@sistemas-de-processamento-paralelo

Mestrado em Informática - UFPR

Diogo Cezar Teixera Batista
Fabiano da Silva

Curitiba

17 de agosto de 2010

1 Informações da Disciplina

- Nome: *Sistemas de Processamento Paralelo - opt. CI316A*;
- Código: *CI 728*;
- Professor: *Fabiano da Silva*;
- Horários: *3ª das 15:30 as 17:10 / 6ª das 15:30 as 17:10*;
- Método de Avaliação:
  - Artigo;
- Página da disciplina: http://www.inf.ufpr.br/fabiano/ci316

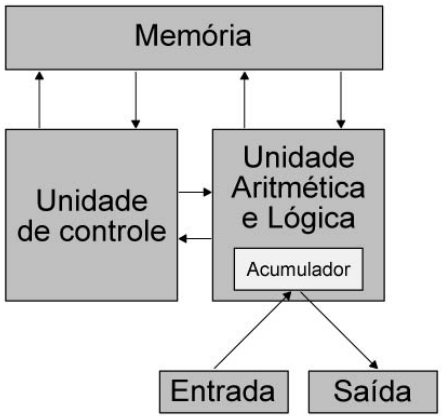
2 Arquitetura de Von Neumann

Se caracteriza pela possibilidade de uma máquina digital armazenar seus programas no mesmo espaço de memória que os dados, podendo assim manipular tais programas.

A máquina proposta por Von Neumann reúne os seguintes componentes:

- uma memória,
- uma unidade aritmética e lógica (ALU),
- uma unidade central de processamento (CPU), composta por diversos registradores, e
- uma Unidade de Controle (CU), cuja função é a mesma da tabela de controle da Máquina de Turing universal: buscar um programa na memória, instrução por instrução, e executá-lo sobre os dados de entrada.

Figura 1: Arquitetura completa de Von Neumann



3 Referencial Teórico

3.1 Visão Geral sobre Pipeline

Pipeline é uma técnica de implementação de processadores que permite a sobreposição temporal das diversas fases de execução de instruções. Há a técnica monociclo, mais simples, que executa uma instrução, exclusivamente, por vez. A técnica de pipeline é dividir o processo em estágios distintos, normalmente quatro ou seis, e processá-los de forma paralela (não processando dois estágios iguais ao mesmo tempo, obviamente).

Exemplo:

Um processador que tem suas instruções executadas em cinco estágios:

1. Busca da instrução na memória;
2. Leitura dos registradores enquanto a instrução é decodificada (estamos simulando um processador que permite que a leitura e a decodificação ocorram ao mesmo tempo);
3. Execução de uma operação ou cálculo de um endereço;
4. Acesso a um operando na memória;
5. Escrita do resultado em um registrador.

Em condições ideais, o ganho devido ao pipeline é igual ao número de estágios do pipe. No nosso caso acima, ele seria cinco vezes mais rápido que um processador similar que utilizasse a técnica de monociclo. Na prática, podemos perceber que o balanceamento não é tão preciso. É bom deixar claro que o pipeline melhora a performance aumentando o número de instruções executadas na unidade de tempo, e não por meio da diminuição do tempo de execução de uma instrução individual.

3.2 Componentes de Uma Arquitetura Superescalar

Uma arquitetura superescalar deve possuir uma série de componentes especiais para executar mais de uma instrução por ciclo, que relacionamos a seguir:

- Unidade de Busca de Instruções: capaz de buscar mais de uma instrução por ciclo. Possui também um preditor de desvios, que deve ter alta taxa de acerto, para poder buscar as instruções sem ter que esperar pelo resultados dos desvios.
- Unidade de Decodificação: capaz de ler vários operandos do banco de registradores a cada ciclo. Note que cada instrução sendo decodificada pode ler até dois operandos do banco de registradores.
- Unidades Funcionais Inteiras e de Ponto Flutuante: em número suficiente para executar as diversas instruções buscadas e decodificadas a cada ciclo.

3.2.1 Limitações de Uma Arquitetura Superescalar

A abordagem superescalar depende da habilidade de executar várias instruções em paralelo. O termo paralelismo no nível de instruções diz respeito ao nível no qual as instruções de um programa podem ser executadas de forma paralela (em média).

Limitações de uma estrutura superescalar:

- Dependência de dados verdadeira;
- Dependência de desvios;
- Dependência de recursos.

Dependência de Dados Verdadeira - Considere a seguinte seqüência de instruções:

- `add r1, r2` - carregar registrador r1 com a soma dos conteúdos de r1 e r2
- `move r1, r3` - carregar registrador r3 com o conteúdo de r1

A segunda instrução pode ser buscada e decodificada antecipadamente, mas não pode ser executada até que seja completada a execução da primeira instrução. A razão é que a segunda instrução requer dados produzidos pela primeira instrução. Essa situação é conhecida como dependência de dados verdadeira (também chamada dependência de fluxo ou dependência de escrita-leitura).

Dependência de Desvios - A presença de desvios condicionais em uma sequência de instruções complica a operação do pipeline. A instrução seguinte a um desvio condicional (tomado ou não) depende dessa instrução de desvio. Esse tipo de dependência também afeta uma pipeline escalar, mas a conseqüência desse tipo de dependência é mais severa em uma pipeline superescalar, porque o número de instruções perdidas em cada atraso é maior. Se forem usadas instruções de tamanho variável, surge ainda um outro tipo de dependência. Como o tamanho de uma instrução particular não é conhecido, uma instrução deve ser decodificada, pelo menos parcialmente, antes que a instrução seguinte possa ser buscada. Isso impede a busca simultânea de instruções, requerida em uma pipeline superescalar. Essa é uma das razões pelas quais técnicas supersescalares são mais diretamente aplicáveis a arquiteturas RISC ou do tipo RISC, que possuem instruções de tamanho fixo.

Dependência de Recursos - Um conflito de recurso ocorre quando duas ou mais instruções competem, ao mesmo tempo, por um mesmo recurso. Exemplos de recursos incluem memórias, caches, barramentos, portas de bancos de registradores e unidades funcionais (por exemplo, o somador da ULA). Em termos de pipeline, um conflito de recurso apresenta um comportamento semelhante ao de uma dependência de dados. existem, entretanto, algumas diferenças. Por um lado, conflitos de recursos podem ser superados pela duplicação de recursos, enquanto uma dependência de dados não pode ser eliminada. Além disso, quando uma operação efetuada em uma dada unidade fucional consome muito tempo para ser completada, é possível minimizar os conflitos de uso dessa unidade por meio de sua implementação como uma pipeline.

3.3 MMX

A primeira extensão SIMD para micro-processadores da família x86 veio através do MMX (MultiMedia eXtensions). Esta extensão incorporou aos micro-processadores Pentium uma unidade SIMD para inteiros, sem suporte a ponto flutuante. Esta unidade é composta por vetores de 64 bits, armazenados em 8 registradores MMX nomeados de MM0 a MM7. As combinações de pacotes suportadas pelo MMX permitem aproveitar os registradores com diferentes precisões para inteiros, conforme a listagem abaixo:

- packed byte, com 8 bytes empacotados em 64 bits;
- packed word, com 4 words empacotados em 64 bits;
- packet doubleword, com 2 double-words empacotadas em 64 bits;
- packed quadword, com um único componente de 64 bits.

O suporte ao MMX na arquitetura x86 trouxe ao conjunto de instruções destes processadores uma nova variedade de instruções para suportar esta extensão. No total foram adicionadas 57 novas instruções, que levam um formato convencional de programação, composto de uma instrução e dois operandos, como no exemplo hipotético:

```
InstruçãoMMX mmreg1, mmreg2
```

Nesta instrução, os registradores mmreg1 e mmreg2 são ambos operandos de origem, enquanto o resultado é salvo no próprio mmreg1. Isto traz alguns inconvenientes, pois caso deseja-se utilizar o valor de mmreg1 antes da instrução ter sido executada é necessário copiá-lo para outro registrador.

Alguns detalhes de implementação do MMX ainda impedem a operação simultânea com ponto flutuante nas mesmas rotinas em que estes registradores são usados. Isto ocorre pelo fato de que o espaço de endereçamento do MMX e do x87 (responsável pelas operações em ponto flutuante) é compartilhado, permitindo o uso apenas de um destes em um dado momento (Peleg e Weiser 1996). Esta decisão foi tomada para evitar que o suporte no sistema operacional precisasse ser modificado para salvar o estado dos registradores MMX durante trocas de contexto.

Apesar de não apresentar características interessantes para o uso na computação genérica, como a falta de suporte a unidades de ponto flutuante, o MMX serviu como modelo para a criação de extensões como o SSE, que expandiu este conjunto de instruções com suporte a ponto flutuante em registradores de precisão superior.

3.4 SSE

As extensões SSE foram introduzidas na família de micro-processadores Pentium III. As instruções SSE operam em valores empacotados de ponto flutuante de precisão simples, conforme o padrão IEEE (P754 1985), e encontram suporte na arquitetura através da adição de novos registradores especiais. O layout deste tipo de registrador é o mesmo visto nas Figuras 2.2 e 2.3, onde é permitido o armazenamento de 4 valores de ponto flutuante, representando um dado escalar ou vetorial, em um único registrador.

Com a extensão SSE foram adicionados 8 novos registradores de 128 bits de dados na arquitetura do Pentium III, denominados XMM, acessíveis pelos nomes XMM0-XMM7. Entretanto, diferente do espaço de endereçamento do MMX, os registradores XMM não compartilham o mesmo espaço com o x87, permitindo o uso conjunto de instruções SSE com MMX ou da FPU do x87. Para o suporte à instruções SIMD sobre inteiros, o SSE utiliza-se dos registradores MMX, que foram mantidos nas novas arquiteturas de microprocessadores. Para suportar esta nova extensão SIMD, foram adicionadas duas unidades de hardware dedicado à CPU. O Pentium III apresenta duas unidades independentes de precisão simples para ponto flutuante; uma opera sobre instruções de multiplicação SIMD, através da expansão do hardware de multiplicação de ponto flutuante já existente, e outra para realizar somas sobre pacotes de dados (Corporation 2004).

Estas novas unidades de processamento independentes adicionaram ao IA-32 um novo estado de execução. Isto implica que o sistema operacional tenha conhecimento dos recursos oferecidos pelo processador, de forma que o estado de execução do SSE seja salvo em trocas de contexto de processos no kernel, da mesma forma que é realizado para salvar o estado da FPU do x87 e do MMX. As instruções suportadas no SSE são sub-divididas nos seguintes grupos:

- Instruções de movimento de dados;
- Instruções aritméticas;
- Instruções lógicas;
- Instruções de comparação.
- Instruções de shufle (combinação de 2 registradores em 1 único);
- Instruções de conversão de dados.

As instruções aritméticas permitem realizar operações como soma, subtração, multiplica ção, divisão, raiz quadrada, máximo e mínimo, entre outras, sobre dados empacotados ou escalares. Nota-se que o uso delas apresenta a mesma característica vista no MMX: como o registrador de destino também armazena um dos valores de entrada para a operação, seu valor deve ser copiado para outro registrador caso ele precise ser usado no futuro. A interface disponibilizada para acessá-las é semelhante à do MMX:

```
InstruçãoSSE xmmreg1, xmmreg2
```

3.5 DSP

DSP significa *Digital Signal Processor*, ou Processador de Sinais Digital. O que faz? Na realidade o DSP é uma espécie de microcontrolador, com maior poder de processamento, incorporando os recursos dos microcontroladores e fornecendo novas funções, desenvolvido para suportar tarefas numéricas intensivas, de alto desempenho e repetitivas. Esse desempenho superior é devido a:

- Capacidade de multiplicação e registro no acumulador em um único ciclo. DSPs de alta performance possuem dois multiplicadores que permitem realizar duas multiplicações ao mesmo tempo por ciclo de instrução. Alguns possuem quatro ou mais multiplicadores;
- Modos especializados de endereçamento: pré e pós-modificação dos ponteiros de endereços, endereçamento circular e endereçamento do tipo bit-reversed.
- Arquitetura de múltiplo acesso à memória, que possibilita o acesso completo a memória on-chip em um único ciclo;
- Instruções especiais de controle de execução, como por exemplo, instrução de loop, não necessitando assim instruções de atualização e teste de contadores de loop;
- Conjunto irregular de instruções, o que geralmente possibilita mais de uma operação em um único ciclo de instrução (por exemplo, em um DSP com arquitetura de 32 bits, 16 podem ser destinados a operações como multiplicação e adição e o resto para os dados manipulados).

3.6 GPU

GPUs são processadores especializados em operações relacionadas com computação gráfica 3D. São extremamente poderosos devido a sua arquitetura paralela e sua eficiência, tanto no

acesso a memória como nas operações vetoriais e de interpolação. Atualmente, devido o grande aumento de flexibilidade da arquitetura e das linguagens de programação, as GPUs estão sendo usadas para substituir a CPU na resolução de diversos algoritmos clássicos (GPU for Generic Programming - GPGPU). O uso cada vez mais freqüente da GPU para programação genérica é devido à grande eficiência que o processador gráfico possui em determinados tipos de operações, superando o desempenho da CPU.

3.7 Hyper-Threading

Hyper-Threading ou hiperprocessamento é uma tecnologia usada em processadores que o faz simular dois processadores tornando o sistema mais rápido quando se usa vários programas ao mesmo tempo. Esse processo todo rende um acréscimo de até 20

A simulação do segundo processador é feito utilizando partes não aproveitadas do processador na previsão de desvio do pipeline. Estas partes são conhecidas como bolhas do pipeline e não teriam utilidade nenhuma desperdiçando ciclos.

É uma tecnologia desenvolvida pela Intel e foi primeiramente empregada no processador Pentium 4 de núcleo Northwood, de 32 bit.Atualmente na nova família de processadores, denominada Nehalem, o processador Core i7 usufrui dessa tecnologia proporcionando até 8 núcleos totais.

4 Modelos de Computação Paralela

Cada um dos elementos apresentados na figura 1 é realizado à custa de componentes físicos independentes, cuja implementação tem variado ao longo do tempo, consoante a evolução das tecnologias de fabricação, desde os reles eletromagnéticos, os tubos de vácuo (ou válvulas), até aos semicondutores, abrangendo os transistores e os circuitos eletrônicos integrados, com média, alta ou muito alta densidade de integração.

As interações entre os elementos exibem tempos típicos que também têm variado ao longo do tempo, consoante as tecnologias de fabricação. As memórias centrais têm tempos típicos de acesso da ordem da dezena de nanosegundos. As unidades de entrada e saída exibem tempos típicos extremamente variáveis, mas que são tipicamente muito superiores à escala do nanosegundo.

O grande interesse por problemas cada vez mais complexos tem levado a necessidade de computadores cada vez mais potentes para resolvê-los. Entretanto, limitações **físicas** e **econômicas**

têm restringido o aumento da velocidade dos computadores seqüenciais, ou seja, computadores que executam instruções em série, uma após a outra pela CPU . Por outro lado, os problemas computacionais usualmente podem ter algumas de suas partes dividida em pedaços que poderiam ser solucionados ao mesmo tempo, ou processados em paralelo. Processamento paralelo é então *uma forma pela qual a demanda computacional é suprida através do uso simultâneo de recursos computacionais* como processadores para solução de um problema.

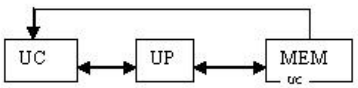
A computação paralela é caracterizada pelo uso de várias unidades de processamento ou processadores para executar uma computação de forma mais rápida. É baseada no fato de que o processo de resolução de um problema pode ser divido em tarefas menores, que podem ser realizadas simultaneamente através de algum tipo de coordenação. O conceito foi originalmente introduzido no CDC 6600 em 1964 pela CDC (Control Data Corporation). No tópico a seguir descreverá os modelos de computação paralela existentes.

Os modelos de arquitetura de computadores são classificados pelo fluxo de instruções e dados que se apresentam. Essa classificação é definida como taxonomia de Flynn. Ela fica divida em quatro categorias: SISD, SIMD, MISD e MIMD. A seguir será descrito mais detalhadamente essas quatro categorias:

4.0.1 SISD (SINGLE INSTRUCTION SINGLE DATA)

Conhecido como fluxo único de instruções sobre um único conjunto de dados é o caso das máquinas convencionais com uma CPU. Essa arquitetura é conhecida também como Von Neumann. A Figura 2 demonstra essa arquitetura SISD.

Figura 2: Arquitetura SISD



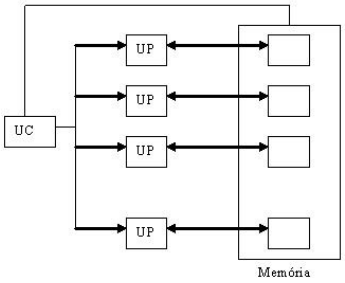
4.0.2 SIMD (SINGLE INSTRUCTION STREM MULTIPLE DATA STREAM)

Corresponde ao caso das arquiteturas vetoriais onde a mesma operação é executada sobre múltiplos operandos. A Figura 3 demonstra essa arquitetura SIMD.

Código 1: Execução de Algoritmo com Arquitetura SIMD rodando em Paralelo

```
1 if (x[i] == 0){
2         y[i] = a[i];
```

Figura 3: Arquitetura SIMD




```
3 }
4 else{
5         y[i] = a[i]/2;
6 }
```

O Código 1 mostra a execução de um trecho de código adotando o modelo SIMD de forma paralela. No código i é o indexador do processador que deve executar determinada instrução.

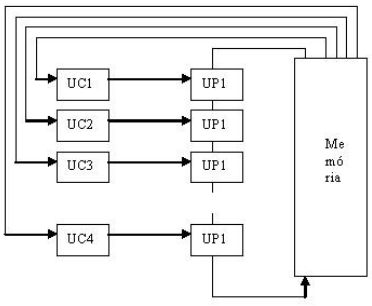
Mas nem sempre todos os processadores assumem que $x[i] == 0$ ao mesmo tempo.

Por isso os que assumem a sentença como verdadeira executam o procedimento que está indicado como verdadeiro enquanto o restando dos processadores aguardam, no próximo *clock* os que estavam aguardando executam, e os que estavam executando aguardam.

4.0.3 MISD (MULTIPLE INSTRUCTION STREAM SINGLE DATA STREAM)

Um pipeline de processadores seria um caso aonde os dados vão sendo processados e passados para o processador seguinte. A proposta de implementação que mais se aproxima desta categoria é a da máquina de fluxo de dados. A Figura 4 demonstra essa arquitetura MISD.

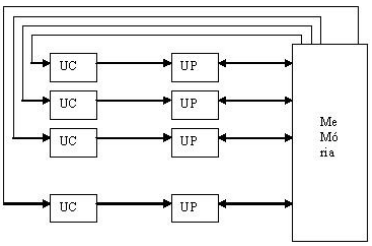
Figura 4: Arquitetura MISD



4.0.4 MIMD (MULTIPLE INSTRUCTION STREAM MULTIPLE DATA STREAM)

Os multiprocessadores são um caso onde várias instruções podem ser executadas ao mesmo tempo em unidades de processamento diferentes controladas por unidades de controle independentes (uma para cada unidade de processamento). A Figura 5 demonstra essa arquitetura MIMD.

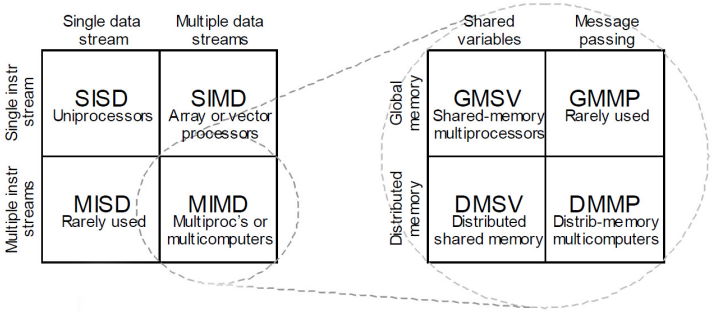
Figura 5: Arquitetura MIMD



De acordo com a estrutura de memória e a comunicação/sincronização, a arquitetura MIMD de Flynn pode ser subdividida em quatro sub-classes:

- GMSV (Global Memory SharedVariable);
- DMSV (Distributed Memory Shared Variable);
- DMMP (Distributed Memory Message Passing);
- GMMP (Global Memory Message Passing).

Figura 6: Arquitetura MIMD



A classe GMSV é referenciada como sistemas multiprocessados com memória compartilhada, considerados sistemas fortemente acoplados.

A classe DMMP é conhecida como sistemas multicomputadores com memória distribuída e fracamente acoplados. Utilizada em *cluster* e *grids*, apresente a desvantagem de um gargalo quase inviável pelo tipo de barramento de comunicação que normalmente é TCP/IP.

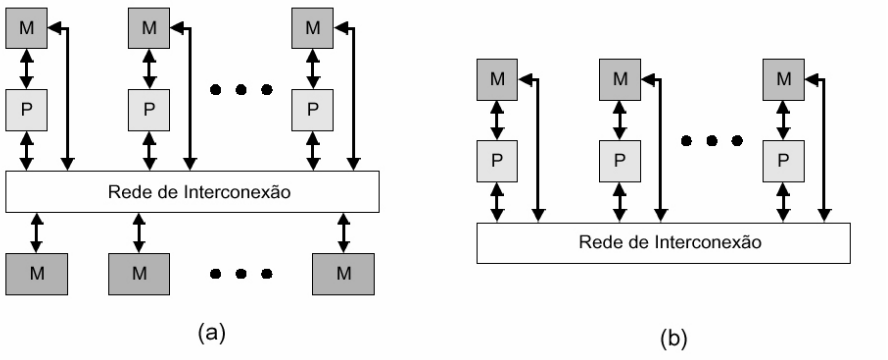
A classe DMSV, que se tornou popular por combinar a implementação de uma memória distribuída com a facilidade de programação com variáveis compartilhadas, é conhecida como memória compartilhada distribuída.

A categoria GMMP não é muito utilizada.

4.1 NUMA (NON-UNIFORM MEMORY ACCESS)

No modelo NUMA (Non-Uniform Memory Access), cada processador possui uma memória local, a qual é agregada ao espaço de endereçamento global da máquina. Dessa forma, podem existir até três padrões de acesso à memória compartilhada. O primeiro, e o mais rápido, é aquele onde a variável compartilhada está localizada na memória local do processador. O segundo padrão refere-se ao acesso a um endereço na memória central. Já o terceiro, e o mais lento, diz respeito ao acesso a uma posição localizada em uma memória local de outro processador. Dois modelos alternativos de máquina NUMA são mostrados na Figura 8.

Figura 7: Modelo NUMA: (a) com memória central; (b) sem memória central.



O modelo mostrado na Figura 8b é também chamado de multiprocessador de memória compartilhada distribuída (DSM - Distributed Shared Memory), pois toda a memória do sistema é distribuída entre os processadores da máquina, não havendo uma memória central.

5 Modelo PRAM

O modelo PRAM (Parallel Random Access Machine) é uma extensão do modelo seqüencial RAM e o mais conhecido modelo de computação paralela. Pode ser descrito como sendo um conjunto de processadores operando de modo síncrono, sob o controle de um relógio comum.

Cada processador é identificado por um índice único e possui uma memória local própria, podendo comunicar-se com os demais processadores através de uma memória global compartil-

hada.

Essa forma de comunicação possibilita o acesso (leitura ou escrita) simultâneo de vários processadores a uma mesma posição da memória global.

5.1 EREW (Exclusive Read, Exclusive Write)

Este modelo não permite qualquer acesso simultâneo a uma mesma posição da memória por mais de um processador, seja para leitura ou escrita;

5.2 ERCW

Somente 1 dos processadores lê, mas o acesso é simultâneo para a escrita.

5.3 CREW (Concurrent Read, Exclusive Write)

Este modelo permite somente leitura simultânea de uma mesma posição da memória por mais de um processador, não sendo a mesma operação permitida para a escrita simultânea;

5.4 CRCW (Concurrent Read, Concurrent Write)

Este modelo permite tanto a leitura, como a escrita concorrente em uma mesma posição da memória por mais de um processador. Para evitar os possíveis conflitos do acesso simultâneo, temos os seguintes critérios:

- CRCW comum: Na escrita simultânea, todos os processadores devem escrever o mesmo valor;
- CRCW prioritário: Os valores escritos simultaneamente podem ser diferentes. Ficará armazenado na posição da memória o valor escrito pelo processador de maior prioridade (assumimos como sendo o processador com menor índice);
- CRCW arbitrário: : Os valores escritos simultaneamente podem ser diferentes e apenas um, entre todos os processadores, poderá escrever, não importando qual deles.

Derivam-se algumas subclassificações para esse modelo, descritas a seguir.
Ao tentar se escrever concorrentemente,

- CRCW-U - marca-se a posição de memória como *undefined* (indefinido);

- CRCW-D - indica-se que houve uma tentativa de escrita concorrente e marca-se a célula como colisão;
- CRCW-C - persiste o valor comum na tentativa de gravação;
- CRCW-R - um dos valores é atribuído aleatoriamente;
- CRCW-P - o processador com maior prioridade (menor índice) grava o valor na memória em questão;
- CRCW-M - o maior ou menor valor dentre os concorrentes é gravado;
- CRCW-REDUCE - a partir de operadores associativos (soma, subtração, divisão e multiplicação) os dados são combinados e grava-se o seu resultado.

6 Crivo de Eratóstenes

O Crivo de Eratóstenes é um algoritmo simples e prático para encontrar números primos até um certo valor limite. Segundo a tradição, foi criado pelo matemático grego Eratóstenes.

6.1 Explicação do Algoritmo

Para exemplificá-lo, vamos determinar a lista de números entre 1 e 30.

- Inicialmente, determina-se o maior número a ser checado. Ele corresponde à raiz quadrada do valor limite, arredondado para baixo. No caso, a raiz de 30, arredondada para baixo, é 5;
- Crie uma lista de todos os números inteiros de 2 até o valor limite: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30;
- Encontre o primeiro número da lista. Ele é um número primo, 2;
- Remova da lista todos os múltiplos do número primo encontrado. No nosso exemplo, a lista fica: 2 3 5 7 9 11 13 15 17 19 21 23 25 27 29;
- O próximo número da lista é primo. Repita o procedimento. No caso, o próximo número da lista é 3. Removendo seus múltiplos, a lista fica: 2 3 5 7 11 13 17 19 23 25 29. O próximo número, 5, também é primo; a lista fica: 2 3 5 7 11 13 17 19 23 29. 5 é o último número a ser verificado, conforme determinado inicialmente. Assim, a lista encontrada contém somente números primos.

6.2 Implementação Sequencial

O algoritmo para implementação sequencial pode ser definido por:

Código 2: Algorítmo Sequencial do Crivo de Eratóstenes

```
1  /* Primeiro passo */
2  recebe valorLimite
3
4  /* Segundo passo */
5  raiz <- sqrt{valorLimite}
6
7  /* Terceiro passo */
8  para i <- 2 até valorLimite
9      vetor[i] <- i
10 fim-para
11
12 /* Quarto passo */
13 para i <- 2 até raiz
14     se vetor[i] = i
15         imprima "O número " i " é primo."
16         para j <- i+i até valorLimite, de i e i
17             vetor[j] <- 0
18         fim-para
19     fim-se
20 fim-para
```

Código 3: Implementação Sequencial em C do Crivo de Eratóstenes

```
1  #include <stdio.h>
2  #include <math.h>
3  #define NMAX 1000000
4  int main() {
5  int i, j, vetor[NMAX];
6  int valorMaximo, raiz;
7  // Primeiro passo
8  scanf("%d", &valorMaximo);
9  // Segundo passo
10 raiz=sqrt(valorMaximo);
11 // Terceiro passo
```

```
12 for (i=2; i<=valorMaximo; i++){
13         vetor[i]=i;
14 }
15 // Quarto passo
16 for (i=2; i<=raiz; i++){
17         if (vetor[i]==i) {
18                 printf("%d é primo!n", i);
19                 for (j=i+i; j<=valorMaximo; j+=i){
20                         vetor[j]=0; // removendo da lista
21                 }
22         }
23 }
24 return 0;
25 }
```

6.3 Implementação Paralela

Implementar as seguintes soluções paralelas para o algoritmo:

- Cada processador ataca n passo. *Paralelismo de Controle no Crivo*;
- Divide-se o vetor e cada processador se encarrega por resolver parte da sequencia. *Paralelismo de Dados.*

6.3.1 Paralelismo de Controle no Crivo

Dois passos:

- Encontrar o próximo número primo;
- Retirar da lista os múltiplos deste primo, começando com o seu quadrado;

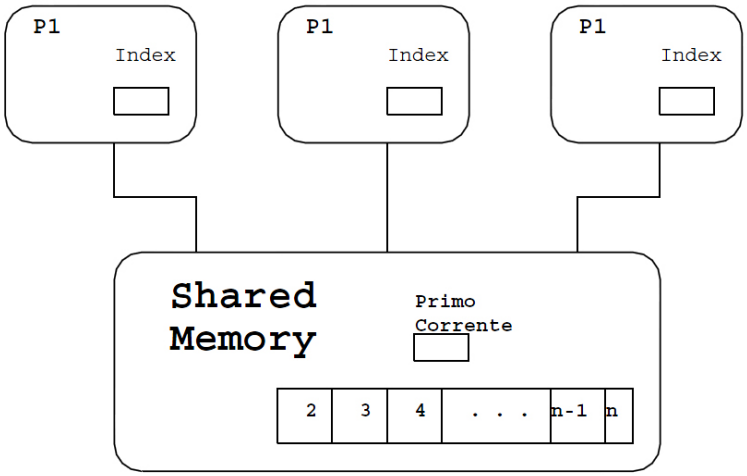
Estratégia:

- Processadores diferentes são responsáveis por marcar múltiplos de primos diferentes;
  - processador 1 marca os múltiplos de 2;
  - processador 2 marca os múltiplos de 3;

Como?

Problemas:

Figura 8: Implementação paralela do Crivo de Eratóstenes



Se um grupo de processadores executando assincronamente compartilham acesso à mesma estrutura de dados de uma maneira desstruturada, inferências ou erros podem ocorrer.

6.3.1.1 Problema 1 - Mesmo Primo Pode acontecer de dois processadores usarem o mesmo primo para marcar os seus múltiplos. O algoritmo não executa erradamente, mas desperdiça processamento.

Normalmente um processador acessa o valor do primo corrente e começa procurando no vetor até encontrar outra célula não marcada (novo primo).

Se um segundo processador acessa o valor do primo corrente antes do primeiro processador atualizá-lo, então ambos os processadores adotarão o mesmo valor de primo para marcar múltiplos.

6.3.1.2 Problema 2 - Número Composto Pode acontecer de um processador marcar múltiplos de um número composto (não primo).

Assuma que um processador A é responsável por marcar os múltiplos de 2, mas antes dele marcar qualquer célula, um processador B encontra o próximo primo 3, e um processador C busca pela próxima célula não-marcada.

Como a célula 4 ainda não foi marcada, o processador C retornará com o valor 4 como o primo mais recente!

6.3.1.3 Algoritmo Paralelo Sempre que um processador está desocupado, ele pega o próximo primo e marca seus múltiplos.

Todos os processadores continuam neste procedimento até que o primeiro primo maior que $\sqrt{n}$ seja encontrado.

Encontrando todos os primos menores que 1000:

- 2 processadores gastam 706 unidades de tempo;
- 3 processadores gastam 499 unidades de tempo;
- Mais de 3 processadores não implica em tempos menores.

6.3.2 Paralelismo de Dados (Crivo)

Processadores trabalham juntos para marcar múltiplos de cada novo primo encontrado.

Cada processador será responsável por um segmento do vetor representando os números naturais até n.

Por que paralelismo de dados?

- Cada processador aplica a mesma operação (marcar múltiplos de um primo) a sua porção do conjunto de dados;

Mecanismo de comunicação:

- Memória compartilhada;
- Troca de mensagens.

6.3.2.1 Crivo com Troca de Mensagens

P processadores.

Cada processador recebe $[\frac{n}{p}]$ naturais.

$p \ll \sqrt{n}$

Todos os primos menores que $\sqrt{n}$ assim como o primeiro primo maior que $\sqrt{n}$ estão na lista de números naturais controlada pelo primeiro processador.

Funcionamento: processador 1 encontrará o próximo primo e divulgará seu valor para todos os outros processadores (broadcast).

Então cada processador marca na sua lista de números compostos todos os múltiplos do novo primo divulgado. Este processo continua até que o primeiro processador encontre um primo maior que $\sqrt{n}$, então o algoritmo termina.

6.4 Soluções para implementação

Dentre as soluções discutidas em aula para implementação do Crivo de Eratóstenes chegou-se as seguintes conclusões:

Para o *Paralelismo de Controle no Crivo*,

- A solução para um modelo síncrono seria disparar a procura pelos múltiplos de n pelo processador P_n assim que o próximo número primo fosse descoberto pelo processador P_{n-1};
- Essa solução requer apenas o modelo EREW (Exclusive Read, Exclusive Write), pois:
  - Ao disparar a procura pelos múltiplos de n o processador P atacar uma gama de dados na memória compartilhada que não será atacado ao mesmo tempo por $P_1, P_2, \ldots, P_N$;
  - Isso acontece pelo fato da buscar ser pelos múltiplos do quadrado do primo corrente, quanto maior o primo, mais loge ele irá atacar os dados;

Para o *Paralelismo de Dados (Crivo)*,

- Assumindo que não há troca de mensagens entre $P_0, P_1, P_2, \ldots, P_N$;
- Há a necessidade de se dividir a tarefa entre um processador mestre P_0 e e os demais processadores escravos $P_1, P_2, \ldots, P_N$;
- P_0 é responsável por através do método de Eratóstenes achar os primos de um conjunto inicial de dados que satisfaça a regra $\sqrt{n} < \alpha$, onde,
  - n é o número limite do vetor de dados;
  - α é o último número primo que deve ser achado pelo processador mestre P_0;
- Por existir um problema de concorrência, ao resgatar qual primo cada processador escravo deve processar seus múltiplos, um modelo indicado para essa solução seria: CREW (Concurrent Read, Exclusive Write);
- Uma solução viável para implementar em um modelo EREW (Exclusive Read, Exclusive Write) seria criar um vetor com n posições igual ao número de processadores, e cada processador lê apenas o primo corrente que está em seu respectivo endereçamento de memória.

7 Game of Life

O jogo da vida é um autômato celular desenvolvido pelo matemático britânico John Horton Conway em 1970.

O jogo foi criado de modo a reproduzir, através de regras simples, as alterações e mudanças em grupos de seres vivos, tendo aplicações em diversas áreas da ciência.

As regras definidas são aplicadas a cada nova "geração"; assim, a partir de uma imagem em um tabuleiro bi-dimensional definida pelo jogador, percebem-se mudanças muitas vezes inesperadas e belas a cada nova geração, variando de padrões fixos a caóticos.

As regras são:

1. Qualquer célula viva com menos de dois vizinhos vivos morre de solidão;
2. Qualquer célula viva com mais de três vizinhos vivos morre de superpopulação;
3. Qualquer célula com exatamente três vizinhos vivos se torna uma célula viva;
4. Qualquer célula com dois vizinhos vivos continua no mesmo estado para a próxima geração.

É importante entender que todos os nascimentos e mortes ocorrem simultaneamente. Juntos eles constituem uma geração ou, como podemos chamá-los, um "instante" na história da vida completa da configuração inicial.

7.1 Solução para o Game of Life

7.1.0.2 SOLUÇÃO 01: Utilizando o modelo CREW (Concurrent Read, Exclusive Write) poderíamos assumir que cada processador p cuidasse de uma única célula da matriz. Dessa forma p se preocuparia somente com as 8 células ao redor de sí.

Essa solução parece ser viável para matrizes onde o número de células possa ser simulado pelo número de processadores disponíveis, entretanto ao se trabalhar com uma matriz maior (por exemplo $M[1000][1000]$) o sistema teria um grande custo de sincronização entre as células e pouco custo de processamento.

7.1.0.3 SOLUÇÃO 02: É possível dividir ma matriz maior M em matrizes menores $N_1, N_2, \ldots, N_n$ afim de que cada processador cuide de uma das matrizes menores. Um grande problema para esse tipo de implementação está na sincronização. É necessário que todos os processadores estejam sincronizados dentro de uma mesma geração para que só assim passem a processar a próxima.

8 Técnicas para Paralelizar a Manipulação de Dados

Essas técnicas são utilizadas para dividir a carga de trabalho (manipulação de dados na memória) entre p processadores.

8.1 Broadcast 1 para N - EREW (Exclusive Read, Exclusive Write)

O Código 4 mostra uma técnica para copiar um elemento x localizado inicialmente em uma posição de um vetor $M[0]$, para as outras posições desse vetor $M[1], M[2], M[3], \ldots, M[n]$ onde n é o tamanho limite do vetor que se esta trabalhando.

Para executar essa operação são necessários um número de processadores p igual a um número de elementos do vetor n.

Essa técnica garante que não haverá leitura concorrente.

O Código 4 simula uma cópia redundante dos valores de x afim de que esses valores possam ser lidos por cada um dos processadores p em um processamento futuro, garantindo a leitura exclusiva, pois cada processador acessará o valor de memória exclusivo para sua indexação.

Código 4: Broadcast 1 para N

```
1 for k := 0 to [lg p] -1, proc_j, 0 <= j < p do
2         copy M[j] into M[j+(2^k)];
3 endfor
```

A Linha 1 do Código 4 mostra um *for* que será executado por todos os processadores que satisfazerem a condição $0 \leq j < p$, ou seja, o processador atual indexado pelo índice j deve ter seu índice maior ou igual a 0 e menor que p. Caso o processador p_j satisfaça essa condição executará a linha do *for*. O escopo de trabalho do laço vai de $k = 0$ até $k = [(\log_2 p) - 1]$.

Isso significa que em um primeiro passo a posição $M[0]$ será copiada para $M[1]$, a posição $M[1]$ para $M[2]$ e assim sucessivamente. Em um segundo passo a posição $M[0]$ é copiada para $M[2]$, a posição $M[1]$ para $M[3]$ e assim sucessivamente. Em um terceiro passo a posição $M[0]$ seria copiada para $M[4]$, a posição $M[1]$ para $M[5]$ e a posição $M[2]$ para $M[6]$.

Essa técnica apresenta uma desvantagem, pois copia o lixo de uma posição de memória para a outra, até que x chegue nessa célula para ser copiado.

8.2 Broadcast 1 para N Otimizado - EREW (Exclusive Read, Exclusive Write)

Afim de otimizar a perda demonstrada pelo Código 4, propõe-se um algoritmo que resolve o problema da cópia de lixo para os demais processadores, esse código copia somente as posições relevantes.

A utilização de variáveis locais utiliza o espaço dos registradores internos de cada processador, então não há gasto de trabalho para executar pequenas atribuições.

Código 5: Broadcast 1 para N Otimizado

```
1 proc_i write into M[i];
2 s := 1;
3 while s<p, proc_j, 0 <= j < min(s,(p-s)) do
4         copy M[j] into M[j+s];
5         S := 2*s;
6 endwhile
7 proc_j, 0 <= j < p, read M[j];
```

O Código 5 funciona de forma parecida com o Código 4, entretanto, algumas restrições impedem que os processadores que não vão manipular x copiem algo para um outro endereço de memória.

8.3 Broadcast N para N - EREW (Exclusive Read, Exclusive Write)

Em determinadas situações pode-se considerar interessante a distribuição de dados de todos os processadores para todos os outros.

Essa técnica necessita de um vetor $M[n]$ onde n é igual ao número de processadores p.

A técnica consiste em cada um dos processadores armazenar em uma posição de memória o que ele tem a compartilhar.

Código 6: Broadcast N para N

```
1 proc_j, 0 <= j < p, write into M[j];
2 for k := 1 to (p-1), proc_j, 0 <= j < p do
3         read B[(j+k) mod p];
4 endfor
```

Em 1 passo todos os processadores podem armazenar seus respectivos valores na matriz M, isso pode ser observado na Linha 1 do Código 6.

Para garantir a leitura exclusiva, é necessário p passos para que todos os processadores leiam as informações alocadas.

A utilização de MOD garante que a leitura será exclusiva e cíclica, ou seja, enquanto p_0 lê o valor de $M[1]$, p_1 lê a informação de $M[2]$ e o último elemento p_n lê a informação de $M[0]$. No próximo passo os valores se incrementam e, enquanto p_0 lê o valor de $M[2]$, p_1 lê a informação de $M[3]$, o penúltimo $p_{(n-1)}$ lê a informação de $M[0]$ e o último elemento p_n lê a informação de $M[1]$.

8.4 Ordenação Paralela

Uma das formas de se ordenar um conjunto de dados é achar qual a posição do elemento n no vetor.

Para isso sabendo-se qual é o número a ser realocado, pode-se contar quantos são os números menores que ele na coleção de dados.

O Código 7 apresenta uma solução semelhante, na qual cria um *ranking* para eleger quais são os maiores elementos.

Código 7: Broadcast N para N

```
1 proc_j, 0 <= j < p, write 0 into R[j]
2 for k := 1 to (p-1), proc_j, 0 <= j < p do
3         l := (j+k) MOD p
4         if M[l] < M[j] or (M[l] = M[j] and l < j) then
5                 A[j] := A[j] + 1;
6         endif
7 endfor
8 proc_j, 0 <= j < p write M[j] into M[A[j]]
```

- A matriz M contém os dados a serem ordenados;
- A matriz A contém o ranking dos números atacados.

Para isso é necessários uma matriz auxiliar que irá incrementar o *ranking* de um determinado número, até que todos sejam cotados. Depois disso basta ordenar o vetor baseando-se no *ranking*.

Mas nem sempre a solução paralela é ideal, nesse exemplo tivemos mais comparações que um processamento de ordenação sequencial.

9 Semigrupos

Um semi-grupo pode ser definido como:

1. um conjunto G dotado de uma operação binária para a qual valem as seguintes propriedades:
   (a) fechamento: dado $a, b \in G$ o elemento resultante da composição de a e b pertence a $G(a * b \in G)$;
   (b) associatividade: para todos $a, b, c \in G$ vale $(a * b) * c = a * (b * c) = a * b * c$;

A condição primária para efetuar uma operação é que a ordem das operações não influencia no resultado final. Por exemplo, tendo o seguinte grupo de elementos a serem calculados:

$\Sigma = \{p \otimes q \otimes r \otimes s \otimes t\}$;

Tanto faz a ordem com que os elementos $\{p, q, r, s, t\}$ serão calculados, o resultado não se alterará.

O seguinte algoritmo propõe uma solução para o calculo de semigrupos com p processadores igual a n número de elementos.

Código 8: Semigrupos

```
1 proc_j, 0 <= j < p, copy X[j] into M[j];
2 s := 1;
3 while  s < p, proc_j, 0 <= s < p-s do
4         M[j+s] := M[j] <operador> M[j+s]
5         s := s*2;
6 endwhile
7 broadcast M[p-1] to all proc
```

Na primeira linha do Código 8 cada um dos processadores j copia o valor de seu índice do vetor $X[j]$ para $M[j]$.

Na segunda linha a variável s é inicializada em 1.

Esse algoritmo possui um laço que restringe a execução de $p - s$ processadores a cada iteração. Isso faz com que em um primeiro passo o vetor M tenha os seguintes valores: $M[0] = \sum\{0, 0\}, M[1] = \sum\{1, 1\}, M[2] = \sum\{2, 2\}, \ldots, M[n] = \sum\{n, n\}$. Em um segundo passo cada processador calcula e armazena o valor de $[p-1 : p]$, isso faz com que o vetor fique com os seguintes valores: $M[0] = \sum\{0, 0\}, M[1] = \sum\{0, 1\}, M[2] = \sum\{1, 2\}, \ldots, M[n] = \sum\{n-1, n\}$.

Em um terceiro passo cada processador calcula e armazena o valor de $[p-2:p]$ isso faz com que o vetor fique com os seguintes valores: $M[0] = \sum\{0,0\}, M[1] = \sum\{0,1\}, M[2] = \sum\{0,2\}, \ldots, M[n] = \sum\{n-2,n\}$. E assim sucessivamente.

9.1 Análise de Algoritmo

O custo do algoritmo apresentado para quantidade de números n igual a 32 e a quantidade de processadores p também igual a 32 é de 5 passos, ou seja em 5 passos a última posição do vetor M irá possuir a soma de todos os elementos. Entretanto além de uma grande quantidade de processadores não trabalharem o custo em processamento é maior que o mesmo algoritmo de forma sequencial que apesar de 31 passos, faz menos operações do tipo $\otimes$.

Para cada algoritmo é possível gerar um coeficiente ζ chamado de ganho. Para gerar esse coeficiente é necessário computar o custo de cada uma das instruções que o algoritmo executa, bem como quantos processadores estão executando esse algoritmo.

9.1.0.4 SITUAÇÃO 1: A primeira situação trabalha com o algoritmo exibido no Código 8 utilizando 32 processadores p para um conjunto de 32 números n.

Nesse caso a fórmula que gera o ganho é:

1. $$\zeta = 1 + \lg p + \lg p \equiv$$

2. $$\zeta = 2 * (\lg p) + 1 \equiv$$

3. $$\zeta = \frac{p}{2 * \lg p + 1}$$

A equação 1 é obtida tendo 1 como o gasto para efetuar a cópia do vetor X para M como o número de processadores é igual ao conjunto numérico, essa cópia é feita em 1 passo. O primeiro $\lg p$ é o número de de iterações necessárias para completar o laço que faz as operações, e o segundo $\lg p$ é o gasto necessário para fazer o *broadcast* mostrado na linha 7.

O ganho então desse algoritmo é de:

1. $$\zeta = 1 + \lg 32 + \lg 32$$

2.
$$\zeta = 2 * (\lg 32) + 1$$

3.
$$\zeta = \frac{32}{2 * \lg 32 + 1}$$

4.
$$\zeta = \frac{32}{(2 * 5) + 1}$$

5.
$$\zeta = \frac{32}{11}$$

6.
$$\zeta = 2.90$$

9.1.0.5 SITUAÇÃO 2: A segunda situação trabalha com o algoritmo exibido no Código 8 utilizando 4 processadores p para um conjunto de 32 números n.

Nesse caso a fórmula que gera o ganho é:

1.
$$\zeta = \frac{n}{p} + \lg p + \lg p \equiv$$

2.
$$\zeta = 2 * (\lg p) + \frac{n}{p} \equiv$$

3.
$$\zeta = \frac{n}{2 * \lg p + \frac{n}{p}}$$

Nesse caso a diferença está no custo para a execução da cópia do vetor X para o vetor M. O custo nesse caso é $\frac{n}{p}$.

Calculando o ganho:

1.
$$\zeta = \frac{32}{4} + \lg 4 + \lg 4 \equiv$$

2.
$$\zeta = 2 * (\lg 4) + \frac{32}{4} \equiv$$

3.
$$\zeta = \frac{32}{2 * \lg 4 + \frac{32}{4}}$$

4.
$$\zeta = \frac{32}{8 + 4}$$

5.
$$\zeta = \frac{32}{12}$$

6.
$$\zeta = 2.66$$

9.1.0.6 COMPARAÇÃO: É possível observar que nem sempre aumentando o número de processadores, aumentaremos também a performance. Para cada algoritmo em específico, temos um número limite de processadores que elevam ao máximo o ganho.

10 Salto de Apontadores

Imaginando um lista de n elementos que possuem um apontador para $n_i + 1$, vamos estudar a possibilidade de paralelizar a geração de um *ranking* para tal lista.

A seguinte lista é apresentada em seu estado inicial:

Tabela 1: Salto de Apontadores em Paralelo

ID	Elemento	Aponta
0	A	4
1	B	0
2	C	1
3	D	2
4	E	5
5	F	5

Nessa lista temos uma identificação ID dos elementos, uma rótulo para cada um deles (A,B,C,...) e um indicador de ponteiro, que marca a qual id o elemento atual aponta, assim poderíamos representar essa lista de forma gráfica por:

$$D \rightarrow C \rightarrow B \rightarrow A \rightarrow E \rightarrow F$$

A idéia do algoritmo é alterar a cada passo, os sucessores dos elementos até que o sucessor de todos os elementos seja o último elemento F.

Para classificar o ranking do conjunto de elementos, basta a cada passo somar a distância atual, mais a distância do elemento que acabou de ser apontado.

A seguinte tabela verdade mostra os passos percorridos para classificar os elementos:

Tabela 2: Salto de Apontadores em Paralelo: Tabela Verdade

ID	Elemento	Aponta	P1	P2	P3	R0	R1	R2	R3
0	A	4	5	5	5	1	2	2	2
1	B	0	4	5	5	1	2	3	3
2	C	1	0	5	5	1	2	4	4
3	D	2	1	4	5	1	2	4	5
4	E	5	5	5	5	1	1	1	1
5	F	5	5	5	5	0	0	0	0

Inicialmente todos os elementos, menos o último, assumem a distância 1, enquanto o último assume a distância 0. Tabela 2 R_0.

P_n Representa a alteração dos ponteiros a cada passo, e R_n representa a construção da tabela de *ranking*.

Em R_1, o elemento A com ID 0 passa a valer 2 pois é obtido da soma de R_0 elemento A com ID 0 que vale 1 mais o valor de *ranking* de quem era apontado por A, no caso, E com valor também 1, obtendo-se 2.

Essa mesma regra é aplicada para os demais elementos nos demais passos.

Quando um elemento já se encontra na posição de *ranking* correta, seu ranking não é incrementado pois ele sempre irá somar 0 que é o valor de *ranking* da última posição.

Um pseudo algoritmo que representa essa situação é:

Código 9: Salto de Apontadores

```
1 proc j, 0 <= j < p, do
2         if next[j] = j then
3                 rank[j] := 0;
4         else
5                 rank[j] := 1;
6         endif
7         while rank[next[head]] <> 0, proc j, 0 <= j < p do
```

```
8                   rank[j] := rank[j] + rank[next[j]];
9                   next[j] := next[next[j]];
10          endwhile
11  end
```

Esse problema que tinha características de um problema sequencial pode ser resolvido com p processadores igual a n elementos.

Entretanto, nem sempre teremos o número de processadores igual ao número de elementos, nesse caso, é possível se adotar uma técnica de produtor-consumidor para elaborar um algoritmo onde o produto a ser gerado são os saltos de apontadores e o consumo é a própria operação que efetua o salto e incrementa o *ranking*.

Essa técnica apesar de muito útil na maioria dos casos onde não se enxerga outra maneira de paralelizar o problema, apresenta um contratempo pois a pilha de ítens (que armazena o que deve ser feito) terá seu acesso concorrido pelos p processadores.

Outra técnica adotada foi a de se armazenar os saltos no próprio vetor, e não mais em uma pilha. Essa técnica em um primeiro momento pareceu causar sobre-escritas, pois os dados que dependeriam dos valores antigos para a computação, adotaria dados já computados, tornando o algoritmo inviável. Ao testar a técnica, observou-se uma conversão em tempo menor e de forma correta, pois adotava-se a relação dos saltos para saltos que já haviam sido atualizados. É importante lembrar que considerou-se que a ordem dos elementos pode alterar o tempo de conversão, entretanto o mesmo algoritmo não é mais custoso que o produtor-consumidor.

11 Multiplicação de Matriz

A multiplicação de matrizes é dada pela fórmula:

$$C_{ij} = \sum_{k=0}^{m-1} a_{ij} * k_{ij}$$

Além das técnicas utilizadas a seguir, que utilizam a matriz completa, é possível subdividir a matriz em quadrantes e delegar cada um dos quadrantes para p processadores, uma boa abordagem é a utilização de $\sqrt{p}$ para divisão, entretanto ainda é necessário em todos os casos garantir a leitura exclusiva através da técnica de Broadcast 1 para N Otimizado tratada anteriormente.

11.1 Multiplicação de Matriz com M^2 Processadores

É dada pelo código PRAM:

Código 10: Multiplicação de Matriz com M^2 Processadores

```
1 proc (i,j), 0 <= i,j < m do
2         t := 0;
3         for k := - to m-1 do
4                 t:= t + a[i][k] * B[k][j];
5         endfor
6         c[i][j] := t;
7 end
```

A indexação dos processadores foi assumida como 2 dimensões.

11.2 Multiplicação de Matriz com M Processadores

É dada pelo código PRAM:

Código 11: Multiplicação de Matriz com M Processadores

```
1 for j := 0 to m-1, proc i, 0 <= i < m do
2         t := 0;
3         for k := 0 to m-1 do
4                 t := t + a[i][k] + b[k][j]
5         endfor
6 endfor
```

12 Ordenação

Algumas abordagens foram discutidas para se classificar um conjunto de n elementos.

Uma das técnicas discutidas foi a de se dividir a sequência de n elementos, até que essa subsequência esteja ordenada. Nos piores casos teríamos um conjunto de apenas dois elementos. Até o passo de divisão dos elementos o idéia de paralização era viável, pois basta dividir a sequência em partes e entregar cada parte para um thread.

O problema identificado foi no momento de se fazer a junção dos pedaços menores. (*merge*).

Esse problema gerou algumas idéias, e chegou-se a conclusão que a única maneira de se executar esse procedimento era com uma busca binária, onde procurava-se num conjunto p_2 o lugar do elemento e do conjunto p_1. Com a pressuposição de que os conjuntos p_1 e p_2 já estavam ordenados.

12.1 Proposta de Batcher

Utilizando uma das propostas de Batcher, foi possível melhorar a idéia do algoritmo anterior e achar o custo mínimo para uma ordenação em paralelo que é dada por $\Theta(n \lg^4 n)$.

Sua proposta é a de separar os números pares dos números ímpares.

Supondo um conjunto:

$$\{x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3\}$$

Onde:

$$x_0 \leqslant x_1 \leqslant x_2 \leqslant x_3$$
$$y_0 \leqslant y_1 \leqslant y_2 \leqslant y_3$$

Elementos x são pares e elementos y são ímpares.

Adota-se então a estratégia de se comparar os elementos pares $\{x_0, y_0\}$, $\{x_2, y_2\}$ e os elementos ímpares $\{x_1, y_1\}$, $\{x_3, y_3\}$.

ESTRATÉGIA PARA COMPARAÇÃO: Deve-se adotar as seguintes regras:

- Para uma comparação $\{a, b\}$ gera-se uma resultado $\{r_0, r_1\}$, onde r_0 é $MIN(a, b)$ e r_1 é $MAX(a, b)$;
- Depois de geradas as combinações $r_0 \ldots r_n$ e $s_0 \ldots s_n$ se obtém um conjunto ordenado $v_0, v_1, v_2, \ldots, v_n$, onde $v_0 = p_0$ e $v_n = i_n$;
- Os elementos entre v_0 e v_n são obtidos pela regra: v_1 é uma comparação entre $\{r_1, s_0\}$ e v_2 é o outro elemento da comparação.

EXEMPLO: Ordenar:

$$\{2, 6, 10, 12, 14, 1, 5, 7, 9, 11\}$$

$$\{x_0 = 2\}\{x_1 = 6\}\{x_2 = 10\}\{x_3 = 12\}\{x_4 = 14\}\{x_5 = 18\}$$
$$\{y_0 = 1\}\{y_1 = 5\}\{y_2 = 7\}\{y_3 = 9\}\{y_4 = 11\}\{y_5 = 15\}$$

Compara-se elementos de x e y com índices par $0, 2, 4$.

$$r_0 = MIN(x_0, y_0) \therefore r_0 = MIN(2, 1) \therefore r_0 = 1$$
$$r_1 = 2$$
$$s_0 = MIN(x_2, y_2) \therefore s_0 = MIN(10, 7) \therefore s_0 = 7$$
$$s_1 = 10$$
$$t_0 = MIN(x_4, y_4) \therefore t_0 = MIN(14, 11) \therefore t_0 = 11$$
$$t_1 = 14$$
$$v_0 = r_0 \therefore 1$$
$$v_1 = MIN(r_1, s_0) \therefore v_0 = MIN(2, 7) \therefore v_0 = 2$$
$$v_2 = 7$$
$$v_3 = MIN(s_1, t_0) \therefore v_2 = MIN(10, 11) \therefore v_2 = 10$$
$$v_4 = 11$$
$$v_5 = t_1 \therefore 14$$

Obtém-se então: $v_0 = 1, v_1 = 2, v_2 = 7, v_3 = 10, v_4 = 11, v_5 = 14$.

Compara-se elementos de x e y com índices ímpares $1, 3, 5$.

$$g_0 = MIN(x_1, y_1) \therefore g_0 = MIN(6, 5) \therefore g_0 = 5$$
$$g_1 = 6$$
$$h_0 = MIN(x_3, y_3) \therefore h_0 = MIN(12, 9) \therefore h_0 = 9$$
$$h_1 = 12$$
$$i_0 = MIN(x_5, y_5) \therefore i_0 = MIN(18, 15) \therefore i_0 = 15$$
$$i_1 = 18 w_0 = g_0 \therefore 5$$
$$w_1 = MIN(g_1, h_0) \therefore w_0 = MIN(6, 9) \therefore w_0 = 6$$
$$w_2 = 9$$
$$w_3 = MIN(h_1, i_0) \therefore w_3 = MIN(12, 15) \therefore w_3 = 12$$
$$w_4 = 15$$
$$w_5 = i_1 \therefore 18$$

Obtém-se então: $w_0 = 5, w_1 = 6, w_2 = 9, w_3 = 12, w_4 = 15, w_5 = 18$.

Basta agora juntar os dois grupos utilizando a mesma lógica para a formação de v e w.

Temos então:

$$z_0 = v_0 \therefore 1 z_1 = MIN(v_1, w_0) \therefore z_1 = MIN(2, 5) \therefore z_1 = 2$$
$$z_2 = 5 z_3 = MIN(v_2, w_1) \therefore z_3 = MIN(7, 6) \therefore z_3 = 6$$
$$z_4 = 7 z_5 = MIN(v_3, w_2) \therefore z_5 = MIN(10, 9) \therefore w_0 = 9$$
$$z_6 = 10 z_7 = MIN(v_4, w_3) \therefore z_7 = MIN(11, 12) \therefore w_0 = 11$$
$$z_8 = 12 z_9 = w_5 \therefore 18$$

O conjunto ordenado fica: $z_0 = 1, z_1 = 2, z_3 = 6, z_4 = 7, z_5 = 9, z_6 = 10, z_7 = 11, z_8 = 12, z_9 = 18$.

Outra proposta de Batcher, conhecida como Bitônica, também com o mesmo custo de $\Theta(n \lg^4 n)$, é a utilização de dois conjuntos de dados separados, em um conjunto c_1 têm-se números ordenados crescentemente e em um conjunto c_2 é dado por uma ordenação decrescente.

O algoritmo para a ordenação Bitônica pode ser visto na página 140 do livro *Introduction do Parallel Processing*.

13 Utilizando um modelo PRAM CRCW-Sum para Ordenação

Uma máquina CRCW-Sum garante leitura e escrita concorrente, sendo que se p processadores tentarem gravar um dado d em um mesmo espaço de memória o resultado final nesse passo será de $\sum c$.

Lembrando isso, é possível estudar o seguinte código PRAM:

Esse código faz a ordenação de n elementos em $\Theta(1)$ com $p = n^2$.

Código 12: Utilizando um modelo PRAM CRCW-Sum para Ordenação

```
1 proc p[i,j], 1 <= i,j <= n do
2         if A[i] > A[j] OR (A[i] = A[j] AND i<j) then
3                 C[i] = 1;
4         else
5                 C[i] = 0;
6         endif
7
8 proc p[i,j], 1 <= i < n do
```

```
9           A[C[i]+1] = A[i];
```

A idéia do código é que se conte quantos elementos menores existem em relação ao elemento que o processador $p[i, j]$ está cuidando.

14 Radix sort

O Radix sort é um algoritmo de ordenação que pode ser usado para ordenar ítens que estão identificados por chaves únicas.

Cada chave é uma cadeia de caracteres ou número.

Computadores, na sua maioria, representam internamente todos os tipo de dados como números binários, por isso processar os dígitos na forma de inteiros em grupos representados por dígitos binários se torna mais conveniente.

Existem duas classificações do radix sort, que são:

- Dígito menos significativo - Começa do dígito menos significativo até o mais significativo, ordenando tipicamente da seguinte forma: chaves curtas vem antes de chaves longas, Isso coincide com a ordem normal de representação dos inteiros, como a seqüência "1, 2, 3, 4, 5, 6, 7, 8, 9, 10";
- Dígito mais significativo - Trabalha no sentido contrário, usando sempre a ordem lexicográfica, que é adequada para ordenação de strings, como palavras, ou representações de inteiros com tamanho fixo. A seqüência "b, c, d, e, f, g, h, i, j, ba" será ordenada lexicograficamente como "b, ba, c, d, e, f, g, h, i, j".

15 Bucket sort

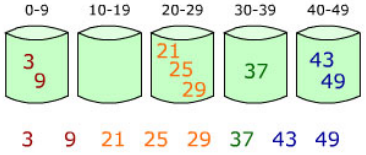
Bucket sort, ou bin sort, é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipientes. Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente, ou usando o algoritmo bucket sort recursivamente. O bucket sort tem complexidade linear $\Theta(n)$ quando o vetor a ser ordenado contém valores que são uniformemente distribuídos.

Bucket sort funciona do seguinte modo:

- Inicialize um vetor de "baldes", inicialmente vazios;
- Vá para o vetor original, incluindo cada elemento em um balde.

- Ordene todos os baldes não vazios.
- Coloque os elementos dos baldes que não estão vazios no vetor original.

Figura 9: Exemplo bucket sort



16 Algoritmo de Prim

O algoritmo de Prim é um algoritmo em teoria dos grafos que busca uma árvore geradora mínima para um grafo conexo com pesos. O algoritmo de Prim é um exemplo de um algoritmo guloso.

O subconjunto S forma uma única árvore, e a aresta segura adicionada a S é sempre uma aresta de peso mínimo conectando a árvore a um vértice que não esteja na árvore.

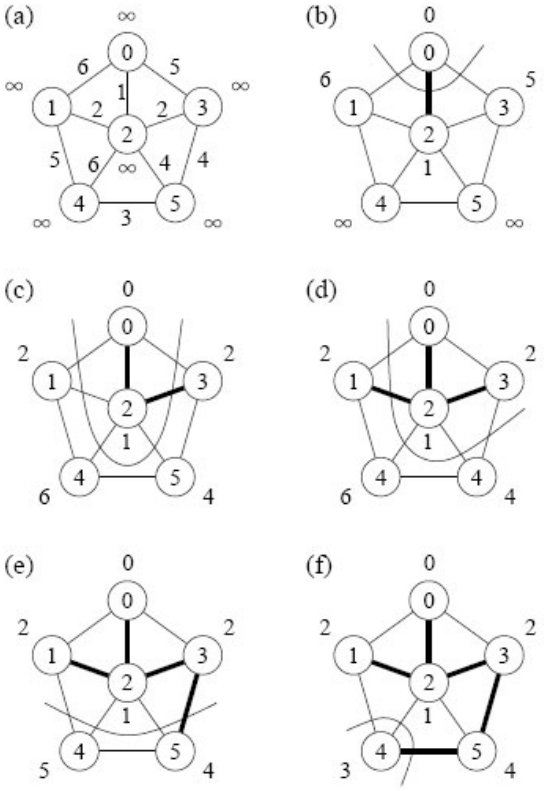
A árvore começa por um vértice qualquer e cresce até que "gere" todos os vértices em V.

A cada passo, uma aresta leve é adicionada à árvore S, conectando S a um vértice de $GS = (V; S)$.

De acordo com o teorema anterior, quando o algoritmo termina, as arestas em S formam uma árvore geradora mínima.

A ordem de complexidade para o algoritmo de Prim é $\Theta(|E| \log |V|)$, em que E é o conjunto de arestas e V é o conjunto de vértices do grafo.

Figura 10: Exemplo algoritmo de Prim



17 Leis de Amdahl e Gustafson

Esses leis descrevem o speedup ganhado por um algoritmo escrito de forma paralela.

Sejam,

s *speedup*;

n o número de processadores;

p a porcentagem paralela de um algoritmo;

As fórmulas são dadas por:

Lei de Amadahl:

$$s(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Assim sendo, podemos podemos analisar a lei de Amadahl atribuindo valores teste para $\{s, n, p\}$.

$p = 0,5$		$p = 0,95$	
n	s	n	s
1	1	1	1
2	1.33	2	1.94
4	1.6	512	19.28
8	1.77	4096	19.40
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	$\frac{1}{0,5+\xi}$	n	20

Pode-se observar que existe um limite de convergência que é atingido. Essa lei não leva em consideração o tamanho da entrada de dados, assim sendo, não expressa a real representação de um algoritmo paralelo, mas pode ser tomada como limiar inferior para medir o *speedup* de um código paralelo.

Lei de Gustafson:

$$s(n) = n - (1 - p) * (n - 1)$$

Assim sendo, podemos podemos analisar a lei de Gustafson atribuindo valores teste para $\{s, n, p\}$.

$p = 0,5$		$p = 0,95$	
n	s	n	s
1	1	1	1
2	1.6	2	1.95
4	2.5	32	30.45
8	4.5	512	486.45
16	8.5	4096	3891.25
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Essa lei mostra que o s está fixado com o valor de n, e sabemos que nem sempre isso é verdade, mas pode ser tomada como um limiar superior para medir o *speedup* de um algoritmo paralelo.

Uma boa prática é testar se o algoritmo paralelo gerado, se encaixa em algo próximo do limite inferior (lei de Amadahl) e o limite superior (lei de Gustafson).

17.1 Como achar p

Uma boa técnica para se encontrar porcentagem paralela do algoritmo escrito é a contagem de tempo de todos os trechos (sequenciais e paralelos) e estabelecer uma diferença entre eles.

18 Algoritmo de Boruvka

O algoritmo de Boruvka (publicado em 1926) se apoia nas condições de otimalidade de MSTs para encontrar uma MST de um grafo com custos nas arestas.

Vamos nos restringir a grafos conexos. Essa restrição simplifica a discussão, embora seja irrelevante para o algoritmo. (Se for aplicado a um grafo desconexo, o algoritmo produzirá uma MST em cada componente.)

Para descrever o algoritmo, é importante lembrar que cada aresta um grafo consiste em dois arcos anti-paralelos. Vamos supor (em consonância com nossa estrutura de dados) que o custo de cada um desses dois arcos é igual ao custo da aresta. O algoritmo manipula os dois arcos de cada aresta independentemente. Assim, durante a execução do algoritmo, pode ocorrer que apenas um dos arcos de algumas arestas é escolhido para fazer parte da árvore geradora. No fim, entretanto, ambos os arcos de cada aresta da árvore estarão presentes.

Será necessário fazer uma adaptação previsível do conceito de franja. Diremos que a franja (= fringe) de uma subárvore T de nosso grafo G é o conjunto de todos os arcos que saem de T, ou seja, têm ponta inicial em T e ponta final fora de T. Diremos, ainda, que um arco é externo a uma floresta geradora F de G se tiver pontas em duas componentes diferentes de F. É claro que cada arco externo a F está na franja de exatamente uma das componentes de F.

Cada iteração do algoritmo de Boruvka começa com uma floresta geradora F de G. (No início da primeira iteração, cada componente de F tem apenas um vértice.) Cada iteração consiste no seguinte:

Código 13: Algoritmo de Boruvka

```
1 se existe algum arco externo a F
2         então  para cada componente T de F
3                          escolha um arco mínimo na franja de T
4                  seja B o conjunto de todos os arcos escolhidos
5                  seja B' um subconjunto maximal de B tal que
6                          para F+B' não tem ciclos de comprimento >=3
7                  seja B'' o conjunto dos arcos anti-paralelos aos de B'
```

```
8                 comece nova iteração com  F+B'+B''  no papel de F
9           senão  pare
```

Esse algoritmo produz uma MST de G (uma vez que estamos supondo G conexo).

Diremos que B é um conjunto de Boruvka. Cada componente da floresta F contribui um arco de sua franja para esse conjunto. (É curioso o sabor de processamento paralelo desse algoritmo.)

Se o grafo $F+B$ não tiver ciclos de comprimento $\geqslant 3$, então B' será igual a B. Caso contrário, será preciso descartar alguns arcos de B para obter B'. O cálculo de B' e B'' e a substituição de F por $F + B' + B''$ são realizados, simultaneamente, pelo seguinte processo iterativo: examine os arcos de B um a um; se um arco for externo à floresta corrente, acrescente-o à floresta juntamente com seu arco anti-paralelo; senão, descarte-o.

Exemplo: A tabela abaixo define um grafo e os custos de suas arestas.

Aresta	0-1	0-2	1-2	1-3
Custo	0.1	0.1	0.1	0.2

No início da primeira iteração do algoritmo de Boruvka, cada componente da floresta F tem um único vértice. Suponha que o algoritmo escolhe os seguintes arcos:

componente	0	1	2	3
arco na franja	0-2	1-0	2-1	3-1

O grafo $F + B$ contém o ciclo $0 - 2 - 1 - 0$, de comprimento $\geqslant 3$. Para obter B', basta eliminar qualquer um dos três arcos do ciclo. Se eliminarmos o arco $2 - 1$, a próxima iteração começará com a floresta que tem arestas $0 - 2$, $0 - 1$ e $1 - 3$. (Se o algoritmo tivesse escolhido o conjunto de arcos abaixo, o grafo $F + B$ não teria ciclos de comprimento $\geqslant 3$, e nesse caso B' seria igual a B.)

componente	0	1	2	3
arco na franja	0-2	1-0	2-0	3-1