# Implementação e Avaliação de Técnicas de Paralelização no Algoritmo de Hirschberg para Sistemas Multicore

**Mario João Jr.**[1], **Alexandre C. Sena**[2] **e Vinod E. F. Rebello**[3]

[1] Laboratório Médico de Pesquisas Avançadas, UERJ – Rio de Janeiro – RJ – Brasil

[2]Instituto de Matemática e Estatística, UERJ – Rio de Janeiro – RJ – Brasil

[3]Instituto de Computação, UFF – Niterói – RJ – Brasil

junior@lampada.uerj.br, asena@ime.uerj.br, vinod@ic.uff.br

***Resumo.*** *Descobrir a maior subsequência comum entre duas sequências em um tempo razoável é fundamental para solucionar diversos problemas. Para garantir que a solução ótima seja encontrada, algoritmos baseados em programação dinâmica são necessários. O algoritmo de Hirschberg possui complexidade linear de espaço, podendo ser usado para comparar sequências longas. Porém, devido à sua complexidade quadrática de tempo, o uso do paralelismo é fundamental. Assim, o objetivo deste trabalho é implementar e avaliar técnicas de paralelismo para o algoritmo de Hirschberg que permitam a comparação de sequências de caracteres longas. Para alcançar este objetivo, três estratégias de paralelismos são implementadas e investigadas em cima de melhorias na versão sequencial do algoritmo. Os resultados mostram que é possível executar mais eficientemente o algoritmo de Hirschberg em máquinas* multicore*, especialmente para grandes cadeias de caracteres, sendo possível alcançar um desempenho até* 33 *vezes melhor do que a versão sequencial original.*

## 1. Introdução

Encontrar a maior subsequência comum (*Longest Common Subsequence - LCS*) entre duas cadeias de caracteres em um tempo aceitável é fundamental para várias aplicações, como por exemplo, comparação de sequências de nucleotídeos ou proteínas para investigar a similaridade entre espécies, comparação de arquivos para identificar as diferenças entre os mesmos, reconhecimento de padrões, entre outras [Gusfield 1997]. Embora existam ferramentas heurísticas para achar a maior subsequência comum entre duas cadeias, as mesmas não garantem o resultado ótimo, podendo conter erros não desprezíveis [Martins et al. 2001]. Assim, nos casos onde seja necessário achar o valor exato, algoritmos que garantem a solução ótima propostos por Needleman e Wunsch [Needleman and Wunsch 1970] ou Hirschberg [Hirschberg 1975] são necessários. Apesar desses dois algoritmos serem baseados em programação dinâmica, o algoritmo de Needleman e Wunsch possui complexidade $O(m \times n)$ em tempo e espaço, assumindo duas cadeias de tamanhos $m$ e $n$ com $n \geq m$, enquanto o algoritmo de Hirschberg possui também complexidade de tempo $O(m \times n)$, mas complexidade de espaço linear $O(n)$, sendo necessário manter apenas uma linha da matriz de similaridade em memória.

Um algoritmo de ordem quadrática no espaço limita bastante o tamanho das cadeias a serem comparadas por sistemas computacionais. Por exemplo, para analisar

duas cadeias de $100.000$ caracteres (onde são necessários $4$ *bytes* para armazenar os dados relativos ao cálculo para cada caracter) são utilizados aproximadamente 40 GB de espaço de armazenamento apenas para guardar a matriz de similaridade. Por ser linear no espaço, o algoritmo de Hirschberg não possui essa limitação, podendo ser utilizado para cadeias com milhões de caracteres. Porém, por possuir complexidade de tempo quadrática, o tempo para comparar sequências grandes é bastante elevado, sendo necessário computação de alto desempenho para se conseguir comparar grandes cadeias em tempos aceitáveis para os usuários.

Com o fim da era dos processadores baseados em uma única unidade de processamento, em função do problema da dissipação de calor e consumo de energia, todo computador passou a ser uma máquina paralela [Diaz et al. 2012]. Assim, para aproveitar todo potencial dessas novas arquiteturas paralelas com memória compartilhada é necessário desenvolver aplicações paralelas eficientes. Desse modo, o objetivo deste trabalho é implementar e avaliar técnicas de paralelismo para o algoritmo de Hirschberg, de maneira a permitir a comparação de grandes cadeias de caracteres em tempos aceitáveis. Mais especificamente, três abordagens para arquiteturas *multicore* com memória compartilhada são investigadas e implementadas, assim como melhorias para aumento de desempenho da versão sequencial. Os resultados mostram que é possível executar eficientemente o algoritmo de Hirschberg nessas arquiteturas e, com isso, diminuir bastante o tempo necessário para se conseguir achar a maior subsequência comum exata entre duas cadeias. Os resultados mostram que para cadeias muito longas foi possível atingir um desempenho $33$ vezes melhor ao se comparar a versão paralela otimizada com a versão sequencial original.

O restante deste artigo está dividido da seguinte maneira: na Seção 2 são apresentados alguns trabalhos relacionados enquanto a Seção 3 apresenta o algoritmo de Hirschberg. As estratégias para aumentar o desempenho da versão sequencial são apresentadas na Seção 4. Por sua vez, a Seção 5 descreve as três abordagens paralelas investigadas e seus desempenhos. Por fim, as conclusões e trabalhos futuros são apresentados na Seção 6.

## 2. Trabalhos Relacionados

Encontrar a maior subsequência comum é uma etapa fundamental para o alinhamento de cadeias de nucleotídeos e proteínas e tem sido objeto de estudo desde a década de 70, quando foram propostos os algoritmos exatos de Needleman e Wunsch [Needleman and Wunsch 1970] e Hirschberg [Hirschberg 1975]. Uma revisão sobre os principais algoritmos exatos para o alinhamento de cadeias par a par pode ser visto em [Sandes et al. 2016]. Além de descrever e classificar os algoritmos, o artigo descreve os principais avanços, as arquiteturas em que os mesmos foram executados e o tamanho das cadeias alinhadas.

Uma implementação paralela em dois níveis do algoritmo de Hirschberg para busca por sequência de proteínas homólogas (sequências que compartilham um ancestral comum) foi proposta em [Rashid et al. 2007]. No primeiro nível de paralelismo (granularidade grossa), composto de um banco de dados de sequências a serem comparadas, foi utilizado MPI (*Message Passing Interface*) para distribuir essas sequências entre os processadores. Por sua vez, no segundo nível de paralelismo (granularidade fina), cada

instância do algoritmo é paralelizada utilizando Pthreads. Em razão das dependências da matriz de similaridades, a paralelização é bastante trivial e limitada. Os autores simplesmente dividiram a matriz em dois blocos que são computados distintamente. O primeiro bloco, composto pela metade superior da matriz, é computado a partir da primeira linha e coluna da matriz. Por sua vez, no segundo bloco, composto pela metade inferior da matriz, é computado a partir da última linha e coluna da matriz. É importante notar que o segundo nível de paralelismo fica limitado a um *speedup* de no máximo 2.

Apesar do trabalho proposto em [Driga et al. 2003] também apresentar um algoritmo exato com complexidade de espaço linear, na prática, ao invés de manter uma linha da matriz por vez em memória, o algoritmo utiliza um parâmetro $k$ que deve ser informado pelo usuário e, de acordo com os experimentos apresentados, varia em função do tamanho da cadeia e da quantidade de processadores. Assim, diferentemente do algoritmo de Hirschberg que mantêm apenas duas linhas da matriz durante toda a execução, este algoritmo necessita de uma quantidade de espaço muito maior, sendo necessário manter $k$ linhas e colunas, assim como, quadrantes de tamanho $n/k$, onde $n$ é o tamanho da cadeia. Para melhorar o desempenho do algoritmo os autores utilizam o tradicional paralelismo *wavefront* [Anvik et al. 2002] [Mohanty and Cole 2014], que permite executar os elementos das antidiagonais de cada bloco da matriz em paralelo.

Diferentemente dos trabalhos apresentados anteriormente nesta seção, o trabalho apresentado neste artigo implementa e avalia técnicas para melhorar o desempenho do algoritmo de Hirschberg que permitam sua execução para cadeias contendo milhões de caracteres em um tempo aceitável. Enquanto que a paralelização dos algoritmos exatos que usam a matriz de similaridades é tradicionalmente feita através da técnica de *wavefront* [Mohanty and Cole 2014], no algoritmo de Hirschberg a mesma não pode ser usada, uma vez que apenas uma linha da matriz é computada por vez para se conseguir uma complexidade de espaço linear. Os resultados obtidos mostram que a execução com as melhorias propostas reduziram o tempo significantemente, principalmente para grandes sequências.

## 3. O Algoritmo de Hirschberg

Em [Hirschberg 1975], o autor desenvolveu um algoritmo, que recebeu seu nome, para a descoberta da maior subsequência comum (*LCS - Longest Common Sequence*). Tal algoritmo, baseado em programação dinâmica, se preocupa em resolver o problema em $O(m \times n)$ em tempo e $O(n)$ em memória, onde $m$ e $n$ são os tamanhos das respectivas sequências com $n \geq m$.

### 3.1. O Algoritmo

Uma descrição do algoritmo de Hirschberg pode ser vista no Algoritmo 1. O *Caso Trivial* (linhas 2 a 14) se dá quando uma das sequências é vazia ou quando a primeira sequência possui apenas um elemento.

Se não for um *Caso Trivial*, são gerados dois vetores $L_1$ e $L_2$ com os pesos das similaridades (linhas 15 a 17), uma vez que a comparação das sequências $A$ e $B$ é realizada em duas etapas. O primeiro vetor é gerado a partir da comparação da sequência $B$ com a metade superior da sequência $A$, enquanto que o segundo vetor a partir da comparação da sequência $B$ com a metade inferior da sequência $A$. O algoritmo utilizado para a

**Algoritmo 1:** Algoritmo de Hirschberg

```
1  Função Hirschberg (m, n, A, B, C)
       /* Caso Trivial */
2      se n = 0 então
3          C ← ∅;
4          retorna;
5      senão
6          se m = 1 então
7              se ∃j ⩽ n tal que A[1] = B[j] então
8                  C ← A[1];
9              senão
10                 C ← ∅;
11             fim
12             retorna;
13         fim
14     fim
       /* Geração dos vetores */
15     i ← ⌊m/2⌋;
16     Geravet (i, n, A[0..i − 1], B, L1);
17     Geravetinv (m − i, n, A[i..m], B, L2);
       /* Ponto de Divisão Vertical */
18     M ← max(L1[j], L2[n − j]), para 0 ⩽ j ⩽ n;
19     k ← min(j), tal que L1[j] + L2[n − j] = M;
       /* Divisão e Conquista */
20     Hirschberg (i, k, A[0..i], B[0..k], C1);
21     Hirschberg (m − i, n − k, A[i + 1..m], B[k + 1..n], C2);
       /* Retorno */
22     C ← concatena(C1, C2);
```

comparação é o mesmo, mas para a metade inferior da sequência $A$, as sequências são percorridas do fim para o início.

Tendo os vetores $L_1$ e $L_2$, é encontrado $k$ (linhas 18 e 19) para que seja feita a divisão e conquista (linhas 20 e 21) e ao final retornada a LCS (linha 22) como a concatenação de $C_1$ e $C_2$.

### 3.2. Geração dos Vetores de Similaridade

O Algoritmo de Hirschberg gera os vetores de similaridade conforme o Algoritmo 2. Dessa forma, não há necessidade de gerar uma matriz para poder armazenar as similaridades como em [Needleman and Wunsch 1970] [Smith and Waterman 1981], o que faz com que sua complexidade de memória seja linear. Cada elemento $j$ de $K_1$ pode ser calculado da seguinte forma:

**Algoritmo 2:** Geração dos Vetores de Similaridade

```
1  Função Geravet (m, n, A, B, LL)
2      K1[j] ← 0, ∀j 0 ⩽ j ⩽ n;
3      para 1 ⩽ i ⩽ m faça
4          K0[j] ← K1[j], ∀j 0 ⩽ j ⩽ n;
5          para 1 ⩽ j ⩽ n faça
6              se A[i] = B[j] então
7                  K1[j] ← K0[j − 1] + 1;
8              senão
9                  K1[j] ← max(K1[j − 1], K0[j]);
10             fim
11         fim
12         LL ← K1;
13     fim
```

$$K_1[j] = \begin{cases} K_0[j-1]+1, & \text{se A[i] = B[j]} \\ & \text{ou} \\ max(K_1[j-1], K_0[j]), & \text{caso contrário} \end{cases}$$

## 4. Aprimoramento do Desempenho da Versão Sequencial

Antes da utilização de técnicas de paralelização para o algoritmo de Hirschberg, ainda cabem melhorias para o aprimoramento do desempenho na sua versão sequencial. Tais melhorias são abordadas nas subseções a seguir. A saber: duas alterações no algoritmo original e a utilização das instruções SIMD (*Single Instruction, Multiple Data*) disponíveis em processadores modernos. A seção termina com a análise experimental das melhorias supramencionadas.

### 4.1. Remoção da Cópia dos Vetores de Similaridade

Na linha 4 do Algoritmo 2 observa-se a cópia de $K_1$ para $K_0$, uma vez que para gerar o novo $K_1$ o algoritmo necessita do vetor anterior. Tal cópia foi removida e o algoritmo foi alterado de forma que as iterações ímpares do loop iniciado na linha 3 utilizam $K_0$ para gerar $K_1$ e as pares utilizam $K_1$ para gerar $K_0$. Devido a remoção da cópia, ainda foi necessária a alteração na inicialização (linha 2), onde $K_0$ passa a ser inicializado e a alteração no retorno da função (linha 12), onde, após as alterações, faz-se necessário verificar qual é o último vetor gerado e atribuí-lo a $LL$.

### 4.2. Remoção da Dependência na Geração do Vetor de Similaridades

Em [Yang et al. 2010], para o algoritmo de Smith-Waterman [Smith and Waterman 1981], os autores mostram que a dependência na mesma linha para gerar $K_1$ pode ser removida se for utilizada uma matriz auxiliar $P$ baseada no dicionário de itens utilizados. Com isso, a geração de $K_1$ depende apenas de $K_0$.

A matriz $P$ para a sequência $B$ é definida como:

$$P[i,j] = \begin{cases} 0, & \text{se i = 0 ou j = 0} \\ j, & \text{se B[j-1] = C[i]} \\ P[i,j-1], & \text{caso contrário} \end{cases}$$

Onde $C$ é o dicionário de itens, ou seja, um vetor com todos os possíveis símbolos. Por exemplo, para análises de sequências homólogas de DNA, o dicionário $C$ seria $\{A, C, G, T\}$.

Assim, $K_1[j]$ pode ser determinado da seguinte forma:

$$K_1[j] = \begin{cases} 0, & \text{se j = 0} \\ K_0[j], & \text{se P[c,j] = 0} \\ max(K_0[j], K_0[P[c,j]-1]+1), & \text{caso contrário} \end{cases}$$

Onde $c$ é o índice de $A[i]$ em $C$. Dessa forma, cada iteração do loop (linhas 5 a 11) do Algoritmo 2 pode ser executada independentemente, propiciando uma grande oportunidade de ganho de desempenho.

### 4.3. Análise Experimental

A Tabela 1 mostra o tempo médio de execução em segundos e o *speedup* obtido, em relação a versão original, ao se executar as versões sequenciais do algoritmo de Hirschberg, compiladas com Intel ICC versão 16.0.1 20151021 e opção O3, em uma máquina multicore com um total de 36 núcleos (dois processadores Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz) com 128 GB de memória compartilhada. Esse ambiente foi utilizado para todos os experimentos deste trabalho. Foram utilizadas sequências de tamanhos que variam entre 50K e 1600K caracteres. Para cada sequência, cada versão foi executada 3 vezes para se minimizar a influência do ambiente computacional de uso não exclusivo. Para se ter a precisão da média, a tabela apresenta também o coeficiente de variação (Co-Var) para cada média calculada.

Os tempos obtidos ao se executar o algoritmo descrito na Seção 3 são exibidos na primeira linha, linha **Original**, da Tabela 1. Com a melhoria descrita na subseção 4.1 (linha **Sem Cópia** da Tabela 1) observa-se um *speedup* de 1,2 em relação a versão original. Ao se agregar a essa versão a melhoria descrita na subseção 4.2 (linha **Sem Dependência** da Tabela 1), já é possível observar um *speedup* de até 1,4. Por fim, o código fonte foi recompilado para utilizar as instruções de extensão vetorial disponíveis para os processadores utilizados (*Advanced Vector Extensions 2* - AVX2) [Lento 2014] e, combinadas com a versão anterior, obteve-se um *speedup* de até 1,9 em comparação com a versão original (linha **Sem Dependência com SIMD** da Tabela 1). Tal ganho só foi possível em virtude da remoção da dependência descrita na subseção 4.2. É importante ressaltar o baixo valor do coeficiente de variação, o que mostra que os tempos de execução foram muito próximos uns dos outros.

## 5. Paralelização do Algoritmo de Hirschberg

Enquanto que os algoritmos baseados em programação dinâmica para o cálculo da maior subsequência comum utilizam, de maneira geral, a técnica de *wavefront* [Mohanty and Cole 2014], por utilizarem uma matriz de similaridade, o algoritmo de Hirschberg utiliza apenas um vetor para diminuir a complexidade de espaço, não sendo possível a aplicação dessa técnica. Assim, analisando as dependências do algoritmo de Hirschberg, três abor-

**Tabela 1. Tempos de execução em segundos para as versões sequenciais**

| | | 50K | 100K | 200K | 400K | 800K | 1600K |
|---|---|---|---|---|---|---|---|
| Original | Tempo | 13,4 | 55,8 | 227,6 | 922,2 | 3680,3 | 14844,5 |
| | CoVar | 0,9% | 0,2% | 0,0% | 0,0% | 1,2% | 0,0% |
| Sem Cópia | Tempo | 11,4 | 46,1 | 185,5 | 742,4 | 2966,1 | 11857,3 |
| | Speedup | 1,2 | 1,2 | 1,2 | 1,2 | 1,2 | 1,3 |
| | CoVar | 1,4% | 0,4% | 0,1% | 0,0% | 0,0% | 0,0% |
| Sem Dependência | Tempo | 10,0 | 39,6 | 158,4 | 631,7 | 2554,4 | 10427,1 |
| | Speedup | 1,3 | 1,4 | 1,4 | 1,4 | 1,4 | 1,4 |
| | CoVar | 1,5% | 0,3% | 0,0% | 0,1% | 0,6% | 0,0% |
| Sem Dependência com SIMD | Tempo | 7,5 | 29,8 | 118,7 | 469,8 | 1877,4 | 7926,1 |
| | Speedup | 1,8 | 1,9 | 1,9 | 1,9 | 1,9 | 1,9 |
| | CoVar | 1,8% | 0,5% | 0,3% | 0,5% | 0,3% | 0,1% |

dagens paralelas podem ser adotadas. As subseções a seguir descrevem, analisam e avaliam a implementação de cada uma dessas abordagens.

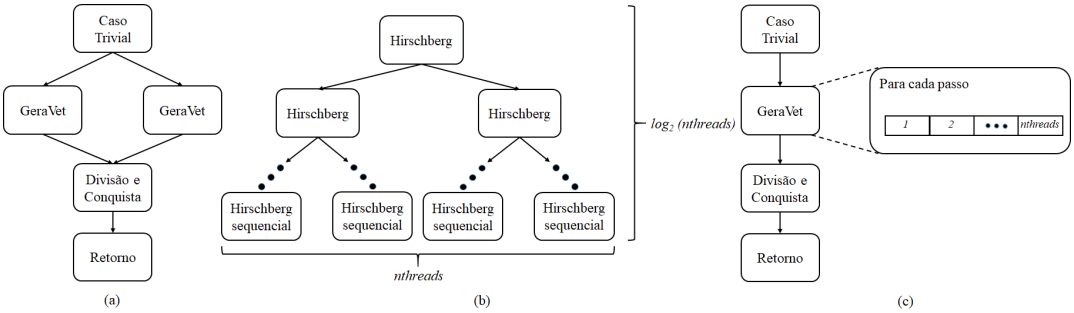


**Figura 1. Abordagens para a Paralelização**

### 5.1. Paralelização da Geração dos Vetores de Similaridade

Entre as linhas 15 e 17 do Algoritmo 1, os vetores de similaridade são gerados de forma independente, suscitando a primeira abordagem de paralelização. Como pode ser visto na Figura 1(a), a cada iteração do algoritmo a função *Geravet*, descrita no Algoritmo 2, é chamada duas vezes, podendo portanto ser realizada, em paralelo, a geração dos dois vetores. Para tal, são criadas duas tarefas, uma para cada uma das chamadas. Somente após término dessas duas tarefas a execução da divisão e conquista pode ocorrer, sendo o único ponto de sincronismo desta abordagem, ficando assim restrita a dois processadores. Esta técnica já foi implementada e avaliada em [Rashid et al. 2007].

### 5.2. Paralelização da Divisão e Conquista

Entre as linhas 20 e 21 do Algoritmo 1 é utilizada divisão e conquista. Cada ramo da recursão é independente, o que sugere a segunda abordagem de paralelização. Conforme pode ser visto na Figura 1(b), a cada divisão o número de tarefas dobra e podem ser processadas em paralelo. Assim, a paralelização adotada foi criar uma *thread* para cada uma dessas tarefas, que são sincronizadas apenas no final da fase de conquista. Neste

caso, seria necessário uma quantidade exponencial de processadores, uma vez que cada ramo se dividirá em dois até que seja atingido um dos casos triviais.

Para limitar a quantidade de *threads* e, com isso, não diminuir significativamente a granularidade das tarefas, a abordagem adotada foi a de utilizar um parâmetro adicional para indicar o nível da chamada recursiva, sendo este iniciado com 0 e incrementado a cada nível da recursividade. Caso $2^{nível}$ seja maior que o número de processadores disponíveis, a recursão prosegue de forma sequencial. Por exemplo, para 16 processadores, a partir do $5^o$ nível, a computação passa a ser sequencial.

### 5.3. Paralelização Fina da Geração dos Vetores de Similaridade

Com a independência obtida pela melhoria descrita na subseção 4.2, é possível executar cada passo da geração dos vetores de similaridade de forma paralela, ou seja, cada elemento do vetor pode ser calculado de maneira independente. Nesta abordagem, a cada passo, os elementos do vetor são calculados em paralelo pelos processadores disponíveis, como mostra a Figura 1(c), havendo um sincronismo a cada passo da geração do vetor. Em cada um desses passos é gerado um novo vetor até que todas as linhas, do que seria a matriz de similaridades, sejam percorridas. Por exemplo, para comparar duas sequências com $10$K caracteres em $16$ processadores, são necessários $10$K passos, onde, a cada passo, cada processador calcula em paralelo $625$ $(10.000/16)$ caracteres.

A medida que o algoritmo de Hirschberg promove a divisão das sequências a serem comparadas, o tamanho dos vetores de similaridade diminui. Por exemplo, para uma sequência *B* inicial de $50$K caracteres processada por $4$ *threads*, no primeiro nível da recursão, cada *thread* seria responsável por calcular $12.500$ elementos. Para o segundo nível, assumindo uma divisão equânime, cada *thread* seria responsável por $6.250$ elementos. O algoritmo segue assim até atingir o caso trivial. Dessa forma, a partir de determinado tamanho da sequência *B*, pode não haver mais ganho em se executar de forma paralela cada elemento do vetor, uma vez que há custos envolvidos na criação e sincronização das *threads* necessárias para tal. Assim, nessa abordagem, foi criado um limite mínimo de corte para o tamanho da sequência *B* a partir do qual a execução é realizada sequencialmente. O valor desse limite é avaliado na subseção 5.4.

### 5.4. Análise Experimental

O presente trabalho se propõe a utilizar técnicas de paralelização baseadas em memória compartilhada, com o intuito de aproveitar da melhor maneira a existência de diversos núcleos (CPUs) nos processadores. Assim, foi utilizado o modelo de programação paralela com memória compartilhada *OpenMP* [Chandra et al. 2000]. Para a implementação das técnicas descritas nas subseções 5.1 e 5.2 foram utilizadas as primitivas *OpenMP* para manipulação de *Tasks*. Já a abordagem da subseção 5.3 foi implementada utilizando a primitiva *Parallel for*.

Na subseção 5.3 foi descrita a utilização de um limite mínimo a partir do qual a geração dos elementos do vetor de similaridade passa a ser sequencial. A Tabela 2 mostra os tempos de execução dessa abordagem para sequências de tamanhos que variam entre $50$K e $1600$K caracteres, executando com $2$, $4$, $8$, $16$ e $32$ *threads* e limites mínimos $1024$, $2048$ e $4096$. Os melhores resultados foram obtidos, em $75\%$ das vezes, para o limite mínimo de $2048$ caracteres. Dessa forma, estes tempos serão utilizados para a comparação entre as melhorias.

**Tabela 2. Tempos de execução em segundos para a abordagem da subseção 5.3**

| | | 50K | | | 100K | | | 200K | | | 400K | | | 800K | | | 1600K | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Thr | Limite | 1K | 2K | 4K | 1K | 2K | 4K | 1K | 2K | 4K | 1K | 2K | 4K | 1K | 2K | 4K | 1K | 2K | 4K |
| 2 | Tempo | 4,5 | 4,5 | 4,9 | 15,6 | 15,4 | 15,7 | 60,2 | 60,2 | 61,1 | 242,5 | 242,8 | 243,8 | 921,2 | 915,4 | 914,7 | 3863,2 | 3917,3 | 3944,5 |
| | CoVar | 3,4% | 4,0% | 6,7% | 1,4% | 1,2% | 1,6% | 0,3% | 0,3% | 1,4% | 0,5% | 0,8% | 1,3% | 0,3% | 0,3% | 0,1% | 1,4% | 0,4% | 1,0% |
| 4 | Tempo | 3.3 | 3,3 | 3,1 | 10,1 | 9,8 | 10,1 | 35,9 | 35,0 | 35,9 | 139,2 | 130,3 | 133,2 | 501,1 | 502,5 | 501,6 | 2078,7 | 2081,2 | 2083,3 |
| | CoVar | 4,5% | 8,2% | 3,2% | 2,8% | 0,5% | 1,9% | 0,8% | 1,9% | 4,3% | 4,5% | 2,2% | 2,8% | 1,1% | 0,7% | 1,5% | 2,0% | 0,7% | 0,4% |
| 8 | Tempo | 2,7 | 2,8 | 2,9 | 7,4 | 7,4 | 7,5 | 23,3 | 23,0 | 23,9 | 78 | 77,4 | 78,5 | 288,7 | 293 | 314,2 | 1123,9 | 1141,2 | 1138,5 |
| | CoVar | 2,7% | 2,9% | 4,5% | 1,3% | 4,7% | 3,8% | 2,3% | 1,7% | 1,6% | 1,3% | 0,3% | 0,9% | 0,2% | 1,4% | 0,6% | 1,3% | 1,0% | 0,3% |
| 16 | Tempo | 2,7 | 2,8 | 2,7 | 6,8 | 6,6 | 6,7 | 18,7 | 18,4 | 18,5 | 56,1 | 55,3 | 55,5 | 180,6 | 179,4 | 186,5 | 665,6 | 660,3 | 668,4 |
| | CoVar | 1,0% | 2,4% | 1,3% | 3,5% | 0,2% | 3,0% | 3,0% | 0,4% | 1,3% | 1,1% | 1,4% | 1,2% | 1,2% | 1,5% | 2,4% | 0,7% | 0,8% | 0,4% |
| 32 | Tempo | 3,3 | 3,2 | 3,3 | 7,4 | 7,2 | 7,5 | 17,9 | 17,6 | 17,6 | 43,6 | 42,6 | 43 | 129,1 | 129,9 | 128,8 | 444,2 | 440,3 | 440,9 |
| | CoVar | 2,3% | 0,7% | 1,4% | 5,2% | 3,2% | 1,3% | 2,5% | 0,6% | 0,4% | 2,8% | 0,9% | 3,9% | 1,4% | 3,4% | 0,9% | 1,3% | 1,4% | 0,7% |

A Tabela 3 mostra os tempos de execução das três abordagens: paralelização da geração dos vetores de similaridade (subseção 5.1 - *VetSim*), paralelização da divisão e conquista (subseção 5.2 - *DivCon*) e paralelização fina da geração dos vetores de similaridade (subseção 5.3 - *ParFin*). São utilizadas sequências de tamanhos que variam entre $50$K e $1600$K caracteres. As abordagens foram executadas com $2$, $4$, $8$, $16$ e $32$ *threads*, com exceção da abordagem *VetSim* que está naturalmente limitada a $2$ *threads*. Nesta abordagem, o *speedup* (em relação à melhor versão sequencial Seção 4.3) ficou entre 1,7 e 1,9, mostrando-se quase linear.

A abordagem *DivCon* apresenta um *speedup* não maior que 1,3, independente do número de *threads* utilizadas. Para se entender tal desempenho, faz-se necessário analisar as etapas responsáveis pelo tempo de execução do Algoritmo 1. Cerca de $70\%$ do tempo total de execução está na geração dos vetores de similaridade (linhas 15 a 17) da primeira chamada recursiva da função. Com isso, só é possível melhorar o desempenho paralelizando os $30\%$ restantes. Assumindo o maior número de *threads* utilizadas neste trabalho ($32$), o tempo de execução da parte restante seria, no melhor caso, dividido por $32$. Utilizando a Lei de Amdahl [Amdahl 1967], o *speedup* máximo teórico que se poderia obter com a paralelização é dado por:

$$Speedup_{Max} = \frac{1}{(1-p) + \frac{p}{t}} \approx 1,4, \text{ para } p = 0,3 \text{ e } t = 32$$

Assim, fica claro que o desempenho obtido está muito próximo do limite teórico.

**Tabela 3. Tempos em segundos e Speedups para as três abordagens propostas**

| | | 50K | | | 100K | | | 200K | | | 400K | | | 800K | | | 1600K | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Thr | | VetSim | DivCon | ParFin | VetSim | DivCon | ParFin | VetSim | DivCon | ParFin | VetSim | DivCon | ParFin | VetSim | DivCon | ParFin | VetSim | DivCon | ParFin |
| 2 | Tempo | 4,4 | 7,0 | 4,5 | 15,3 | 26,3 | 15,4 | 61,4 | 102,3 | 60,2 | 247,0 | 430,2 | 242,8 | 1013,3 | 1631,5 | 915,4 | 4522,2 | 7103,5 | 3917,3 |
| | CoVar | 3,5% | 0,8% | 4,0% | 1,2% | 1,8% | 1,2% | 1,6% | 0,2% | 0,3% | 0,0% | 0,3% | 0,8% | 0,2% | 0,1% | 0,3% | 0,4% | 0,2% | 0,4% |
| | SpeedUp | 1,7 | 1,1 | 1,7 | 1,9 | 1,1 | 1,9 | 1,9 | 1,2 | 2,0 | 1,9 | 1,1 | 1,9 | 1,9 | 1,2 | 2,1 | 1,8 | 1,1 | 2,0 |
| 4 | Tempo | | 6,6 | 3,3 | | 24,1 | 9,8 | | 93,9 | 35,0 | | 396,7 | 130,3 | | 1497,6 | 502,5 | | 6735,4 | 2081,2 |
| | CoVar | | 2,7% | 8,2% | | 0,4% | 0,5% | | 0,2% | 1,9% | | 0,2% | 2,2% | | 0,1% | 0,7% | | 2,5% | 0,7% |
| | SpeedUp | | 1,1 | 2,3 | | 1,2 | 3,0 | | 1,2 | 3,4 | | 1,2 | 3,6 | | 1,3 | 3,7 | | 1,2 | 3,8 |
| 8 | Tempo | | 6,3 | 2,8 | | 23,7 | 7,4 | | 93,7 | 23 | | 390,5 | 77,4 | | 1469,6 | 293 | | 6425,3 | 1141,2 |
| | CoVar | | 1,0% | 2,9% | | 0,2% | 4,7% | | 0,9% | 1,7% | | 0,3% | 0,3% | | 0,0% | 1,4% | | 0,6% | 1,0% |
| | SpeedUp | | 1,2 | 2,7 | | 1,3 | 4 | | 1,3 | 5,2 | | 1,2 | 6,1 | | 1,3 | 6,4 | | 1,2 | 6,9 |
| 16 | Tempo | | 6,5 | 2,7 | | 24,0 | 6,6 | | 92,0 | 18,4 | | 388,0 | 55,3 | | 1460,5 | 179,4 | | 6405,5 | 660,3 |
| | CoVar | | 0,9% | 2,4% | | 1,6% | 0,2% | | 0,1% | 0,4% | | 0,1% | 1,4% | | 0,2% | 1,5% | | 0,8% | 0,8% |
| | SpeedUp | | 1,1 | 2,8 | | 1,2 | 4,5 | | 1,3 | 6,5 | | 1,2 | 8,5 | | 1,3 | 10,5 | | 1,2 | 12 |
| 32 | Tempo | | 6,5 | 3,2 | | 23,8 | 7,2 | | 92,1 | 17,6 | | 386,5 | 42,6 | | 1459,5 | 129,9 | | 6384,9 | 440,3 |
| | CoVar | | 0,6% | 0,7% | | 0,2% | 3,2% | | 0,3% | 0,6% | | 0,2% | 0,9% | | 0,2% | 3,4% | | 0,5% | 1,4% |
| | SpeedUp | | 1,2 | 2,3 | | 1,2 | 4,2 | | 1,3 | 6,8 | | 1,2 | 11,0 | | 1,3 | 14,5 | | 1,2 | 18,0 |

Diferente das duas anteriores, a abordagem *ParFin* obteve *speedups* expressivos, à medida que o tamanho das sequências aumenta, apesar de sua granularidade fina e que varia consideravelmente. Para os experimentos realizados, a granularidade varia entre o mínimo de $64$ caracteres, para $32$ *threads* e o limite mínimo de $2048$, e $800K$, para $2$ *threads* e o tamanho de sequência $1600$K. O desempenho da abordagem *ParFin* pode

ser observado no gráfico da Figura 2, onde fica clara a influência da granularidade fina principalmente no desempenho para 32 *threads*, onde o *speedup* tem um crescimento mais acelerado a partir de sequências com tamanho 200K.

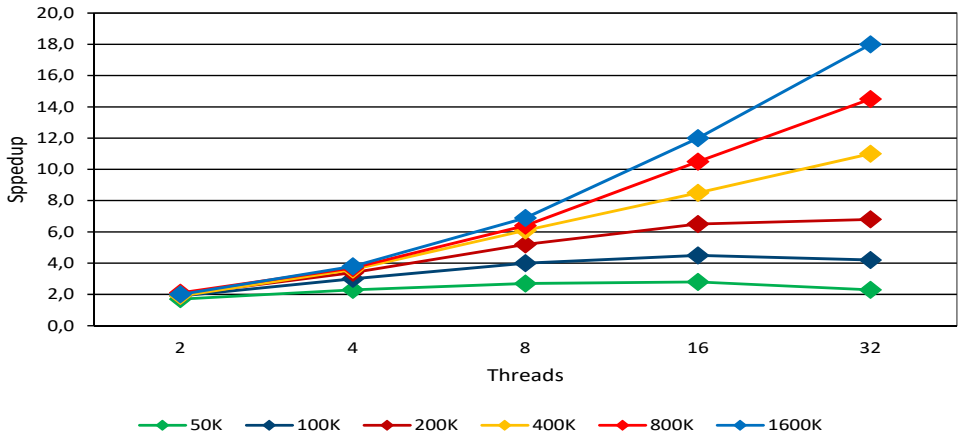


**Figura 2. Speedup da Abordagem ParFin em relação à quantidade de threads**

Para ilustrar o ganho no desempenho total obtido por essa abordagem, acrescido da melhoria na versão sequencial, o gráfico da Figura 3 mostra o *speedup* quando calculado em relação à versão sequencial original do algoritmo. Pode-se observar que este *speedup* chega a 33 para uma sequência de tamanho 1600K executando com 32 *threads*.

É importante destacar que a perda de desempenho, especialmente para as menores sequências, é fruto da sobrecarga da execução paralela, em função da granularidade fina das tarefas, como já explicado anteriormente. Além disso, como a máquina utilizada é do tipo NUMA (*Non-Uniform Memory Access*), ou seja, o acesso a memória não é uniforme, foi observado que parte da perda de desempenho surge em função desse acesso não uniforme. Experimentos preliminares restringindo o acesso a uma parte da memória NUMA com acesso uniforme, apresentou ganhos significativos. A melhora do desempenho nessas arquiteturas será estudada em trabalhos futuros.

Cabe ressaltar também que foram realizadas avaliações iniciais juntando as técnicas de paralelismo avaliadas neste trabalho. O desempenho da versão paralela juntando a técnica *DivCon* com a *ParFin* foi muito ruim, principalmente, em função do desbalanceamento dos ramos de cada divisão da técnica de divisão e conquista. Como cada ramo produz tarefas com tamanhos distintos, a execução de cada ramo é realizada em um tempo distinto, ao invés de forma balanceada, o que prejudica consideravelmente o desempenho.

Com maior potencial para aumentar o desempenho, a versão paralela juntando a técnica *VetSim* com a *ParFin*, apesar de um desempenho bem melhor que a anterior, também não foi melhor que somente a versão *ParFin*. Neste caso, apesar dos dois vetores de similaridades serem balanceados, a perda de desempenho em função do acesso à memória NUMA prejudicou o desempenho. Em trabalhos futuros serão feitas

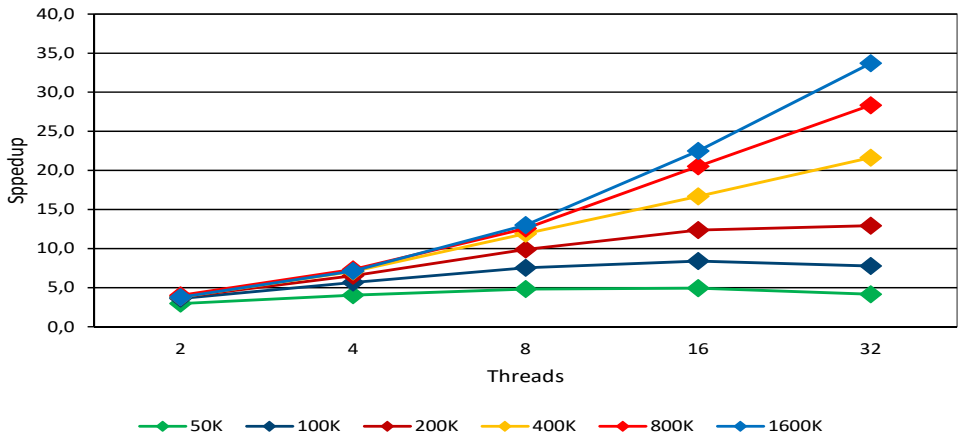


**Figura 3. Speedup da Abordagem ParFin em relação à versão sequencial original**

investigações para avaliar se o melhor uso da memória NUMA pode fazer com que essa combinação de abordagens execute eficientemente.

## 6. Conclusão

Achar a maior subsequência comum (*Longest Common Subsequence - LCS*) entre duas sequências em um tempo aceitável é fundamental para várias aplicações. O algoritmo de Hirschberg permite encontrar a solução ótima do problema, com um gasto linear de memória, o que permite a comparação de sequências longas. Este trabalho implementou e avaliou melhorias de desempenho para versão sequencial do algoritmo de Hirschberg, assim como, técnicas de paralelismo para uso eficiente de máquinas *multicore*. Os resultados mostraram que a versão sequencial ficou aproximadamente duas vezes mais rápida do que a versão original, o que proporcionou um *speedup* de 18 quando executando a versão paralela para sequências longas em 32 *cores*. Se for considerado o *speedup* em relação à versão sequencial original, o *speedup* da nova proposta atinge 33.

Como trabalhos futuros, pretende-se investigar a sobrecarga de acesso à memória em arquiteturas NUMA, assim como, a vetorização manual da abordagem *ParFin* e, com isso, avaliar e implementar técnicas para melhorar ainda mais o desempenho do algoritmo de Hirschberg nesses ambientes.

## Agradecimentos

Os autores agradecem o uso dos recursos computacionais *manycore* mantidos e operados pelo Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP), financiado parcialmente pela Intel, no contexto do projeto Intel/UNESP Modern Code.

## Referências

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), páginas 483–485, New York, NY, USA. ACM.

Anvik, J., MacDonald, S., Szafron, D., Schaeffer, J., Bromling, S., and Tan, K. (2002). Generating parallel programs from the wavefront design pattern. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, páginas 165–172.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann, 1st edition.

Diaz, J., Muñoz-Caro, C., and Niño, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386.

Driga, A., Lu, P., Schaeffer, J., Szafron, D., Charter, K., and Parsons, I. (2003). FastLSA: a fast, linear-space, parallel and sequential algorithm for sequence alignment. In *2003 International Conference on Parallel Processing, 2003. Proceedings.*, páginas 48–57.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343.

Lento, G. (2014). Optimizing performance with intel® advanced vector extensions. Technical report, Intel.

Martins, W. S., del Cuvillo, J., Useche, F. J., Theobald, K. B., and Gao, G. R. (2001). A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Pacific Symposium on Biocomputing*.

Mohanty, S. and Cole, M. (2014). Autotuning wavefront applications for multicore multi-gpu hybrid architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, páginas 1:1–1:9, New York, NY, USA. ACM.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453.

Rashid, N. A., Abdullah, R., and Talib, A. Z. H. (2007). Parallel homologous search with hirschberg algorithm: A hybrid MPI-pthreads solution. In *Proceedings of the 11th WSEAS International Conference on Computers*, páginas 228–233.

Sandes, E. F. D. O., Boukerche, A., and Melo, A. C. M. A. D. (2016). Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification. *ACM Comput. Surv.*, 48(4):63:1–63:36.

Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197.

Yang, J., Xu, Y., and Shang, Y. (2010). An efficient parallel algorithm for longest common subsequence problem on GPUs. In *Proceedings of the World Congress on Engineering*, páginas 499–504.