Uma Experiência de Implementação de Métodos de Ordenação Paralelos em Máquina SIMD

Rodrigo Lima Carceroni* Wagner Meira Júnior† Márcio Luiz Bunte de Carvalho‡

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais
Caixa Postal 702 – Belo Horizonte – MG – CEP 30.161-970

Resumo

O presente trabalho baseia-se na implementação de três algoritmos de ordenação em uma máquina SIMD com 8K processadores de 1 bit. Os algoritmos implementados foram os seguintes: *Odd-Even Transposition Sort*, um híbrido composto a partir do *Odd-Even* e do *Shellsort*, e *Linar Array Sort*. Os testes realizados mostraram que o algoritmo híbrido é o mais eficiente, se o número de elementos a serem ordenados for relativamente elevado (maior que 4096). Em caso contrário, deve-se utilizar o *Odd-Even*. O algoritmo *Linear Array* apresentou um desempenho mais modesto, por ser mais flexível quanto à geração dos itens a serem ordenados. Adicionalmente, mostrou-se que uma duplicação do número de processadores virtuais utilizados permite, em geral, reduzir o tempo gasto na execução dos métodos estudados pela metade. Finalmente, constatou-se que a eliminação total do paralelismo pode provocar aumentos de até 100 vezes nos tempos de execução dos algoritmos em estudo.

Abstract

This work describes the implementation of three sorting algorithms on a SIMD machine with 8K processors of 1 bit each. These algorithms are: Odd-Even Transposition Sort, a hybrid composed by Odd-Even and Shellsort, and Linear Array Sort. The tests performed have shown that, since the number of items to be sorted is relatively large (greater than 4096), the hybrid algorithm is the most efficient. Otherwise, Odd-Even must be used. Linear Array has been the one which performed worst, due to its greater flexibility with relation to the generation of the items to be sorted. In addition, it was shown that doubling the number of virtual processors used, generally reduces the execution time of these sorting methods by 50%. Finally, it was found that eliminating the parallelism at all can cause the running times of the algorithms in study to be increased by a factor of down to 100.

Este trabalho foi desenvolvido com o apoio do CNPq e da FAPEMIG (Tec 1113/90)

*Bacharelando em Ciência da Computação (UFMG); Algoritmos paralelos, redes neuronais, otimização; E-mail: carceron@dcc.ufmg.br

†Mestrando em Ciência da Computação (UFMG); Algoritmos paralelos, redes neuronais, programação matemática; E-mail: meira@dcc.ufmg.br

‡Doutorando em Pesquisa Operacional (UC Berkeley); Algoritmos paralelos, programação matemática, algebra linear numérica, redes neuronais; Professor do Departamento de Ciência da Computação – UFMG; E-mail: mlbc@dcc.ufmg.br

1 Introdução

Ordenação é uma tarefa requerida freqüentemente e de grande importância teórica na Ciência da Computação. Por isso, muito esforço tem sido despendido em pesquisas com a finalidade de tornar essa tarefa mais rápida, por meio da utilização de paralelismo. Entretanto, os algoritmos desenvolvidos fazem, em geral, uma série de suposições a respeito da configuração da rede de processadores a ser utilizada, sendo ineficientes ou mesmo inviáveis se executados em máquinas cuja arquitetura não está em acordo com tais suposições. Em muitos casos, tais algoritmos acabam por ter reduzida utilidade prática, por exigirem a utilização de *hardware* com custo excessivamente elevado.

O objetivo do presente trabalho é estudar de que forma pode-se realizar ordenação em paralelo de forma eficiente, dada a existência de certas restrições na rede de processadores disponível. Mais especificamente, foi utilizada uma determinada máquina SIMD (Single Instruction Multiple Data), que apresenta certas limitações de comunicação entre seus processadores. Tais restrições, assim como outras peculiaridades da máquina utilizada, serão expostas na seção seguinte. Na seção 3 serão caracterizados os algoritmos de ordenação estudados. Na seção 4 serão descritos os testes realizados e analisados os resultados obtidos. Por fim, na seção 5 serão apresentadas as conclusões do presente trabalho.

2 A Máquina SIMD utilizada

2.1 Arquitetura

A *Zephyr-Wavetracer* (WT) é uma máquina SIMD com 8192 processadores de 1 bit cada, os quais são dispostos em duas placas, cada uma com 4K processadores. Essa máquina é ligada a uma *workstation* hospedeira por uma interface SCSI. No caso do presente trabalho, a *workstation* utilizada foi uma *Sun Sparc2*. A memória total da máquina (da ordem de 256 Mbytes) é igualmente dividida entre os processadores, de forma tal que para para cada um deles são reservados 32Kbytes. Além dessa memória externa, cada processador possui uma memória interna de alta velocidade (*cache*) de 2Kbits.

Os processadores podem ser organizados bi ou tridimensionalmente, sendo essa configuração realizada via *software*, pelos programas aplicativos. O arranjo bidimensional tem dimensões 64×128 e o tridimensional $16 \times 32 \times 16$. A essas dimensões do arranjo de processadores denomina-se espaço de solução. Assim, em um arranjo de 64×128 pode-se manipular simultânea e identicamente 8192 elementos.

Uma característica interessante da WT é que ela pode suportar um espaço de solução com dimensões maiores que as dos arranjos de processadores físicos. Isso ocorre por meio de um particionamento automático da memória de cada um desses processadores em um certo número P de partes iguais, a cada uma das quais é associado um ponto do espaço de solução. Com isso, cada processador físico passa a se comportar como P processadores virtuais.

2.2 Linguagem MultiC

As aplicações destinadas a serem executadas na WT devem ser implementadas com o uso da linguagem *MultiC*, que é uma extensão do *Ansi C*. Entre as características incorporadas nessa nova linguagem, a qual é descrita de forma detalhada em [1], destacam-se as seguintes:

- O especificador de tipos multi, que é usado para declarar variáveis com valores numéricos independentes em cada processador virtual.
- O operador de comunicação interprocessador que permite movimentar uma variável multi ao longo das dimensões do espaço de solução
- Os operadores de redução, através dos quais pode-se combinar todos os diferentes valores numéricos de uma variavel multi nos processadores virtuais.

O programa escrito em *MultiC* é então compilado e executado na *workstation* hospedeira, de forma tal que a WT atua como um co-processador responsável pela realização das operações que envolvem variáveis

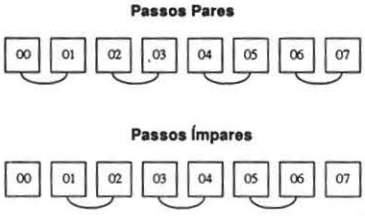

Figura 1: Processo de seleção das duplas do algoritmo *Odd-Even*

do tipo multi. Com isso, o programa de aplicação tem acesso direto às facilidades da *workstation*, como seu sistema de arquivos, facilidades de rede e ambientes gráficos. Além disso, as dimensões do espaço de solução são especificadas em tempo de execução, de forma totalmente independente do arranjo de processadores físicos, o que permite que os programas escritos em *MultiC* possam ser executados em máquinas te diferentes tamanhos, sem necessidade de recompilação.

3 Algoritmos de ordenação estudados

Sabe-se que o tempo mínimo para ordenar N itens por meio de comparações e com um número de processadores não superior a N é $O(logN)$ [2]. No entanto, os algoritmos $O(logN)$ fazem uma série de suposições sobre a topologia da rede de processadores a ser utilizada, às quais não se adequa o arranjo de processadores físicos da WT. Levando-se em consideração as peculiaridades da arquitetura descrita na seção anterior, decidiu-se então por estudar três algoritmos de ordenação $O(N)$, que são descritos a seguir:

3.1 Odd-Even Transposition Sort

Esse algoritmo, descrito em [3], requer N passos para ordenar um vetor com N itens. Cada um desses passos consiste em dividir o vetor em uma série de duplas de itens situados em posições adjacentes, as quais são individualmente ordenadas pela realização de uma comparação e uma possível troca.

A fim de garantir a correção desse método de ordenação, dois critérios de escolha das duplas são utilizados intercaladamente. Sejam os N passos numerados de 0 a $N-1$. Nos passos pares, as duplas são formadas a partir da posição inicial do vetor, de forma tal que o i-ésimo item ($0 \leq i < N$) é associado ao item $i+1$, se i é par (e $i \neq N-1$), e é associado ao item $i-1$, se i é ímpar (e $i \neq 0$). Já nos passos ímpares, as duplas são formadas de forma tal que o i-ésimo item é associado ao item $i-1$, se i é par (e $i \neq 0$), e ao item $i+1$ se i é ímpar (e $i \neq N-1$). O processo de seleção das duplas é ilustrado na Figura 1.

3.2 Híbrido Shellsort/Odd-Even

O algoritmo descrito na seção anterior pode ser adaptado para realizar um número de passos menor do que N. Para tanto, basta que seu critério de parada seja modificado de forma a levar em consideração o fato de que o vetor já estará ordenado, se nenhuma troca for realizada em dois passos consecutivos. De forma geral, entretanto, essa modificação não melhora o desempenho do algoritmo, uma vez que o número de passos realizados continua sendo bastante próximo de N. Decidiu-se então preprocessar o vetor a ser ordenado pela aplicação de uma versão modificada do algoritmo *Shellsort*, com o objetivo de reduzir o número de passos a serem realizados em média, antes que o novo critério de parada seja verificado.

A versão modificada do algoritmo *Shellsort* que foi utilizada consiste da realização de um número de passos variável (K), não superior a $log_2 N$. Sejam esses passos numerados de 0 a $K-1$. O i-ésimo passo é realizado pela divisão do vetor em $M = N \times 2^{-i-1}$ subconjuntos de 2^{i+1} itens, de maneira tal que, entre as posições de cada par de elementos pertencentes a um mesmo subconjunto, exista pelo menos um elemento pertencente

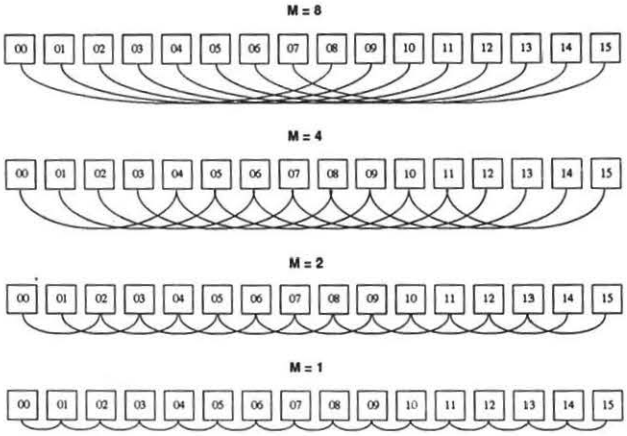

Figura 2: Processo de seleção dos subconjuntos do algoritmo *Shellsort*

a cada um dos demais subconjuntos. Sejam os elementos de cada um desses subconjuntos numerados de 0 a $\frac{N}{M} - 1$. São então realizados $\frac{N}{M} - 1$ subpassos, sendo que o j-ésimo subpasso ($0 \leq j < \frac{N}{M} - 1$ consiste em ordenar, para cada subconjunto, a dupla formada pelos elementos de número $\frac{N}{M} - j - 1$ e $\frac{N}{M} - j - 2$. O funcionamento desse algoritmo, que não ordena totalmente o vetor, mas apenas aproxima os itens de suas posições ideais, é ilustrado na Figura 2.

3.3 Linear Array Sort

Esse algoritmo, descrito em [4] e [5], se caracteriza pela utilização de um arranjo linear de $\frac{N}{2}$ nodos (processadores), onde N é o menor número par que é maior ou igual ao número de itens a serem ordenados. Cada nodo i pertencente ao arranjo, $0 \leq i < \frac{N}{2}$, é conectado bidirecionalmente ao nodo $i-1$ (denominado vizinho da esquerda), se $i > 0$, e ao nodo $i + 1$ (denominado vizinho de direita), se $i < \frac{N}{2} - 1$. O nodo 0 funciona como entrada e como saída para o arranjo, sendo o único nodo diretamente conectado ao mundo exterior.

O processo de ordenação pode ser dividido em duas fases. Na primeira delas, os itens a serem ordenados são apresentados seqüencialmente ao nodo 0, sendo propagados para a direita até que o último deles tenha sido introduzido no arranjo. Já na segunda fase, a propagação ocorre no sentido inverso e os itens ordenados são retirados seqüencialmente do nodo 0. A ordenação termina quando o último item é retirado do arranjo.

Cada nodo do arranjo linear pode armazenar até dois itens em registradores internos, sendo capaz de compará-los, enviar um deles para um de seus vizinhos e receber um novo item do outro vizinho, em um único ciclo. Durante a primeira fase, cada nodo se comporta da seguinte maneira, a cada ciclo: se dois itens estiverem armazenados em seu interior, ele os compara e envia o maior para o seu vizinho da direita, recebendo um novo item do seu vizinho da esquerda. Em caso contrário, o nodo apenas recebe um item possivelmente enviado pelo seu vizinho da esquerda, armazenando-o em um de seus registradores internos. Já na segunda fase, cada nodo ocupado por dois itens realiza, a cada ciclo, a comparação entre eles e envia o menor para o seu vizinho da esquerda, recebendo (possivelmente) um novo item do seu vizinho da direita. Se algum nodo estiver ocupado por um único item, esse item é propagado para o vizinho da esquerda.

A grande motivação para a utilização do algoritmo descrito acima é que ele permite que os itens a serem ordenados sejam gerados seqüencialmente, o que ocorre em aplicações como a emulação de PRAM [6], onde tais itens não se encontram todos disponíveis inicialmente. Nesse caso, pode-se provar que o presente algoritmo requer um número mínimo de ciclos ($2N$) para que o processo de ordenação seja completado.

4 Testes e resultados

Terminada a implementação dos algoritmos descritos anteriormente, foram então realizados testes com o objetivo de comparar o desempenho de cada um deles e avaliar o ganho obtido com o paralelismo. No caso do algoritmo híbrido, no entanto, esse desempenho depende diretamente do valor de K, ou seja, do número de passos a serem realizados pela versão modificada do *Shellsort*. Esse número de passos (K), por sua vez, é determinado diretamente pelo valor atribuido a M (número de subconjuntos a serem formados no último passo). Portanto, inicialmente tentou-se determinar qual o valor a ser atribuido a M de forma a que o desempenho do algoritmo híbrido fosse otimizado.

4.1 Otimização do método híbrido

Uma das razões pelas quais o valor de M influi no desempenho desse algoritmo é que tal valor determina o número médio de passos realizados pelo método *Odd-Even* (com o novo critério de parada). Por isso, um primeiro teste realizado consistiu em observar de que forma esse número médio de passos (N_p) responde a variações do valor de M, para diferentes tamanhos (N) do vetor a ser ordenado. Foram fixados 5 valores distintos para N: 128, 256, 512, 1024 e 2048. Para cada um deles, foram utilizados todos os valores de M correspondentes a uma potência de 2, entre 1 e N. Finalmente, para cada par (N,M), foram realizadas 50 execuções independentes do método híbrido, sendo o vetor a ser ordenado gerado aleatoriamente a cada execução (assim como em todos os demais testes). Foram considerados para propósito de análise os valores médios verificados para N_p. Os resultados obtidos foram sintetizados no gráfico da Figura 3.

Pode-se observar que, de forma genérica, N_p permanece mais ou menos constante, em um patamar próximo de $\frac{N}{2}$, para valores de M inferiores a $\sqrt{N}$. Para os demais valores de M, no entanto, N_p aumenta, aproximando-se bastante do seu limite máximo igual a N. Portanto, é possível inferir que o valor a ser fixado para M não deve ser inferior a $\sqrt{N}$, uma vez que os passos do *Shellsort* realizados abaixo desse limite em nada contribuem para uma melhora do desempenho do método híbrido como um todo.

Um outro teste realizado, ainda com o objetivo de determinar o valor ótimo de M, consistiu em medir o tempo médio gasto por uma execução do processo de ordenação para diferentes pares (N,M). Desta vez, foram fixados 6 valores distintos para N: 128, 256, 512, 1024, 2048 e 4096. Da mesma forma que no teste anterior, M foi variado entre 1 e N. Para cada par (N,M) foram realizadas 5 execuções independentes do processo de ordenação. A partir dos valores médios medidos do tempo gasto, foi então construido o gráfico mostrado na Figura 4.

Em acordo com os resultados obtidos no teste anterior, pode-se verificar que o tempo gasto na ordenação de um vetor de tamanho N cai à medida em que o valor de M cresce de 1 até cerca de $\sqrt{N}$. Além disso, observa-se que quando M é aumentado além desse ponto, o tempo de execução começa a crescer, chegando a atingir, para $M = N$, um valor entre 33% e 53% maior que o mínimo. Portanto, esse teste indica que o valor de M deve ser fixado em $\sqrt{N}$ a fim de que o algoritmo de ordenação híbrido apresente eficiência máxima. Adicionalmente, se for observado que, para $M = N$, nenhum passo do *Shellsort* é realizado, pode-se esperar que o método híbrido seja entre 25% e 35% mais rápido que o *Odd-Even*. No entanto, será visto adiante que isso nem sempre ocorre, pois a verificação do novo critério de parada envolve uma operação relativamente lenta na arquitetura utilizada.

4.2 Comparação do desempenho

Determinado o valor ótimo de M, foi feito então o teste de comparação do desempenho dos três algoritmos estudados, com vetores de diferentes tamanhos (variando de 128 a 32768). Para $N \leq 4096$, foram realizadas $3 \times \frac{4096}{N}$ execuções distintas, das quais foi calculada a duração média. Para os demais valores de N foi medido o tempo gasto por uma única execução. Os resultados obtidos podem ser observados no gráfico da Figura 5. Adicionalmente, na Figura 6 é mostrado o fator pelo qual o tempo de execução é multiplicado quando o número de subvetores dobra.

Observa-se em ambos os gráficos, que os tempos de execução dos algoritmos *Odd-Even* e *Linear Array* crescem de forma bastante acelerada quando N passa, respectivamente, de 2048 a 8192, e de 1024 a 4096. Isso se deve a uma peculiaridade da arquitetura utilizada: a existência de dois tipos de memória. Para

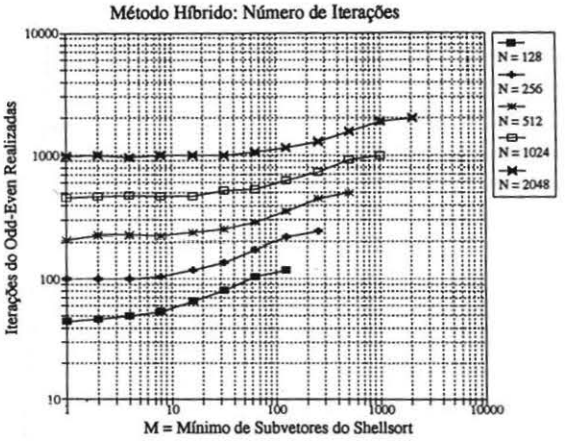

Figura 3: Variação do número de passos do *Odd-Even* no método híbrido

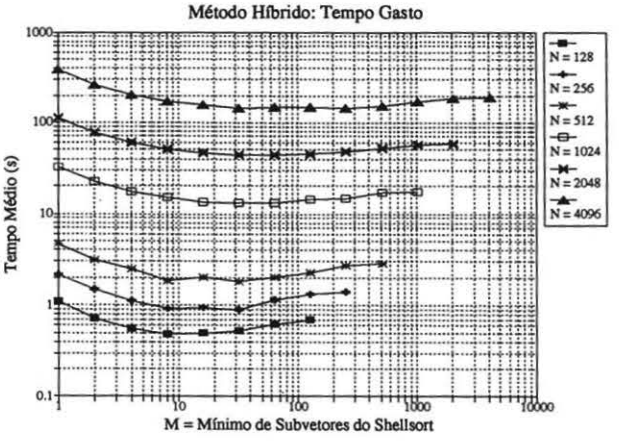

Figura 4: Variação do tempo de execução do método híbrido

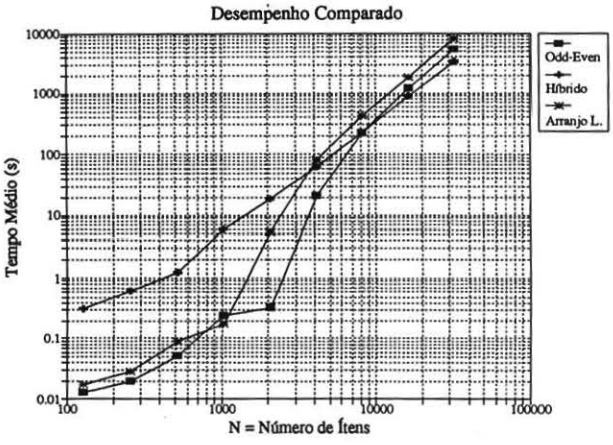

Figura 5: Comparação de desempenho entre métodos estudados

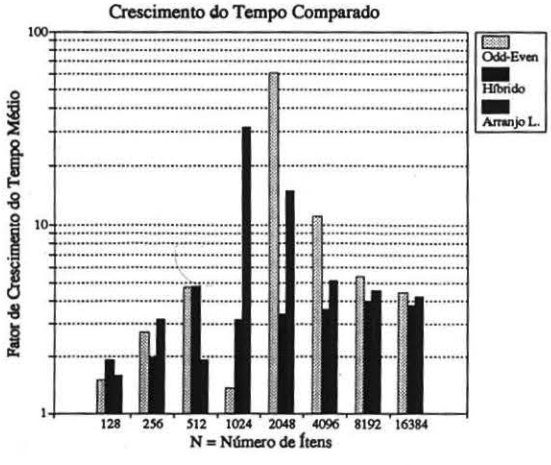

Figura 6: Fator de variação dos tempos com o aumento de N

valores relativamente pequenos de N, é possível manter todos os dados utilizados pelas instruções *multiC* do algoritmo de ordenação na memória rápida (*cache*), interna aos processadores da WT. No entanto, quando N cresce além de um certo limite, torna-se necessário utilizar também a memória externa, bem mais lenta, o que provoca uma queda drástica do desempenho.

Já o tempo de execução do algoritmo híbrido cresce de forma bem mais homogênea, não apresentando a anormalidade descrita acima. Isso se deve ao fato de que esse algoritmo faz uso de uma instrução que é bastante lenta na arquitetura WT, mas cujo tempo parece não aumentar muito quando a memória externa passa a ser utilizada: a redução. Tal operação é necessária para calcular o número de trocas realizadas a cada passo, a fim de que se possa determinar o fim da execução do processo de ordenação. Quando N é suficientemente pequeno para permitir que os dados sejam mantidos na memória interna aos processadores da WT, a velocidade da execução do algoritmo é limitada pela realização de uma redução a cada ciclo, o que faz com que o tempo gasto por esse método de ordenação seja bem maior que o gasto pelos outros dois. Entretanto, quando a memória externa da WT passa a ser utilizada, as outras instruções se tornam relativamente mais lentas, passando a ser o principal fator de limitação da rapidez desse algoritmo.

Pode-se constatar ainda, a partir de uma análise do gráfico da Figura 6, que os tempos de execução de todos os algoritmos apresentam uma tendência de crescimento aproximadamente linear para valores relativamente pequenos de N e aproximadamente quadrática para valores de N relativamente elevados. Essa diferença ocorre porque, a partir de um certo valor de N, a WT passa a fazer uso de todos os seus processadores físicos, tendo que emular novos processadores, o que faz com que o tempo gasto no processo de ordenação passe a aumentar como se este estivesse sendo executado seqüencialmente.

Verifica-se adicionalmente que, para $N \leq 4096$, o algoritmo *Odd-Even* é, em geral, o mais eficiente, sendo superado pelo método híbrido, a partir desse limite. A vantagem do último passa então a crescer continuamente com N, até que o seu tempo de execução se torna cerca de 38% menor que o do primeiro, para $N = 32768$. Já o algoritmo *Linear Array* apresenta um desempenho mais modesto, que se justifica pela sua flexibilidade com relação à geração dos itens a serem ordenados. Para valores de N fora da faixa de transição para a memória externa, seu tempo foi entre 38% e 83% maior que o do *Odd-Even*, o que é bastante razoável já que este realiza apenas N passos, contra $2N$ do *Linear Array*.

4.3 Avaliação do ganho obtido com o paralelismo

Por fim, foram realizados testes com o objetivo de avaliar como a existência de paralelismo contribui para tornar mais eficientes os algoritmos estudados. O primeiro teste consistiu em determinar de que forma o tempo de execução de cada um deles responde a uma variação do número de processadores virtuais utilizados. No caso dos algoritmos *Odd-Even* e híbrido isso foi feito pela segmentação do vetor a ser ordenado (que tinha cada um de seus elementos originalmente associado a um processador virtual diferente) em N_s subvetores de tamanho $\frac{N}{N_s}$. Essa segmentação foi realizada de forma tal que o i-ésimo elemento de cada subvetor ($0 \leq i < \frac{N}{N_s}$) passou a estar associado ao processador virtual i. Com isso, o número de processadores virtuais foi reduzido de N para $\frac{N}{N_s}$. No caso do algoritmo *Linear Array*, a segmenteção foi realizada de forma semelhante sobre o arranjo linear de nodos, de forma que os $\frac{N}{2}$ processadores virtuais existentes originalmente (um para cada nodo), foram reduzidos a $\frac{N}{2N_s}$.

Fixados 6 valores distintos de N (entre 128 e 4096), N_s foi variado exponencialmente entre 1 e 64. Para cada par (N,N_s) resultante, foram realizadas $3 \times \frac{4096}{N}$ execuções distintas de cada um dos três métodos de ordenação em estudo, estando os tempos médios obtidos representados graficamente na Figura 7. Na Figura 8 é mostrado o fator de crescimento do tempo de execução quando N_s é multiplicado por 2.

Nos gráficos correspondentes aos algoritmos *Odd-Even* e *Linear Array*, pode-se observar claramente que o aumento de N_s também provoca variações bruscas no tempo de execução, devido à transição de parte dos dados da memória interna aos processadores da WT para a externa. Isso ocorre pois, à medida em que N_s cresce, um número cada vez maior de itens (ou nodos) passa a estar associado a um mesmo processador virtual, o qual, por sua vez, é mapeado em um dado processador físico com memória interna limitada. Ainda nesses dois gráficos, verifica-se que, fora dessa faixa de transição (ou seja, a partir de $N_s = 16$), o crescimento do tempo de execução é aproximadamente linear em relação a N_s. Isso indica que o paralelismo é explorado de forma bastante eficaz, já que com duas vezes mais processadores, é possível executar a mesma tarefa em aproximadamente metade do tempo. Para valores mais elevados de N, entretanto, esse fator de crescimento

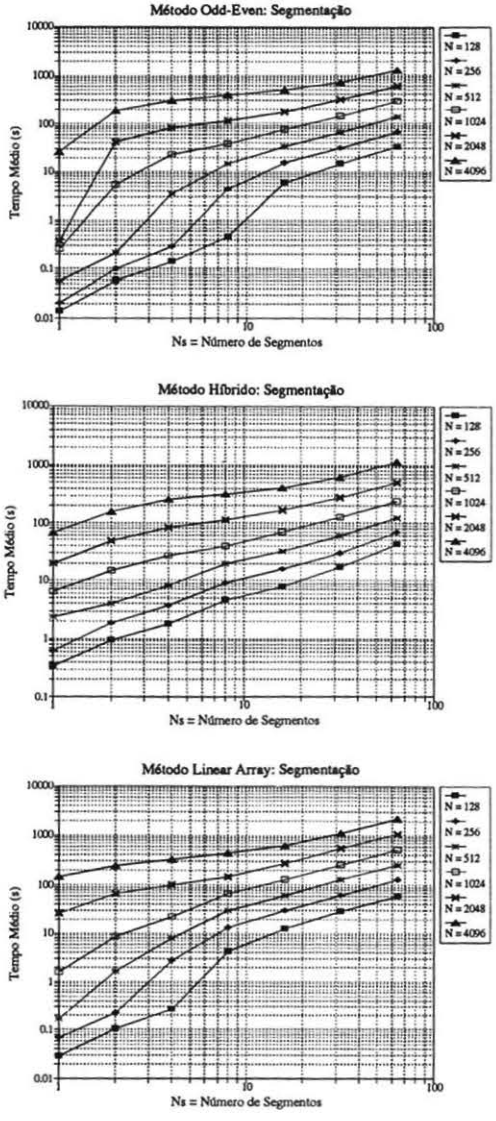

Figura 7: Tempo gasto × Segmentação

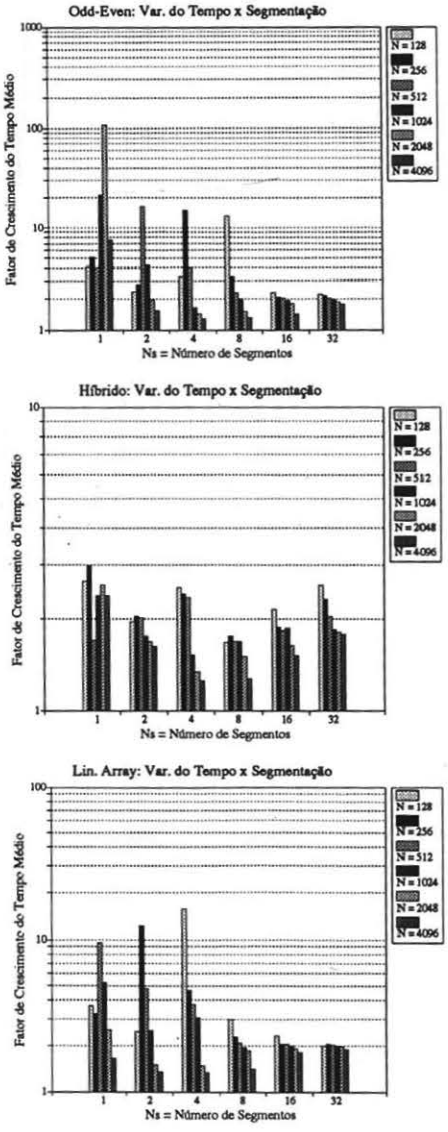

Figura 8: Fator de crescimento do tempo x Segmentação

do tempo parece ser, em geral, um pouco menor, o que se deve ao fato de que, nesses casos, a taxa de utilização dos processadores físicos da WT é maior, sendo em conseqüencia menor a eficiência de novos processadores virtuais criados.

Quanto aos gráficos do algoritmo híbrido, verifica-se em geral fatores de crescimento mais reduzidos, especialmente para valores mais elevados de N. Isso se deve ao fato de que o tempo necessário para realizar uma operação de redução em um dos subvetores resultantes da segmentação parece depender do tamanho desse subvetor (ou seja, do número de processadores virtuais utilizados). Com isso, o crescimento do número de operações individuais a serem realizadas, ocasionado pelo aumento de N_s, é em parte contrabalançado pela diminuição do tempo gasto em cada uma das operações de redução, que são as mais lentas. Além disso, pelos motivos já discutidos na subseção anterior, não se verificam variações muito grandes no tempo de execução quando ocorre a transição de parte dos dados entre as duas memórias da WT.

Finalmente, foi realizado um teste com o objetivo de avaliar qual seria a perda em eficiência decorrente da eliminação total do paralelismo. Foram então implementadas versões sequenciais de cada um dos algoritmos estudados, as quais foram executadas em uma *workstation Sun Sparc2*. O teste foi executado da mesma forma que a avaliação do desempenho dos algoritmos paralelos: N foi variado entre 128 e 32768, sendo realizadas $3 \times \frac{4096}{N}$ execuções independentes de cada um dos métodos de ordenação para $N \leq 4096$, e uma única execução para $N > 4096$. A comparação entre os tempos médios dos algoritmos paralelos e seqüenciais pode ser observada na Figura 9 e 10.

Os algoritmos *Odd-Even* e *Linear Array* paralelos apresentaram um desempenho sempre superior ao de seus respectivos concorrentes seqüenciais. Para os valores de N nos quais apenas a memória interna aos processadores da WT era utilizada, a razão entre o tempo gasto pelo seqüencial e o gasto pelo paralelo variou entre 10 e 99 para o *Odd-Even*, e entre 9 e 61 para o *Linear Array*. Entretanto, quando a memória externa passou a ser utilizada, essa razão caiu drasticamente, chegando a cerca de 2,17 no caso do *Odd-Even* e 1,67 no caso do *Linear Array*.

O algoritmo híbrido paralelo, por outro lado, apresentou um desempenho inferior ao de seu concorrente seqüencial, para $N \leq 1024$, devido à lentidão das operações de redução. Entretanto, nesse caso a razão entre o tempo de execução do algoritmo seqüencial a e do paralelo foi normalmente crescente com em relação a N, chegando a cerca de 2,18 para $N = 16384$.

5 Conclusões

O presente trabalho contribui no sentido de determinar formas eficientes de se realizar ordenação em máquinas SIMD cuja arquitetura apresenta certas restrições de comunicação interprocessador. Dentre os três algoritmos de ordenação estudados, o *Odd-Even* foi o que apresentou o melhor desempenho, para vetores de até 4096 elementos. Acima desse limite, o algoritmo híbrido desenvolvido a partir do *Shellsort* e do próprio *Odd-Even* se mostrou mais eficiente. No entanto, se a geração dos itens a serem ordenados for relativamente lenta e ocorrer seqüencialmente, a utilização do algoritmo *Linear Array* pode ser vantajosa (especialmente se o número de itens for pequeno), já que sua execução poderá ser iniciada assim que o primeiro item for gerado.

Foi mostrado ainda que os algoritmos estudados exploram de forma bastante eficaz a existência de paralelismo, uma vez que (fora da faixa de transição entre as duas memórias da WT) seu tempo de execução pode ser reduzido aproximadamente pela metade, por meio da duplicação do número de processadores virtuais utilizados. Outra prova da eficiência dos algoritmos paralelos foi dada quando o desempenho destes foi comparado ao de seus concorrentes seqüenciais. As versões seqüenciais do *Odd-Even* e do *Linear Array* foram sempre mais lentas que as paralelas, por fatores nunca inferiores a 2,17 e 1,67, respectivamente. Nos casos em que não foi necessário utilizar a memória externa aos processadores da WT, entretanto, tais fatores foram bem mais elevados, chegando a 99 e 61. Além disso, a versão paralela do método híbrido foi mais rápida que sua concorrente seqüencial, para todos os valores de N para os quais sua utilização é mais vantajosa que a do *Odd-Even* (acima de 4096).

Por fim, cabe ressaltar que o presente trabalho também permitiu que fosse estudado o impacto de diversas peculiaridades da arquitetura *Zephyr-Wavetracer* sobre os programas aplicativos, como a redução de desempenho ocasionada pela utilização da memória externa aos processadores e a influência relativamente elevada das operações de redução sobre o tempo de execução.

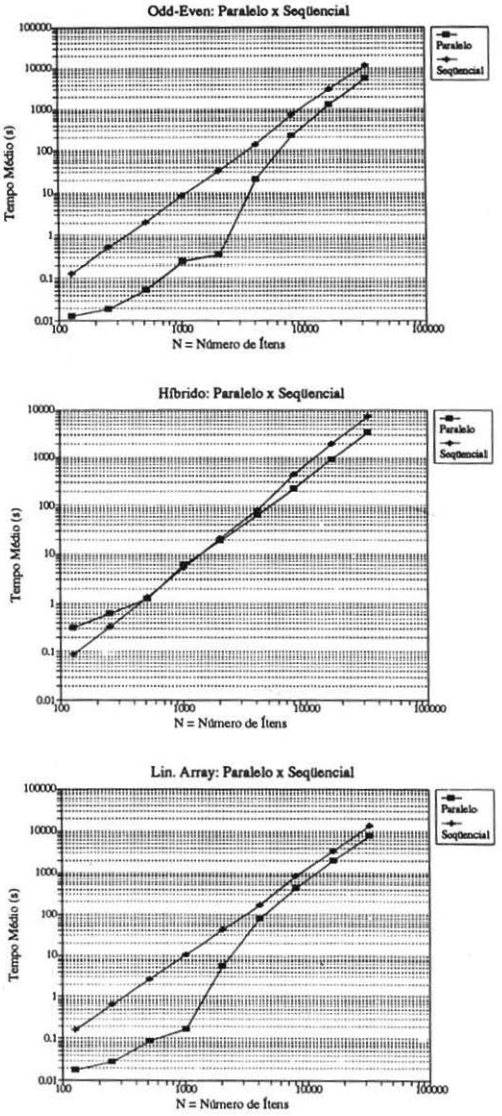

Figura 9: Seqüencial × Paralelo: comparação entre os tempos

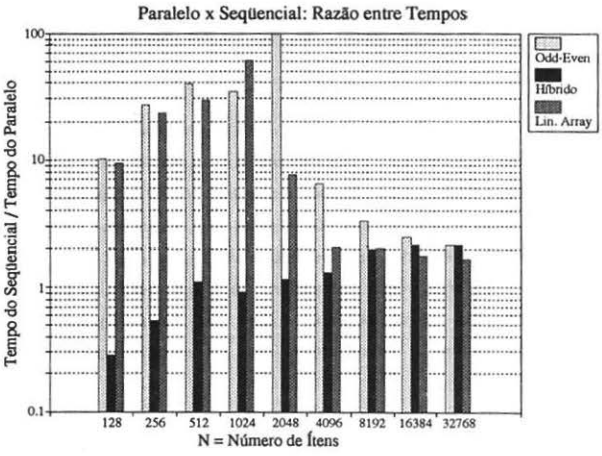

Figura 10: *Speedup*

Referências

[1] WAVETRACER, Inc. *The MultiC programming language - user documentation.* Acton, Massachusetts, 1991.

[2] M. Ajtai, J. Komlós, and E. Szmerédi. *An O(n log n) sorting network.* In *Proceedings of the 15th ACM Annual Symposium on Theory of Computing*, pp 1-9. New York, 1983.

[3] D. E. Knuth. *The art of computer programming*, Vol. 3: *Sorting and searching.* Addison-Wesley Publishing Co., 1973.

[4] F. T. Leighton, B. Maggs, and S. Rao. *Universal packet routing algorithms.* In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pp 256-269, 1988.

[5] F. Albohassan, J. Keller, and D. Scheerer. *Optimal sorting in linear arrays with minimum global control.* Preprint.

[6] F. Abolhassan, J. Keller, and W. J. Paul. *On the cost-effectiveness of PRAMs.* In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pp 2-9, 1991.

[7] R. S. Barr, and B. L. Hickman. *Reporting computational experiments with parallel algorithms: issues, measures and experts' opinions.* In *ORSA Journal on Computing*, Vol. 5, No. 1, pp 2-32, 1993.