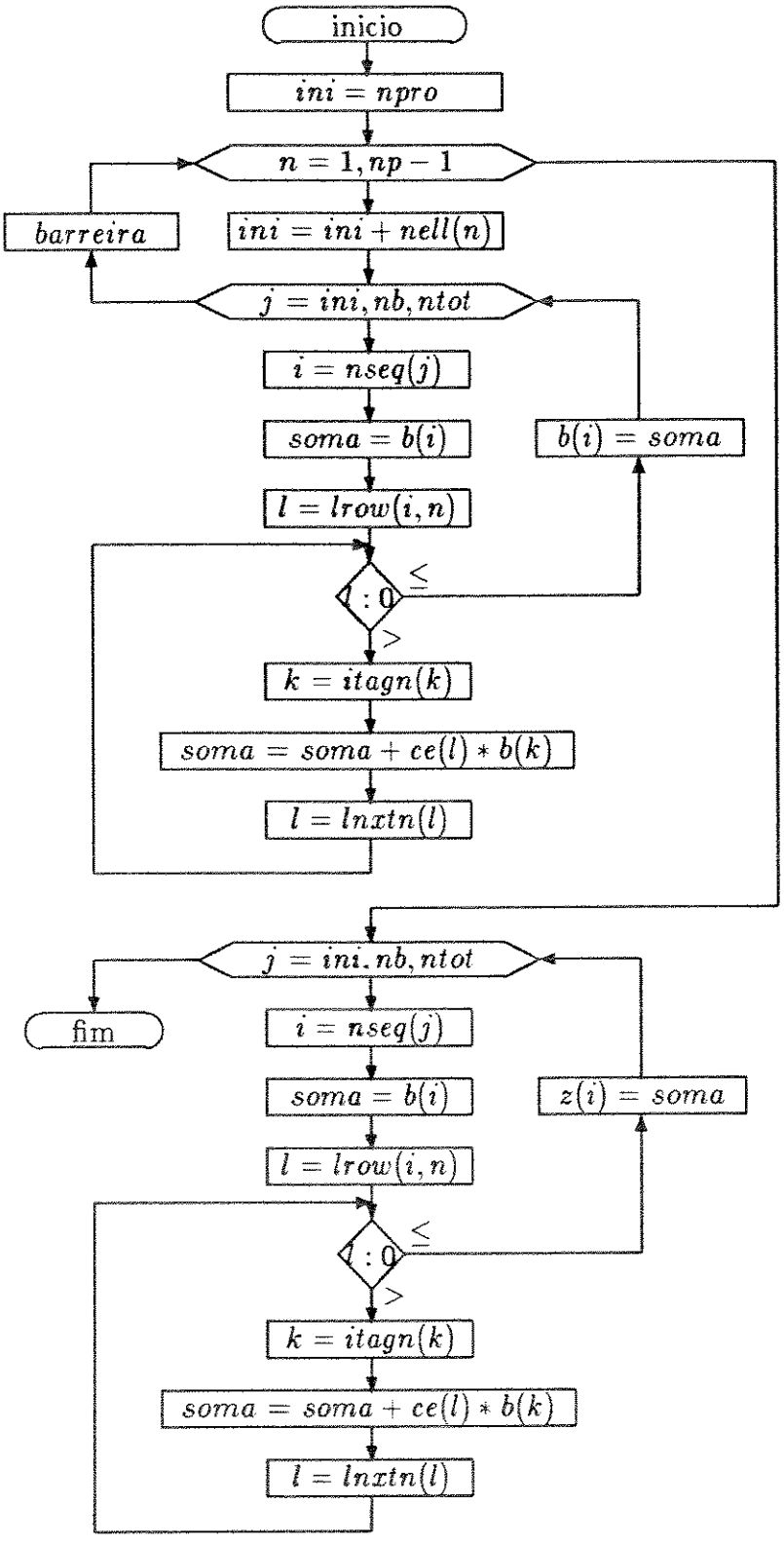


Figura D5 - Diagrama da substituição "forward" paralela sem W_{up}

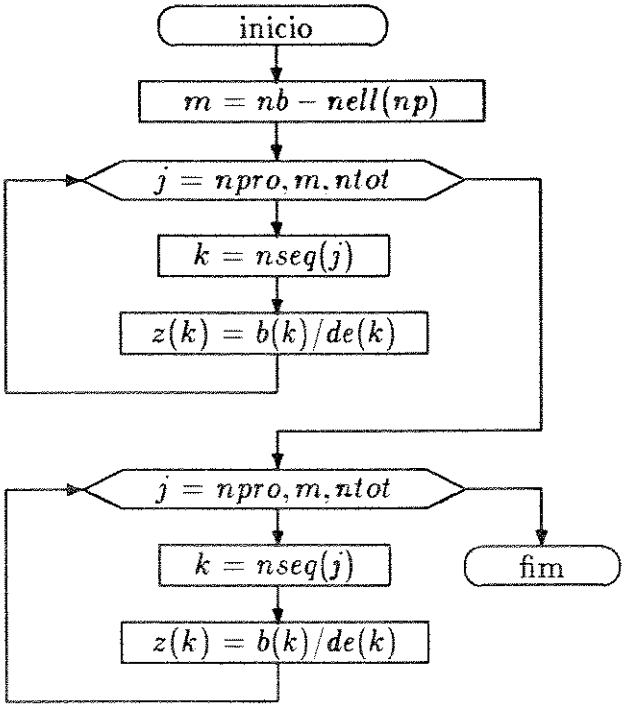


Figura D6 - Diagrama da divisão pela diagonal

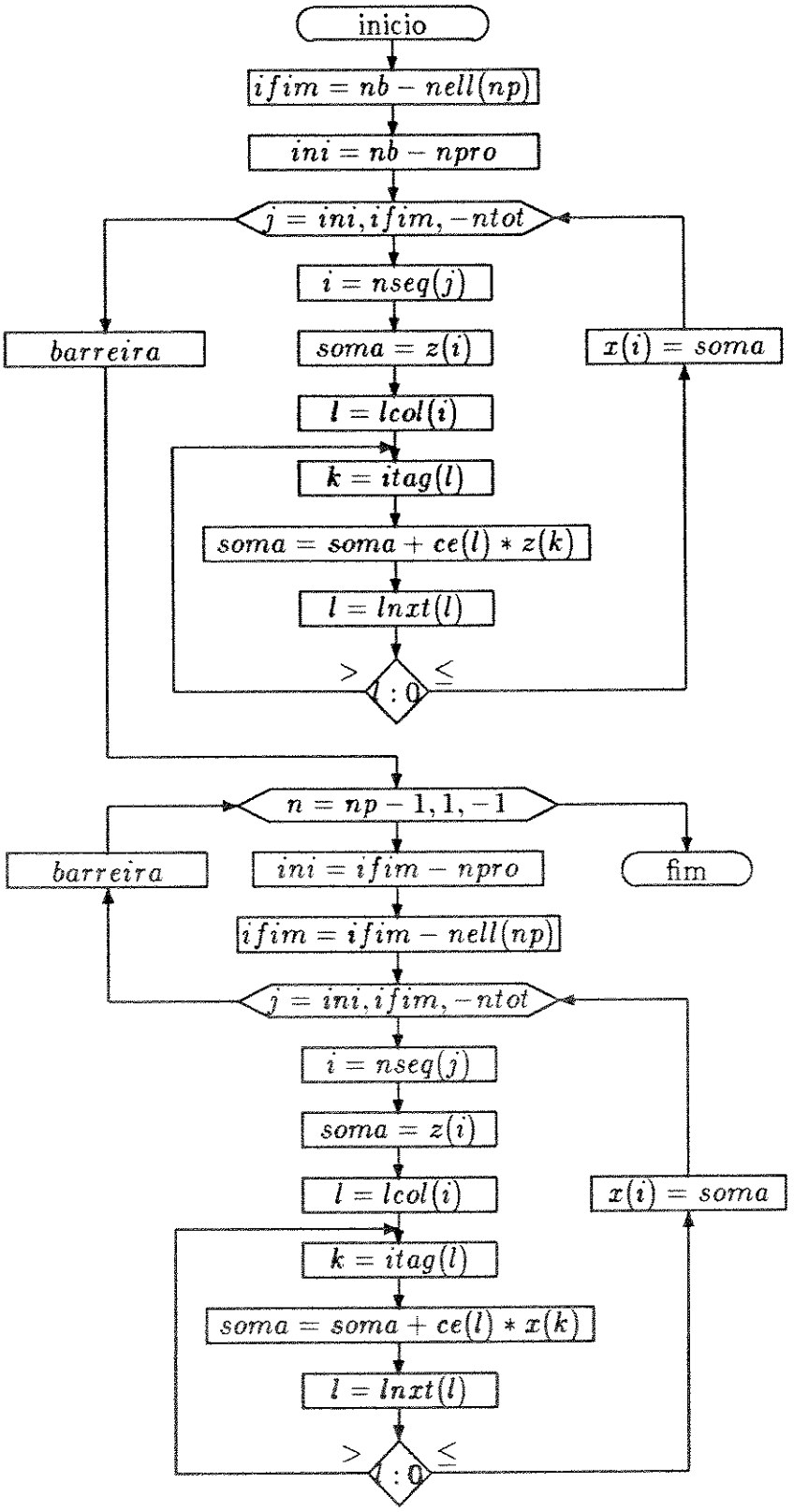


Figura D7 - Diagrama da substituição "backward" paralela sem W_{up}

D.3.2 Solução paralela calculando-se W_{up}

Armazenamento compacto

O armazenamento dos fatores das $n-1$ primeiras partições é feito exatamente como para a solução anterior. A última partição agora é uma matriz cheia e terá seu armazenamento na matriz W_{up} que deverá ser dimensão pelos menos igual aos número de nós da última partição. Para o exemplo, a matriz W_{up} é 3×3.

Diagramas de blocos para "forward", divisão pela diagonal e "backward"

A Figura D8 mostra a etapa "forward" para as $n-1$ primeiras partições, sendo o resultado acumulado no próprio vetor b. Na Figura D9 mostra-se a etapa de divisão pela diagonal para os nós das $n-1$ primeiras partições e a multiplicação da matriz W_{up} pelo vetor b, sendo que o resultado destas operações é colocado no vetor z. Na Figura D10 mostra-se as operações da "backward", sendo que o resultado é colocado no vetor x.

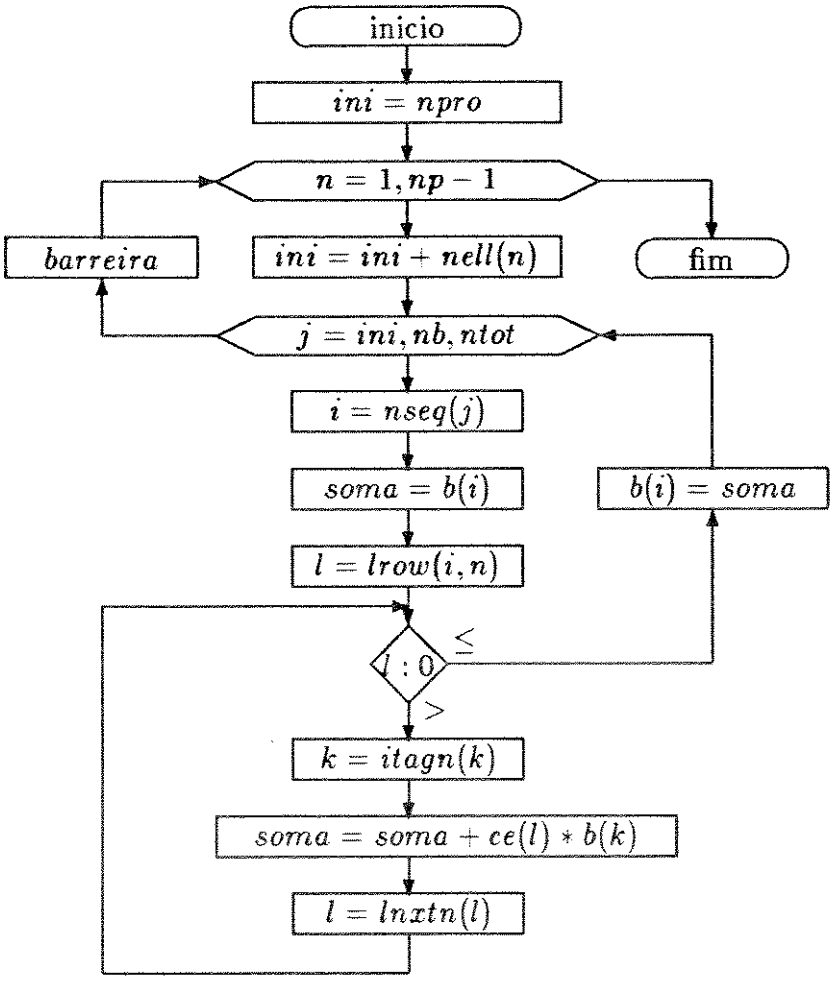


Figura D8 - Diagrama da substituição "forward" paralela com W_{up}

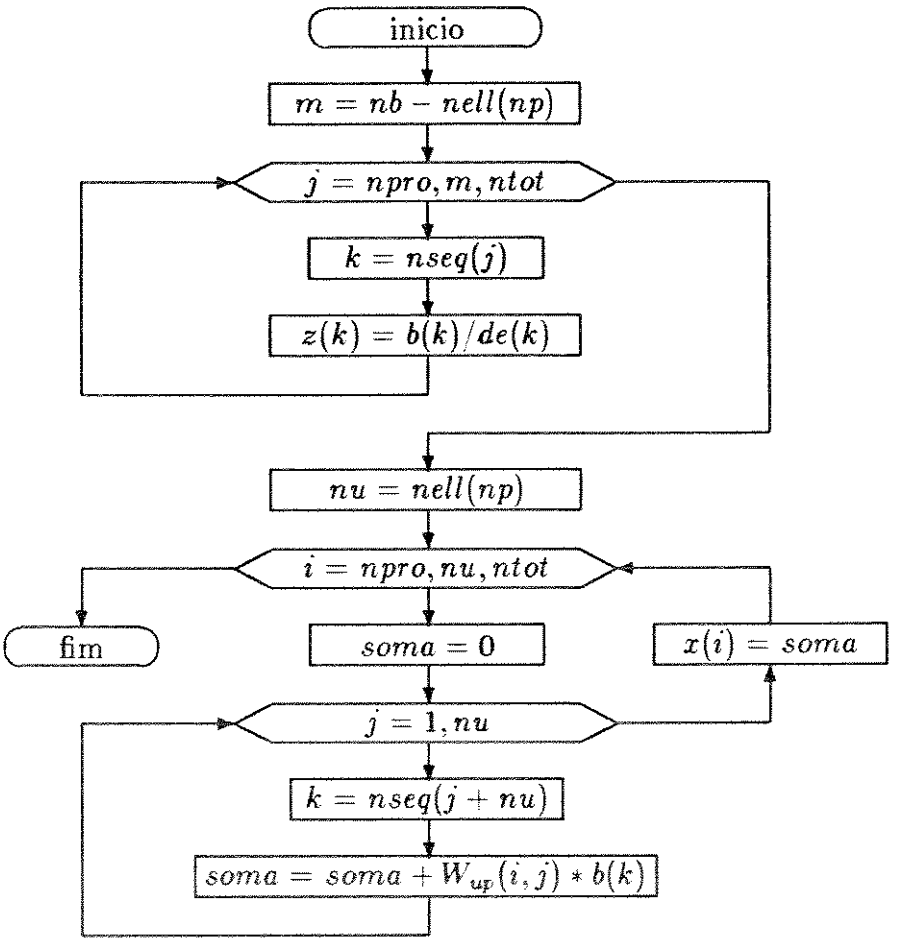


Figura D9 - Diagrama da divisão pela diagonal e multiplicação de W_{up} por b

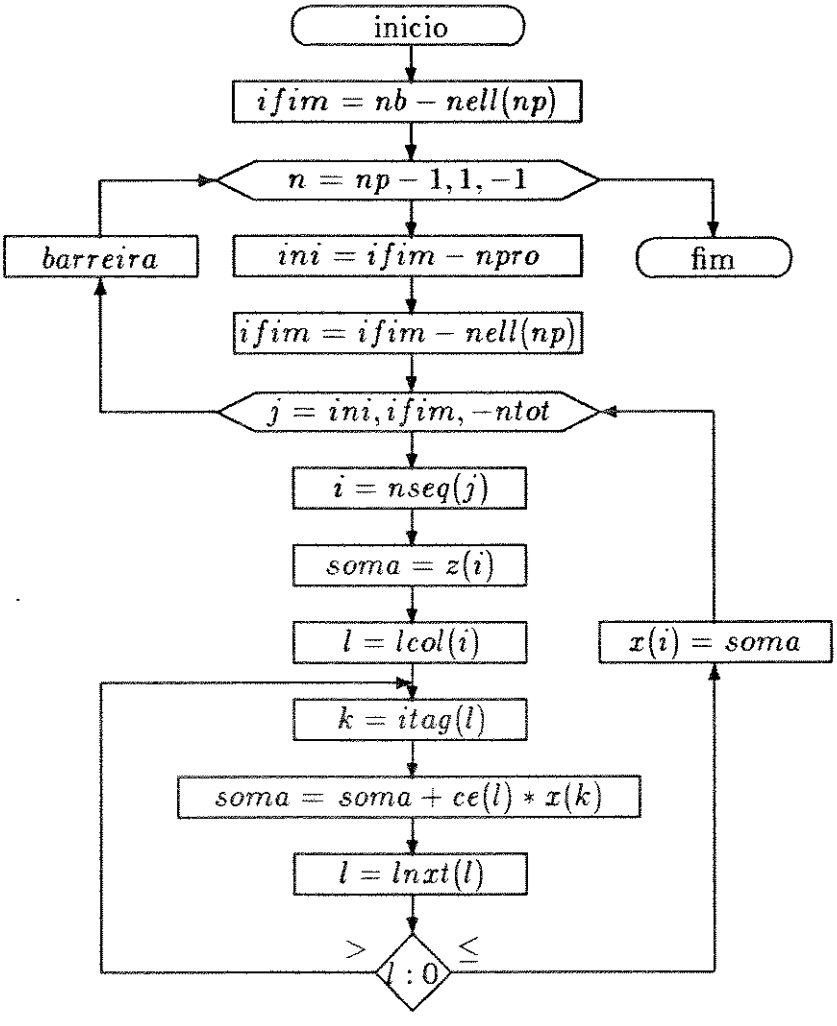


Figura D10 - Diagrama da substituição "backward" paralela com W_{up}

Apêndice E

"Paper" a ser Apresentado em Congresso do IEEE

A seguir mostra-se uma cópia do "paper" a ser apresentado no IEEE 1991 Summer Power Meeting.

A W-MATRIX METHODOLOGY FOR SOLVING SPARSE NETWORK EQUATIONS ON MULTIPROCESSOR COMPUTERS

A. Padilha
UNESP-Ilha Solteira-Brazil

A. Morelato
UNICAMP-Campinas-Brazil

Abstract: This paper describes a methodology for solving efficiently the sparse network equations on multiprocessor computers. The methodology is based on the matrix inverse factors (W-matrix) approach to the direct solution phase of $Ax = b$ systems. A partitioning scheme of W-matrix , based on the leaf-nodes of the factorization path tree, is proposed. The methodology allows the performance of all the updating operations on vector b in parallel, within each partition, using a row-oriented processing. The approach takes advantage of the processing power of the individual processors. Performance results are presented and discussed.

Keywords: *Power Flow, Stability, Parallel Processing, Sparsity, Direct Solutions.*

INTRODUCTION

It is a matter of fact that low-cost multiprocessor computers are now available featuring supercomputer-like performances. For instance, new powerful multiprocessor machines have been announced which can offer peak performance ranging from 480 to 7600 Mflops with 8-128 processors interconnected using a message passing architecture [1] or ranging from 2 to 130 Gflops, up to about 65,000 processors, using a shared memory architecture [2].

Even though peak performance is not straightforward to achieve because it depends on how the parallel algorithms are mapped to the parallel architectures, there is no doubt that the power industry may be one of the first to take advantage of this computer technology. Many power systems problems require the repetitive solution of a large set of sparse linear equations and this task represents the most time consuming part of the overall solution. For this reason, the effort for solving efficiently the linear network equations on multiprocessor hardware is one of most promising ways to take advantage of parallel processing in power systems applications.

It has been recognized that the parallel solution of sparse network equations is not a smooth task, because the great number of precedence relations among arithmetic operations leads the solution processing to have a large amount of idle time [3,4]. If one processor has assigned very simple tasks, a multiplication for example, the precedent relationship web will make the communication and/or synchronization overheads increase a lot and so offset the advantages of parallel processing.

This paper describes a methodology for decomposing the repeat solution process of the equation $Ax = b$ into independent tasks to be done in parallel. The approach allows the task unit to be assigned to each processor to be somewhat more coarse-grained than elemental arithmetic operations. Therefore, it is possible to take advantage of the powerful processors that constitutes a multiprocessor environment and, at the same time, to reduce the amount of communication and synchronization overheads involved in the solution process.

The methodology presented here is based on the matrix inverse factors (W-matrix) approach to the direct solution phase of $Ax = b$ systems, as proposed in [5]. The W-matrix approach does not seem to be advantageous for processing of the repeat solution on conventional serial machines, as shown in the discussion of [6]. On the other hand, it seems very promising for parallel processing methods, considering its properties of decoupling the elementary operations.

Problem Formulation

A set of linear network equations is usually expressed as:

$$Ax = b \tag{1}$$

where A is a nonsingular matrix of order n, the vector x is the unknown solution and b is the given independent vector, which will be considered a full vector. The matrix A is assumed to be sparse and symmetric.

The standard solution method for (1) comprises a sparsity-oriented LDU decomposition of A followed by forward, diagonal and backward operations on the vector b:

$$A = LDU \tag{2}$$

$$x = U^{-1}D^{-1}L^{-1}b \tag{3}$$

where L and $U = L^t$ are lower and upper triangular matrices with unity elements in the diagonals and D is a diagonal matrix. The L-matrix can be expanded as $L = L_1L_2 \ldots L_n$ such that each L_i is an identity matrix except for the i^{th} column, which contains the column i of L. The computation of L and U is preceded by ordering of rows and columns of the matrix, usually to minimize the number of nonzero elements.

The solution process described by equation (3) may be seen as an ordered sequence of updating operations applied to the right hand side vector. In this paper the problem to be tackled is how to perform efficiently the solution (3) on a multiprocessor environment in such a manner to take advantage of its

potential computing speedup and so to realize the solution in the least time. More specifically, the challenges are: how to decompose the repeat solution process into independent tasks which have an appropriate size, and how to schedule the tasks on the processors in such a way as to reduce the communication and synchronization overheads, achieving a minimum solution time.

BASIS OF THE METHODOLOGY

In this section the foundations on which the proposed approach is based will be presented briefly. The 20-node system [7], whose network is shown in Figure 1, will be taken as a tutorial example with no loss of generality.

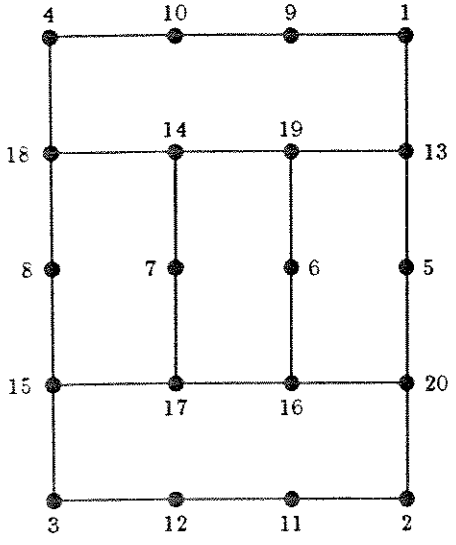


Figure 1: The 20-node example network

Factorization Paths and Ordering Schemes

As is well known, the precedence relations among operations on columns and rows to perform the LDU decomposition and forward/backward solutions can be depicted by the factorization path graph [7]. It is also known that different ordering schemes can change the shape of the path tree and alter the number of fill-in elements of L and L^{-1}. Figure 2 shows the path tree of the example network obtained from the L-matrix structure using the ML-MD ordering scheme [8]. Using the MD-MNP scheme [9], the path tree for the example network is, by chance, exactly the same as the classical MD scheme.

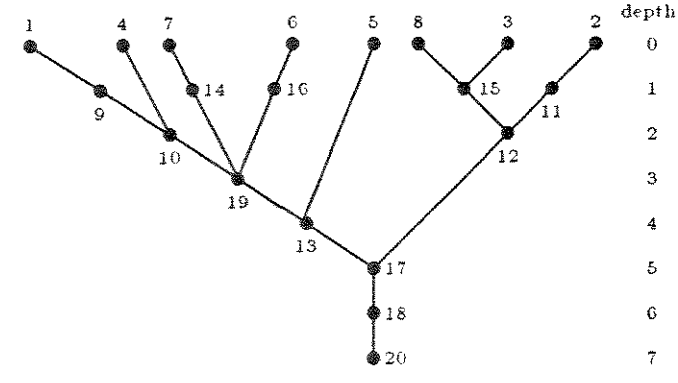


Figure 2: Path tree after ML-MD ordering

A node of the factorization path tree is to be said a *leaf* if it has no predecessor. The *depth* (or *level*) of a node can be defined as the maximum number of nodes in the tree which precede it. Both concepts can be associated and so it can be said that a *leaf of depth i* is a leaf provided that all the nodes of depth *i-1* have been reduced. For example, in the path tree illustrated in Figure 2 the nodes {1,2,3,4,5,6,7,8} are leaves of depth 0 and the nodes {9,11,14,15,16} are leaves of depth 1. Table A.1 in Appendix A shows the number of leaves at the different depths obtained from two test networks and using several ordering criteria.

The performance of the methodology presented in the paper is not significantly affected by the ordering schemes, though some schemes can be more friendly than others. The ML-MD scheme provides an ordered sequence of nodes that matches automatically the leaves of successive depths. Otherwise, the MD-MNP scheme requires extra work after ordering to rearrange the nodes.

Partitioning Scheme

Assuming that the LDU decomposition has been carried out in an appropriate way, the goal is to perform in parallel the operations on vector b described by equation (3). This equation can be transformed and expanded according to the idea of reference [5] as:

$$x = W^t D^{-1} W b = W_1^t W_2^t \dots W_n^t D^{-1} W_n \dots W_2 W_1 b \qquad (4)$$

where $W_i = L_i^{-1}$, $i = 1, \dots, n$.

The adjacent matrices W_i can be combined in different ways giving rise to different *partitions* of matrix W. The advantage of this approach is that all multiplications between inverse factors and components of vector b within each partition become independent and can be performed in parallel. Unfortunately, the necessary additions to update each component of b cannot all be performed simultaneously. On the other hand, the partitions may be utilized to keep under control the additional fills that can be created in the W-matrix. Reference [5] proposes to perform the direct solution phase by breaking it up into partitions whose number depends on the number of additional fill-in elements allowed within each partition. This partitioning technique has been enhanced in reference [6].

The partitioning scheme proposed in this paper consists of breaking the W-matrix according the depths of the factorization path tree, as follows:

- each partition p is assigned to a depth of the path tree, i. e. all the nodes which are leaves of depth p belong to the partition;
- the assignment proceeds until some previously chosen depth (called *break-in depth*) is achieved and then all the remaining nodes of the tree are gathered into one partition (called *last partition*).

The choice of the *break-in depth* is heuristic but not critical for the solving methodology because the number of leaves

by depth decreases quickly at the top but slowly after a few depths, as shown in Table A.I in the Appendix A. A simple heuristic rule relating the number of nodes in the last partition with the number of available processors may be used, for example.

Figure 3 shows the W-matrix structure from the 20-node example after the application of the partitioning scheme proposed, assuming the path tree ML-MD illustrated in Figure 2 and taking the break-in depth as 2. Note that the matrix is divided into three partitions composed of nodes {1,2,3,4,5,6,7,8}, {9,11,14,15,16} and {10,12,13,17,18,19,20} respectively. Furthermore, note the one additional fill-in element in the last partition whereas there are no additional fills in the two first partitions, as would be expected.

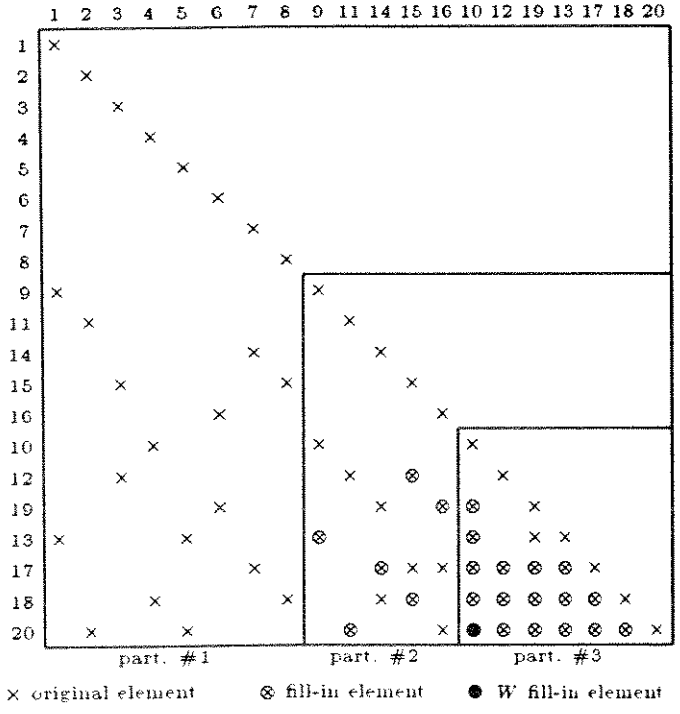


Figure 3: Partitioning of W-matrix

In this scheme, all the information needed to generate the partitions can be obtained straightforwardly from the network factorization path tree. The partitioning algorithm is simple and easy to implement. The elements of W_i matrices, except the last partition, can be obtained directly from L-matrix elements, not requiring extra work.

The proposed scheme guarantees that additional fills will be created only in the last partition. As a matter of fact, if a partition does not have any two nodes which are predecessors of each other, then no additional fills are produced in the partition matrix. This condition can be easily validated remembering what is occurring with the network when the factorization proceeds, node by node, according to the path tree. It should be noted that the condition is sufficient but not necessary. If two nodes of different depths are put together into the same partition, it is not assured that additional fills will be yielded because it may happen that the place has been already filled. This is the case of the element (13,10) in Figure 3.

Reference [10] presents a partitioning technique based on the levels of the factorization path graph but the difference is twofold: the goal to be achieved and the criterion to create a partition. In reference [10] the objective of partitioning is to generate the minimum number of extra fill-in, aiming to minimize the number of elementary arithmetic operations to be performed on a vector computer. Otherwise, our partitioning scheme aims at generating the minimal number of partitions with the maximum number of independent tasks to be assigned to processors on a multiprocessor computer. The idea behind our proposed scheme is to take advantage of the great number of leaf-nodes in the first partitions because they correspond to independent tasks. When the leaf-nodes begin to fade, because the precedence relationships become stronger, then all the remaining nodes are gathered disregarding the extra fills. This action gives rise to new independent tasks in the last partition but the price to be paid is to perform the extra fills.

The last partition concentrates all the additional fills, but it can be shown that those extra fills do not delay the parallel processing of the partition, considering that the lowest row of the partition is always full. Indeed, the number of nonzero elements of the lowest row of the L^{-1} matrix corresponds to the number of nodes which precede the ultimate node in the factorization path tree. Therefore, the time spent to process the lowest row determines the least time to process the last partition.

It is possible to use partitioning schemes that combine nodes from more than one level in a given partition with no extra fills, as shown in [6] and [10]. Particularly, [10] describes a simple but very effective way to find out the proper nodes, although it was valid only for MD ordering. Those procedures can slightly rearrange the nodes belonging to the first partitions but it is not clear that this leads to smallest solution time in general.

Formation of the W-matrices

The computation of the W-matrices elements is straightforward for the first $(p-1)$ partitions because they are equal to the off-diagonal elements of L-matrix with the sign reversed. This is valid providing that all nodes within each partition are at the same level in the factorization tree. The computation of the last partition elements requires some extra work. They might be directly obtained from the multiplication of the adjacent matrices W_i corresponding to the last partition. The effort is reasonable remembering that a matrix W_i is an identity matrix except for the *i-th* column. This is the approach used in the paper. Otherwise, the computation of the W-matrix for the last partition can be done in parallel utilizing, e.g. the technique proposed in [11]. However, the construction of the W-matrices is one step that must be performed only once in the repeat solution process and it may not be worth doing it in parallel. It has been recognized that parallelism is best used for steps that require a significant number of cycles.

TASK SCHEDULING

This section deals with the decomposition of the repeat solution process into independent tasks, i. e. the amount of work which can be done at the same time, in any order, and without communicating with another task. Also is discussed what is the best size of the task to be assigned to each processor on multiprocessor environments. Moreover, the task

scheduling strategy for distributing the tasks among the processors is addressed.

Independent Tasks

The repeat solution comprises essentially an ordered sequence of updating tasks operating on the components of the vector b, where each one is formed by arithmetic operations of multiplication and addition. It is possible to assign each one of these elementary arithmetic operations to a processor in order to exploit the maximum amount of parallelism. However, it is not clear that the solution with the maximum parallelism is always the fastest. The communication or synchronization overheads may increase a lot producing significant delays. Unfortunately, this is the case of the repeat solution process due to the large number of precedence relations among the operations, i. e. the elementary operations are not independent of each other.

The application of the W-matrix approach to the repeat solution allows all multiplications between inverse factors and elements of b to be independent each other - within each partition - and could be carried out in parallel. However, the additions cannot all be performed at the same time. This fact does not mean that there is no parallelism even in the additions but that some of them cannot be performed simultaneously because each one has to take the most updated value of b, which is under processing at the same time. In other words, some additions (not previously known) are not independent and to find out its precedent relationships is time consuming.

It should be noted that the degree of dependence of the updating tasks depends upon whether the solution is performed by columns or by rows. To illustrate this point take the 20-node example system and suppose the forward solution is to be performed by columns (conventional forward). The operations on vector b corresponding to partition 1 are:

task number	arithmetic operations
#1	$b_9 = b_9 + w_{9,1}b_1$
#2	$b_{13} = b_{13} + w_{13,1}b_1$
#3	$b_{11} = b_{11} + w_{11,2}b_2$
#4	$b_{20} = b_{20} + w_{20,2}b_2$
#5	$b_{15} = b_{15} + w_{15,3}b_3$
#6	$b_{12} = b_{12} + w_{12,3}b_3$
#7	$b_{10} = b_{10} + w_{10,4}b_4$
#8	$b_{18} = b_{18} + w_{18,4}b_4$
#9	$b_{13} = b_{13} + w_{13,5}b_5$
#10	$b_{20} = b_{20} + w_{20,5}b_5$
#11	$b_{16} = b_{16} + w_{16,6}b_6$
#12	$b_{19} = b_{19} + w_{19,6}b_6$
#13	$b_{14} = b_{14} + w_{14,7}b_7$
#14	$b_{17} = b_{17} + w_{17,7}b_7$
#15	$b_{15} = b_{15} + w_{15,8}b_8$
#16	$b_{18} = b_{18} + w_{18,8}b_8$

where $w_{i,j}$ is the element (i,j) of partition 1. It can be seen that the additions to update b_{13}, as indicated in tasks #2 and #9, cannot be processed simultaneously to assure correct results and so they are not independent. The same kind of conflict can be found in the rows 15, 18 and 20. In the backward solution there is no problem because the conventional backward is usually performed by rows.

The idea presented here is to gather *by rows* the operations to be performed. In doing this, all the updating tasks within each partition become completely independent of each other. For the example system, the operations on vector b, referring to partition 1, can be grouped as:

task number	arithmetic operations
#1	$b_9 = b_9 + w_{9,1}b_1$
#2	$b_{11} = b_{11} + w_{11,2}b_2$
#3	$b_{14} = b_{14} + w_{14,7}b_7$
#4	$b_{15} = b_{15} + w_{15,3}b_3 + w_{15,8}b_8$
#5	$b_{16} = b_{16} + w_{16,6}b_6$
#6	$b_{10} = b_{10} + w_{10,4}b_4$
#7	$b_{12} = b_{12} + w_{12,3}b_3$
#8	$b_{19} = b_{19} + w_{19,6}b_6$
#9	$b_{13} = b_{13} + w_{13,1}b_1 + w_{13,5}b_5$
#10	$b_{17} = b_{17} + w_{17,7}b_7$
#11	$b_{18} = b_{18} + w_{18,4}b_4 + w_{18,8}b_8$
#12	$b_{20} = b_{20} + w_{20,2}b_2 + w_{20,5}b_5$

Note that there are no conflicts here because each component of b is updated only once within the partition. Therefore, the chunk of updating tasks corresponding to a row of each partition is independent and can be done in parallel. However, the forward and backward processing by rows requires double sets of tables for indices of rows and columns of the lower and upper triangular W-matrices.

Scheduling Strategy

The strategy proposed here is to schedule on each processor the operations corresponding to a row of each partition. It should be kept in mind that multiprocessor environments are equipped with powerful unit processors and then it seems a sound strategy to perform the mult-add elementary operations inside the hardware in order to exploit its computing efficiency. This strategy seeks to match the parallel algorithm to the parallel architecture. The precedent relations - that give rise to delays - are replaced by mult-add operations performed inside the processor node without external communication. Parallelism within a row could be exploited if the processor node is provided with binary adder facilities.

The scheduling strategy should take into account the number of available processors. A simple but efficient strategy consists of assigning several rows to one processor using a wrap mapping style, so that processor zero gets row 1, nproc+1, 2nproc+1, ... , where nproc is the number of available processors. This is the strategy used in taking the results presented further. It is evident that if the number of processors is greater than the number of rows to be performed in a partition, then the maximum load of one processor corresponds to just one row.

It should be noted that the size of the row-oriented tasks is not uniform because the number of elements per row is not the same, ranging from 1 to 10 for the networks simulated. Load unbalancing is not an issue if the goal is the solution in minimum time. The solution time spent to solve a partition is always given by the processor which has assigned the row with the greatest number of elements. It is does not matter if some processors remain idle for a while because they cannot speedup the solution. The solution with the best load balance is not necessarily the fastest. The goal is not to keep busy all

the processors but using the available processors in such a way to achieve the fastest solution.

THE LAST PARTITION

Taking into account that the direct solution will be performed by rows, it becomes advantageous to handle the last partition as an unique one.

According to the partitioning scheme equation (4) can be expressed as:

$$x = W_1^t W_2^t \dots W_n^t D_{p-1}^{-1} D_{l_p}^{-1} W_n \dots W_2 W_1 b \qquad (5)$$

where the diagonal operations D^{-1} were split into D_{p-1}^{-1}, concerning the early $(p-1)$ partitions, and $D_{l_p}^{-1}$, concerning the last partition.

As the operations corresponding to the last partition are not affected by the operations D_{p-1}^{-1}, equation (5) is equivalent to:

$$x = W_1^t W_2^t \dots W_{n-1}^t D_{p-1}^{-1} W_{l_p} W_{n-1} \dots W_2 W_1 b \qquad (6)$$

where $W_{l_p} = W_n^t D_{l_p}^{-1} W_n$ represents the forward, diagonal and backward aggregated operations on the last partition.

In the last partition, the forward, diagonal and backward solutions may be gathered, and so all the operations can be expressed as the product of matrix W_{l_p} by the updated components of vector b. The result is that the tasks of updating the last partition elements of b become independent if they were performed by rows, as the other partitions are. Moreover, the total number of operations is smaller than in the conventional procedure and the number of serial steps is reduced. Though W_{l_p} is a full matrix, its additional fills do not increase the solution time of the last partition because it depends essentially on the number of nonzero elements in the lowest row (it is always full) and the number of processors available. On the contrary, the solution time decreases because the serial steps of forward, diagonal and backward solutions will be performed at the same time as the forward step.

PERFORMANCE RESULTS

The methodology proposed here can be used either on a message-passing or shared-memory architecture, even though the characteristics of the approach are best fitted to shared-memory or hybrid machines (each processor accesses local memory too). In this case, the vector b is stored in the shared memory which can be accessed for every processor through global variables. The access time is the same as the local memory references, neglecting common bus contention. Therefore, the processors are able to update the vector b, within each partition, without communication overhead. Before carrying out the next partition, all the processors involved must overcome one synchronization barrier. As for message-passing machines, the need of broadcasting the updated values of vector b among the processors, before carrying out the next partition, introduces a communication overhead which is inherent to this architecture. Anyway, the coarse-grained parallelism achieved by the presented approach is adequate to minimize the influence of hardware overheads in both architectures.

The performance of the methodology can be evaluated considering the simulation results depicted in Table I and Table II showing the maximum number of mult-add operations corresponding to each serial step of the solution for several numbers of partitions (not counting the operations with diagonal elements). The ML-MD ordering was used. In Table I (IEEE-118 system) it can be seen that the first serial step requires at most 4 mult-add operations, i. e. the greatest number of nonzero elements in one row of the partition is 4. If it is assumed, for simplicity, that the number of processors available is large enough, then the total solution time (including the idle time) to process this step is the time spent to do 4 multiplications and 4 additions. The last partition step is the bottleneck because the lowest row of W_{l_p} is always full. The impact of the last partition could be reduced if the number of partitions is increased, however the increase in the backward operations may offset this gain and increase the total solution time. The number in parentheses denotes the additional fills

Table I - 118 node system.

Serial steps	Partitions 2 (1704)	3 (237)	4 (151)	5 (34)	6 (7)	7 (0)
1 (W_1)	4	4	4	4	4	4
2 (W_2)	–	4	4	4	4	4
3 (W_3)	–	–	4	4	4	4
4 (W_4)	–	–	–	3	3	3
5 (W_5)	–	–	–	–	2	2
6 (W_6)	–	–	–	–	–	2
7 (W_{l_p})	62	35	22	14	10	8
8 (W_6^t)	–	–	–	–	–	8
9 (W_5^t)	–	–	–	–	9	9
10 (W_4^t)	–	–	–	6	6	6
11 (W_3^t)	–	–	6	6	6	6
12 (W_2^t)	–	3	3	3	3	3
13 (W_1^t)	3	3	3	3	3	3
total	69	49	46	47	54	62

Table II - 1729 node system.

Serial steps	Partitions 4 (23088)	5 (9171)	6 (3858)	7 (1886)	8 (981)	9 (570)
1 (W_1)	9	9	9	9	9	9
2 (W_2)	7	7	7	7	7	7
3 (W_3)	5	5	5	5	5	5
4 (W_4)	–	4	4	4	4	4
5 (W_5)	–	–	4	4	4	4
6 (W_6)	–	–	–	4	4	4
7 (W_7)	–	–	–	–	4	4
8 (W_8)	–	–	–	–	–	3
9 (W_{l_p})	221	143	97	72	56	46
10 (W_8^t)	–	–	–	–	–	17
11 (W_7^t)	–	–	–	–	15	15
12 (W_6^t)	–	–	–	10	10	10
13 (W_5^t)	–	–	10	10	10	10
14 (W_4^t)	–	9	9	9	9	9
15 (W_3^t)	7	7	7	7	7	7
16 (W_2^t)	7	7	7	7	7	7
17 (W_1^t)	6	6	6	6	6	6
total	262	197	165	154	157	167

produced in the lower triangular matrix of the last partition. Note that the no-fills condition, attained with 7 partitions, does not produce the speedest solution time. In summary, Table I shows that the direct solution calculations for the IEEE-118 network can be performed in the time equivalent to 46 floating point operations of multiplication and addition if the number of processors is large enough (62 processors in this case). By comparison, if the partitioning scheme of [10] was used, no extra fills would be created but the solution time would be equivalent to 68 flops (12 partitions), considering the IEEE-118 network. For a 1729-node system, Table II shows that using 7 partitions the calculations correspond to only 154 floating point mult-add operations. Figure 4 illustrates how the number of processors can affect the fastest solution time for different network sizes. The gain is given by $G = ts/tp$, where ts is the total mult-adds operations for the serial solution (using MD ordering), and tp is the number of operations obtained for the parallel solution (using ML-MD ordering). The *maximum usable number* denotes the number of processor that must be available to get the least solution time. It is given by the greater of two numbers: the number of rows of the first partitions with nonzero elements (forward) or the number of leaf-nodes of the path tree (backward). The maximum usable number is 62, 161, 402 and 956 processors for the simulated networks, respectively. This kind of graphics allows finding the least solution time posssible to achieve if the number of available processors is previously fixed.

The methodology was implemented in a multiprocessor computer and the performance results are presented in Appendix B.

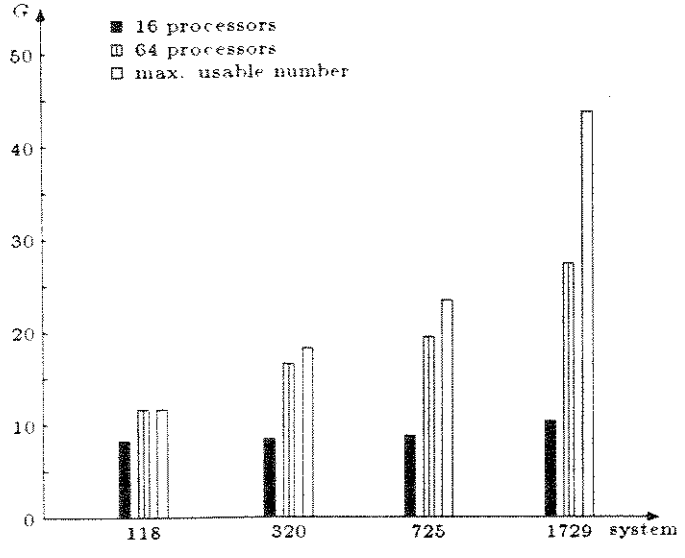


Figure 4: Gain *versus* system size.

CONCLUSIONS

A methodology has been described for solving sparse network linear equations on a multiprocessor computer using a partitioned W-matrix approach to the direct solution phase. The methodology allows all the rows of each partition to be processed in parallel and leads to a task scheduling strategy fitted to multiprocessor computer architectures. The approach takes full advantage of the powerful megaflops features more and more offered by the processor unit hardware. The performance results show that the potential speedup of the solution time is essentially bounded by the floating point operation capability of each processor, denoting that the methodology is a suitable way to exploit the growing power of the computing technology.

REFERENCES

[1] "Intel builds new supercomputer around I860", Computer, pp.101, March 1990.

[2] Rettberg R.D., Crowther W.R., Carvey P.P. & Tomlinson R.S., "The Monarch Parallel Processor Hardware Design", Computer, pp.18-28, April 1990.

[3] Brasch Jr. F.M., Van Ness J.E. & Kang S.C., "Simulation of a Multiprocessor Network for Power System Problems", IEEE Transactions on PAS, vol. PAS-101, n. 2, pp. 295-301, February 1982.

[4] Van Ness J.E., discussion in reference [5].

[5] Enns M.K., Tinney W.F. & Alvarado F.L., "Sparse Matrix Inverse Factors", IEEE Transactions on Power Systems, vol. 5, n. 2, pp. 466-473, May 1990.

[6] Alvarado F.L., Yu D.C. & Betancourt R., "Partitioned Sparse A^{-1} Methods", IEEE Transactions on Power Systems, vol. 5, n. 2, pp. 452-459, May 1990.

[7] Tinney W.F., Brandwajn V. & Chan S.M., "Sparse Vector Methods", IEEE Trans. on PAS, vol. 104,n. 2, pp.295-301, February 1985.

[8] Betancourt R., "An Efficient Heuristic Ordering Algorithm for Partial Matrix Refactorization", IEEE Trans. on Power Systems, vol. 3, n. 3, pp 1181-1187, August 1988.

[9] Gomes A. & Franquelo L.G.,"An Efficient Ordering Algorithm to Improve Sparse Vector Methods", IEEE Trans. on Power Systems, vol. 3, n. 4, pp. 1538-1544, November 1988.

[10] Gomes A. & Betancourt R., "Implementation of the Fast Decoupled Load Flow on a Vector Computer", IEEE Trans. on Power Systems, vol. 5, n. 3, pp. 977-983, August 1990.

[11] Betancourt R. & Alvarado F.," Parallel Inversion of Sparse Matrices", IEEE Transactions on Power Systems", vol. 1, n. 1, pp. 74-81, February 1986.

Biographies

Antonio Padilha Feltrin was born in Meridiano, Brazil in 1956. He received the B.Sc. degree in 1980 from EFEI and the M.Sc. degree in 1986 from Unicamp. Since 1981 he has been an Assistant Professor at Unesp-Ilha Solteira. He is currently a candidate to Ph.D. degree at Unicamp.

Andre L. Morelato França received the B.Sc. degree in 1970 from ITA and the Ph.D. degree in 1982 from Unicamp, Brazil, where he is currently an Associate Professor of Electrical Engineering. His general research interests are in the area of control and stability of electrical power systems and distribution automation.

APPENDIX A

Table A.I - Distribution of leaves at different depths for several orderings and systems.

Depth	118-node 179			1729-node 2154		
	MD (86)	MLMD (122)	MDMNP (86)	MD (1201)	MLMD (1732)	MDMNP (1211)
0	48	56	52	807	956	875
1	25	27	26	347	380	364
2	11	13	12	184	172	168
3	6	8	8	107	78	98
4	6	4	5	68	46	57
5	5	2	3	45	25	37
6	3	1	3	34	16	26
7	3	1	2	24	10	20
8	3	1	2	18	7	11
9	2	1	2	15	4	8
10	1	1	1	9	4	7
11	1	1	1	8	3	6
12	1	1	1	7	3	5
13	1	1		5	3	3
14	1			3	3	3
15	1			3	2	3
16				3	2	2
17				3	2	2
18				2	1	2
19				2	1	2
20				2	1	2
21				2	1	2
22				2	1	2
23				2	1	2
24				2	1	2
25				2	1	2
26				2	1	2
27				1	1	2
28				1	1	1
⋮				⋮	⋮	⋮
max. depth	15	13	12	47	30	41

The number in square brackets denotes the original off-diagonal nonzero elements in L-matrix.

The number in parentheses denotes the fill-in elements.

APPENDIX B

The methodology described in the paper was implemented in a multiprocessor computer (Parallel Preferential Processor - P3) under developing by the Federal Telecommunications R&D Center (CPqD) in Brazil. The version we are using consists of eight iAPX 286/287 based boards, connected in a common-bus 10 Mbytes/s bandwidth architecture. The common-bus uses a daisy-chain scheme for controlling the concurrent accesses. Each board contains a local memory of 512K bytes RAM and can access 2048K bytes of shared memory. Each cpu board is loaded with DOS operating system, extended with concurrent processing facilities.

The tests were carried out by solving a I=YV system, where Y is the complex nodal admittance matrix, I is the current injection vector and V is the voltage vector. All variables are stored in the local memories except vector V which is only stored in the shared memory. This data structure takes advantage of the machine architecture allowing the bus contention to be minimized. Figure B.1 compares the gain obtained using P3 with theoretical gains, regarding 50 repeated solutions of equation I=YV for the four networks. The theoretical gain here mentioned is slightly different from the gains shown in Figure 4, for the operations with the diagonal elements are now included whereas they were neglected before.

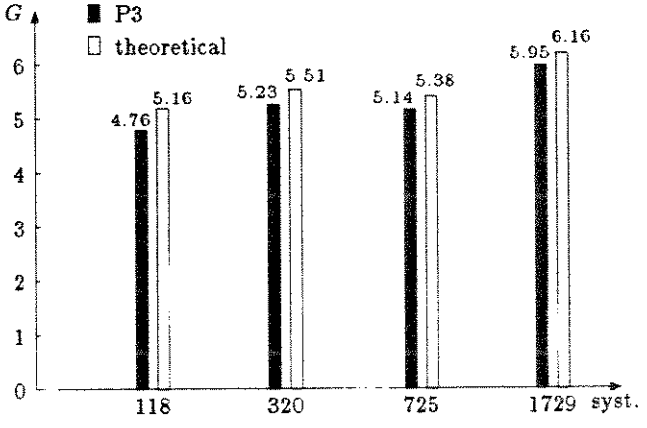


Figure B.1: Experimental results using eight processors

The experimental results were obtained using eight processors, the ordering scheme was MLMD and the last partition was handled as one block. The number of partitions used are 7,8,11 and 12, respectively. The codes were written in Fortran 77, extended with library functions to support parallelism. The results show that the measured gains are very close to the theoretical ones, indicating that the communication overhead and common-bus contention are not significant.

Bibliografia

[1] *ABUR A.* A parallel scheme the forward backward substituions in solving sparse linear equations, **IEEE Transactions on Power Systems**, Vol. 3, n. 4 pp.1471-1478, November 1988.

[2] *ALMEIDA H.L.S., KASZKUREWICZ E. & FALCÃO D.M.* Utilização de técnicas de processamento paralelo no problema da simulação de transitórios eletromagnéticos em sisteams de potência, **In: 8o Congresso Brasileiro de Automática**, Belém-Pa 10-14 setembro 1990 Vol.2 pp.667-672.

[3] *ALVARADO F.L.* Parallel solution of transient problems by trapezoidal integration, **IEEE Transactions on Power Apparatus and Systems**, Vol. 98, pp. 1080-1090, May/june 1979.

[4] *ALVARADO F.L., YU D.V. & BETANCOURT R.* Partitioned sparse A^{-1} methods, **IEEE Transactions on Power Systems**, Vol. 5, n. 2, pp. 452-459, May 1990.

[5] *ARNOLD C.P., PARR M.I. & DEWE M.B.* An efficient parallel algorithm for the solution of large sparse linear matrix equations, **IEEE Transactions on Computers**, Vol. 32, n. 3, pp.265-272, March 1983.

[6] *BETANCOURT R.* Efficient parallel processing technique for inverting matrices with random sparsity, **IEE Proceedings**, Vol. 133, Pt. E, n. 4, July 1986.

[7] *BETANCOURT R.* An efficient heuristic ordering algorithm for partial matrix refatorization. **IEEE Transaction on Power Systems**, Vol. 3, n. 3, pp. 1181-1187 August 1988.

[8] *BETANCOURT R. & ALVARADO F.* Parallel inversion of sparse matrices, **IEEE Transactions on Power Systems**, Vol. 1, n. 1, February 1986.

[9] *BRASCH Jr. F.M., VAN NESS J.E. & KANG S.C.* Simulation of a multiprocessor network for power system problems, **IEEE Transactions on Power Apparatus and Systems**, Vol. 101, n, 2, pp. 295-301, February 1982.

[10] *CAVALLI E. & ZABEU M.C.* Sistema hardware para processamento paralelo, **In: I Simpósio Brasileiro de Arquitetura de Computadores – Processamento Paralelo**, Gramado-RS 13-15 maio 1987, pp. 91-100.

[11] *CHAI J.S., ZHU N., BOSE A. & TYLAVSKY D.J.* Parallel Newton type methods for power system stability analysis using local and shared memory multiprocessors, **In: IEEE PES 1991 Winter Meeting**, New York, February 3-7, 1991.

[12] *CHAN, S. & BRANDWAJN, V.* Partial matrix refactorization. **IEEE Transactions on Power System**, Vol. PWRS-1(1), pp. 193-200, feb. 1986.

[13] *CROW M. L. & ILIĆ* The parallel implementation of the waveform relaxation method for transient stability simulations, **IEEE Transactions on Power Systems**, Vol. 5, n. 3, August 1990.

[14] *DUNCAN R.* A Survey of Parallel Computer Architectures, **COMPUTER**, pp. 5-16, February, 1990.

[15] *ENNS M.K., TINNEY W.F. & ALVARADO F.L.* Sparse matrix inverse factors, **IEEE Transactions on Power Systems**, Vol. 5, n. 2, pp. 466-473, May 1990.

[16] *FLYNN M. J.* Very high speed computing systems, **Proceedings of the IEEE**, Vol. 54, pp. 1901-1909, 1966.

[17] *GOMES A. & BETANCOURT R.* Implementation of the fast decoupled load flow on a vector computer, **IEEE Transactions on Power Systems**, Vol. 5, n. 3, pp. 977-983, August 1990.

[18] *GOMES A. & FRANQUELO, L.G.* Node ordering algorithms for sparse vector method improvement. **IEEE Transactions on Power Systems**, Vol. 3, n. 1, pp. 73-79 February 1988.

[19] *GOMES A. & FRANQUELO, L.G.* An efficient ordering algorithm to improve sparse vector methods.**IEEE Transactions on Power Systems**, Vol. 3, n. 4, pp. 1538-1544, November 1988.

[20] *HAPP H.H.* Diakoptics – The solution of system problems by tearing, **Proceedings of the IEEE**, Vol. 62, n. 7, pp. 930-940, July 1974.

[21] *HWANG K. & BRIGGS F.A.* **Computer architectures and parallel processing**, McGraw-Hill Book Company, Singapore, 1985.

[22] Intel builds new supercomputer around I860, **COMPUTER**, pp. 101, March 1990.

[23] *KARP A.H.* Programming for parallelism, **COMPUTER**, pp:43-57, May, 1987.

[24] *MONTICELLI A.* **Fluxo de carga em redes de energia elétrica**, Editora Edgard Blucher, São Paulo, 1983.

[25] *ORTEGA J.M.* **Introduction to parallel and vector solution of linear systems**, Plenum Press, 1988.

[26] *PADILHA FELTRIN A. & MORELATO FRANÇA A.L.* Estabilidade transitória de sistemas de energia elétrica: comportamento do esquema simultâneo implícito, **In: 6o Congresso Brasileiro de Automática**, Belo Horizonte-MG 25-28 novembro 1986, Vol. 1 pp.95-100.

lel languages respond to the needs of scientific programmers?, **COMPUTER** pp. 13-23, December 1990.

[28] *RETTBERG R.D. & THOMAS R.* Contention is no obstacle to shared-memory multiprocessing, **COMMUNICATIONS OF THE ACM**, Vol. 29, n. 12 pp.1202-1212, December 1986.

[29] *RETTBERG R.D.; CROWTHER W.R.; CARVEY P.P. & TOMLINSON R.S.* The Monarch parallel processor hardware design, **COMPUTER**, pp.18-28, April 1990.

[30] *RODRIGUES M.* Comunicações orais, 1990

[31] *SALEH A.R. et al* Parallel circuit simulation on supercomputer, **Proceedings of the IEEE**, Vol. 77, n. 12, pp. 1915-1931, December 1989.

[32] *STENSTRON P.* Reducing contention in shared-memory multiprocessors, **COMPUTER** pp.26-37, November 1988.

[33] *STONE H. S.* **High-performance computer architectures**, Addison-Wesley Publishing Company, USA, 1987.

[34] *TINNEY W.F., BRANDWAJN V. & CHAN S.M.* Sparse vector methods, **IEEE Transactions on Power Apparatus and Systems**, Vol. 104, n. 2, pp.295-301, February 1985.

[35] *TINNEY W. F., & WALKER, J. W.* Direct solutions of sparse network equations by optimally ordered triangular factorization. **Proceedings of the IEEE**, Vol. 55, pp. 1801-1809, 1967.

[36] *TERRY L.A.; TEIXEIRA M.J. & ROMÉRO S.P.* Uma nota sobre solução paralela de sistemas lineares esparsos. **In: II Simpósio Brasileiro de Arquitetura de Computadores - Processamento Paralelo** Águas de Lindóia-SP 1988, pp. 11.A.1.1-11.A.1.8.

[37] *VAN NESS J.E.* Discussão da referência [15]

[38] *VAN NESS J.E. & MOLINA G.* The use of multiple factoring in the parallel solution of algebraic equations, **IEEE Transactions on Power Apparatus and Systems**, Vol. 102, n. 10, pp.3433-3438, October 1983.

[39] *YU D.C.* & *WANG H.* A new approach for the forward and backward substitutions of parallel solution of sparse linear equations - basead on dataflow architecture, **IEEE Transactions on Power Systems**, Vol. 5, n. 2, pp.621-627, May 1990.

[40] *ZOLLENKOPF K.* Bi-factorisation basic computational algorithm and programming techniques. In: REID, J.K. ed. **Large sparse sets of linear equations**. London, Academic Press, 1971. p.75-96. (Conference of Institute of Mathematics and Its Applications, Oxford, 5-8 April, 1970)