Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Masa-StarPU: Estratégia com Múltiplas Políticas de Escalonamento de Tarefas para Alinhamento de Sequências com Pruning

Rafael Alvares da Silva Lopes

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Alba Cristina Magalhaes Alves de Melo

Brasília
2020

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Masa-StarPU: Estratégia com Múltiplas Políticas de Escalonamento de Tarefas para Alinhamento de Sequências com Pruning

Rafael Alvares da Silva Lopes

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Alba Cristina Magalhaes Alves de Melo (Orientador)
CIC/UnB

Prof. Dr. Eduardo Adilo Pelinson Alchieri
CIC/UnB

Prof. Dr. Cristiana Barbosa Bentes
FEN/UERJ

Prof. Dr. Genaina Nunes Rodrigues
Coordenador do Programa de Pós-graduação em Informática

Brasília, 29 de Setembro de 2020

Agradecimentos

Agradeço aos meus pais, avós e irmãos, pelo apoio constante e contribuição na minha formação, principalmente a minha mãe que me deu forças em momentos em que eu quiz desistir e ao meu avô Atila que durante esta trajetória infelizmente faleceu, porém, sempre foi uma pessoa de extrema importância para mim e teve grande participação na minha formação. A família é a base que permite o alcance de sonhos cada vez maiores.

À minha orientadora, Alba Melo, pelo acompanhamento e suporte sempre precisos, incentivando e indicando a melhor forma de conduzir este trabalho. Aos professores deste Departamento, pela dedicação empregada na árdua tarefa da formação.

Aos amigos queridos, que me incentivaram e apoiaram neste caminho desde o começo. E finalmente, mas em especial, agradeço à minha namorada, Kelly, pela ajuda na criação das figuras presentes neste documento, na edição de slides de apresentação e video gravado sobre o presente trabalho, além de estar sempre presente, alguém que eu posso sempre contar em momentos difíceis e tenho imenso carinho.

Resumo

A comparação de sequências biológicas é uma tarefa importante executada com frequência na análise genética de organismos. Algoritmos que realizam este procedimento utilizando um método exato possuem complexidade quadrática de tempo, demandando alto poder computacional e uso de técnicas de paralelização. Muitas soluções têm sido propostas para tratar este problema utilizam aceleradores como GPUs e FPGAs, porém poucas soluções utilizam apenas CPUs. O MASA é uma ferramenta multiplataforma específica para realizar a comparação de sequências biológicas. Uma de suas maiores virtudes é a otimização *block pruning* que realiza a poda da matriz de programação dinâmica em tempo de execução acelerando o processamento, porém introduzindo um problema de desbalanceamento de carga. O StarPU é uma ferramenta de programação paralela que possui implementações de diversas políticas de escalonamento dinâmico de tarefas. Neste trabalho, propomos e avaliamos o MASA-StarPU, uma ferramenta que utiliza a estrutura do MASA para realizar a comparação de sequências biológicas e as políticas do StarPU adequadas ao *block pruning* com o objetivo de eliminar o problema de desbalanceamento de carga. O MASA-StarPU foi testado em dois ambientes, avaliando pares de sequências de DNA cujos tamanhos variam entre 10 KBP (milhares de pares de bases) e 47 MBP (milhões de pares de bases), e as políticas de escalonamento de tarefas foram avaliadas em diferentes casos. Quando comparado com outras soluções da literatura que utilizam apenas CPU, o MASA-StarPU obteve o melhor resultado para todas as comparações. O MASA-StarPU atingiu o máximo de 18,4 GCUPS (bilhões de células atualizadas por segundo).

Palavras-chave: Comparação de sequências biológicas, Ambientes de programação paralela

Abstract

The comparison of biological sequences is an important task performed frequently in the genetic analysis of organisms. Algorithms that perform this procedure using an exact method have quadratic time complexity, demanding high computational power and, consequently parallelization techniques. Many solutions have been proposed to address this problem using accelerators such as GPUs and FPGAs, but few solutions use only CPUs. MASA is a domain-specific platform for performing biological sequence comparison. One of its greatest virtues is the optimization block pruning. Which prunes the dynamic programming matrix at run time introducing load imbalance. StarPU is a generic parallel programming tool that provides several dynamic task scheduling policies. In this work, we propose and evaluate MASA-StarPU, a tool that uses the MASA structure to carry out the comparison of biological sequences and uses the StarPU policies to accelerate the computation. MASA-StarPU was tested in two environments, evaluating pairs of DNA sequences whose sizes vary between 10 KBP (thousands of base pairs) and 47 MBP (millions of pairs of bases), and multiple task scheduling policies were evaluated in different cases. When compared to other solutions in the literature that use only CPU, MASA-StarPU obtained the best result for all comparisons and reached a maximum of 18.4 GCUPS (billions of cells updated by second).

Keywords: Biological Sequence Comparison, Parallel Programming Environments

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

1.1 Motivação

A Bioinformática é um campo interdisciplinar que envolve Ciência da Computação, Biologia, Matemática e Estatística visando analisar e interpretar dados biológicos [8]. Uma das operações mais básicas e importantes da Bioinformática é a comparação de sequências biológicas [8] que permite organizar as sequências biológicas para que os biólogos identifiquem claramente regiões de similaridades e diferenças.

As sequências biológicas podem ser de DNA, RNA ou proteínas. Sequências de proteína e RNA são bastante pequenas, tendo tamanhos que chegam até dezenas de milhares de caracteres. Por outro lado, as sequências de DNA podem ser muito longas, muitas vezes compostas por dezenas ou centenas de milhões de caracteres, ou Pares de Bases (MBP - milhões de pares de base). Além disso, os repositórios de sequências de proteínas, DNA e RNA têm crescido exponencialmente nos últimos anos, chegando a milhões de sequências.

Um dos algoritmos mais usados para calcular o alinhamento ótimo local entre duas sequências é o Smith-Waterman (SW) [9]. O algoritmo SW utiliza programação dinâmica para alinhar duas sequências S_0 e S_1 de tamanhos m e n. A complexidade do algoritmo é quadrática ($O(mn)$) tanto para tempo quanto para espaço e, por isso, a comparação de duas sequências biológicas longas pode demorar muito tempo para ser concluída.

O processamento paralelo tem sido usado para acelerar a obtenção de alinhamentos ótimos. Existem propostas na literatura implementadas em CPU [10] [11], GPU [12] [4] e FPGA [13] [14]. Dentre essas, o MASA(*Multi-Platform Arquitecture for Sequence Aligners*) [4] é uma solução que permite a integração de diferentes ambientes de programação paralela e diversas plataformas de hardware para a obtenção de alinhamentos ótimos locais, globais e semi-globais. O MASA aplica uma otimização chamada *Block Pruning* para reduzir a área calculada da matriz de programação dinâmica. Em sua versão mais recente,

o MASA não utiliza o *Block Pruning* em execuções com mais de um dispositivo, como por exemplo duas máquinas, cada qual com uma GPU, pois essa otimização faz a poda da matriz de programação dinâmica em tempo de execução, introduzindo desbalanceamento de carga.

O escalonamento dinâmico de tarefas pode reduzir significativamente o desbalanceamento de carga, atribuindo tarefas aos nodos menos carregados em um determinado instante da execução. O problema do escalonamento de tarefas foi provado NP-difícil [15]. Com isso, diversas políticas de escalonamento dinâmico de tarefas que usam métodos heurísticos existem na literatura e cada política é adequada a casos específicos, não havendo política que seja a mais adequada a todos os casos [16].

Na literatura, existem várias ferramentas de programação paralela que implementam políticas de escalonamento de tarefas adequadas ao balanceamento de carga. Dentre elas, o StarPU [17] é um ambiente de programação paralela que possui múltiplas políticas de escalonamento de tarefas. Para tanto, disponibiliza filas globais e locais de tarefas, bem como métodos para inserir tarefas na fila (*push*) e retirá-las (*pop*). Usando essas filas e métodos, o StarPU implementa diversas estratégias de escalonamento.

Apesar de existirem várias ferramentas de comparação de sequências otimizadas para aceleradores tais como GPU, FPGA, menor esforço tem sido feito para ambientes *multicore*. No entanto, diversos laboratórios de Bioinformática no Brasil e em outros países em desenvolvimento somente dispõem de plataformas *multicore* para suas execuções. Por essa razão, o escopo foi limitado a *multicore*.

1.2 Objetivo

O objetivo principal da presente Dissertação de Mestrado é investigar o uso das estratégias de escalonamento dinâmico do StarPU no MASA de forma a acelerar as execuções com a otimização *block pruning* em diversos cenários para ambientes *multicore*.

Para atingir esse objetivo, integramos o ambiente *domain-specific* MASA com o ambiente genérico de programação StarPU, criando o MASA-StarPU. O MASA-StarPU possui os benefícios do MASA, ou seja, comparação rápida de sequências biológicas e do StarPU, que escalona tarefas com múltiplas políticas de escalonamento. A principal decisão de projeto do MASA-StarPU foi manter a interface dos dois ambientes de programação, sem exigir modificações em quaisquer dos ambientes. Essa decisão foi integralmente cumprida e o MASA-StarPU oferece tanto desempenho como flexibilidade, agregando os benefícios dos dois ambientes. Adicionalmente, fizemos uma avaliação das políticas de escalonamento integradas ao MASA-StarPU para diversas comparações de sequências, com tamanhos e

taxas de poda distintas. O projeto do MASA-StarPU e os resultados obtidos foram publicados no artigo [18] e o anexo I apresenta a sua primeira página.

1.3 Organização do texto

O restante deste documento está organizado como se segue. O Capítulo 2 apresenta uma revisão bibliográfica sobre os algoritmos mais usados na literatura para realizar a comparação de sequências biológicas e as estratégias de poda que podem ser utilizadas em tais comparações. O Capítulo 3 apresenta uma revisão bibliográfica sobre ambientes de programação paralela. O Capítulo 4 apresenta uma análise comparativa de soluções de comparação de sequências biológicas que usam escalonamento dinâmico de tarefas. O Capítulo 5 apresenta o projeto do MASA-StarPU. O Capítulo 6 apresenta os resultados obtidos assim com a análise dos mesmos. Finalmente, o Capítulo 7 apresenta as conclusões e trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

Na Bioinformática, uma da operações mais básicas é a comparação de sequências biológicas. A comparação é feita alinhando-se as sequências de maneira a organizá-las para identificar regiões de similaridade, que podem ser uma consequência de relações funcionais, estruturais ou evolutivas [8]. Na área da Ciência da Computação, existem diversos algoritmos para realizar o alinhamento de sequências biológicas [8]. No presente capítulo, são inicialmente detalhados o alinhamento de sequências e os modelos de *gap*, que são lacunas introduzidas em uma das sequências. A seguir, quatro algoritmos exatos para o alinhamento de sequências são explicados. Finalmente, são apresentadas duas estratégias de poda.

2.1 Alinhamento de sequências biológicas

Sequências biológicas são representadas por uma sequência de caracteres. Para determinar semelhanças no alinhamento de sequências, é preciso estabelecer algumas regras para descrever como o alinhamento deve ser realizado. Chama-se de *match* a pontuação atribuída quando os caracteres são iguais, de *mismatch* quando são diferentes e de *gaps* a inserção de espaços em uma das sequências [8].

Podemos classificar um alinhamento em dois tipos básicos, de acordo com suas características. Um alinhamento é dito global quando possui todos os caracteres das sequências. Neste caso, o objetivo é identificar as similaridades entre as duas sequências como um todo [8]. Um alinhamento é dito local quando só estão presentes as partes das sequências que mais se assemelham. O objetivo desta abordagem é obter a região com maior similaridade entre as sequências. A Figura 2.1 apresenta um exemplo de alinhamento global (esquerda)

e local (direita) entre duas sequências S_0 e S_1, onde os *matches* estão representados pelo símbolo ':', *mismatches* pelo símbolo '.' e *gaps* pelo símbolo '-'.

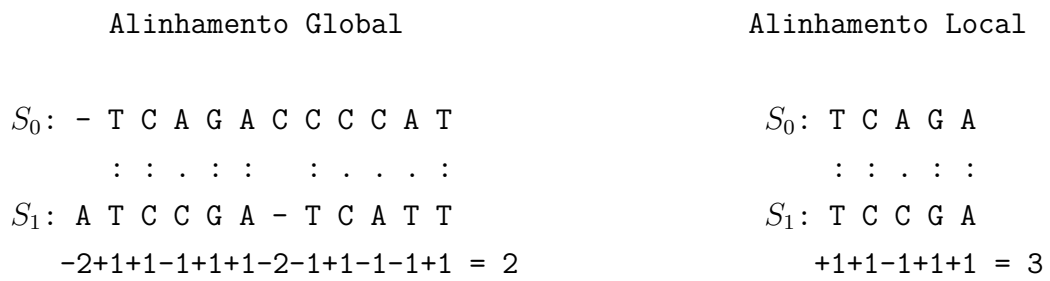


Figura 2.1: Exemplo de alinhamento global e local entre as sequências S_0=TCAGACCCCAT e S_1=ATCCGATCATT

2.2 Modelos de Gap

Cada alinhamento possui um escore associado para *matches*, *mismatches* e *gaps* [8]. O escore dos *gaps* é definido por uma função de penalidade dependente do número de *gaps* consecutivos. Dentre as funções mais comuns encontramos [8]:

- **Linear gap**: A função de penalidade de *gaps* atribui o mesmo valor negativo $-G$ para cada *gap*, ou seja, $\gamma(k) = -k.G$, onde G é o valor do parâmetro de *gap penalty* e k é comprimento de uma sequência consecutiva de *gaps*.
- **Affine gap**: A função de penalidade de *gaps* atribui uma penalidade maior $-G_{first}$ para o primeiro *gap* de uma sequência de *gaps* e uma penalidade menor $-G_{ext}$ para os demais *gaps* dessa sequência. Assim, a penalização de uma sequência de k *gaps* consecutivos é dada pela fórmula $\gamma(k) = -G_{first} - (k-1).G_{ext}$, sendo G_{first} a penalidade do primeiro *gap* e G_{ext} a penalidade por estendê-lo. O linear *gap* pode ser considerado um caso específico do *affine gap*, onde $G_{ext} = G_{first}$.

2.3 Algoritmos Exatos

Os algoritmos exatos que realizam a comparação de sequências biológicas visam encontrar o escore ótimo e seu respectivo alinhamento, dentre todos os possíveis alinhamentos de duas sequências, considerando o sistema de pontuação escolhido.

Nesta seção serão descritos alguns dos principais algoritmos de comparação de sequências biológicas que sempre retornam a solução ótima.

2.3.1 Needleman-Wunsch (NW)

O algoritmo Needleman-Wunsch(NW) [19] foi publicado em 1970 e é um dos primeiros algoritmos de programação dinâmica para comparar sequências biológicas. O algoritmo NW obtém o alinhamento global ótimo entre as sequências S_0 e S_1, de tamanhos m e n, respectivamente, permitindo que *gaps* sejam inseridos para melhorar o alinhamento.

É executado em duas fases: (a) Cálculo da matriz de programação dinâmica; e (b) Obtenção do alinhamento ótimo por meio do *traceback*.

Calculo da matriz de programação dinâmica

Seja H a matriz de programação dinâmica calculada pelo algoritmo, onde $H_{i,j}$ é o escore do alinhamento global entre as subsequências $S_0[0...i]$ e $S_1[0...j]$. Primeiramente, inicia-se o elemento $H_{0,0}$ com o valor zero para representar o escore das sequências vazias. A seguir, os campos $H_{i,0}$ e $H_{0,j}$ são inicializados com os valores $-i.G$ e $-j.G$ respectivamente, onde G é o valor de penalidade linear dos *gaps* (Seção 2.2).

A equação de recorrência do algoritmo NW fundamenta-se no fato de que $H_{i,j}$ é o maior valor para o alinhamento global entre duas subsequências $S_0[1...i-1]$ e $S_1[1...j-1]$, obtido a partir de três situações: (1) alinhar $S_0[1...i-1]$ e $S_1[1...j-1]$ acrescentando os caracteres $S_0[i]$ e $S_1[j]$; (2) alinhar $S_0[1..i]$ e $S_1[1...j-1]$ e inserir um gap alinhado com $S_1[j]$; (3) alinhar $S_0[1...i-1]$ e $S_1[1...j]$ e inserir $S_0[i]$ alinhado com um gap. Logo, a equação de recorrência de NW é a Equação 2.1 [19] sendo $escore(S_0[i],S_1[j])$ o parâmetro de *match* ou *mismatch* de acordo com a comparação entre os caracteres $S_0[i]$ e $S_1[j]$.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + escore(S_0[i], S_1[j]), \\ H_{i-1,j} - G, \\ H_{i,j-1} - G \end{cases} \tag{2.1}$$

Obtenção do alinhamento ótimo (*traceback*)

Com a matriz H calculada, cada célula $H_{i,j}$ possui um ponteiro para indicar qual célula dentre $H_{i-1,j-1}$, $H_{i-1,j}$ e $H_{i,j-1}$, originou o seu valor, de acordo com a equação de recorrência. Então, percorre-se a sequência de ponteiros começando pela última célula de $H(H_{m,n})$, ou seja, onde i e j contem seus valores máximos, em direção ao início, onde os valores de i e j decrescem, até zero.

A Tabela 2.1 apresenta um exemplo da matriz de programação dinâmica para alinhamento global ótimo com o algoritmo NW. Os parâmetros utilizados foram: *match*: +1, *mismatch*: -1 e *gap*: -2. As células em destaque indicam o percurso reverso (*trace-*

back) capaz de produzir o alinhamento global ótimo com escore -2, conforme ilustrado na Figura 2.2.

```
S0: - T C A G A C C C C A T
      : : . : :   : . . . :
S1: A T C C G A - T C A T T
   -2+1+1-1+1+1-2-1+1-1-1+1 = -2
```

Figura 2.2: Alinhamento global ótimo entras as sequências S_0=TCAGACCCCAT e S_1=ATCCGATCATT

Tabela 2.1: Matriz de programação dinâmica gerada pelo alinhamento global ótimo entre as sequências S_0=ATCCGATCATT e S_1=TCAGACCCCAT.

		T	C	A	G	A	C	C	C	C	A	T
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22
A	**-2**	-1	-3	-3	-5	-7	-9	-11	-13	-15	-17	-19
T	-4	**-1**	-2	-4	-4	-6	-8	-10	-12	-14	-16	-16
C	-6	-3	**0**	-2	-4	-5	-5	-7	-9	-11	-13	-15
C	-8	-5	-2	**-1**	-3	-5	-4	-4	-6	-8	-10	-12
G	-10	-7	-4	-3	**0**	-2	-4	-5	-5	-7	-9	-11
A	-12	-9	-6	-3	-2	**1**	**-1**	-3	-5	-6	-6	-8
T	-14	-11	-8	-5	-4	-1	0	**-2**	-4	-6	-7	-5
C	-16	-13	-10	-7	-6	-3	0	1	**-1**	-3	-5	-7
A	-18	-15	-12	-9	-8	-5	-2	-1	0	**-2**	-2	-4
T	-20	-17	-14	-11	-10	-7	-4	-3	-2	-1	**-3**	-1
T	-22	-19	-16	-13	-12	-9	-6	-5	-4	-3	-2	**-2**

Complexidade de Tempo e espaço

No algoritmo NW, precisa-se calcular toda a matriz de programação dinâmica. Esta matriz contem $(m+1)\times(n+1)$ elementos sendo que m é o tamanho de S_0 ($m = |S_0|$) e n é o tamanho de $S_1(n = |S_1|)$. Considerando que o tamanho das duas sequências são próximos, o algoritmo possui complexidade $O(n^2)$ para o cálculo da matriz de programação dinâmica tanto em tempo de processamento como de memória. A complexidade do *traceback* é proporcional ao tamanho do alinhamento, portanto possui um limite superior de $O(m)$ [19].

Para sequências muito grandes o algoritmo exige uma grande quantidade de tempo de processamento e memória. Por exemplo, duas sequências de DNA com 10 MBP (10

Milhões de Pares de Bases) demandaria 400 *Tera bytes* de memória caso cada célula de matriz ocupasse 4 bytes.

Caso o escore ótimo seja suficiente como resultado, ou seja, caso não seja necessário fazer o *traceback*, não é preciso armazenar completamente a matriz de programação dinâmica, pois cada linha e coluna depende apenas de suas antecessoras imediatas para serem calculadas. Assim, armazena-se apenas duas linhas ou colunas para obter o escore ótimo, tornando a complexidade do algoritmo linear $O(m)$ em termos de espaço [19]. O escore sem o alinhamento é útil em casos de comparação de uma sequência com um banco de sequências, com o objetivo de encontrar as sequências mais similares. Em contrapartida, para uma análise mais detalhada, o estudo biológico não é feito apenas com o escore, mas sim pela análise do alinhamento como um todo. Logo, mesmo na comparação de uma sequência com um banco de sequências, a obtenção do alinhamento pode ser importante.

2.3.2 Smith-Waterman (SW)

Outra operação muito comum na área de Bioinformática é a aquisição do alinhamento local ótimo entre duas sequências S_0 e S_1, que pode ser alcançado pelo algoritmo Smith-Waterman [9].

Este algoritmo foi proposto em 1981 e é muito parecido com o Needleman-Wunsch (Seção 2.3.1), com três diferenças. Primeiramente, todas as células da primeira linha e primeira coluna da matriz são inicializadas com zero. Em segundo lugar, na equação de recorrência utiliza-se o valor zero para impedir que existam números negativos na matriz de programação dinâmica. Desta forma, escreve-se a equação de recorrência do Smith-Waterman de acordo com a Equação 2.2 [9] sendo escore($S_0[i]$,$S_1[j]$) o parâmetro de *match* ou *mismatch* de acordo com a similaridade entre os caracteres $S_0[i]$ e $S_1[j]$ e G a penalidade de *gap*.

$$H_{i,j} = max \begin{cases} 0 \\ H_{i-1,j-1} + escore(S_0[i], S_1[j]), \\ H_{i-1,j} - G, \\ H_{i,j-1} - G \end{cases} \tag{2.2}$$

A ocorrência de um valor zero representa o início de um novo alinhamento, pois se o alinhamento até tal ponto possui escore negativo, então é mais vantajoso começar um novo alinhamento local nesta posição [9].

A terceira diferença em relação ao Needleman-Wunsch é na forma de como é calculado o *traceback*. Ao contrário do Needleman-Wunsch (Seção 2.3.1), o *traceback* do Smith-Waterman começa a partir do maior valor da matriz de programação dinâmica, ou seja,

onde o valor de $H_{i,j}$ é máximo. O percurso é feito até se encontrar uma célula com o valor zero. Desta forma, obtém-se o alinhamento local ótimo entre duas subsequências. A complexidade de tempo e espaço são as mesmas do algoritmo Needleman-Wunsch (Seção 2.3.1).

A Tabela 2.1 apresenta um exemplo de matriz para o alinhamento local ótimo com escore +3. Os parâmetros utilizados foram *match*: +1 *mismatch*: −1 e *gap*: −2.

Tabela 2.2: Matriz de programação dinâmica gerada pelo alinhamento local ótimo entre as sequências S_0=ATCCGATCATT e S_1=TCAGACCCCAT.

		T	C	A	G	A	C	C	C	C	A	T
	0	0	0	0	0	0	0	0	0	0	0	0
A	**0**	0	0	1	0	1	0	0	0	0	1	0
T	0	**1**	0	0	0	0	0	0	0	0	0	2
C	0	0	**2**	0	0	0	1	1	1	1	0	0
C	0	0	1	**1**	0	0	1	2	2	2	0	0
G	0	0	0	0	**2**	0	0	0	1	1	1	0
A	0	0	0	1	0	**3**	1	0	0	0	2	0
T	0	1	0	0	0	1	2	0	0	0	0	3
C	0	0	2	0	0	0	2	3	1	1	0	1
A	0	0	0	3	1	1	0	1	2	0	2	0
T	0	1	0	1	2	0	0	0	0	1	0	3
T	0	1	0	0	0	1	0	0	0	0	0	1

A células destacadas indicam um alinhamento local ótimo entre as sequências S_0 = ATCCGATCATT e S_1 = TCAGACCCCAT produzido pelo *traceback*. O mesmo alinhamento local ótimo está representado na Figura 2.2.

```
S0: T C C G A
    : : . : :
S1: T C A G A
   +1+1-1+1+1 = 3
```

Figura 2.3: Alinhamento local ótimo entras as sequências S_1 = TCAGACCCCAT e S_0 = ATCCGATCATT

2.3.3 Gotoh

O algoritmo Gotoh [20] foi proposto em 1982 e é um algoritmo de programação dinâmica que calcula o alinhamento global ótimo para o modelo *affine gap* (Seção 2.2). Substitui-se

a constante de *gap* G pela função $\gamma(k) = -G_{first} - (k-1).G_{ext}$ (Seção 2.2). O algoritmo de Gotoh usa 3 matrizes de programação dinâmica e calcula o alinhamento global ótimo com complexidade de tempo e espaço $O(n^2)$.

Para cada célula da matriz de programação dinâmica (i, j), são mantidas 3 células correspondentes a três situações: (1) $S_0[i]$ alinhado com $S_1[j]$; (2) um *gap* alinhado com $S_1[j]$ e (3) um *gap* alinhado com $S_0[i]$. Definem-se então três matrizes H, E e F para cada situação. As equações de recorrência de NW (Seção 2.3.1) são modificadas, gerando as equações 2.3, 2.4 e 2.5 [20]. Para adaptar o algoritmo para a obtenção do alinhamento local, é suficiente incluir o valor zero em H, limitando assim o valor mínimo da matriz H.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + escore(S_0[i], S_1[j]), \\ E_{i,j}, \\ F_{i,j} \end{cases} \tag{2.3}$$

$$E_{i,j} = max \begin{cases} E_{i,j-1} - G_{ext}, \\ H_{i,j-1} - G_{fist} \end{cases} \tag{2.4}$$

$$F_{i,j} = max \begin{cases} F_{i-1,j} - G_{ext}, \\ H_{i-1,j} - G_{fist} \end{cases} \tag{2.5}$$

2.3.4 Myers e Miller

Os algoritmos vistos nas Seções 2.3.1, 2.3.2 e 2.3.3 possuem complexidade quadrática de memória, o que inviabiliza a aquisição do alinhamento entre sequência muito longas. Para resolver tal problema, algoritmos foram desenvolvidos utilizando espaço linear, ou seja, na ordem de $O(m+n)$, onde m e n são os tamanhos da sequências S_0 e S_1, respectivamente.

Hirschberg [21] desenvolveu em 1975 um algoritmo para resolver o problema da Maior Subsequência Comum (LCS – *Longest Common Subsequence*) [22] com complexidade de espaço linear. Este algoritmo, posteriormente, foi adaptado por Myers e Miller [1] em 1988 para transformar o algoritmo de Gotoh (Seção 2.3.3) em uma solução de espaço linear.

Sendo assim, embora o algoritmo de Gotoh possa alinhar sequências de comprimentos n e m em nm passos, estas operações ainda requerem o armazenamento de três matrizes de tamanho nm em memória. Myers e Miller [1] reduziram este uso de memória, armazenando informações suficientes para calcular o escore em n unidades de memória.

Considerando-se a obtenção de um alinhamento global, a ideia é encontrar o ponto médio, chamado de *crosspoint*, pelo qual passa o alinhamento ótimo. Para encontrar este ponto, o cálculo é feito em duas partes, conforme definido por Hirshberg [21]. Na primeira

parte, o cálculo é feito em espaço linear linha a linha, do início para o meio da matriz até a linha central ($m/2$). Na segunda parte, o processamento é feito sobre as duas sequências invertidas do final para o meio, até que a mesma linha central $m/2$ seja calculada. Soma-se então as posições correspondentes das duas linhas centrais, ou seja, posição n na linha central calculada no sentido original e somada à posição n na linha central calculada no sentido inverso. A posição em $m/2$ onde a soma é máxima é o *crosspoint*. Essa posição é usada para dividir a matriz em 4 partes e o procedimento é repetido de maneira recursiva no quadrante superior esquerdo e no quadrante inferior direito. A Figura 2.4 apresenta dois níveis de recursão do algoritmo.

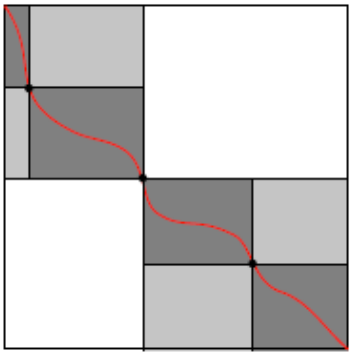


Figura 2.4: Dois níveis de recursão do algoritmo de Myers e Miller [1].

A linha vermelha representa onde passa o alinhamento global ótimo e os pontos marcados ao longo da linha são os *crosspoints*. Depois que um *crosspoint* é encontrado, o problema é recursivamente dividido em subproblemas menores até que problemas triviais sejam encontrados.

2.4 Estratégia de poda (*pruning*)

À medida em que as pesquisas na área de comparação de sequências biológicas foram evoluindo, os pesquisadores perceberam que, em alguns casos, não é necessário o cálculo de toda a matriz de programação dinâmica para obter os alinhamentos ótimos.

A seguir, serão apresentadas duas das principais estratégias de *pruning* utilizadas para otimizar os cálculo de alinhamento de sequências.

2.4.1 Fickett

Fickett [2] desenvolveu em 1984 uma otimização capaz de reduzir a complexidade de tempo e memória para $O(nk)$, onde k é a largura da faixa de diagonais que contém

todo o alinhamento ótimo. Sequências muito similares tendem a possuir poucos *gaps* e, consequentemente, possuem uma faixa pequena. Logo o tempo de processamento pode ser consideravelmente reduzido, para sequências similares, utilizando esta técnica. A Figura 2.5 ilustra a técnica de Fickett [2].

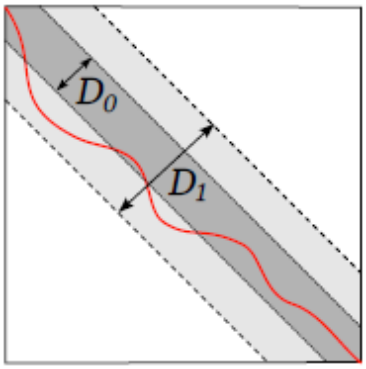


Figura 2.5: Representação do algoritmo de Fickett [2]. O alinhamento ótimo não está inicialmente contido na área delimitada pelo valor D_0, mas está dentro da área delimitada por D_1.

A ideia básica por trás da técnica de Fickett é fazer o cálculo da matriz de programação dinâmica somente em diagonais próximas à diagonal principal. O valor D_0 na Figura 2.5 representa a distância inicial onde o alinhamento é calculado. Caso o alinhamento não esteja totalmente presente na faixa delimitada por D_0, uma nova faixa, representada por D_1, é tomada como delimitador e o cálculo é refeito. Isso é feito sucessivamente até que todo o alinhamento possa ser encontrado.

2.4.2 Block *Pruning*

O *block pruning* foi proposto em 2013 [3] e aplicado inicialmente ao alinhamento local. Baseia-se no fato de que, caso o escore máximo calculado até o momento possua um valor suficientemente grande, existem partes da matriz de programação dinâmica com valores de escore tão pequenos, onde é matematicamente impossível que seja produzido um escore maior do que o já obtido. Sendo assim, essas partes da matriz não precisam ser calculadas. A Figura 2.6 ilustra essa situação. A determinação da área de poda é feita por blocos de células, e por isso a estratégia foi chamada de *block pruning*.

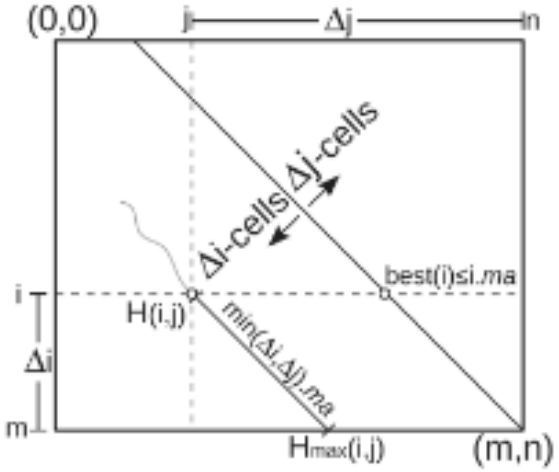


Figura 2.6: Cálculo de uma célula *prunable* [3].

Para reduzir o número de células processadas, devemos identificar inicialmente uma célula *prunable* antes mesmo de calcularmos o seu valor. Seja m o menor valor entre Δi (distancia entre a linha atual e a linha final) e Δj (distancia entre a coluna atual e a coluna final) mostrado na Figura 2.6. Caso o valor de $H_{i,j} + m$ seja menor do que o escore máximo atual, $H_{i,j}$ é uma célula *prunable*. Caso as células $(i-1, j)$, $(i, j-1)$ e $(i-1, j-1)$ sejam *prunable*, então a célula (i, j) também será [3].

Uma janela não-*prunable* é dada pelo intervalo $[k_s...k_e]$ de colunas que deve ser processado em uma determinada linha. Inicialmente, $[k_s...k_e] = [0...n]$. Para cada linha i, calcula-se todas as células $(i, j \in [k_s...k_e])$ e, então, atualiza-se a janela não-*prunable* para os valores $[k'_s...k'_e]$ onde os valores k'_s e k'_e são, respectivamente, a primeira e a última célula não-*prunable* desta linha. Assumindo que as células $(i-1, j \in [0...k_s])$ são *prunable*, então todas as células $(i, j \in [0...k_s])$ também serão e, por indução, todas as células $(i' \geq i, j \in [0...k_s])$ também serão [3].

O *block pruning* possui resultados expressivos quando as sequências são similares. A Figura 2.7 apresenta, em cinza, a área de poda da matriz, que não foi calculada na obtenção do alinhamento local ótimo entre sequências similares.

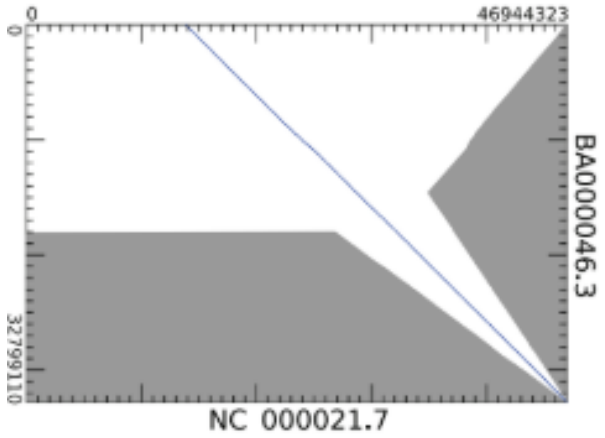


Figura 2.7: Área não calculada da matriz de programação dinâmica (em cinza) [4].

Capítulo 3

Ambientes de programação paralela

Para realizar a comparação entre duas sequências biológicas muito longas, os algoritmos apresentados no Capítulo 2 podem levar muito tempo para serem executados. Por isso, ambientes de programação paralela são utilizados a fim de obter o resultado em tempo hábil.

Sendo assim, neste capítulo, serão apresentados alguns dos principais ambientes de programação paralela em CPU.

3.1 OpenMP

O OpenMP (*Open Multi-Processing*) é uma interface de programação (API) que suporta programação paralela utilizando memória compartilhada [23].

O OpenMP é uma implementação de *multithreading* que, em seu uso tradicional, utiliza um método de paralelização pelo qual uma *thread* mestre bifurca em *threads* filhas onde as tarefas do sistema são divididas entre elas (modelo *fork-join*) [24]. As *threads* então são executadas simultaneamente, enquanto o ambiente as aloca em diferentes processadores (ou *cores*). A Figura 3.1 apresenta uma versão simplificada da hierarquia de *threads* do OpenMP.

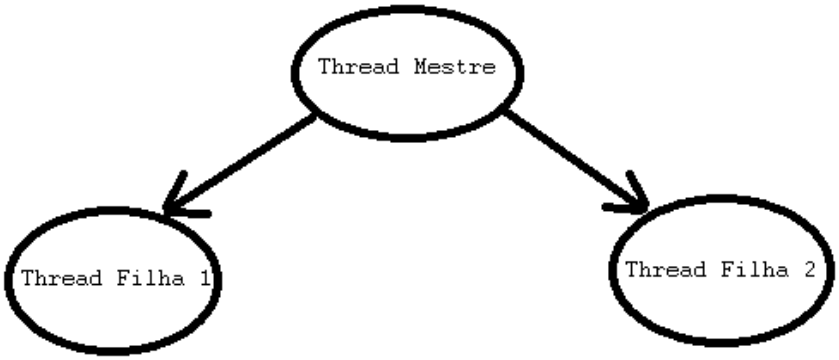


Figura 3.1: Hierarquia de *threads* do OpenMP .

A seção de código que deve executar em paralelo é geralmente marcada com uma diretiva de compilador que fará com que as *threads* sejam criadas antes que uma seção seja executada. Cada *thread* tem um identificador (ID) anexado a ela, que pode ser obtido usando uma função chamada `omp_get_thread_num()`. O ID da *thread* é um número inteiro, e a *thread* mestre possui um ID igual a 0. Após a execução do código em paralelo, as *threads* se juntam novamente à *thread* mestre, que continua até o final do programa.

Em C/C++, o OpenMP utiliza `#pragmas` para a crição de *threads*. O `#pragma omp parallel` é usado para forçar a criação de *threads* para executar o trabalho incluído na construção em paralelo. O Codigo 3.1 apresenta um código OpenMP onde cada *thread* imprime "hello world".

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(void)
5 {
6     #pragma omp parallel
7     printf("Hello world.\n");
8     return 0;
9 }
```

Código 3.1: *Hello World em OpenMP*

Na linha 6, o `#pragma omp parallel` é utilizado para criar novas *threads* para executar a linhas seguintes. Em um computador com dois núcleos, o Código 3.1 imprime como saída dois "Hello world".

3.2 OpenCL

O OpenCL (*Open Computing Language*) é um *framework* para escrever programas executados em plataformas heterogêneas como: CPUs, GPUs, processadores digitais de sinais (DSPs), FPGAs (*field-programmable gate arrays*) e outros processadores ou aceleradores [25]. O OpenCL suporta linguagens de programação baseadas em C99 e C++ 11 para programar esses dispositivos e interfaces de programação (APIs) para controlar a plataforma e o paralelismo.

O OpenCL considera um computador como um conjunto de diferentes dispositivos, que podem ser unidades de processamento central (CPUs) ou aceleradores, como unidades de processamento gráfico (GPUs). As funções executadas em um dispositivo OpenCL são chamadas de *kernels*. Um único dispositivo consiste tipicamente em várias unidades de processamento (núcleos de processamento). Uma única execução do *kernel* pode ser feita em todas ou em subconjunto destas unidades.

Além de sua linguagem de programação C, o OpenCL define uma API que permite que programas executem o *kernel* nos dispositivos e gerenciem a memória, que é separada da memória do *host*.

O OpenCL define uma hierarquia de memória de quatro níveis para seus dispositivos: a) **memória global**: compartilhada por todos os elementos de processamento, mas com alta latência de acesso (*global*); b) **memória somente leitura**: menor, baixa latência, gravável pela CPU do *host*, mas não pelos dispositivos (*constant*); c) **memória local**: compartilhada por um grupo de elementos de processamento (*local*); d) **memória privada por elemento** (*register*, *private*): específica de cada elemento de processamento sem compartilhamento.

Nem todo dispositivo precisa implementar cada nível da hierarquia de memória no hardware. Além disso, os dispositivos podem ou não compartilhar a memória com a CPU do *host*. A API do *host* fornece acesso aos *buffers* de memória do dispositivo e funções para transferir dados entre o *host* e os dispositivos.

A linguagem de programação que é usada para escrever *kernels* é chamada de OpenCL C e é baseada em C99, mas adaptada para ajustar o modelo de dispositivos em OpenCL. Os buffers de memória residem em níveis específicos da hierarquia de memória e os ponteiros são anotados com os qualificadores de região `__global`, `__local`, `__constant` e `__private`, refletindo isso. Em vez de um programa de dispositivo ter uma função *main*, as funções OpenCL C são marcadas como `__kernel` para sinalizar que são pontos de entrada no programa a ser chamado a partir do programa *host*.

O programa *host* possui uma função *main* que pode criar *kernels* através da função `clCreateKernel`. O *kernel* deve ser delegado e executado por um dispositivo dentro da

lista de dispositivos mapeados pelo OpenCL. O Código 3.2 apresenta uma implementação de um programa que imprime "Hello World" em OpenCL.

```
1  __kernel void HelloWorld(){
2      std::cout("Hello World\n");
3  }
4
5  int main(){
6
7      std::vector<cl::Platform> platforms;
8      cl::Platform::get(&platforms);
9
10     auto platform = platforms.front();
11     std::vector<cl::Device> devices;
12     platform.getDevices(CL_DEVICE_TYPE_CPU, &devices);
13
14     auto device = devices.front();
15
16     cl::Context context(device);
17     cl::Program program(context);
18
19     auto err = program.build("-cl-std=CL1.2");
20
21     cl::Kernel kernel(program, "HelloWorld", &err);
22
23     cl::CommandQueue queue(context, device);
24     queue.enqueueTask(kernel);
25     queue.enqueueReadBuffer(0, GL_TRUE, 0, sizeof(0), 0);
26 }
```

Código 3.2: *Hello World em OpenCL*

Da linha 1 até a linha 3 está escrito o código do *kernel* que apenas imprime a frase "Hello World" e termina. Da linha 7 à linha 12 é feita a verificação de quantas CPUs estão disponíveis no dispositivo e criada uma lista destas CPUs. A linha 14 obtêm a primeira CPU da lista. A linha 16 cria o contexto em que esta CPU deve executar. Este contexto define alguns parâmetros que a CPU pode utilizar, como por exemplo o tamanho e as permissões da memória utilizada. No entanto, como o código do *kernel* apenas imprime uma *string* que não está em memória, estes parâmetros não precisam ser configurados. A linha 17 cria um programa a partir deste contexto, ou seja, um código que pode ser compilado. A linha 19 compila este programa. Na linha 21 a rotina `kernel` faz o papel da função `clCreateKernel` porém em C++. Ela um cria um *kernel* a partir do código

de uma função e o contexto de um programa. Por fim, as linhas 23, 24 e 25 preparam o *kernel* para a CPU executar.

3.3 OmpSs

O propósito do OmpSs é oferecer um modelo único de programação abrangendo as diferentes arquiteturas homogêneas e heterogêneas [26]. O modelo de programação OmpSs é baseado no modelo básico do OpenMP (Seção 3.1) com algumas modificações: o modelo de execução é diferente, um número maior de construtores é oferecido e permite a implementação de dependências entre as tarefas.

Enquanto o OpenMP tradicional possui um modelo *fork-join*, o OmpSs define um modelo de *pool* de *threads* onde todas as *threads* existem desde o começo da execução. Destas *threads*, apenas a *thread* mestre começa a executar o código do usuário enquanto as outras *threads* serão executadas quando seus parâmetros de *input* estiverem prontos (modelo *dataflow*).

Três cláusulas são adicionadas à construção da *thread*, que vem do modelo do StarSs [27]: `input`, `output` e `inout`. As três aceitam uma expressão que avalia um conjunto de *lvalues*, que são expressões que se referem a um objeto.

Se uma tarefa criada tiver uma cláusula de `input` que seja avaliada para um determinado *lvalue*, a tarefa não estará qualificada para ser executada enquanto uma tarefa criada anteriormente com uma cláusula `output` que se aplica ao mesmo *lvalue* não terminou sua execução.

Se uma tarefa criada tiver uma cláusula de `output` que seja avaliada para um determinado *lvalue*, a tarefa não estará qualificada para execução, se uma tarefa criada anteriormente com uma cláusula de `input` ou `output` aplicada ao mesmo *lvalue* não tiver terminado sua execução.

Se uma tarefa criada tiver uma cláusula `inout` avaliada para um determinado *lvalue*, ela será considerada como se tivesse uma cláusula de `input` e uma cláusula de `output` avaliada para esse mesmo *lvalue*. A Figura 3.2 demonstra uma versão simplificada do modelo de hierarquia de *threads* proposto pelo OmpSs.

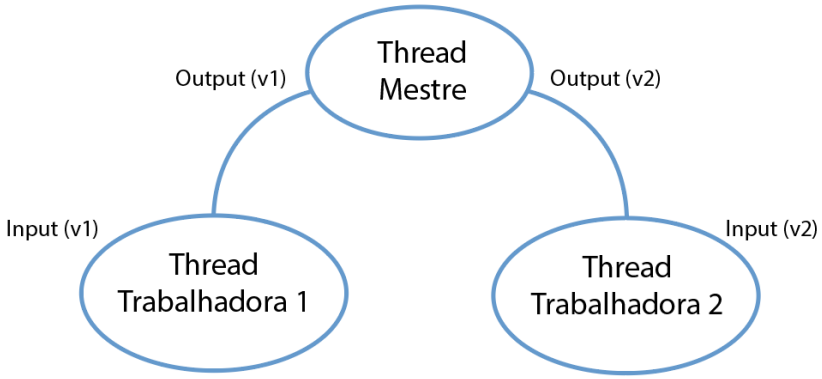


Figura 3.2: Modelo de hierarquia de *threads* do OmpSs.

A construção das tarefas permite expressar dependências de dados entre tarefas usando as cláusulas `in`, `out` e `inout`. Elas permitem especificar para cada *thread* no programa quais dados uma tarefa está aguardando. Cada vez que uma nova tarefa é criada, suas dependências de entrada e saída são comparadas com as das tarefas existentes. O Código 3.3 imprime a frase "Hello World!", porém ele está dividido em dois métodos em que a execução de um depende do outro.

```
1  void hello(){
2      printf("Hello ");
3      done = 1;
4  }
5
6  void world(){
7      printf("World!\n");
8  }
9
10 int main(){
11     int done = 0;
12
13     #pragma omp task input(hello)
14     world();
15
16     #pragma omp task output(hello)
17     hello();
18 }
```

Código 3.3: *Hello World em OmpSs*

Os métodos `hello()` e `world()` são responsáveis por imprimir as palavras "Hello" e "World" respectivamente. No entanto, o método `hello()` precisa executar antes do

`world()`. As linhas 13 e 14 são responsáveis por paralelizar estes métodos e realizar este comportamento. Na linha 13 o `input (hello)` indica que o método `world()` só poderá ser executado depois que houver um `output (hello)` o que acontece na linha 16 após executar o método `hello()`.

3.4 StarPU

Escrever código portátil que é executado em diferentes dispositivos é atualmente uma importante tarefa, especialmente se a aplicação precisar explorar vários tipos tecnologias de aceleradores, como GPUs, ao mesmo tempo. Para ajudar a resolver esse problema, foi criado o StarPU, um ambiente de execução que fornece uma interface unificando a execução em aceleradores e processadores multicore [17].

O StarPU define uma abstração de tarefa que pode ser executada em um núcleo de processamento ou descarregada em aceleradores de forma assíncrona [17]. Os programadores podem implementar tarefas por meio das linguagens de programação (por exemplo, CUDA). Do ponto de vista da programação, o StarPU é responsável por executar as tarefas e se encarrega do balanceamento de carga entre as mesmas.

3.4.1 Declaração de tarefas e dependências de dados

A estrutura de tarefas do StarPU inclui uma descrição de alto nível de cada parte dos dados manipulados pela tarefa e como eles são acessados, ou seja, R (*read only*), W (*write only*), R / W (*read / write*). Também é possível expressar dependências de tarefas no StarPU através de grafos de interdependências. Como as tarefas são lançadas de forma assíncrona, isso também permite StarPU reorganizá-las para melhorar o desempenho.

O StarPU vem com um plug-in GCC que estende a linguagem de programação C com pragmas e atributos que facilitam a transformação de um programa C sequencial em um programa paralelo.

Antes de enviar qualquer tarefa para o StarPU, a chamada `starpu_init()` deve ser feita. As tarefas podem então ser enviadas até o término do StarPU, feito por uma chamada para `starpu_shutdown()`. Para criar uma tarefa é preciso utilizar a função `starpu_task_create()`. Esta função cria a tarefa com suas configurações padrão mas não envia a tarefa para o StarPU executar.

O StarPU define um conceito chamadado de *codelet*, uma estrutura que representa um núcleo computacional. Tal *codelet* pode conter uma implementação do mesmo núcleo em diferentes arquiteturas como CUDA e x86. A tarefa criada pela função `starpu_task_create()` possui um ponteiro `starpu_task::cl` para o *codelet* em que a tarefa irá executar. Em outras palavras, a estrutura do *codelet* descreve a qual tipo de

núcleo computacional a tarefa deve ser transferida, considerando-se as diferentes arquiteturas. A tarefa é um conjunto de instruções que o *codelet* deve executar.

Após alocada a tarefa e definido o seu *codelet*, a função `starpu_task_submit()` é chamada para que a tarefa seja executada. Ao serem submetidas ao StarPU para execução, todas as novas tarefas criadas passam a ser filhas da tarefa que as criou. A função `starpu_shutdown()` não garante que tarefas assíncronas tenham sido executadas antes de retornar: `starpu_task_wait_for_all()` pode ser usada para esse efeito.

Por padrão, o StarPU considera as tarefas na ordem em que são submetidas. Caso alguma tarefa deva ser executada com prioridade, por exemplo, sua saída é necessária para muitas outras tarefas e pode, portanto, ser um gargalo se não for executada primeiro, o campo `starpu_task::priority` pode ser configurado para transmitir as informações de prioridade para o StarPU. O Código 3.4 imprime a frase "Hello world" utilizando os recursos do StarPU.

```
1
2  void HelloWorld(void *buffers[], void *cl_arg){
3      printf("Hello world\n");
4  }
5
6  struct starpu_codelet cl = {
7      .cpu_funcs = { HelloWorld },
8      .nbuffers = 0
9  };
10
11 int main (){
12     starpu_init(NULL);
13
14     struct starpu_task *task = starpu_task_create();
15     task->cl = &cl;
16     task->synchronous = 1;
17
18     starpu_task_submit(task);
19
20     starpu_shutdown();
21     return 0;
22 }
```

Código 3.4: *Hello World em StarPU*

A linha 2 declara a tarefa que imprime a frase "Hello World" enquanto a linha 3 a implementa. Da linha 6 até a linha 9 está definido o *codelet* utilizado. O campo `starpu_codelet :: nbuffers` especifica o número de buffers de dados que são manipu-

lados pelo *codelet*. No Codigo 3.4 o *codelet* não acessa ou modifica nenhum dado que é controlado pela biblioteca de gerenciamento de dados por isso seu valor é 0. Este *codelet* só pode ser executado em CPUs (linha 7). Quando um núcleo da CPU executar um *codelet*, ele chamará a função definida pelo campo *cpu_func* que no caso é a `HelloWorld`. A linha 12 inicializa o StarPU, o parametro `NULL` é para utilizar a configuração padrão. A linha 14 aloca e preenche a estrutura da tarefa com suas configurações padrões. A linha 15 define o `campo starpu_task :: cl`, um ponteiro para o *codelet* que a tarefa executará. A linha 16 preenche o campo `starpu_task :: synchronous` com 1 para que o envio da tarefa será síncrono, ou seja, a função `starpu_task_submit ()` não retornará até que a tarefa tenha sido executada, tornando-a uma chamada blocante. Se `starpu_task :: synchronous` não for definido, `starpu_task_wait()` precisa ser chamado após o envio da tarefa. A linha 18 submete a tarefa para que o StarPU a execute. Finalmente a linha 20 finaliza o StarPU.

As dependências entre as tarefas são definidas utilizando *tags*. Uma *tag* é uma *string* que identifica uma tarefa e o valor desta *string* deve ser definido pela aplicação. A *tag* que identifica uma tarefa pode ser configurada pelo campo `starpu_task :: tag_id`. Além das dependências o StarPU também permite que parâmetros sejam passados para as tarefas. As tarefas possuem um campo `starpu_task :: cl_arg`, um ponteiro do tipo `void` que permite a atribuição de variáveis de qualquer tipo inclusive `structs`. O Código 3.5 imprime a frase "Hello World" utilizando os recursos de dependências de tarefas e passagem de parâmetros para as tarefas.

```
1  void HelloWorld(void *buffers[], void *cl_arg){
2      printf("$s ", static_cast<char[]>(cl_arg));
3  }
4
5  struct starpu_codelet cl = {
6      .cpu_funcs = { HelloWorld },
7      .nbuffers = 0
8  };
9
10 int main (){
11     starpu_init(NULL);
12
13     struct starpu_task *task = starpu_task_create();
14
15     task->cl = &cl;
16     task->use_tag = 1;
17     task->tag_id = "a";
18     task->cl_arg = "Hello";
19     starpu_task_submit(task);
20
21     task->tag_id = "b";
22     task->cl_arg= "World";
23     starpu_tag_declare_deps("b", 1, "a");
24     starpu_task_submit(task);
25
26     starpu_task_wait_for_all();
27     starpu_shutdown();
28     return 0;
29 }
```

Código 3.5: *dependências de tarefas no StarPU*

Nas linhas 1, 2 e 3 está definida a função que as tarefas vão executar. Esta função funciona da mesma forma da função definida no Código 3.4 com apenas uma diferença: o parâmetro `void cl_arg` possui um valor, por isso, o comando `static_cast` (linha 2) está identificando o tipo do parâmetro passado, no caso, `char[]`. O restante do código funciona da mesma maneira do Código 3.4 com exceção das linhas 16 a 18 e 21 a 23. A linha 16 indica que a tarefa usará o sistema de *tags*. A linha 17 configura a *tag* identificadora da tarefa como a *string* "a" e a linha 18 passa como parâmetro a *string* "Hello". Na linha 21 começa a ser definida uma nova tarefa e esta tarefa será identificada pela *tag* "b". A linha 22 passa como parâmetro a *string* "world". Na linha 23 o comando `starpu_tag_declare_deps` define a dependência entre as duas tarefas. O primeiro parâmetro deste comando é a *tag* identificadora da tarefa a qual terá suas dependências

definidas, o segundo parâmetro é a quantidade de tarefas das quais esta tarefa depende e o resto dos parâmetros são das *tags* das tarefas as quais a tarefa do primeiro parâmetro depende. Neste caso, a tarefa com a *tag* "b" depende de uma tarefa e esta tarefa possui a *tag* "a". Logo, por causa das dependências a tarefa "a" sera executada antes da tarefa "b" e a frase "Hello world" será impressa na ordem correta.

3.4.2 Estratégias de escalonamento de tarefas

Do ponto de vista da programação, o StarPU é responsável por executar as tarefas que foram submetidas e diferentes estratégias podem ser adotadas para realizar o escalonamento dessas tarefas. No StarPU, a aplicação pode escolher qual estratégia de escalonamento de tarefas utilizar. No contexto de plataformas heterogêneas ou na execução de aplicações que possuem grafos complexos de tarefas, o desempenho varia muito de acordo com as arquiteturas e de acordo com a carga de trabalho. Portanto, é crucial considerar a especificidade das aplicações ao escalonar uma tarefa [17].

O StarPU utiliza um sistema de dicas opcionais adicionadas pelo programador para auxiliar no escalonamento das tarefas. A primeira dica consiste em especificar o nível de prioridade de uma tarefa. Tais prioridades impedem que tarefas cruciais tenham sua execução atrasada por muito mais tempo [17]. Outra adição é, sempre que possível, especificar um modelo de desempenho para atribuir pesos ao grafo de dependência entre as tarefas. Várias técnicas estão disponíveis para permitir que o programador faça previsões de desempenho ou use técnicas de amostragem para determinar automaticamente os custos desse modelo, a partir de medições reais [17]. No StarPU, isso pode ser feito por meio de uma execução de pré-calibragem, usando os resultados de execuções anteriores ou até mesmo adaptando dinamicamente o modelo com relação à execução atual.

O StarPU utiliza o modelo *master-slave* para alocação de tarefas. Neste modelo, existe um processo denominado mestre que é responsável por atribuir tarefas para os demais processos chamados de escravos. O StarPU utiliza modelos diferentes de filas para realizar a alocação de tarefas, podendo haver uma fila compartilhada para todos os escravos ou uma fila por escravo. Duas operações podem ser executadas nestas filas: envio da tarefa para a fila (*push*) e solicitação de uma tarefa para executar (*pop*).

A política adotada pelo mestre para realizar a alocações de tarefas é geralmente responsável por garantir o balanceamento de carga [17]. Distribuir de maneira uniforme as tarefas para os escravos pode não ser uma boa política quando as tarefas não são iguais, quando as máquinas onde as tarefas se executam não são iguais e quando há dependências entre as tarefas. No StarPU, o mestre pode adotar diversas estratégias diferentes para realizar a alocação das tarefas e neste trabalho foram utilizadas sete estratégias. Como todas as estratégias foram implementadas para utilizar filas, ou seja, todas elas utilizam

o paradigma de programação guloso para filas, quando uma tarefas pronta (*ready*) é submetida ao StarPU ela é inserida diretamente em uma das filas [17]. Uma tarefa é dita pronta todas as suas dependências foram resolvidas.

Cinco estratégias (*eager*, *prio*, *random*, *ws* e *lws*) utilizam o modelo básico de decisão, ou seja, as decisões são tomadas pelo mestre não levam em conta modelos de desempenho. A política *eager* usa uma fila central compartilhada entre todos os escravos e estes recuperam tarefas da fila em uma base de auto-agendamento *self-scheduling* [28]. A política *prio* também usa uma fila central, mas as tarefas são organizadas por ordem de prioridades fornecidas pelo programador. No StarPU 1.2.9, usado neste trabalho, o intervalo de prioridade é $[-5, 5]$ e 5 é a prioridade mais alta. No StarPU 1.3 e além, o intervalo é arbitrário. A política *random* utiliza filas locais e distribui as tarefas de forma aleatória. Existem duas políticas baseadas em *work stealing* [29]: (a) *ws* usa uma fila local para cada escravo e atribui tarefas ao escravo que libera a tarefa; se um escravo estiver ocioso, ele rouba tarefas do escravo mais carregado; e (b) *lws* se comporta como *ws*, mas rouba tarefas dos escravos vizinhos primeiro e, em seguida, dos escravos que estão executando mais longe na topologia da arquitetura.

Duas políticas (*dmda* e *dmdas*) que utilizam o modelo baseado em desempenho foram utilizadas nesta Dissertação. Neste caso, o programador pode especificar um modelo de custo para cada tarefa para complementar o sistema de dicas. Este modelo se baseia no tempo de término mais próximo (*Heterogenous Earliest finish Time* - HEFT) [30]. Dada a sua duração esperada nas várias arquiteturas, uma tarefa é atribuída à unidade de processamento que minimiza o tempo de término, com respeito à quantidade de trabalho já atribuída a este escravo. *Dmda* (*dequeue modeling data aware*) é uma política baseada em [31] minimizar o tempo de encerramento das tarefas e agendar tarefas quando elas estiverem disponíveis, levando também em consideração o tempo de transferência de dados. A política *dmdas* (*dequeue modeling data aware sorted*) classifica as tarefas por prioridade e usa *dmda* para agendar tarefas da mesma prioridade. Ao contrário de *prio*, não há um intervalo predefinido de valores de prioridade.

3.5 Arquitetura MASA

O principal objetivo do *Multiplatform Architecture for Sequence Aligners* (MASA) [4] é fornecer uma infraestrutura flexível e personalizável para desenvolver alinhadores de sequências em várias plataformas de hardware/software. O MASA propõe e implementa um conjunto de módulos independentes de plataforma que podem ser reutilizados por várias implementações específicas da plataforma. Ao contrário dos ambientes genéricos

de programação discutidos nas seções 3.1 a 3.4, o MASA é uma ferramenta específica para aplicações de alinhamento de sequências.

No projetar do MASA, foi analisado o código do CUDAlign (Seção 4.1) para determinar quais partes do código são independentes ou específicas da plataforma. A maior parte do tempo de execução do CUDAlign é gasto calculando as equações de recorrência (SW/NW (Seção 2.3.1/2.3.2)) e esta parte é específica da plataforma.

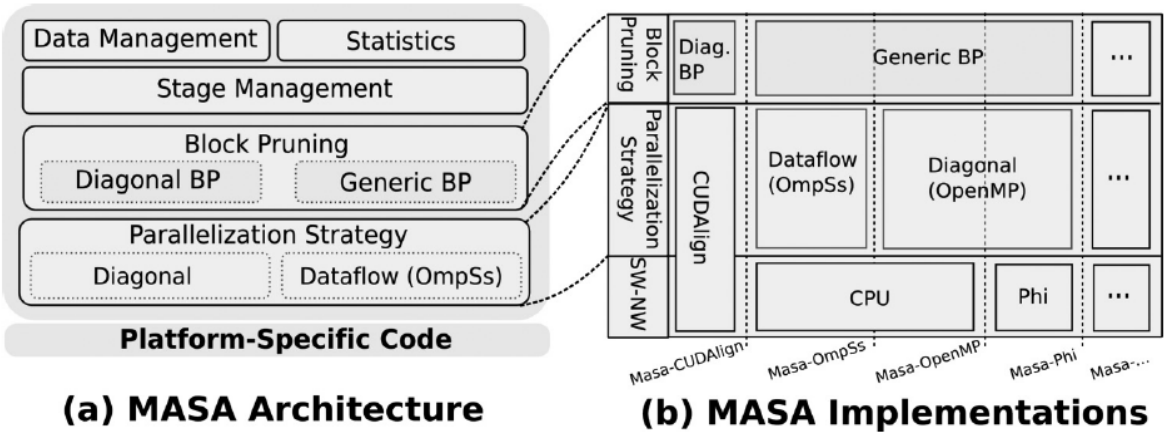


Figura 3.3: Visão geral do *framework* MASA [4].

Na Figura 3.3, (a) apresenta os módulos MASA, incluindo otimizações independentes de plataforma que são selecionadas pelo desenvolvedor, como a estratégia de poda *block pruning*, bem como o código específico da plataforma executado no processador de destino para calcular as recorrências de NW/SW; (b) mostra vários alinhadores desenvolvidos sobre o MASA com sua escolha de estratégias de poda, paralelização, ambientes de programação e plataformas de hardware.

Para criar uma nova extensão MASA, é preciso desenvolver uma classe chamada de alinhador (*aligner*) e fornecer uma instância dessa classe como entrada para o MASA. Em seguida, o código independente da plataforma deve processar os argumentos, ler as sequências e coordenar a execução. Quando a equação de recorrência de SW/NW precisar ser calculada, o objeto alinhador é chamado. Ele recebe as coordenadas de limite da partição (retângulo correspondente à sub-matriz de programação dinâmica que deve ser calculada), a primeira coluna/linha da partição e calcula a equação de recorrência de SW/NW para a partição. Em seguida, o alinhador usa funções MASA para retornar para a parte independente o maior escore, a última coluna da partição calculada (coluna à direita) e a última linha da partição calculada (linha abaixo).

Capítulo 4

Trabalhos Relacionados

Neste capítulo, serão inicialmente discutidas soluções para comparação de sequências biológicas que usam ambientes paralelos. Em seguida será apresentada uma discussão mais detalhada sobre as soluções que implementam balanceamento de carga dinâmico.

4.1 Comparação de sequências Biológicas em plataformas de alto desempenho

Devido à complexidade quadrática do algoritmo SW (Seção 2.3.2) e suas variantes, o tempo de execução pode ser muito grande. Por essa razão, plataformas de alto desempenho vem sendo utilizadas para a execução desses algoritmos há mais de duas décadas. Nesta seção será apresentado uma categorização de várias destas soluções encontradas na literatura.

4.1.1 Categorização utilizada

Serão apresentadas doze soluções recentes para acelerar o algoritmo de Smith-Waterman utilizando plataformas de alto desempenho. Estas soluções foram categorizadas de acordo com o modelo proposto por [32]. Segundo este modelo, uma solução é dividida em três aspectos: a definição do problema, a descrição do algoritmo usado para resolver o problema e plataforma paralela em que a solução foi implementada.

Na definição do problema, são categorizados o escopo e restrições aplicadas ao problema. Este aspecto é dividido em 5 sub-aspectos: tipos de sequências comparadas, tamanho da menor sequência, tipo de alinhamento, penalidades de *gap* e saída fornecida.

Tendo definido o problema a ser resolvido, o próximo passo é escolher o algoritmo apropriado. Este aspecto é dividido em 4 subpartes: o algoritmo em si, a complexidade da memória da fase de *traceback*, o cálculo geral da matriz onde avalia-se se a matriz de

programação dinâmica é calculada por completo ou foi utilizada alguma técnica de poda, e a estratégia de paralelização. Naturalmente, a definição do problema limitará o tipo de algoritmo usado.

O último aspecto da categorização é a plataforma, que é dividida nos seguintes 3 sub-aspectos: tipo de unidade de processamento, número de unidades de processamento e desempenho. Os dois primeiros sub-aspectos referem-se à infraestrutura, o último subaspecto fornece uma maneira de comparar, em um sentido amplo, o desempenho das soluções.

A Tabela 4.1 apresenta os dados coletados categorizados de acordo com a definição do problema.

Projetos	**Sequencias**	**Menor sequência**	**Alinhamento**	**Modelo de Gap**	**Saida**
ASW [33]	DNA	3.000 K	Local	Affine	Score
Spark-SW [34]	Proteina	4 K	Local	Linear	Score
SWiFOLD [13]	DNA	67.000 K	Local	Affine	Score
SWHybrid [5]	Proteina	5 K	Local	Affine	Score
OSWALD [35]	Proteina	35 K	Local	Affine	Score
DSA [36]	Proteina	4 K	Local	Linear	Score
FPGASW [14]	DNA	8 K	Local	Affine	Score e Alinhamento
[37]	Proteina	6 K	Local	Linear	Score e Alinhamento
[6]	Proteina	1 K	Local	Affine	Score
MASA-CUDAlign [12]	DNA	228.000 K	Local	Affine	Score e Alinhamento
MASA [4]	DNA	47.000 K	Local, Global e Semi-Global	Affine	Score e Alinhamento
[7]	DNA	8.000 K	Local	Affine	Score
MASA-StarPU	DNA	5.000 K	Local	Affine	Score

Tabela 4.1: Dados categorizados de acordo com a definição do problema

Conforme mostrado na Tabela 4.1, seis das soluções analisadas fazem comparações entre sequências pequenas (proteínas). As soluções que comparam sequências de DNA tratam de sequências de 8 KB a 220 MB , enquanto as soluções que comparam proteínas variam de 1 KB até 35 KB. Todas as soluções, com exceção do MASA, que implementa os alinhamentos local, global e semi-global, implementam somente o alinhamento local. Apenas o SparkSW, o DSA e o [37] utilizam o modelo de *gap* linear, enquanto todas as outras utilizam o modelo *affine*. Todas as soluções retornam o escore máximo obtido, porém, apenas o FPGASW, o [37], o MASA-CUDAlign e o MASA retornam, também, o alinhamento. Cabe ressaltar que, em 2014, o CUDAlign foi re-estruturado e seu nome agora é MASA-CUDAlign. Os resultados em [12] são então do MASA-CUDAlign. A ultima linha da Tabela 4.1 apresenta a categorização do MASA-StarPU, que compara sequências de DNA com o algoritmo SW, com *affine gap* e fornece o escore ótimo. O tamanho máximo de sequência comparada foi 5M.

A Tabela 4.2 apresenta os dados categorizados de acordo com a descrição do algoritmo.

Como o objetivo desta desta dissertação é avaliar o impacto das politicas de escalonamento em soluções paralelas que usam o algoritmo Smith-Waterman, a coluna de algoritmo que consta em [32] foi retirada e acrescentada a de escalonamento de tarefas (estático ou dinâmico). Nas soluções que oferecem escalonamento dinâmico, geralmente o balanceamento de carga é considerado.

Projetos	**Memoria (TraceBack)**	**Calculo da matriz**	**Estratégia de paralelismo**	**Escalonamento de tarefas**
ASW [33]	-	Completa	Fine	Estático
Spark-SW [34]	-	Completa	Coarse	Estático
SWiFOLD [13]	-	Completa	Fine	Estático
SWHybrid [5]	-	Completa	Coarse	Dinâmico
OSWALD [35]	-	Completa	Coarse	Estático
DSA [36]	-	Completa	Coarse	Estático
FPGASW [14]	Linear	Completa	Fine	Estático
[37]	Linear	Completa	Coarse	Estático
[6]	-	Completa	Coarse	Dinâmico
MASA-CUDAlign [12]	Linear	Completa	Fine	Estático
MASA [4]	Linear	*block pruning*	Fine	-
[7]	-	Completa	Fine	Estático e Dinâmico
MASA-StarPU	-	*block pruning*	Fine	Dinâmico

Tabela 4.2: Dados categorizados de acordo com o algoritmo

Como apenas o FPGASW, [37], o MASA-CUDAlign e o MASA retornam o alinhamento, logo eles também são os únicos que possuem a fase do *traceback* e todos eles implementam esta parte utilizando complexidade linear de memória. Na literatura, somente o MASA utiliza o *block pruning* com estratégia de poda da matriz de programação dinâmica, todos os outros fazem o cálculo completo. Metade das soluções faz uso da estratégia de paralelismo *Fine* enquanto a outra metade faz da *Coarse*. No paralelismo fine, o cálculo de uma única matriz de programação dinâmica é dividido entre várias *threads* e, no paralelismo *coarse*, cada *thread* calcula uma matriz de programação dinâmica distinta, sem comunicação com as outras *threads*. Apenas o SWHybrid, [6] e [7] implementam o escalonamento dinâmico. Todas as outras, soluções, implementam o escalonamento estático. A última linha apresenta o MASA-StarPU. Como somente o escore ótimo é obtido, não é feito o *traceback*. O cálculo da matriz usa a otimização *block pruning* com paralelismo a grão fino e escalonamento dinâmico de tarefas. Dentre os projetos analisados o MASA-StarPU é o único que usa tanto *block pruning* como escalonamento dinâmico de tarefas.

A Tabela 4.3 apresenta os dados coletados categorizados de acordo com a plataforma.

Projetos	Unidade de processamento	Numero de unidades de processamento	Performance(GCUPs)
ASW [33]	CPU e GPU	1 CPU + 8 GPUs	72,00
Spark-SW [34]	CPU	4 CPU (128 Nucleos)	0,29
SWiFOLD [13]	CPU e FPGA	2 CPUs + FPGAs(ARRIA 10)	125,00
SWHybrid [5]	CPU, GPU e Intel phi	1 CPU (2 nucleos) + (1 GPU(Titan X) + 1 phi(7110) ou 2 phi(7110) ou 2 GPU(Titan X)) ou 1 phi (7210)	1000,00
OSWALD [35]	CPU, FPGA e GPU	2 CPUs(28 Nucleos) + 2 FPGAs(Stratix V) + 1 GPU(Tesla K20c)	442,00
DSA [36]	CPU	8 CPUs (16 nucleos)	N/A
FPGASW [14]	FPGA	1 FPGA (Virtex7)	105,90
[37]	CPU e GPU	1 CPU + 2 GPUs (GTX295, GTX 400)	29,07
[6]	CPU	1 Phi(61 nucleos)/1 Phi(57 nucleos)	730,00
MASA-CUDAlign [12]	GPU ou CPU	384 GPUs (Tesla M2090)	10.370,00
MASA [4]	CPU, GPU ou Intel phi	2 CPUs(12 nucleos) ou 1 GPU (Tesla M2090)	32,10
[7]	CPU	3 CPUs (18 nucleos)	N/A
MASA-StarPU	CPU	2 CPUs (24 nucleos)	18,41

Tabela 4.3: Dados categorizados de acordo com a plataforma

Conforme a Tabela 4.3, cinco das soluções utilizam GPUs como aceleradores, sendo que duas utilizam FPGAs e duas Intel Xeon Phi. As performances variam de 0,29 GCUPS com SparkSW até 10.320 com o CUDAlign. A métrica GCUPs (bilhões de células atualizadas por segundo) é comumente utilizada para comparar soluções sequências de alinhamento de sequências. O GCUPS é calculado multiplicando-se o tamanho da matriz de programação dinâmica (m * n) e dividindo-se esse valor pelo tempo de execução em segundos multiplicado por 10^9. Dentre as soluções analisadas, o MASA-CUDAlign obteve o melhor desempenho (10,37 GCUPS). A última linha da tabela apresenta o MASA-StarPU. Como pode ser visto, dentre soluções analisadas que usam somente CPU ([33], [36] e [7]) o MASA-StarPU obtém o melhor desempenho.

4.2 Escalonamento dinâmico em comparação de sequências biológicas

Nessa seção, são explanadas três soluções que fazem escalonamento dinâmico de tarefas. Cabe ressaltar que muito poucas soluções na literatura utilizam escalonamento dinâmico para a comparação de sequências biológicas (Tabela 4.2).

4.2.1 SWHybrid

O SWHybrid [5] utiliza instruções *Single Instruction, Multiple Data* (SIMD), com vetores que operam sobre uma única instrução. O SWHybrid tem implementações para CPU, GPU e Intel Xeon Phi. Como visto na Tabela 4.3, esta solução utiliza a estratégia *coarse*, ou seja, compara uma sequência chamada de *query* com várias outras sequências para encontrar as mais semelhantes. Não existe dependência de dados entre as diversas comparações.

O framework SWHybrid é baseado no método inter-sequencial [5]. Nesta abordagem, notou-se que o desempenho piora quando sequências de diferentes comprimentos estão

agrupadas. Para eliminar este problema, sequências de igual comprimento são agrupadas dentro de um vetor. Seja Ω o mínimo múltiplo comum do número de vetores SIMD de um dispositivo em uma determinada plataforma. As comparações são divididas em conjuntos e cada conjunto com Ω sequências é chamado de lote global.

Inicialmente, as sequências são divididas em lotes globais. Quando um *Worker* recebe um lote global, este pode ser dividido em lotes locais menores para aumentar o paralelismo. A Figura 4.1 exemplifica o esquema de partição de lote.

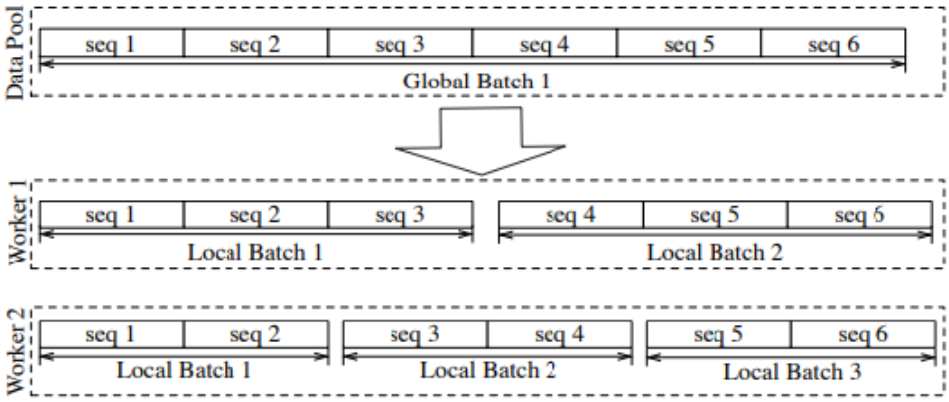


Figura 4.1: Esquema de partição de lote em um sistema com um dispositivo vetorial de três faixas e um dispositivo de duas faixas [5].

O SWHybrid tem como objetivo alcançar extensibilidade em nível de dispositivo com código nativo da plataforma, a fim de usar funções de *kernel* otimizadas. Para isso, uma hierarquia de classes foi proposta para separar as operações gerais das operações específicas da arquitetura. A partir desta hierarquia, o sistema é dividido em três blocos principais: *data pool*, *worker pool* e funções especificas de *kernel*. O *worker pool* associa um *Worker* a cada dispositivo de computação. Ele solicita lotes de dados do *data pool* e cria uma tabela de para acesso rápido aos dados. O objeto *Worker* também controla a transferência de dados entre a memória do *host* e a memória do dispositivo, para que a função do *kernel* possa se concentrar na tarefa de processamento dos lotes recebidos. Cada dispositivo possui um *buffer* de memória que é preenchido pelo o *Worker* toda vez que novos lotes são requisitados.

O *Worker* é projetado para processar a tarefa de comparação para um lote de sequências e retornar os resultados correspondentes. O processamento de cada bloco inclui as seguintes tarefas: (1) Preencher o *buffer*; (2) Transferir de dados entre o *host* e o dispositivo; (3) Processar; (4) Transferir de volta os resultados do processamento.

O escalonamento dinâmico é tratado pelo *data pool*, que está ciente do progresso geral de comparação das sequências presentes no banco de dados. Os *buffers* de dispositivo

são preenchidos com lotes globais pelo *data pool*. Na prática, o tamanho do lote global é pequeno comparado ao tamanho do *buffer*. A solução emprega um esquema de balanceamento dinâmico de carga, onde os *Workers* se revezam. Todos os *Workers* são mantidos ocupados até que o final do processamento do banco de sequências seja atingido. Um caso crítico ocorre quando um *Worker* recebe seu último lote do banco enquanto outros *Workers* já terminaram seu trabalho. Quando isso acontece os outros *Workers* podem ficar ociosos enquanto um *Worker* processa este último lote. Para resolver este problema o *data pool* preenche os últimos blocos de dados com lotes globais menores do que o solicitado, a fim de alcançar um melhor balanceamento de carga. O tempo de atraso de um *Worker* com relação aos demais é calculado utilizando a Equação 4.1.

$$Tempo\_Atraso\_Max = \frac{Tamanho\_seq\_query \times Tamanho\_bloco}{Performance\_em\_GCUPs} \tag{4.1}$$

O tempo de atraso é proporcional ao comprimento da sequência de *query* e o tamanho do lote de dados ao qual o respectivo *Worker* está responsável. A Figura 4.2 apresenta o esquema de balanceamento de carga utilizado pelo SWHybrid.

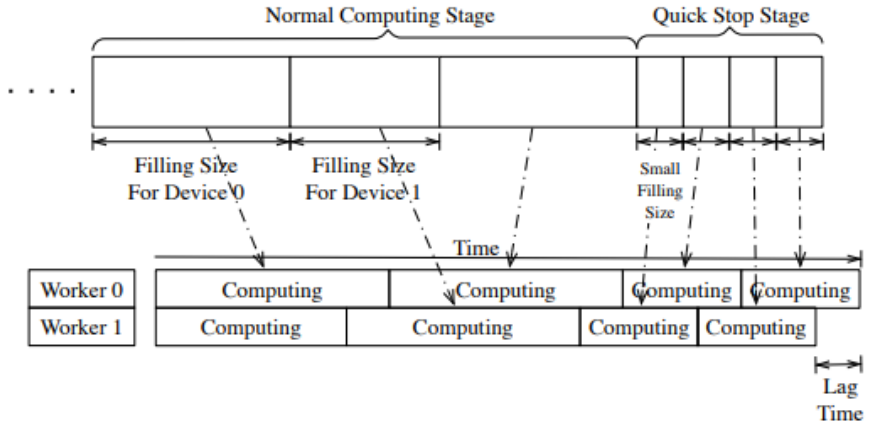


Figura 4.2: Esquema de balanceamento de carga do SWHybrid [5].

4.2.2 *Large-scale sequence comparison on Xeon Phi*

A solução proposta em [6] executa a comparação de uma sequência de proteína (*query*) com um banco de sequências a partir usando unidades SIMD do Xeon Phi. Da mesma maneira que a solução apresentada em Seção 4.2.1, não existe dependência de dados entre as diversas comparações e as comparações são *coarse*. Cada *vector processor unit* (VPU) no Xeon Phi pode executar várias operações de números inteiros em um modo SIMD. Com isso, o processo de comparação do banco de sequências é particionado, em um único nó,

em duas partes: (1) a parte em nível de dispositivo é executada em uma CPU chamada *host*, dividindo o banco de sequências em vários lotes que podem ser distribuídos para aceleradores CPU e Xeon Phi; (2) a parte em nível de *thread* é usada para processar lotes de dados localmente.

A CPU *host* organiza as sequências do banco de dados por ordem de tamanho para então dividi-las em lotes menores. A tarefa de comparação das sequências de um lote com a sequência *query* é feita pelos dispositivos. Esta tarefa pode ser dividida em dois níveis: o nível de *thread* é implementado em dispositivos de CPU e o nível de VPU em dispositivos de Xeon Phi. No nível de *thread*, o processo de comparar a sequência *query* S_i às sequências do lote $S = (S_{(i+1)}, ..., S_n)$ é agrupado em uma tarefa e cada tarefa é processada por uma *thread*. No nível de VPU, comparações *coarse* são realizadas em paralelo nas VPUs. Neste nível, o lote S é empacotado em um *buffer* 2D que possui 16 canais, o que significa que a sequência S_i pode ser alinhada a 16 sequências diferentes no *buffer* de 16 canais em paralelo.

A distribuição de tarefas é feita utilizando o paradigma *master-worker*. O mestre particiona o banco de sequências em lotes em um estágio de pré-processamento e então os envia para os *Workers*. Os *Workers* recebem os lotes do mestre, executam os cálculos de programação dinâmica para o alinhamento local e retornam o resultado para o mestre.

Para fazer fazer a distribuição dos lotes, dois tipos de despachadores foram implementados, um estático e outro dinâmico.

O despachador estático primeiramente divide o banco de dados em vários blocos com relação ao número total de nós no estágio de pré-processamento. Os blocos do banco de dados são então enviados ao nó correspondente para a comparação local. Como o poder computacional de todos os nós de computação pode variar, o tamanho de cada subconjunto de bancos de dados também pode variar. Para obter balanceamento de carga estático entre todos os nós, foi implementado um método de teste de amostra [6]. Neste método, no estágio de pré-processamento, primeiramente é realizado um teste de amostra para explorar o poder computacional de todos os nós. Fatores de desempenho de diferentes nós são então obtidos. Com o fator de desempenho, poder computacional P_i, é possível calcular o tamanho apropriado do subconjunto do banco de sequência alocado para o nó i. A Figura 4.4 ilustra o comportamento do despachador estático.

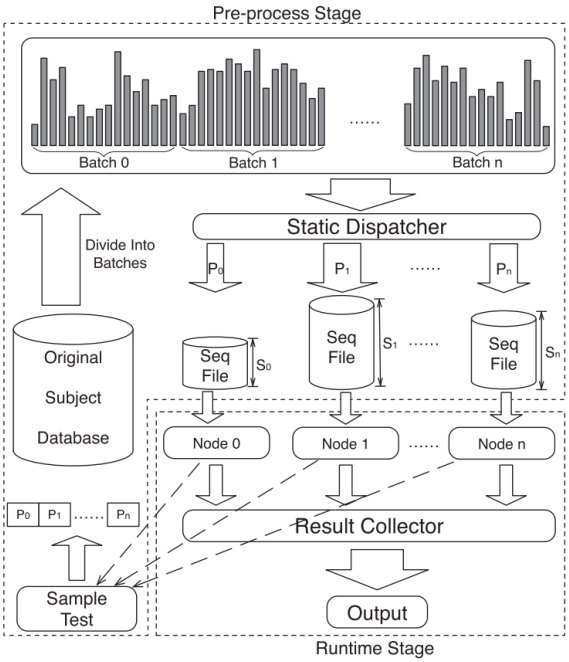


Figura 4.3: Esquema despachador estático [6].

O despachador dinâmico primeiro divide o banco de sequências em um conjunto de tarefas que são organizadas como uma *pool* de tarefas. Em seguida, várias tarefas são enviadas para cada nó. Depois que as tarefas alocadas forem processadas, cada nó enviará uma solicitação de novas tarefas para executar ao despachador. Este procedimento continuará até que todas as tarefas sejam processadas. A Figura 4.4 ilustra o comportamento do despachador dinâmico.

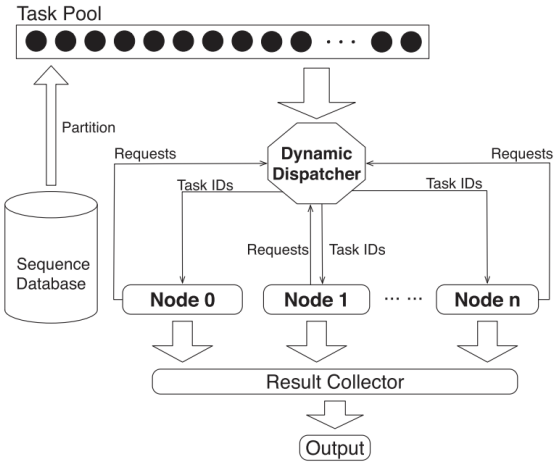


Figura 4.4: Esquema despachador dinâmico [6].

4.2.3 *Adaptive grid*

O *Adaptive grid* [7] é uma das poucas soluções existentes na literatura que utiliza o escalonamento dinâmico de tarefas para sequências biológicas longas de DNA. Este projeto faz uso de duas abordagens, uma com balanceamento de carga estático e outra dinâmico. A seguir, será feita uma breve descrição destas duas abordagens.

Balanceamento Estático

Nesta abordagem, o mapeamento de tarefas possui dois níveis de particionamento. Em primeiro lugar, a matriz de programação dinâmica é dividida em partes, com colunas adjacentes iguais ao número de *clusters*, onde um *cluster* consiste de vários núcleos de processamento. Em seguida, a sub-matriz de dentro de cada *cluster* é particionada. O cálculo da matriz de programação dinâmica é então realizado da mesma maneira como mostrado na Figura 4.5 (quatro processadores), seguindo o padrão *wavefront*.

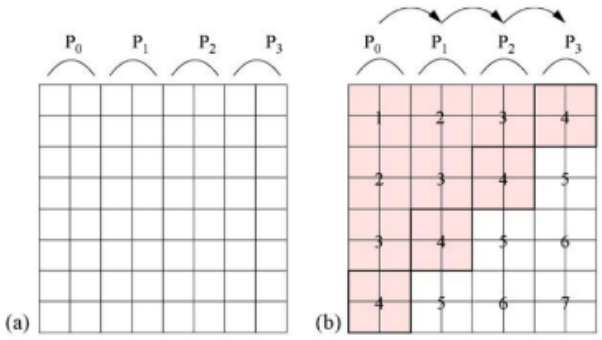


Figura 4.5: Calculo da matriz de programação dinâmica [7].

Na Figura 4.5, (a) ilustra a divisão baseada em colunas de uma matriz 8×8 usando quatro núcleos de processamento e (b) ilustra o cálculo baseado em *basic wavefront* para 4 núcleos de processamento, oito colunas e um tamanho de bloco 2×2. A matriz 8×8 completa pode então ser computada em sete etapas de iteração. Para evitar gargalos na arquitetura, o número de colunas atribuídas a cada *cluster* depende da sua capacidade computacional.

Balanceamento Dinâmico

Para o balanceamento dinâmico de carga, o paradigma *master-slave* foi utilizado. Neste caso, existe um processo mestre que atribui tarefas para os demais processos chamados de escravos. Quando um nó escravo termina sua execução, ele notifica o nó mestre para

receber uma nova tarefa. O mestre então responde com uma nova tarefa e este modelo se repete até que todas as tarefas terminem.

A implementação do alinhamento de sequências longas de DNA no *adaptive grid* com esta abordagem funciona da seguinte forma. A matriz de similaridade é dividida em blocos retangulares. O cálculo de um bloco retangular é atribuído pelo mestre a um escravo disponível. Isto requer que a coluna à esquerda e a linha superior do bloco sejam enviadas para o escravo como entrada. O escravo, então, retorna a coluna à direita e a linha inferior do bloco computado para o mestre. Esse conceito é similar ao conceito de processamento por partições do MASA (Seção 3.5).

Para atingir o balanceamento dinâmico de carga, uma técnica chamada *scheduler–slave*, proposta nesta solução, foi utilizada. Nesta técnica, os escravos relatam seu desempenho computacional para o escalonador toda vez que terminam uma tarefa. O escalonador, em seguida, produz uma nova alocação de tarefas, dependendo do desempenho de cada nó e transmite-a para cada escravo. As novas alocações de tarefas são implementadas pela troca de dados direta entre os escravos. No caso do alinhamento da sequências de DNA, a transferência de dados para reorganizar trabalhos só acontece entre dois processos vizinhos. A Figura 4.6 demonstra uma versão simplificada da técnica *Scheduler-Worker*.

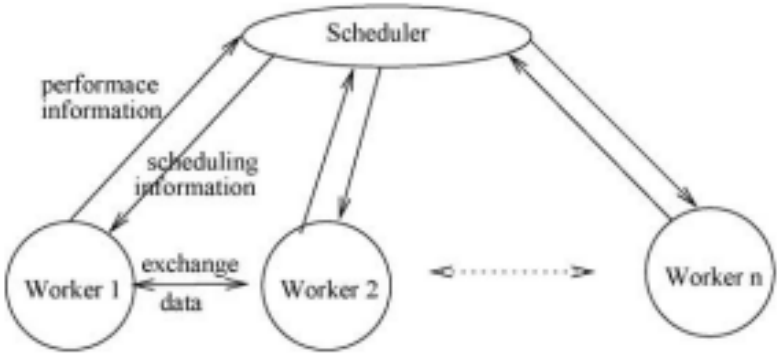


Figura 4.6: Técnica *Scheduler-Worker* [7].

Inicialmente, a matriz de programação dinâmica é particionada da mesma maneira que na abordagem de balanceamento de carga estática. Durante os cálculos, cada escravo reporta seu desempenho para o escalonador. A Equação 4.2 define o desempenho de um determinado no i [7]. Onde NP_i é o desempenho do nó i, SB_i denota o tamanho do bloco atribuído ao trabalhador i e T o tempo para terminar o cálculo deste bloco. Os valores NP descrevem o poder de computação atualmente disponível em um nó. O escalonador julga se uma nova alocação de trabalho é necessária considerando todos os NPs.

$$NP_i = \frac{SB_i}{T} \tag{4.2}$$

4.3 Considerações

Neste capítulo, apresentamos na Seção 4.1 uma categorização de trabalhos que fazem comparação de sequências biológicas em plataformas com mais de um elemento de processamento. Através de três tabelas comparativas, mostramos as características do MASA-StarPU, proposto na presente Dissertação, em relação a 12 trabalhos na literatura. A seguir, na Seção 4.2, apresentamos 3 trabalhos na literatura que fazem escalonamento dinâmico de tarefas. Dois destes trabalhos realizam a comparação *coarse*, executando tarefas sem dependência de dados. O terceiro trabalho, apresentado na Seção 4.2.3, faz a comparação a grão fino com escalonamento dinâmico. A nosso conhecimento, o MASA-StarPU é o primeiro trabalho da literatura que faz a poda da matriz de programação dinâmica em conjunto com escalonamento dinâmico de tarefas.

Capítulo 5

Projeto do MASA-StarPU

O MASA (Seção 3.5) atualmente possui resultados expressivos na área de comparação de sequencias biológicas [4], principalmente em GPUs. Em máquinas com somente CPUs, por exemplo, consideramos que devem ser desenvolvidas novas estratégias para obter resultados ainda melhores.

Uma das grandes virtudes do MASA é a sua estratégia de poda chamada *block pruning* (Seção 2.4.2) que possui grande impacto na redução do tempo para obter o alinhamento. Em sua versão mais recente, o MASA não utiliza o *block pruning* em execuções com mais de um dispositivo pois essa otimização faz a poda da matriz em tempo de execução, introduzindo um problema de balanceamento de carga. Aditivamente, como observado no Capítulo 4, poucas soluções vêm sendo propostas para tratar o problema da comparação de sequências utilizando escalonamento dinâmico de tarefas. A maioria destas (Seções 4.2.1, e 4.2.2), implementam a comparação de sequências pequenas.

O objetivo da presente Dissertação de Mestrado é investigar o comportamento de diversas estratégias de escalonamento dinâmico de tarefas adequadas ao *block pruning* (Seção 2.4.2). O StarPU (Seção 3.4) é um ambiente de programação paralela que possui suporte à dependência de tarefas e agrega implementações de diversas estratégias para escalonar tarefas de forma dinâmica. Sendo assim, na presente Dissertação de Mestrado, foi criada uma nova extensão do *framework* MASA que incorpora as diferentes estratégias da plataforma StarPU para lidar com o problemas de balanceamento de carga, criando assim a extensão MASA-StarPU.

Dentre as ferramentas avaliadas no Capítulo 3, o ambiente de programação paralela OmpSs (Seção 3.3) é o que mais se aproxima do StarPU. Tanto o OmpSs quanto o StarPU possuem suporte a dependência de tarefas e implementam estas dependências em forma de grafos. Além disso, ambas as plataformas possuem suporte a prioridades entre as tarefas e os dois ambientes possuem suporte para implementações em CPU. Devido a estas similaridades a extensão MASA-StarPU foi criada a partir da análise da extensão

MASA-OmpSs.

A arquitetura MASA (Seção 3.5) foi projetada requerendo que apenas o de processamento da matriz de similaridade baseado em variantes do algoritmo Smith-Waterman (Seção 2.3.2) seja implementado. Desta forma, a extensão MASA-StarPU foi criada sem que a parte independente da plataforma fosse modificada. Por consequência, as otimizações já existentes na arquitetura, como *block pruning*, também são utilizadas na extensão MASA-StarPU.

5.1 Análise do MASA-OmpSs

O código do MASA-OmpSs divide o processo de comparação de sequências (Capítulo 2) em cinco estágios, sendo primeiro estágio a obtenção do escore máximo e o quinto estágio o *traceback*. Para que o código original, desenvolvido na plataforma OmpSs, do MASA-OmpSs pudesse ser adaptado na geração da versão MASA-StarPU o passo inicial foi a análise das funções existentes relacionadas ao processamento dos estágios além da avaliação geral do código e classes existentes. Em paralelo, foi necessário identificar todas as chamadas a funções providas pela plataforma OmpSs, pois elas tiveram que ser substituídas e/ou adaptadas para a plataforma StarPU.

O escopo definido para o MASA-StarPU foi a implementação do primeiro estágio do MASA-OmpSs, que calcula o escore ótimo, retornando seu valor e posição na matriz programação dinâmica. Este estágio é o que possui maior impacto de desempenho na comparação de sequências biológicas, sendo também o que permite maior paralelização. A maioria dos trabalhos avaliados na bibliografia (Capítulo 4) produz o escore como resultado final, ratificando a relevância deste estágio.

Visando adequar o MASA-OmpSs ao escopo delimitado, uma versão simplificada da solução foi obtida. Neste código otimizado, apenas o primeiro estágio do programa é executado, inibindo a chamada aos procedimentos do segundo estágio, e consequentemente aos demais estágios. Além disso, todas as otimizações feitas para os demais estágios também foram desabilitadas, visto que não são necessárias para realização do cálculo do escore ótimo. Com exceção destas mudanças, todas as demais otimizações existentes no MASA-OmpSs persistem nesta versão simplificada, incluindo a otimização de poda de blocos (*Block Pruning*).

Após avaliação do código, observou-se que o *framework* MASA e sua extensão OmpSs foram desenvolvidos na linguagem C++, baseando-se nos conceitos de orientação a objeto. O código é bastante modularizado e dividido em duas partes: (a) independente da plataforma e (b) dependente da plataforma (Seção 3.5). Foram identificadas 34 classes

independentes da plataforma envolvidas no processamento do primeiro estágio. Algumas das classes principais disponíveis e suas funcionalidades podem ser vistas na Tabela 5.1.

Classe	Funcionalidade
AbstractAligner	Classe abstrata que realiza o procedimento de alinhamento
BlockAlignerParameters	Classe que contém os parâmetros utilizados no alinhamento
Configs	Classe com funcionalidades de configuração da aplicação
AbstractBlockPruning	Classe que implementa o procedimento de poda de blocos
job	Classe que gerencia os estágios e status do trabalho de alinhamento

Tabela 5.1: Principais classes plataforma-independentes que executam o primeiro estágio do MASA

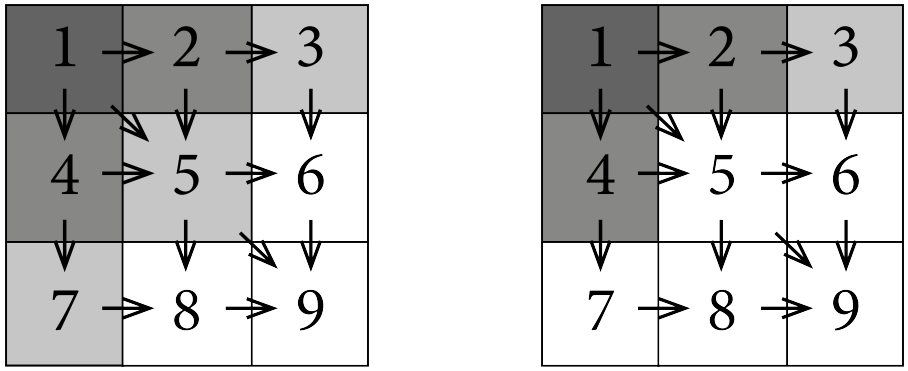
A parte dependente da plataforma possui apenas duas classes: (1) `OmpSsAligner`, que é uma classe filha da classe `AbstractBlockAligner` que por sua vez é filha de `AbstractAligner` (Tabela 5.1). Enquanto os métodos da classe-pai se preocupam em realizar o alinhamento, esta classe tem o objetivo de criar as tarefas a serem executadas em paralelo utilizando a plataforma OmpSs; (2) `OmpSsalinerParameters`, que é uma classe filha de `BlockAlignerParameters` (Tabela 5.1). Nesta classe estão presentes os parâmetros de blocos como: número de blocos na coluna da matriz, número de blocos na linha da matriz, o tamanho da altura do bloco, em número de células, e o tamanho da largura do bloco. Se a plataforma não necessita de nenhum parâmetro específico, como é o caso do OmpSs, os parâmetros são definidos pela classe-pai.

Enquanto a classe `AbstractBlockAligner` é responsável por realizar o alinhamento de um bloco, a classe `job` está ciente do progresso e do status de execução do programa como um todo. Esta classe também é responsável por dividir a matriz em partições de blocos de acordo com os parâmetros definidos pela classe `BlockAlignerParameters` (Tabela 5.1). Uma vez definidos os blocos, estes são entregues para a classe `OmpSsAligner` para que sejam criadas as tarefas para alinhar estes blocos em execução paralela. Para cada bloco é criada uma tarefa. Como descrito na Seção 2.3, não é possível calcular o alinhamento de uma partição de blocos sem que seus adjacentes acima e a esquerda tenham sido calculados. Logo existe uma dependência de dados entre os blocos.

A plataforma OmpSs utiliza a diretiva `#pragma` para criar as tarefas e definir suas dependências (Seção 3.3). No MASA-OmpSs, as dependências entre as tarefas são criadas considerando a linha e a coluna finais dos blocos anteriores como `input` seguindo o modelo *dataflow*. Na Figura 5.1 a matriz (a) ilustra o modelo de execução *wavefront* e a matriz (b) apresenta o modelo de execução *dataflow*.

O modelo *wavefront* impõe que cada anti-diagonal seja processada em paralelo e, além disso, que o processamento da anti-diagonal i seja concluído antes que o processamento

da anti-diagonal $i + 1$ seja iniciado. O modelo *dataflow* não impõe restrições além das dependências de dados que existem na equação de recorrência, ou seja, $H_{i,j}$ depende de $H_{i-1,j}$, $H_{i-1,j-1}$,$H_{i,j-1}$. Com isso, mais paralelismo pode ser explorado.



(a) *Wavefront* (b) *Dataflow*

Figura 5.1: Modelos de execução *wavefront* e *dataflow*

Na Figura 5.1 as setas representam as dependências; os quadrados, os blocos; e os números são identificadores dos blocos. Neste exemplo os blocos 1, 2 e 4 são dependentes do bloco 1 e o bloco 5 depende dos blocos 4 e 2. No modelo *Wavefront* o bloco 1 executaria sozinho, em seguida o 2 e o 4 em paralelo, depois o 3, 5 e 7 em paralelo e assim por diante. Porém, pela maneira que as dependências são implementadas na plataforma OmpSs, caso o bloco 2 termine de ser executado antes do bloco 4, o bloco 3 poderá ser executado. Esta variação na ordem de execução dos blocos e segue o modelo *Dataflow*.

Idealmente, todas as tarefas devem ser criadas no OmpSs no início da execução do programa. No entanto, devido a problemas relacionados à capacidade da memória, o OmpSs possui um mecanismo chamado *hysteresis throttle* que limita o número de tarefas no grafo de tarefas, reduzindo a quantidade de memória usada pela plataforma [4]. A Figura 5.2 apresenta em (a) a definição das prioridades das tarefas e em (b) a ordem de criação das tarefas segundo o padrão *square* com *hysteresis throttle*.

Ao se estudar o *block pruning* em detalhe [38] percebeu-se que o processamento segundo o padrão *square* (Figura 5.2) resultava em uma área de poda maior do que o padrão *wavefront*. Por isso, o MASA-OmpSs empregou este padrão de processamento dentro do modelo *dataflow*.

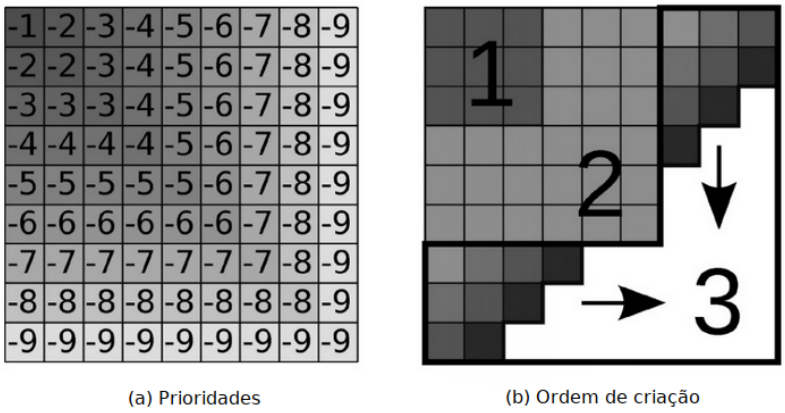


Figura 5.2: Prioridades e ordem de criação da tarefas do Masa-OmpSs.

Com o mecanismo *hysteresis throttle*, quando o número de tarefas atinge um limite, a criação de tarefas é interrompida até cair abaixo de um determinado valor. Para evitar a perda de paralelismo durante a fase suspensa da *hysteresis*, a ordem de criação das tarefas foi alterada para seguir prioridades. Na plataforma OmpSs, pode-se definir uma prioridade para uma tarefa através de um parâmetro chamado `priority`. Quanto maior o valor de `priority` maior a prioridade da tarefa. O MASA-OmpSs utiliza então a prioridade do OmpSs para criar uma ordem de execução preferível em ondas quadradas, pois esse padrão aumenta a taxa de poda. Considerando o bloco `(bx, by)`, sua prioridade é `min (-bx, -by)`(Figura 5.2 (a)). Assim, as tarefas são criadas nas faixas (Figura 5.2 (b)). A largura das faixas é o número de *threads*, que é igual ao número de núcleos no ambiente em execução.

Outro aspecto relevante do MASA-OmpSs é o nível de parametrização da aplicação, tanto em constantes existentes no código fonte como em argumentos informados na linha de comando de sua execução, que permitem que sejam escolhidos alguns parâmetros para o processamento das sequências. Todas as opções de compilação e execução que se aplicam ao primeiro estágio estão também disponíveis no MASA-StarPU.

A análise detalhada do código e das funções indicou pelo menos 4 métodos ou procedimentos utilizados pela plataforma OmpSs. Estes procedimentos tiveram que ser substituidos para que pudessem ser executados na plataforma StarPU. O StarPU utiliza a linguagem C, por isso novos métodos e estruturas foram criados para adaptar o modelo de execução. Ademais, todas as classes e métodos com prefixo OmpSs no nome foram substituídos para utilizar o prefixo StarPU.

Adicionalmente, as diretivas de criação de tarefas da plataforma OmpSs precisaram ser adaptadas para seguir a implementação da plataforma StarPU. A forma como as duas

plataformas trabalham para criar as tarefas e suas dependências é muito diferente. Enquanto o OmpSs cria as dependências a partir de valores lógicos de entrada e saída (`input` e `output`)(Seção 3.3), o StarPU utiliza um sistema de `TAGs`(Seção 3.4) em que cada tarefa possui uma `TAG` identificadora e um ponteiro para uma ou mais `TAGs` de outras tarefas indicando as dependências. Por isso, o método `alignBlock` da classe `StarPuAligner`, responsável pela criação das tarefas, precisou ser completamente alterado. Este método foi reconstruído seguindo o mesmo modelo de execução (*dataflow*), o mesmo modelo de prioridades e segue a mesma ordem de criação das tarefas.

5.2 Arquitetura da Solução MASA-StarPU

Após a conclusão do mapeamento de todo o código do MASA-OmpSs, foi possível identificar o escopo do projeto do MASA-StarPU e sua interface com o *framework* MASA, propiciando a definição da arquitetura da solução. A opção pelo desenvolvimento da aplicação como uma extensão do MASA mostrou-se a alternativa mais adequada, uma vez que a utilização da arquitetura de classes existente simplificaria a implementação, objetivando avaliar o desempenho com diferentes estratégias para alocação de tarefas.

Foi então, desenvolvida uma nova implementação do *framework* MASA utilizando a plataforma StarPU, chamada de MASA-StarPU. O StarPU foi utilizado na implementação da estratégia de paralelização e do algoritmo de Smith-Waterman (Seção 2.3.2). Considerando a modelagem da arquitetura, esta extensão foi adicionada à Figura 3.3 ilustrada anteriormente, como pode ser visto na Figura 5.3.

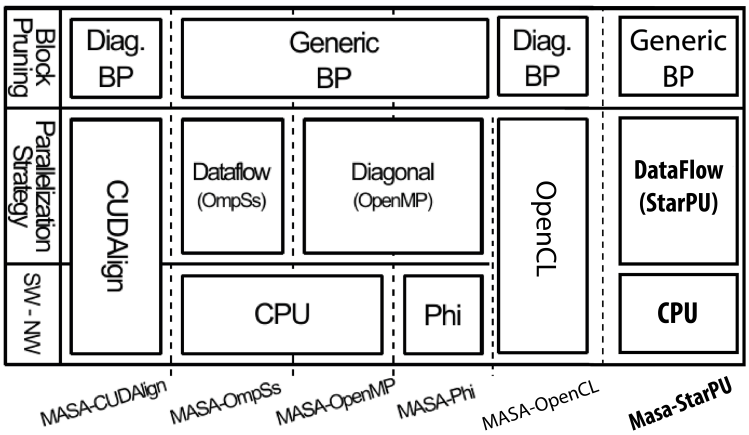


Figura 5.3: Arquitetura MASA com a extensão MASA-StarPU.

Como apresentado na Figura 5.3 a nova extensão da arquitetura MASA, MASA-StarPU, foi desenvolvida apenas em CPU além de utilizar a mesma estratégia de paralelização, *Dataflow* (Seção 5.1), da extensão MASA-OmpSs e utiliza a estratégia padrão de poda de blocos (*block pruning*).

5.3 Desenvolvimento da Solução MASA-StarPU

Um dos objetivos deste trabalho é avaliar o desempenho da solução MASA-StarPU com diferentes políticas de alocação dinâmica de tarefas adequadas ao *block pruning*, avaliando-se o tempo que a aplicação leva para realizar a comparação e a quantidade de células da matriz de programação dinâmica atualizadas em um segundo (GCUPS). Para tanto, optou-se pelo desenvolvimento de código utilizando a plataforma StarPU que permitisse a alternância entre as diferentes estratégias de alocação dinâmica de tarefas de modo simples.

Um dos pré-requisitos para o desenvolvimento do MASA-StarPU foi a escolha da versão da plataforma StarPU a ser utilizada. A última versão disponibilizada é o StarPU 1.3 [39]. A nova versão possui implementações de mais políticas de alocação dinâmica de tarefas em relação à versões anteriores (1.1 e 1.2). Contudo, o sistema operacional utilizado nos ambientes de teste foi o Ubuntu [40], uma distribuição Linux livre e de código aberto baseada em Debian [41]. O StarPU 1.3 necessita da versão 18 ou mais atual do Ubuntu, no entanto, nos ambientes de teste disponíveis estavam instaladas apenas as versões 14 e 16. Devido a esta limitação a versão 1.3 do StarPU não pode ser utilizada e optamos por usar o StarPU 1.2.

O próximo passo no desenvolvimento foi o mapeamento inicial das principais modificações a serem realizadas no pseudo-código do MASA-OmpSs para produzir a solução de alinhamento MASA-StarPU. Estas linhas estão marcadas com asterisco no Algoritmo 1.

O método `InitializeStructures()` teve que ser modificado para inicializar as estruturas do StarPU (Seção 3.4). Nele, são inicializados o *codelet* utilizado, o modelo de performance e a própria plataforma pela chamada `starpu_init(NULL)`. O método `StarPuAligner::scheduleBlocks(grid_width, grid_height)` (linha 5) é responsável por definir a ordem de criação das tarefas e os parâmetros *grid_width* e *grid_height* representam a largura e a altura, em número de células, da matriz de programação dinâmica, respectivamente. O método `createTasks()` (linha 6) possui o *loop* de criação das tarefas no formato *square* (Seção 5.1). A cada iteração é feita uma chamada ao método `AbstractBlockAligner::alignBlock(i, j)`. Este método é responsável por criar uma tarefa para alinhar um bloco, por isso, precisou ser completamente alterado para utilizar as funções do StarPU. Os parâmetros i e j representam a posição do bloco

Algoritmo 1 Implementação MASA-StarPU - funções gerais

```
1: procedure StarPuAligner::alignBlock
2:     initializeStructures()***
3: end procedure
4:
5: procedure StarPuAligner::scheduleBlocks(grid_width, grid_height)
6:     createTasks(i, j)
7:     starpu_task_wait_for_all()***
8:     starpu_shutdown()***
9: end procedure
10:
11: procedure main(args)
12:     MASA::EntryPoint(args , new StarPuAliner())
13: end procedure
```

na matriz. A chamada `starpu_task_wait_for_all()` (linha 8) foi adicionada para que a plataforma espere todas as tarefas terminem antes de prosseguir e a chamada `starpu_shutdown()` foi adicionada para terminar finalizar a plataforma e atualizar alguns parâmetros controlados pelo StarPU como o modelo de performance. A função `MASA::EntryPoint(args,new StarPuAliner())` (linha 12) indica o início do programa e recebe o parâmetro *args* por argumentos de linha. Neste argumentos são passados os caminhos para os arquivos em que as sequências a serem comparadas se encontram. A chamada `MASA::EntryPoint` foi herdada da parte plataforma-independente da biblioteca MASA.

O Algoritmo 2 apresenta o pseudo código com mais detalhes da função de inicialização.

Algoritmo 2 Implementação MASA-StarPU - função de inicialização

```
1: procedure initializeStructures
2:     starpu_codelet_init(cl)
3:     cl.cpu_func := StarPuAligner::callBack
4:
5:     cl.model := masa_perf_model
6:     masa_perf_model.type := STARPU_HISTORY_BASED
7:     masa_perf_model.symbol := "masa_perf_model"
8:
9:     starpu_init(NULL);
10: end procedure
```

No Algoritmo 2, a linha 2 apresenta a inicialização de um `struct starpu_codelet cl` (Seção 3.4). A linha 3 indica qual função o StarPU, utilizando CPU, deve executar caso um tarefa seja submetida a este *codelet*. Como o foco desta dissertação é execução apenas em CPU, apenas a função de CPU foi definida. O StarPU utiliza a linguagem C, por

isso, o parâmetro `cpu_func` do *codelet* espera uma função C. No entanto, o *framework* MASA foi escrito utilizando a linguagem C++. Na linguagem C++ um método estático é considerado uma função C. Por isso, foi atribuído ao parâmetro `cpu_func` um método estático `StarPuAligner::callBack` cuja a única função é chamar um método plataforma-independente chamado `AbstractBlockAligner::processBlock(bx, by)`, que por sua vez é responsável por realizar o alinhamento de um bloco.

A linha 5 inicializa o parâmetro `model` que indica qual o modelo de performance `struct starpu_perfmodel masa_perf_model` sera usado pelo *codelet*. As linha 6 indica como este modelo deve ser atualizado. No caso, a constante `STARPU_HISTORY_BASED` especifica que este modelo deve ser atualizado utilizando o histórico de execuções. A linha 7 indica qual o nome do arquivo que descreve o modelo de performance, caso este modelo não exista, ele será criado após a primeira chamada do `starpu_shutdown()`, com as informações desta última execução. Por fim, a linha 9 inicializa a plataforma StarPU.

O Algoritmo 3 apresenta o pseudo código com mais detalhes do método responsável por criar as tarefas (`StarPuAligner::alignBlock(bx, by)`).

Algoritmo 3 Implementação MASA-StarPU - método de criação de tarefas

```
 1: procedure StarPuAligner::alignBlock(bx, by, i0, j0, i1, j1)
 2:     priority = min(−i0, −j0);
 3:     if not isBlockPruned(bx, by) then
 4:
 5:         params.bx := bx
 6:         params.by := by
 7:         params.i0 := i0
 8:         params.j0 := j0
 9:         params.i1 := i1
10:         params.j1 : = j1
11:         params.func := this
12:
13:         task = starpu_task_create()
14:         task.cl = cl;
15:         task.cl_arg = params
16:         task.use_tag = 1
17:         task.tag_id = TAG(bx, by)
18:         task.priority = priority
19:
20:         express_deps(bx,by)
21:         starpu_task_submit(task)
22:     end if
23: end procedure
24:
```

No Algoritmo 3 as variáveis $i0$, $i1$, $j0$ e $j1$ são parâmetros que definem o tamanho do bloco e correspondem a posição inicial da primeira linha do bloco, posição final da última linha do bloco, posição da primeira coluna do bloco e a posição da última coluna do bloco respectivamente. Os parâmetros *bx* e *by* são as coordenadas do bloco na matriz. A rotina `TAG` retorna uma *string* em hexadecimal a partir de dois números.

A linha 2 define a prioridade da tarefa da mesma maneira descrita na Seção 5.1. Caso o bloco em questão não seja *prunable* (Seção 2.4.2) uma tarefa é criada para realizar o alinhamento deste bloco (linhas 3 a 23). Inicialmente, os parâmetros a serem passados para o *codelet* (Seção 3.4) são inicializados com os dados do bloco (linhas 5 a 10). O parâmetro `func` (linha 11) guarda uma referência do objeto atual. Todos estes parâmetros são passados pelo *codelet* à função de *callback* para que se possa restaurar o contexto. Em seguida, a tarefa é criada e seus parâmetros são atualizados (linhas 13 a 18). Nas linhas 16 e 17 os parâmetros de configuram o uso de *tags* que são utilizados para definir as dependência entre as tarefas (Seção 3.4). Em seguida o método `express_deps(bx,by)`(linha 21) é chamado para definir as dependências. Por fim, a tarefa é submetida para execução (linha 21).

O Algoritmo 4 apresenta o pseudo código com mais detalhes do método que define as dependências entre as tarefas `express_deps(bx,by)`.

Algoritmo 4 Implementação MASA-StarPU - método de definição das dependências

```
1: procedure EXPRESS_DEPS(i,j)
2:     if i = 0 and j > 0 then
3:         starpu_tag_declare_deps(TAG(i,j), 1, TAG(i,j-1))
4:     else if i > 0 and j == 0 then
5:         starpu_tag_declare_deps(TAG(i,j), 1, TAG(i-1,j))
6:     else if i > 0 and j > 0 then
7:         starpu_tag_declare_deps(TAG(i,j), 2, TAG(i-1,j), TAG(i,j-1))
8:     end if
9: end procedure
```

No Algoritmo 4 o método `express_deps` cria as dependências das tarefas seguindo o modelo *wavefront* (Seção 5.1). A chamada `starpu_tag_declare_deps` cria as dependências de uma tarefa a partir de *tags*. O primeiro parâmetro é a *tag* da tarefa para qual estão sendo definidas as dependências; o segundo parâmetro determina quantas dependências esta tarefa possui e o terceiro parâmetro contem as *tags* das tarefas das quais esta tarefa depende.

O diagrama de fluxo ilustrado na Figura 5.4 contem o fluxo de execução do MASA-StarPU, que engloba os algoritmos de 1 a 4 e mostra claramente quais atividades são responsabilidades do MASA e quais são do StarPU.

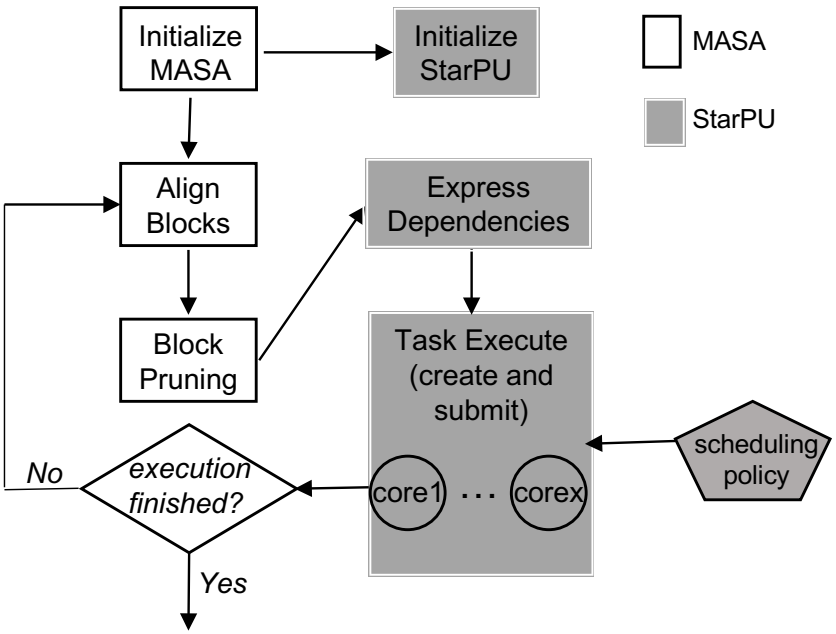


Figura 5.4: Em branco estão as atividades do MASA e em cinza do StarPU.

5.4 Resumo do Projeto do MASA-StarPU

Baseando-se nas decisões de projeto e desenvolvimento discutidas, algumas características principais podem ser identificadas na solução MASA-StarPU. Inicialmente, cabe destacar que a solução foi projetada como uma extensão da arquitetura MASA, utilizando como base o ambiente de programação paralela StarPU. O objetivo foi desenvolver uma ferramenta com diversas políticas de escalonamento dinâmico de tarefas que fossem apropriadas a um ambiente de comparação de sequências biológicas com poda. Foi utilizada na implementação a versão e 1.2.9 do StarPU, e a biblioteca de funções existentes nestas versões em linguagem C. Adicionalmente, cabe destacar que o foco da aplicação é o cálculo do escore ótimo entre duas sequências longas de DNA, provendo também suas coordenadas na matriz de programação dinâmica. A solução possui boa flexibilidade, permitindo que sejam escolhidos parâmetros (quantidade de blocos ou quantidade de *threads*) a serem informados na linha de comando de execução do programa, que alteram a forma de processamento da matriz de programação dinâmica.

Finalmente, cabe destacar que a forma que a solução foi projetada de maneira a permitir que ela possa ser aprimorada em versões futuras com reuso das funções implementadas, como, por exemplo, para possibilitar o desenvolvimento dos demais estágios necessários para prover o alinhamento ótimo, ou a ampliação para execução em ambientes heterogêneos.

Capítulo 6

Resultados Experimentais

Neste capítulo estão descritos os testes realizados para avaliar o desempenho do MASA-StarPU bem como de suas diferentes estratégias de alocação dinâmica de tarefas, ressaltando as condições de execução e resultados obtidos. Na Seção 6.1, os ambientes e sequências de DNA utilizados nos testes são detalhados, na Seção 6.2, são apresentados e discutidos os tempos de execução, GCUPS e taxas de poda. A Seção 6.3 apresenta a comparação com o MASA-OmpSs e o MASA-OpenMP.

6.1 Plataformas e sequências utilizadas nos testes

Nos nossos testes, foram utilizadas duas plataformas de execução: (a) PLaFRIM (Federative Research Platform in Computer Science and Mathematics), uma plataforma hospedada no INRIA / França, composta por vários ambientes de computação paralela. Neste trabalho, foi usado um multicore (24 núcleos) na máquina Miriel (2 x 12 núcleos Haswell Intel® Xeon® E5-2680v3), 2,5 GHz, 64 GB RAM, com Linux CentOS versão 7.1.1503, compilador gcc 8.4 e StarPU versão 1.2.9; e (b) um notebook Acer (4 núcleos) com Intel i7-7700HQ, 2,8 GHz, 16 GB RAM, com Linux Ubuntu 16.04, compilador gcc 8.4 e StarPU versão 1.2.9.

Sequências de diferentes tamanhos foram selecionadas para avaliar as execuções do MASA-StarPU, variando de milhares de pares de bases (KBP) a milhões de pares de bases (MBP), refletindo os mesmos casos de testes utilizados em [4]. As sequências foram obtidas no site do National Center for Biotechnology Information (NCBI) [42], distribuídas em pares de comparações e avaliadas em sua similaridade através do escore obtido no alinhamento ótimo. A Tabela 6.1 resume as informações relativas às sequências usadas nos testes de execução, onde as colunas “Id de Acesso” contém os identificadores de acesso das sequências no NCBI, as colunas “tamanho” apresentam o tamanho em K (Kilobytes) ou M (Megabytes) das sequências e coluna “Escore” representa o valor do escore

ótimo do alinhamento local entre as sequências, baseando-se nos parâmetros utilizados no processamento da matriz de programação dinâmica.

Comparação	**Sequência 1**		**Sequência 2**		**Escore**
	Id de Acesso	**Tamanho**	**Id de Acesso**	**Tamanho**	
10K	AF133821.1	10K	AY352275.1	10K	5091
50K	NC_001715.1	57K	AF494279.1	57K	52
150K	NC_000898.1	162K	NC_007605	172K	18
500K	NC_003064.2	543K	NC_000914.1	536K	48
1M	CP000051.1	1M	AE002160.2	1M	88353
3M	BA000035.2	3M	BX927147.1	3M	4226
5M	AE016879.1	5M	AE017225.1	5M	5220960

Tabela 6.1: Sequências selecionadas para testes

Os valores dos parâmetros utilizados para obtenção dos alinhamentos foram os mesmos adotados na solução MASA-OmpSs: *match*: +1; *mismatch*: 3; primeiro *gap*: 5; extensão de *gap*: 2 e o tamanho do bloco foi definido empiricamente como 2048 x 2048.

Conforme explicado na Seção 5.1, apenas o primeiro estágio do alinhamento de sequências (que retorna o escore ótimo obtido) foi inicialmente implementado no MASA-StarPU, portanto apenas os tempos de execução relativos à inicialização do programa e execução efetiva do estágio 1 foram considerados na análise. As sete políticas explicadas na Seção 3.4 foram usadas nos testes: *eager*, *prio*, *random*, *ws*, *lws*, *dmda*, *dmdas*.

6.2 GCUPS, Tempos de Execução e Taxas de Poda

Para efeito de comparação, o desempenho de aplicações é normalmente expresso em unidades de tempo de execução (segundos ou milissegundos). Entretanto, devido às características do processamento da matriz de programação dinâmica e o grande volume de dados, a métrica mais usada para medição de desempenho para aplicações de comparação de sequências biológicas é expressa em milhões de células atualizadas por segundo (*MCUPS - Mega Cells Updated per Second*) ou em bilhões de células atualizadas por segundo (*GCUPS - Giga Cells Updated per Second*). A métrica CUPS é obtida dividindo-se a quantidade de células da matriz (resultante do produto entre os tamanhos das sequências comparadas) pelo tempo de execução total em segundos.

Tomando-se por base duas sequências de tamanhos m e n, e sendo t o tempo medido em segundos, o desempenho em CUPS é obtido pela Equação 6.1. Para o cálculo do GCUPS, o tempo t na Equação 6.1 é multiplicado por 10^9.

$$CUPS = \frac{m \times n}{t} \tag{6.1}$$

As Tabelas 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 e 6.8 apresentam os resultados de percentual de poda, GCUPS e tempo de execução para as sete comparações apresentadas na Seção 6.1, as 7 estratégias de alocação de tarefas e nos dois ambientes de teste. Os Gráficos 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, apresentam os GCUPS para cada estratégia de alocação de tarefas nos dois ambientes de testes.

6.2.1 Comparação de 10k

A Tabela 6.2 e o Gráfico 6.1 apresentam os resultados obtidos para a comparação de 10k nas duas plataformas.

	10k					
	Acer Notebook			**PLaFRIM**		
	Poda (%)	GCUPS	TEMPO(s)	Poda (%)	**GCUPS**	**TEMPO(s)**
dmda	18.05	1.14	0.08	40.31	1.26	0.08
dmdas	15.27	1.07	0.09	33.47	0.51	0.19
eager	18.05	0.91	0.11	41.90	1.42	0.07
lws	15.27	1.14	0.08	40.54	0.86	0.11
prio	**20.83**	**1.15**	**0.08**	**43.71**	**1.65**	**0.06**
random	13.88	0.78	0.13	38.74	1.60	0.06
ws	15.27	1.15	0.08	41.56	0.51	0.19

Tabela 6.2: Percentual de poda, GCUPS e tempo de execução do MASA-StarPU com 7 estrategias para a comparação de 10k.

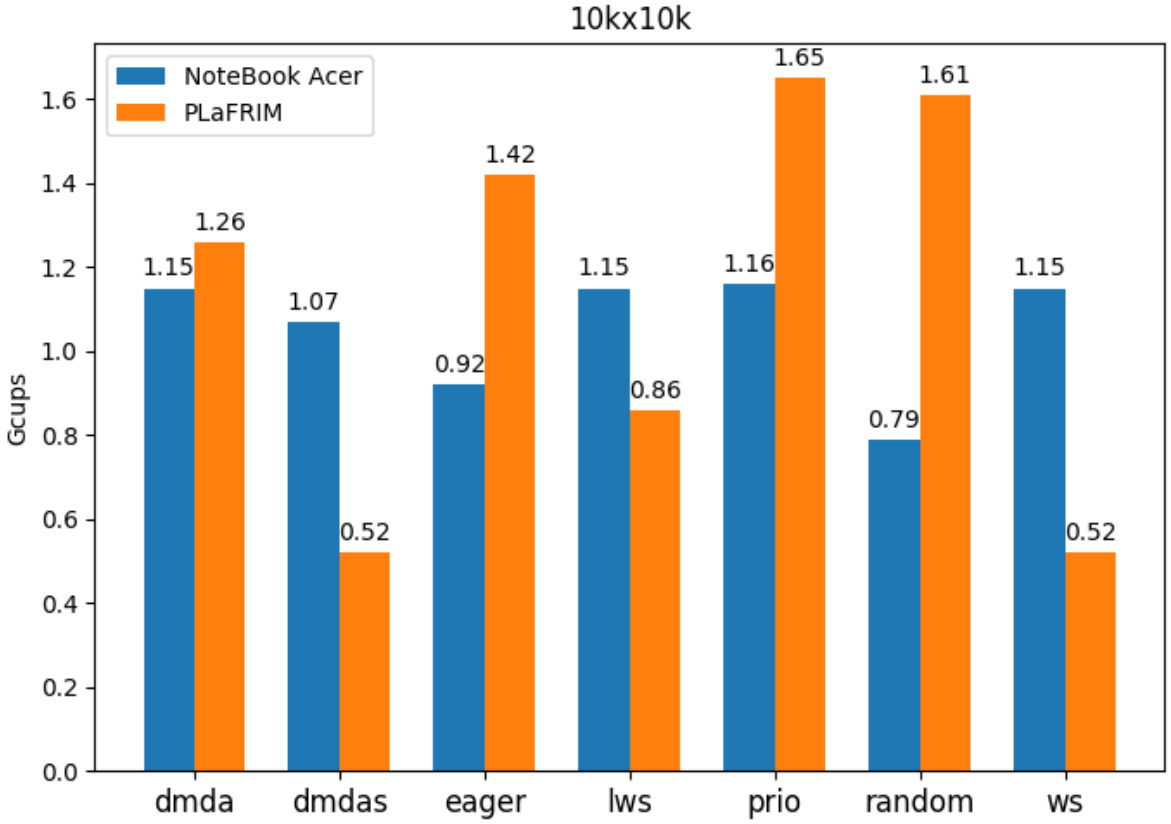


Figura 6.1: GCUPS da comparação de 10K para diferentes políticas de escalonamento

Conforme os resultados obtidos para a comparação de 10k, o percentual de poda apresenta uma variação dependendo da estratégia de alocação de tarefas nos dois ambientes de testes, com o maior percentual (20.83) obtido pela estratégia *prio*, e o menor (13.88) pela a *random* no Acer Notebook. No PLaFRIM o maior percentual de poda (43.71) também foi da estratégia *prio*, porém o menor (33.47) foi da estratégia *dmdas*. A estratégia com o melhor valor de GCUPS e tempo de execução foi a *prio* (1.15 e 0.08s) e a pior (0.78 e 0.13s) foi a *random* no Acer Notebook. Na plataforma PLaFRIM os melhores GCUPS e taxa de poda (1.65 e 0.06s) foram também obtidos com a política *prio* e as piores (0.51 e 0.19s) foram obtidos com *ws* e a *dmdas* .

6.2.2 Comparação de 50k

A Tabela 6.3 e o Gráfico 6.2 apresenta os resultados obtidos para a comparação de 50k.

	50k					
	Acer Notebook			PLaFRIM		
	Poda (%)	**GCUPS**	**TEMPO(s)**	**Poda (%)**	**GCUPS**	**TEMPO(s)**
dmda	0	1.17	2.40	0	4.29	0.71
dmdas	0	1.16	2.42	0	4.00	0.89
eager	0	1.16	2.40	0	4.74	0.65
lws	0	1.22	2.35	0	**4.77**	**0.66**
prio	0	**1.32**	**2.33**	0	2.97	1.09
random	0	0.94	3.15	0	2.53	1.78
ws	0	1.18	2.39	0	3.97	1.06

Tabela 6.3: Percentual de poda, GCUPS e tempo de execução do MASA-StarPU com 7 estrategias para a comparação de 50k.

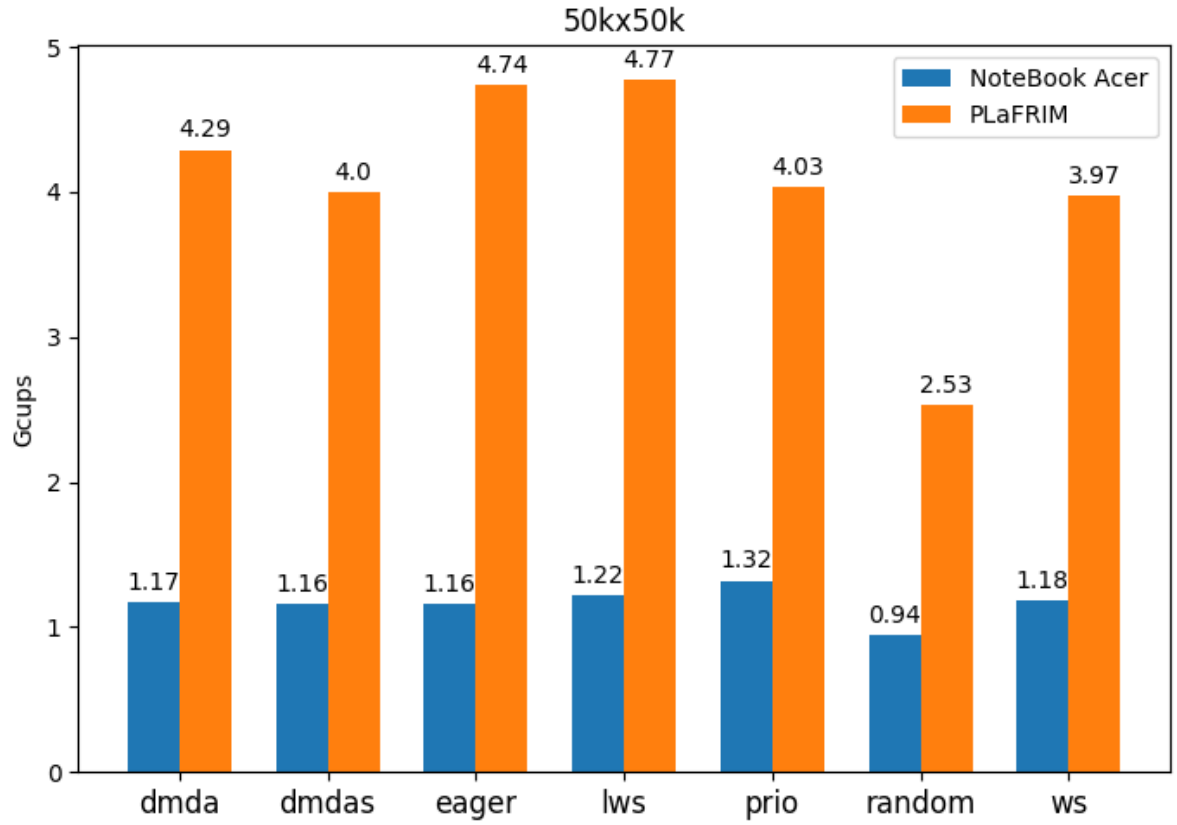


Figura 6.2: GCUPS da comparação de 50K para diferentes políticas de escalonamento

Como as sequências da comparação de 50k é possuem similaridade muito baixa, o percentual de poda é sempre 0, independente da estratégia e da plataforma de teste. A estratégia com o melhor valor de GCUPS e tempo de execução no notebook foi a *prio*

(1.32 e 2.33s) e a pior (0.94 e 3.15s) foi a *random*. No PLaFRIM, a melhor (4.77 e 0.66s) foi *lws* e a pior (2.53 e 1.78s) *random*.

6.2.3 Comparação de 150k

A Tabela 6.4 e o Gráfico 6.3 apresentam os resultados obtidos pelo MASA-StarPU para a comparação de 150k.

	150k					
	Acer Notebook			**PLaFRIM**		
	Poda (%)	**GCUPS**	**TEMPO(s)**	**Poda (%)**	**GCUPS**	**TEMPO(s)**
dmda	0	1.39	19.97	0	**4.88**	**5.70**
dmdas	0	1.39	19.99	0	4.62	6.01
eager	0	1.37	20.31	0	3.89	7.15
lws	0	1.40	19.84	0	3.84	7.25
prio	0	1.38	20.12	0	3.69	7.53
random	0	1.21	22.84	0	2.41	11.54
ws	0	**1.40**	**19.82**	0	3.86	7.21

Tabela 6.4: Percentual de poda, GCUPS e tempo de execução do MASA-StarPU com 7 estrategias a para comparação de 150k.

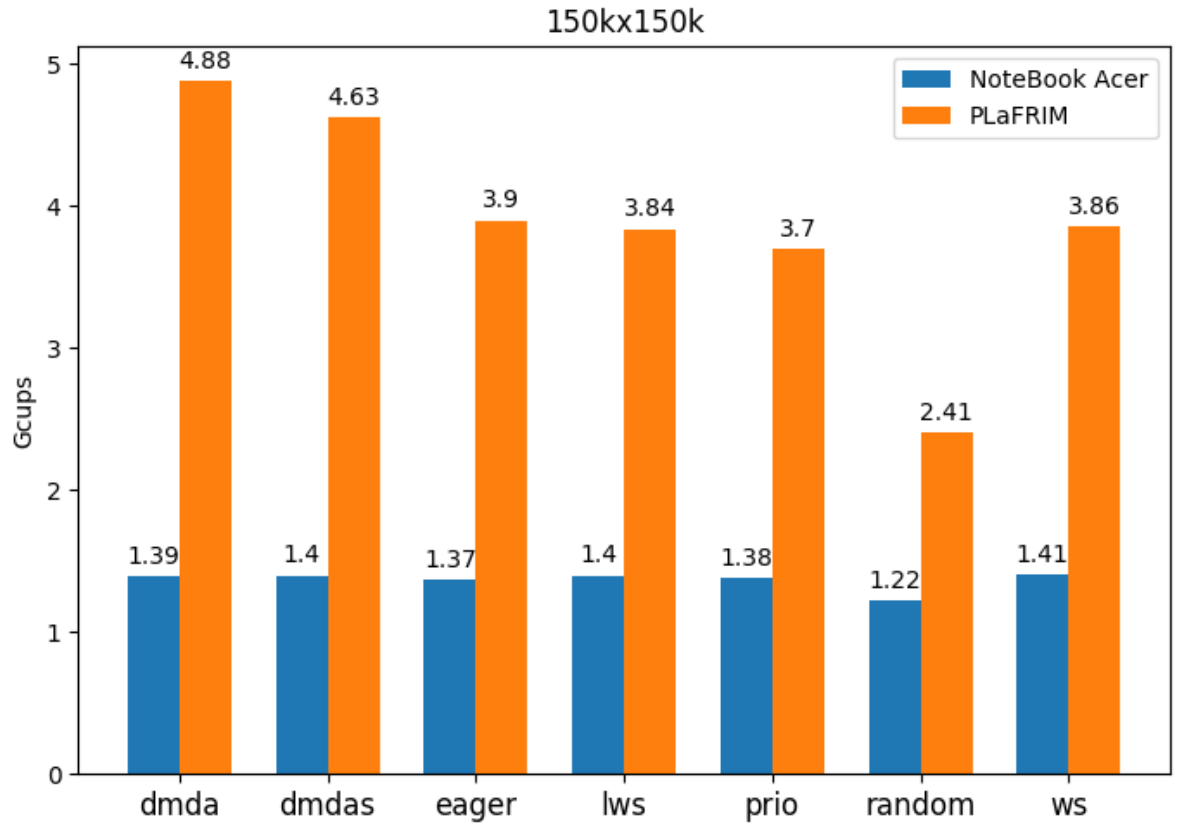


Figura 6.3: GCUPS da comparação de 150K para diferentes políticas de escalonamento

A comparação de 150k é também é uma comparação de baixa similaridade e, por isso, o percentual de poda também é sempre 0, independente da estratégia e da plataforma de teste. A estratégia com o melhor GCUPS e tempo de execução no notebook foi a *ws* (1.40 e 19.82s) e a pior (1.21 e 22.84s) foi a *random*. No PLaFRIM, a melhor estratégia (4.88 e 5.70s) foi *dmda* e a pior (2.41 e 11.54) *random*. Nessa comparação, podemos ver que o aumento no número de cores favoreceu a política *dmda*, que é uma política de escalonamento mais complexa. A política *ws*, que obteve o melhor desempenho no notebook, obteve desempenho médio no PlaFRIM.

6.2.4 Comparação de 500k

A Tabela 6.5 e o Gráfico 6.4 apresentam os resultados obtidos do MASA-StarPU para a comparação de 500k.

	500k					
	Acer Notebook			PLaFRIM		
	Poda (%)	GCUPS	TEMPO(s)	Poda (%)	GCUPS	TEMPO(s)
dmda	0	1.40	207.87	0	**6.21**	**46.80**
dmdas	0	1.40	207.45	0	5.86	49.61
eager	0	1.39	208.67	0	4.20	69.26
lws	0	1.41	206.66	0	4.28	67.89
prio	0	1.40	207.32	0	4.11	70.69
random	0	1.31	221.40	0	3.32	87.59
ws	0	**1.41**	**206.58**	0	4.28	68.07

Tabela 6.5: Percentual de poda, GCUPS e tempo de execução do MASA-StarPU com 7 estrategias para a comparação de 500k.

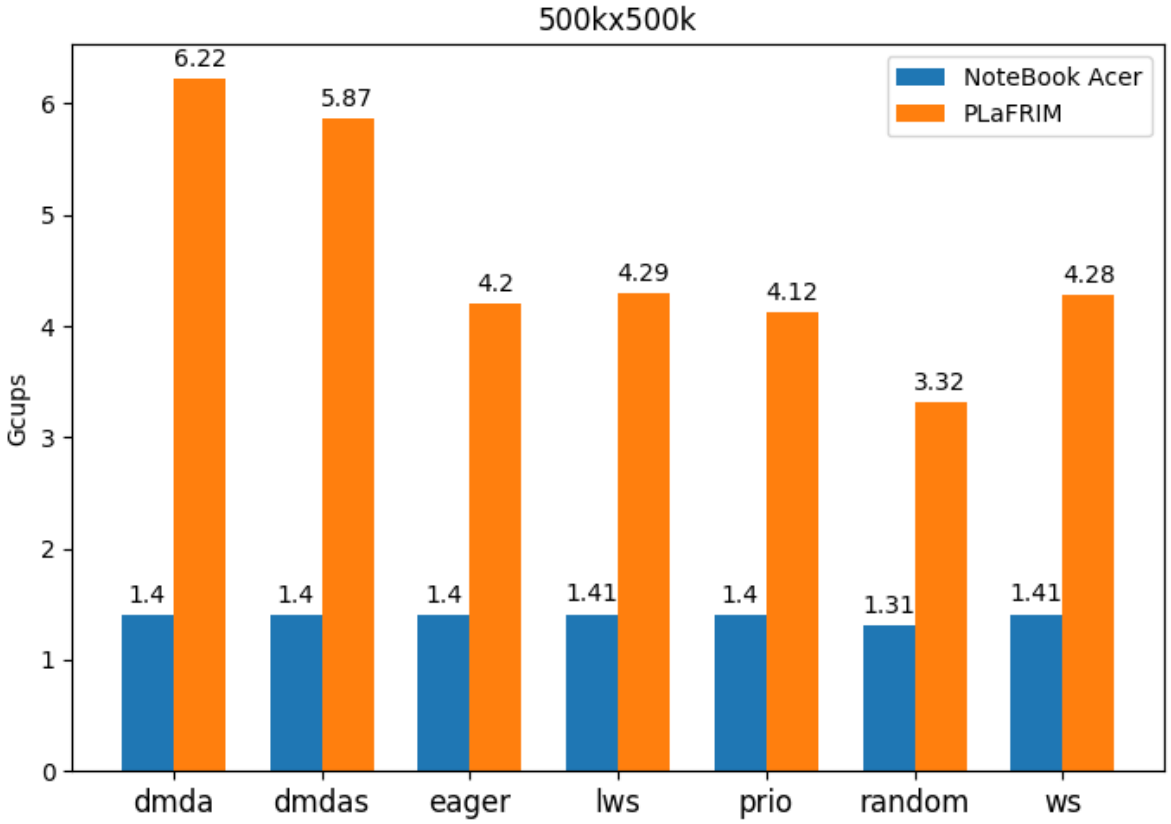


Figura 6.4: GCUPS da comparação de 500K para diferentes políticas de escalonamento

Assim como as comparações de 150k e 50k, a comparação de 500k também possui comparação de baixa similaridade, por isso, o percentual de poda também é sempre 0 independente da estratégia e da plataforma de teste. Para essa comparação todas as políticas obtiveram resultados parecidos, na plataforma Acer Notebook. Na plataforma

PLaFRIM, os resultados apresentam uma variação dependendo da política. A política com o melhor valor de GCUPS e tempo de execução foi a *ws* (1.41 e 206.58) e a pior (1.31 e 221.40s) foi a *random*, no Acer Notebook. No PLaFRIM a melhor política (6.21 e 46.80s) foi *dmda* e a pior (3.32 e 87.07) foi a *random*.

6.2.5 Comparação de 1M

A Tabela 6.6 e os Gráfico 6.5 apresentam os resultados obtidos para a comparação de 1M.

	1M					
	Acer Notebook			**PLaFRIM**		
	Poda (%)	**GCUPS**	**TEMPO(s)**	**Poda (%)**	**GCUPS**	**TEMPO(s)**
dmda	11.57	1.59	703.94	11.51	**7.13**	**157.01**
dmdas	**12.26**	1.60	698.37	**12.18**	7.04	159.12
eager	11.58	1.57	709.15	11.56	4.76	235.33
lws	10.90	1.57	709.12	11.47	4.87	229.71
prio	12.05	**1.60**	**699.23**	11.94	4.82	232.33
random	10.92	1.51	739.22	11.07	4.08	274.49
ws	10.67	1.57	710.30	10.99	4.83	231.96

Tabela 6.6: Percentual de poda, GCUPS e tempo de execução do MASA-StarPU com 7 estrategias para a comparação de 1M.

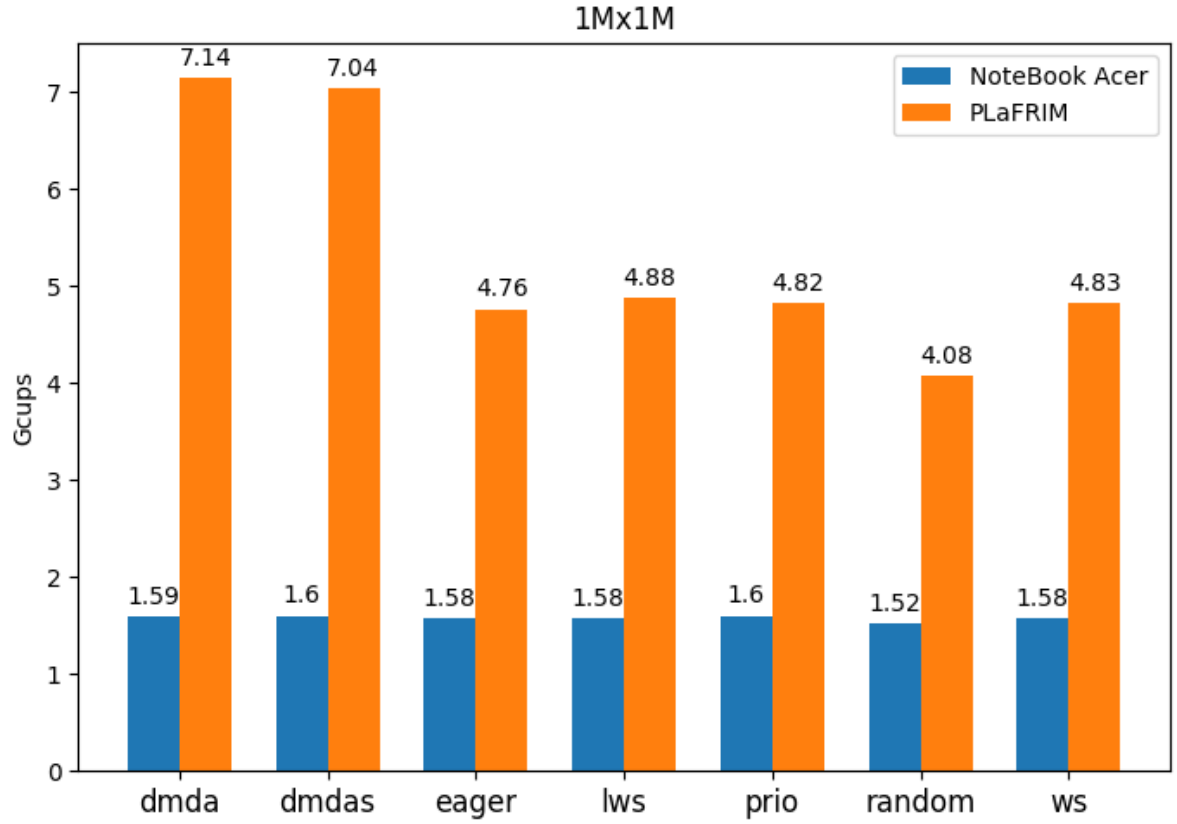


Figura 6.5: GCUPS da comparação de 1M para diferentes políticas de escalonamento

Na comparação de 1M, as sequências comparadas possuem uma certa similaridade, por isso, o percentual de poda é diferente de 0. No entanto, diferente da comparação de 10k, estes percentuais apresentam pouca variação independente da plataforma de testes. Na plataforma Acer Notebook o maior percentual de poda (12.26) foi obtido pela estratégia *dmdas*, e o menor (10.67) pela *ws*. No PLaFRIM o maior percentual de poda (12.18) também foi da estratégia *dmdas*, e o menor também (11.00) foi da estratégia *ws*. A estratégia com o melhor GCUPS e tempo de execução foi a *dmdas* (1.60 e 699.23s) e a pior (1.515 e 739.22s) foi a *random* no Acer Notebook. No PlaFRIM, o melhor percentual de poda (12.19) foi obtido pela *dmdas* e o melhor GCUPS pela *dmda* (7.14). Nessa comparação, as taxas de poda e GCUPS das políticas *dmdas* e *dmda* estão próximas. Nessa plataforma, o pior resultado foi da política *random* (11.07 % de poda e 4.08 GCUPS).

6.2.6 Comparação de 3M

A Tabela 6.7 e o Gráfico 6.6 apresentam os resultados obtidos para a comparação de 3M.

	3M					
	Acer Notebook			PLaFRIM		
	Poda (%)	GCUPS	TEMPO(s)	Poda (%)	GCUPS	TEMPO(s)
dmda	0.12	**1.41**	**7325.20**	0.12	**6.33**	**1629.69**
dmdas	0.12	1.40	7329.45	0.12	6.30	1638.80
eager	0.12	1.41	7389.55	0.12	4.25	2428.11
lws	0.12	1.40	7328.90	0.12	4.34	2377.85
prio	0.12	1.40	7327.21	0.12	4.27	2414.91
random	0.12	1.38	7453.35	0.12	3.99	2585.95
ws	0.12	1.41	7324.98	0.19	4.33	2385.20

Tabela 6.7: Percentual de poda, Gcups e tempo de execução do MASA-StarPU com 7 estrategias para a comparação de 3M.

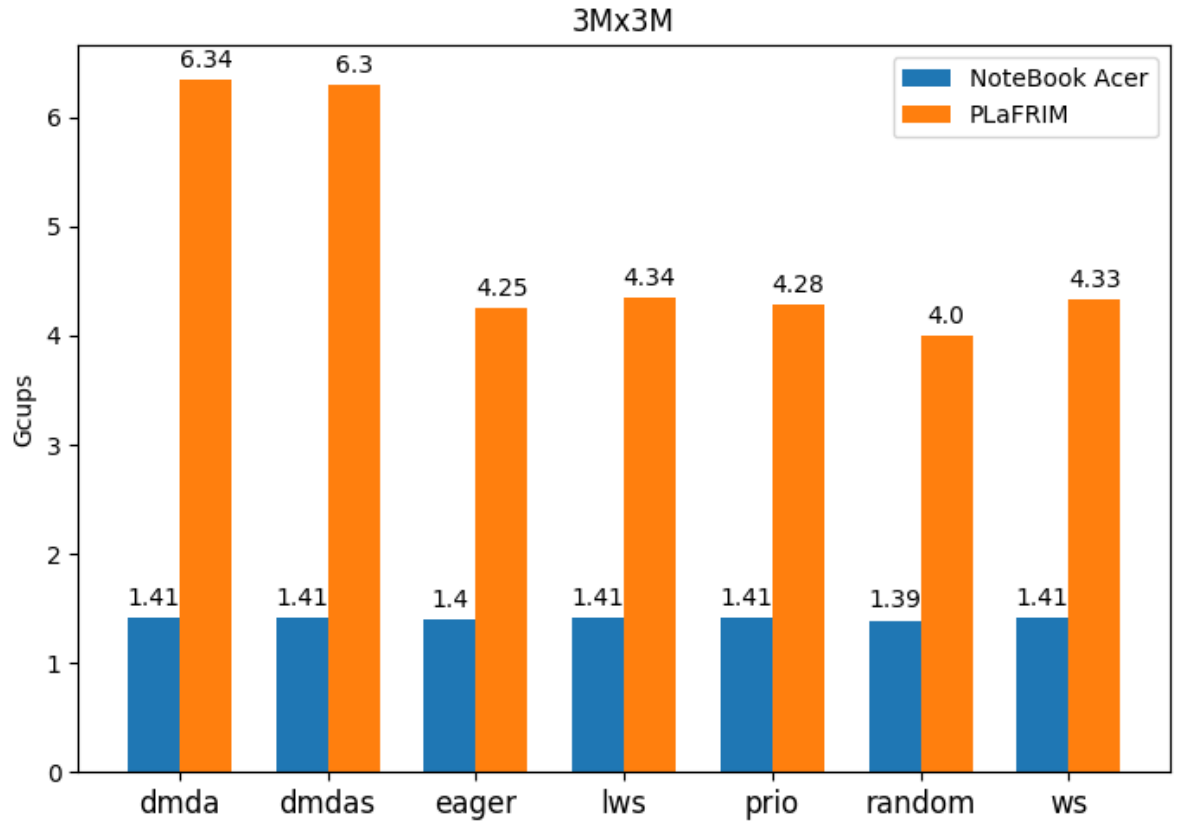


Figura 6.6: GCUPS da comparação de 3M para diferentes políticas de escalonamento

As sequências de 3M possuem baixa similaridade e o percentual de poda está próximo de 0 nas duas plataformas. O melhor GCUPS e tempo de execução foi obtido com a política *ws* (1.41 e 7324.98s) e a pior (1.38 e 7453.35s) foi a *random* no Acer Notebook.

No PLaFRIM, a melhor política (6.33 e 1629.69s) foi *dmda* e a pior (3.99 e 2585.95s) foi a *random*.

6.2.7 Comparação de 5M

A Tabela 6.8 e os Gráficos 6.7 e apresentam os resultados obtidos para a comparação de 5M.

	5M					
	Acer Notebook			**PLaFRIM**		
	Poda (%)	GCUPS	TEMPO(s)	**Poda (%)**	**GCUPS**	**TEMPO(s)**
dmda	57.43	3.31	8262.89	57.34	14.84	1841.60
dmdas	**66.45**	**4.19**	**6519.23**	**66.02**	**18.41**	**1484.08**
eager	57.44	3.28	8329.44	57.43	9.85	2773.10
lws	57.44	2.83	9465.78	43.79	7.68	3555.89
prio	65.93	4.12	6626.11	65.05	12.25	2229.33
random	53.28	2.95	9242.07	52.76	8.35	3272.80
ws	50.10	2.81	9696.65	43.02	7.58	3601.09

Tabela 6.8: Percentual de poda, GCUPS e tempo de execução do MASA-StarPU com 7 estrategias para a comparação de 5M.

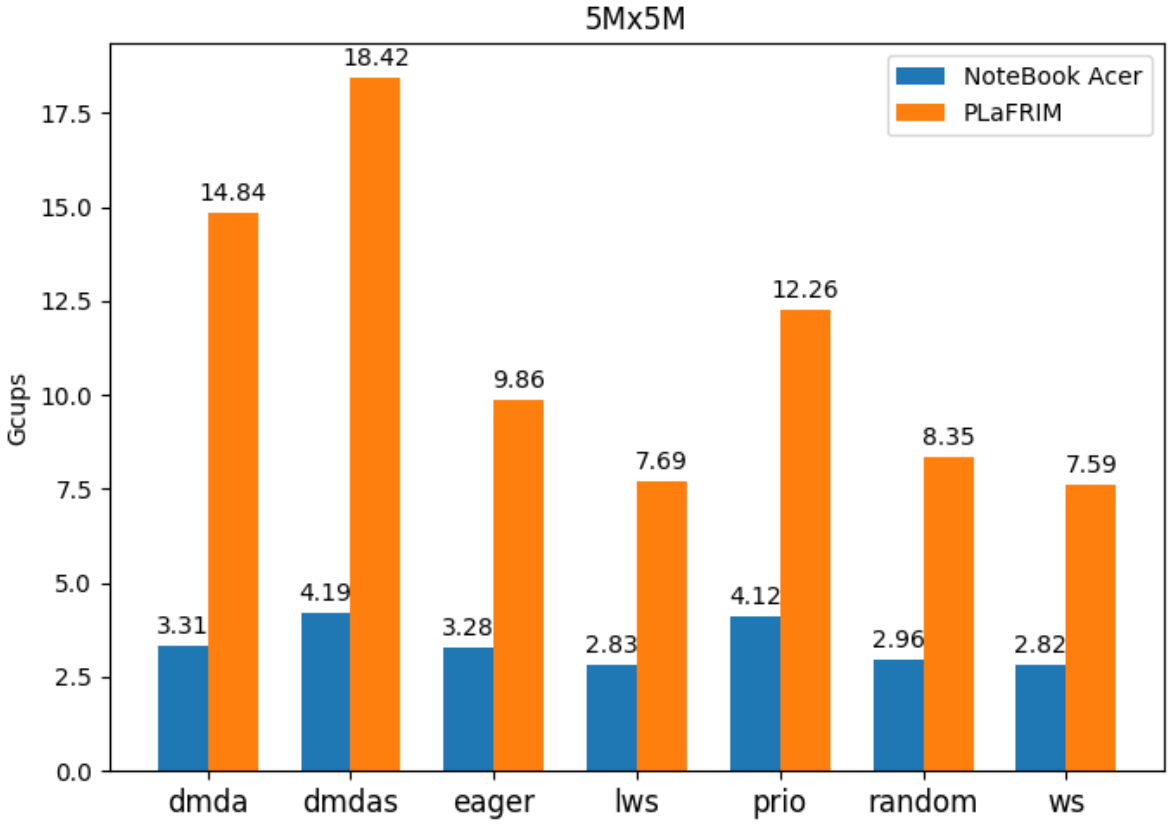


Figura 6.7: GCUPS da comparação de 5M para diferentes políticas de escalonamento

A comparação de 5M é a comparação com maior similaridade. Por isso, a maioria dos percentuais de poda passa de 50%. Nesta comparação estes percentuais apresentam uma grande variação dependendo da política. Na plataforma Acer Notebook, o maior percentual de poda obtido (66.45) foi da política *dmdas* e o menor (50.1058) foi da política *ws*. No PLaFRIM o maior percentual (66.02) foi da também foi da política *dmdas* e o menor (43.02) também da *ws*. A estratégia com o melhor GCUPS e tempo de execução no notebook foi a *dmdas* (4.19 e 6519.23s) e a pior (2.81 e 9696.65s) foi a *ws*. No PLaFRIM a melhor política (18.41 e 1484.08s) foi *dmdas* e a pior (7.58 e 3601.09) foi a*ws*.

6.2.8 Avaliação Conjunta

Nesta seção, será avaliado o impacto das políticas de escalonamento de tarefas, a plataforma de execução e o percentual de poda no tempo de execução e GCUPS para as sete comparações mostradas na Tabela 6.1.

A Figura 6.8 e Figura 6.9 apresentam os resultados conjuntos das sete políticas de escalonamento de tarefas presentes no MASA-StarPU em GCUPS nas plataformas notebook Acer (a) e PlaFRIM (b).

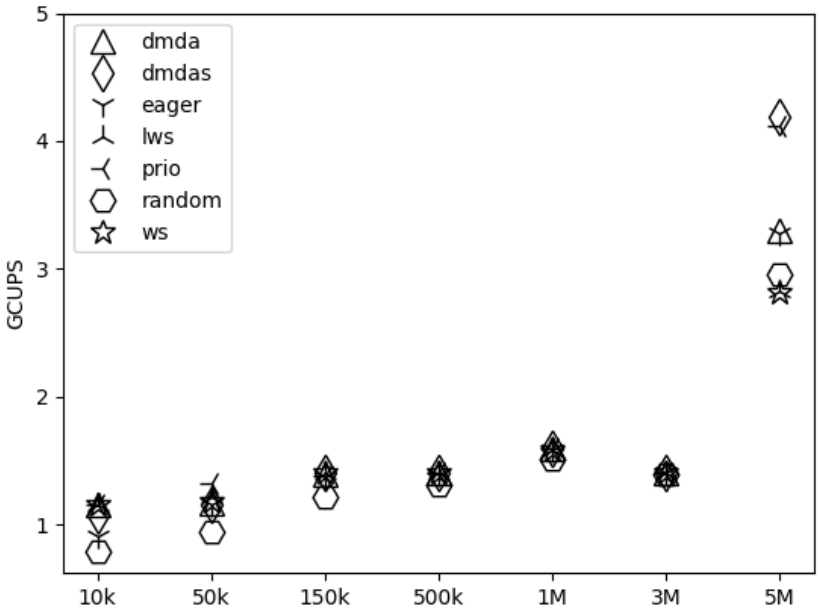


Figura 6.8: (a) Resultado consolidado no notebook Acer

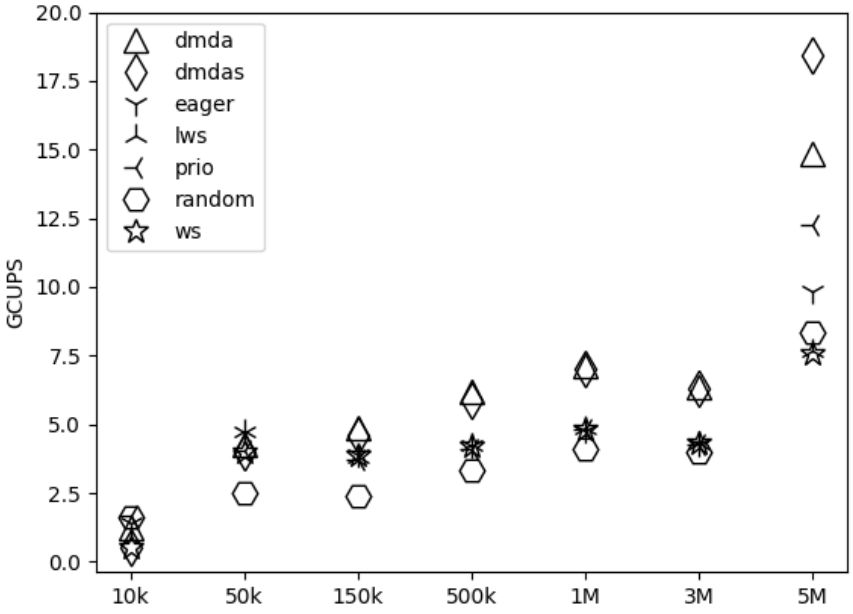


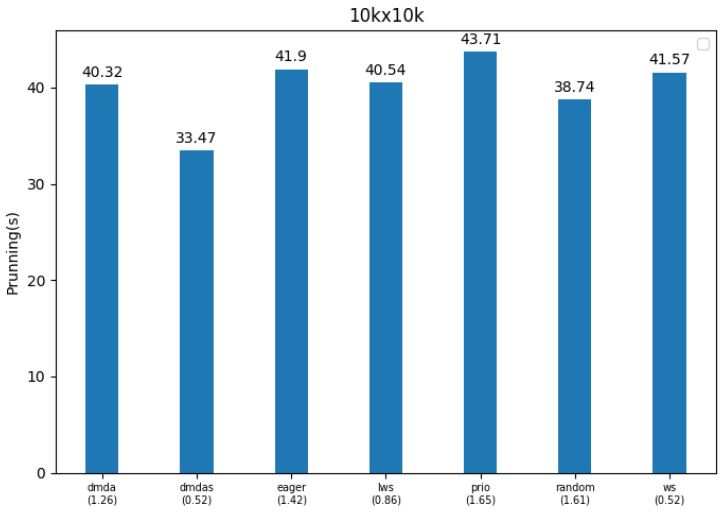
Figura 6.9: (b) Resultado consolidado no PLAFRIM

Embora as formas dos gráficos em ambas as plataformas sejam semelhantes, o comportamento das políticas é diferente. No notebook (Figura 6.8), foi observado que, com

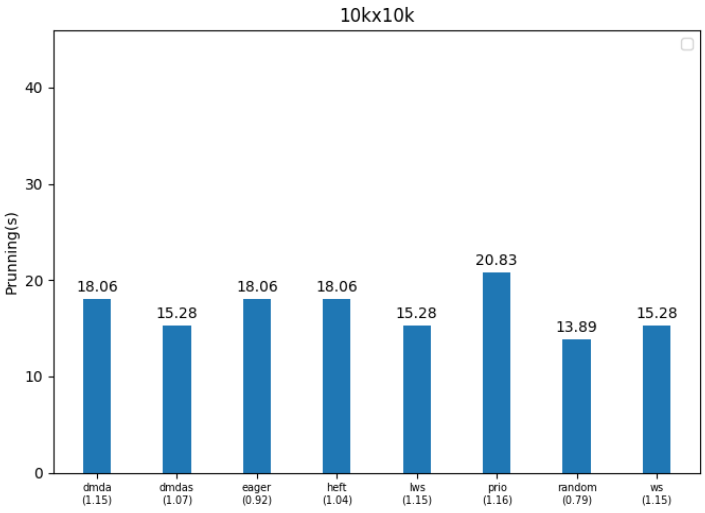
exceção da comparação 5M x 5M, as políticas escalonamento têm GCUPS próximos. Isso acontece porque o número de núcleos(*core*) é pequeno. Mesmo com um número limitado de núcleos, há uma diferença significativa entre as políticas na comparação 5M x 5M. Nesta comparação, dois fatores foram combinados: uma grande matriz de programação dinâmica (5M x 5M) e uma grande taxa de poda (mais de 50 %). Isso levou a diferenças consideráveis. Neste caso, *dmdas* e *prio* têm os GCUPS mais altos, enquanto *ws*, *lws* e *random* têm os mais baixos.

Na plataforma PlaFRIM (Figura 6.9), observamos que a política tem um impacto maior no desempenho. Para as comparações na figura, a diferença no GCUPS é superior a 55 %, se considerarmos o valor mais alto e mais baixo para cada execução com política diferente. A comparação 5M x 5M apresentou a maior variação do GCPUS (139,7 %) e o melhor GCUPS (18,41) foi obtido com a política *dmdas*. Embora a política *prio* tenha um desempenho muito bom no notebook (Figura 6.8), ela teve um desempenho médio no PlaFRIM (Figura 6.9). Nesse caso, o pequeno intervalo de prioridades $[-5, +5]$ teve um impacto negativo no desempenho, quando mais nós são usados. Como esperado, *random* teve resultados ruins em todas as comparações.

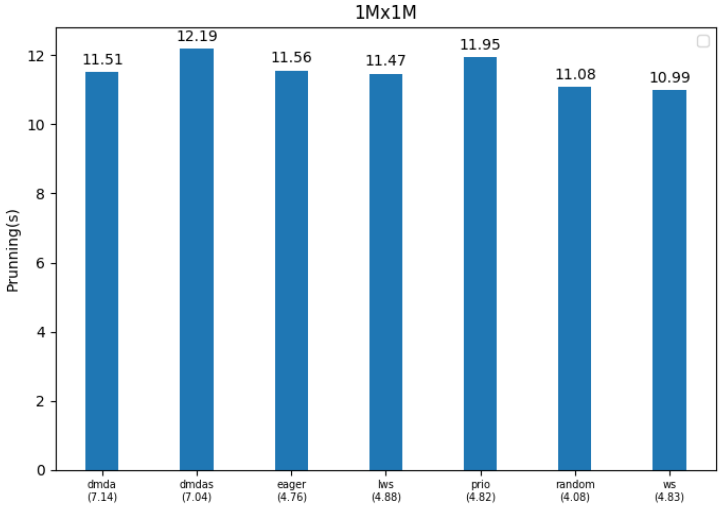
Os gráficos da Figura 6.10 apresentam as taxas de poda por estratégia de alocação de tarefas para as comparações de 10kx10k, 1Mx1M e 5Mx5M.



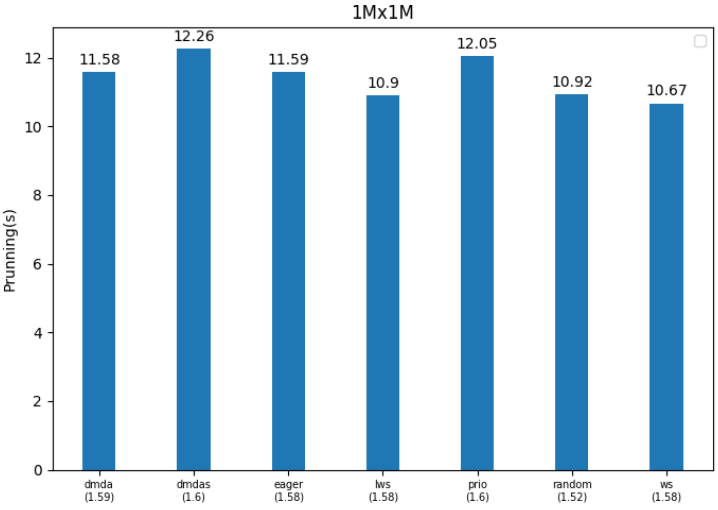
(a) 10k x 10k (PLAFRIM)



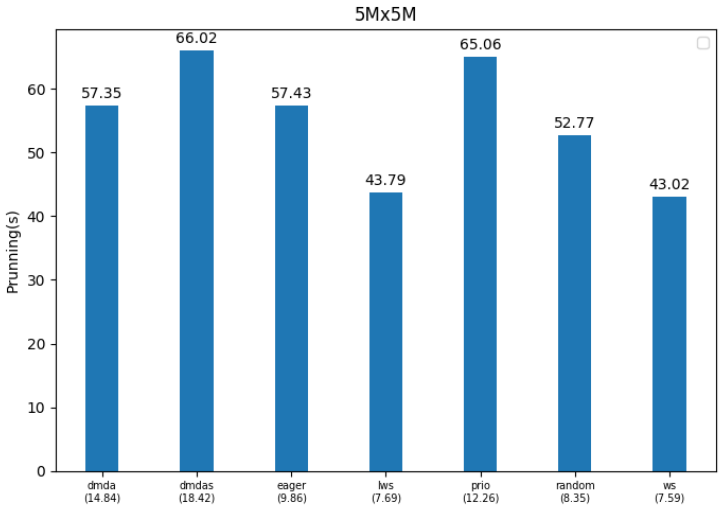
(b) 10k x 10k (PC)



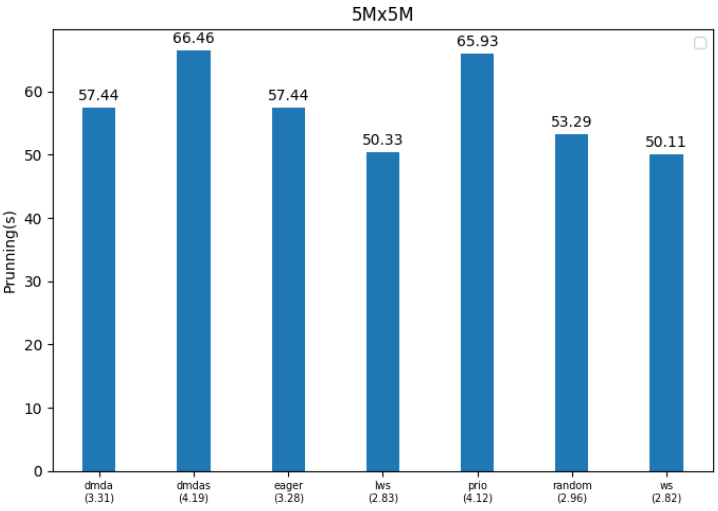
(c) 1M x 1M (PLAFRIM)



(d) 1M x 1M (PC)



(e) 5M x 5M (PLAFRIM)



(f) 5M x 5M (PC)

Figura 6.10: Taxa de poda para as comparações 10Kx10K, 1Mx1M e 5Mx5M. Os GCUPS são mostrados na parte inferior entre parenteses.

Nos gráficos da Figura 6.10, no eixo x estão as políticas de alocação de com tarefas os resultados obtidos em GCUPS e no eixo y está a taxa de poda. Nas demais comparações (50kx50k, 150kx150, 500kx500k, 3Mx3M), as sequências são muito diferentes, por isso, a taxa de poda é 0 ou próxima de 0.

Como apresentado na Figura 6.10, a política de escalonamento de tarefas tem um impacto significativo na taxa de poda. Com exceção da Figura 6.10b, os resultados das taxas de poda em ambas as plataformas são comparáveis aos resultados obtidos para as mesmas seqüências com o MASA-OmpSs [4]. Na comparação 10K x 10K usando o notebook, o número reduzido de núcleos teve um impacto negativo na taxa de poda, que caiu para metade da taxa obtida na plataforma PlaFRIM (Figura 6.10a). Nas duas plataformas, a melhor taxa de poda para a comparação 10K x 10K foi alcançada pela política *prio*.

Embora exista uma variação na taxa de poda para a comparação 1M x 1M, em ambas as plataformas (Figura 6.10c e Figura 6.10d) essa variação é pequena (menor que 2%).

Para a comparação 5M x 5M, os melhores resultados de taxa de poda foram alcançados pelas políticas *dmdas* e *prio*. Não surpreendentemente, essas são as políticas que usam prioridades para agendar tarefas. As políticas baseadas em *work stealing* (*ws* e *lws*) não forneceram bons resultados de poda em nenhuma das plataformas. Para essas políticas, na plataforma PlaFRIM, a taxa de poda foi cerca de 7 % menor do que no notebook.

A Tabela 6.9 mostra os GCUPS e os intervalos de poda (mais baixo-mais alto), bem como a política que obteve o melhor GCUPS para as plataformas Notebook e PlaFRIM.

Comparação	**Notebook (4 cores)**			**PlaFRIM (24 cores)**		
	Taxa de poda (%)	Intervalo de GCUPS	Melhor política GCUPS	Taxa de poda (%)	Intervalo de GCUPS	Melhor política GCUPS
10Kx10K	13.8-20.8	0.78-1.15	prio	33.4-43.7	0.52-1.62	prio
50Kx50K	0-0	0.94-1.32	ws	0-0	2.53-4.77	lws
150Kx150K	0-0	1.22-1.41	ws	0-0	2.41-4.88	dmda
500Kx500K	0-0	1.31-1.41	ws lws	0-0	3.32-6.22	dmda
1Mx1M	10.6-12.2	1.52-1.60	dmdas prio	10.9-12.1	4.08-7.14	dmda
3Mx3M	0.1-0.1	1.39-1.41	dmda ws	0.1-0.1	4.00-6.34	dmda
5Mx5M	50.1-66.4	2.83-4.19	dmdas	43.7-66.0	7.69-18.42	dmdas

Tabela 6.9: GCUPS, Intervalos de poda e melhor política para sete comparações nas plataformas Notebook e PlaFRIM

A escala de cinza nas linhas indica a taxa de poda da matriz de programação dinâmica. As taxas de poda menores que 10 % são brancas e quanto maior a taxa de poda, mais escura a cor. Com relação à taxa de poda, é possível perceber que, com exceção da comparação 10K x 10K, não há grande variação quando alteramos a plataforma (Notebook ou PlaFRIM). Ocorreu uma grande variação na comparação 10K x 10K porque o comprimento das seqüências comparadas é próximo ao tamanho do bloco (2K x 2K). Portanto, uma pequena variação no número de blocos não processados levou a uma grande variação percentual. Em comparações de seqüências mais longas, essa grande variação na taxa de poda entre plataformas não ocorreu.

Com relação à melhor política, as políticas *eager* e *random* não alcançaram o melhor resultado para qualquer comparação em qualquer plataforma. Isso era esperado, pois essas políticas são muito simples e as tarefas que compõem as comparações de sequências biológicas produzem um gráfico de tarefas complexo. Ao considerar cada plataforma sozinha (Notebook ou PlaFRIM), não existe uma política que forneça o melhor resultado (melhor política) para todas as comparações.

Para comparações de sequências menores que 1M e com baixa taxa de poda (50K, 150K e 500K), a política *ws* apresentou uma alta taxa de GCUPS no notebook. Deve-se observar que *ws* não teve bons resultados na plataforma PlaFRIM. Por exemplo, para a comparação de 150K x 150K, o GCUPS da *ws* no PLaFRIM foi 3,86 e o melhor GCUPS foi 4,93 (Tabela 6.9). Para sequências até 500k, a política *dmda* teve bons resultados na plataforma notebook, por exemplo para a comparação de 150K x 150K, o GCUPS foi de 1,39 e o melhor GCUPS foi de 1,40.

Em comparações de sequências maiores ou iguais a 1M com taxa de poda superior a 10 % (1M e 5M), a política *dmdas* tem um desempenho muito bom nas duas plataformas. Para sequências longas com baixa taxa de de poda (3M), as melhores políticas para sequências menores (*lws* e *ws*, para as plataformas PlaFRIM e Notebook, respectivamente) ainda têm desempenho muito bom. Ao comparar sequências 3M no notebook, as políticas *dm* e *dmda* também fornecem resultados muito bons.

Analisando os resultados apresentados nesta seção, é possível fazer algumas observações gerais. Quando se espera que a taxa de poda seja baixa, políticas simples que não levam em consideração as prioridades entre as tarefas parecem ser a melhor opção, pois a sobrecarga de levar em consideração as prioridades não compensa a pequena otimização da poda atingida. No entanto, quando se espera que a taxa de poda seja alta, é realmente importante usar uma estratégia de escalonamento de tarefas mais complexa, levando em conta, por exemplo, as prioridades.

6.3 Comparação com outras ferramentas

Por fim, o desempenho do MASA-StarPU foi comparado com duas ferramentas que compõem o estado da arte de comparação de sequências longas em CPU (MASA-OpenMP e MASA-OmpSs) na plataforma PlaFRIM (24 *cores*). A Tabela 6.10 mostra os tempos de execução da melhor política do MASA-StarPU e os tempos de execução dessas duas ferramentas, bem como a aceleração obtida pelo MASA-StarPU. A aceleração foi calculada dividindo-se o tempo de execução do MASA-StarPU pelo segundo melhor tempo de execução (em negrito na tabela). Os tempos de execução medidos são os tempos do relógio. Portanto, incluem o tempo de inicialização (MASA, StarPU e OmpSs), além de ler as sequências dos arquivos e gravar a saída.

Comp.	MASA-StarPU (s)	MASA-OpenMP (s)	MASA-OmpSs (s)	Speedup	GCUPS MASA-StarPU
10K	**0.05**	**0.07**	0.08	1.40x	1.86
50K	**0.71**	1.37	**1.28**	1.80x	4.54
150K	**5.65**	9.87	**9.20**	1.62x	4.93
500K	**46.80**	**76.73**	77.17	1.63x	6.21
1M	**156.91**	**247.16**	258.23	1.57x	7.14
3M	**1621.42**	**2472.80**	2637.42	1.52x	6.37
5M	**1484.08**	2824.90	**2147.63**	1.44x	18.41

Tabela 6.10: Comparação entre MASA-StarPU, MASA-OpenMP e MASA-OmpSs (24 *cores*)

Pode ser visto na Tabela 6.10 que o MASA-StarPU atinge o menor tempo de execução para todas as comparações. A aceleração em relação à segunda melhor ferramenta (MASA-OpenMP ou MASA-OmpSs) varia de 1,4x a 1,8x. Particularmente, o MASA-StarPU conseguiu atingir 18,41 GCUPS na comparação de 5M que é melhor do que os GCUPS obtidos pelas ferramentas de CPU listados na Tabela 4.3. Na mesma comparação, o MASA-OmpSs, a mesma estratégia de paralelização (*dataflow* em formato quadrado), atinge 12,72 GCUPS.

Capítulo 7

Conclusão

Na presente Dissertação de Mestrado, foi proposto e avaliado o MASA-StarPU, uma solução que usa computação paralela para comparar longas sequências de DNA com poda e suporte para sete estratégias dinâmicas de escalonamento de tarefas. O MASA-StarPU foi desenvolvido com base no *framework* MASA, com o objetivo de avaliar diversas políticas de escalonamento de tarefas adequadas a otimização *block prunning*.

A integração entre os ambientes MASA e Star-PU foi feita de maneira a não haver alteração em quaisquer dos ambientes, utilizando somente as APIs fornecidas pelos dois ambientes. Apesar disso, o MASA-StarPU apresentou um desempenho muito bom, o que mostra que a estratégia de integração foi adequada.

O MASA-StarPU foi executado em dois ambientes: (a) PLaFrim (24 núcleos) e (b) notebook (4 núcleos), obtendo, em geral, variados desempenhos de acordo com a política de escalonamento de tarefas, as seqüências, a taxa de poda e o número de núcleos disponíveis. O impacto desses fatores no tempo de execução e GCUPS para as sete comparações foi avaliado. No notebook, com exceção da comparação 5M x 5M, as políticas de escalonamento obtiveram uma taxa GCUPS próxima e *dmdas* e *prio* tiveram o maior GCUPS enquanto *ws*, *lws* e *random* obtiveram os GCUPS mais baixos. Na plataforma PlaFRIM, a política de escalonamento tem um impacto maior no desempenho. Para todas as comparações, a diferença no GCUPS foi superior a 55 %, se considerar o menor e maior valor. A comparação 5M x 5M apresentou a maior variação do GCPUS (139,7 %) e a melhor taxa de GCUPS (18,41) foi obtido com a política dmdas.

Outro aspecto avaliado foi a variação da taxa de poda. Com exceção da comparação 10K x 10K, não houve grande variação na taxa de poda. Uma grande variação ocorreu na comparação de 10K x 10K porque o comprimento das sequências comparadas estava perto do tamanho do bloco (2K x 2K). Então, uma pequena variação no número de blocos podados levou a uma grande variação percentual.

Não foi possível decidir a melhor política de escalonamento, em termos de tempo

de execução / GCUPS, uma vez que as mudanças na taxa de poda, comprimentos das seqüências ou do ambiente de execução causaram uma ampla variação no desempenho.

Por fim, foi feita a comparação do MASA-StarPU com outras soluções para comparações de sequências longas em CPU. Nessa comparação, o MASA-StarPU atingiu o menor tempo de execução para todas as comparações. A aceleração em relação à segunda melhor ferramenta (MASA-OpenMP ou MASA-OmpSs) variou de 1,4x a 1,8x. Particularmente, o MASA-StarPU conseguiu atingir 18,41 GCUPS. Na mesma comparação, o MASA-OmpSs, que também segue o mesmo modelo de execução (*dataflow* em formato quadrado), atinge 12,72 GCUPS.

Como trabalhos futuros, pretendemos recuperar também o alinhamento local entre as duas sequências. Para atingir esse objetivo, os outros estágios do MASA devem ser incorporados ao MASA-StarPU.

Neste trabalho, foi observado que não foi possível determinar a melhor política de escalonamento para todos os casos. No entanto, acreditamos que é possível avaliar casos específicos e determinar as melhores políticas para esses casos. Para tanto, planeja-se criar uma estratégia baseada em inteligência artificial que tenha como entrada a taxa de poda esperada, os comprimentos das seqüências comparadas, o número de núcleos da ambiente de execução e forneça como saída a política de escalonamento mais adequada.

Finalmente, o MASA-StarPU foi construído com o objetivo de ser executado usando apenas uma máquina. Outro trabalho futuro é, portanto, criar um implementação usando MPI para que MASA-StarPU possa ser executado em mais de uma máquina.

Referências

[1] Myers, Eugene W e Webb Miller: *Optimal alignments in linear space*. Bioinformatics, 4(1):11–17, 1988. viii, 11, 12

[2] Fickett, James W: *Fast optimal alignment*. 1984. viii, 12, 13

[3] Sandes, Edans Flavius O e Alba CMA de Melo: *Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu*. IEEE Transactions on Parallel and Distributed Systems, 24(5):1009–1021, 2013. viii, 13, 14

[4] Sandes, Edans Flavius, F, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro e Alba CMA De Melo: *Masa: a multiplatform architecture for sequence aligners with block pruning*. ACM Transactions on Parallel Computing (TOPC), 2(4):28, 2016. viii, 2, 15, 27, 28, 30, 31, 32, 40, 43, 52, 68

[5] Lan, Haidong, Weiguo Liu, Yongchao Liu e Bertil Schmidt: *Swhybrid: A hybrid-parallel framework for large-scale protein sequence database search*. Em *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, páginas 42–51. IEEE, 2017. viii, 30, 31, 32, 33, 34

[6] Lan, Haidong, Yuandong Chan, Kai Xu, Bertil Schmidt, Shaoliang Peng e Weiguo Liu: *Parallel algorithms for large-scale biological sequence alignment on xeon-phi based clusters*. BMC bioinformatics, 17(9):267, 2016. viii, 30, 31, 32, 34, 35, 36

[7] Chen, Chunxi e Bertil Schmidt: *An adaptive grid implementation of dna sequence alignment*. Future Generation Computer Systems, 21(7):988–1003, 2005. viii, 30, 31, 32, 37, 38

[8] Mount, David W e David W Mount: *Bioinformatics: sequence and genome analysis*, volume 2. Cold spring harbor laboratory press New York:, 2001. 2, 5, 6

[9] Smith, Temple F e Michael S Waterman: *Comparison of biosequences*. Advances in applied mathematics, 2(4):482–489, 1981. 2, 9

[10] Katoh, K. e D. M. Standley: *Mafft multiple sequence alignment software version 7: improvements in performance and usability*. Molecular biology and evolution, 30(4):772–780, 2013. 2

[11] Rucci, Enzo, Carlos Garcia Sanchez, Guillermo Botella Juan, Armando De Giusti, Marcelo Naiouf e Manuel Prieto-Matias: *Swimm 2.0: enhanced smith–waterman on intel's multicore and manycore architectures based on avx-512 vector extensions*. International Journal of Parallel Programming, páginas 1–21, 2018. 2

[12] De Sandes, Edans Flavius O, Guillermo Miranda, Xavier Martorell Bofill, Eduard Ayguadé Parra, George Teodoro e Alba de Melo: *Cudalign 4.0: incremental speculative traceback for exact chromosome-wide alignment in gpu clusters*. IEEE transactions on parallel and distributed systems, 27(10):2838–2850, 2016. 2, 30, 31, 32

[13] Rucci, Enzo, Carlos Garcia, Guillermo Botella, Armando De Giusti, Marcelo Naiouf e Manuel Prieto-Matias: *Swifold: Smith-waterman implementation on fpga with opencl for long dna sequences*. BMC systems biology, 12(5):96, 2018. 2, 30, 31, 32

[14] Fei, Xia, Zou Dan, Lu Lina, Man Xin e Zhang Chunlei: *Fpgasw: Accelerating large-scale smith–waterman sequence alignment application with backtracking on fpga linear systolic array*. Interdisciplinary Sciences: Computational Life Sciences, 10(1):176–188, 2018. 2, 30, 31, 32

[15] Papadimitriou, Christos H e Kenneth Steiglitz: *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998. 3

[16] El-Rewini, Hesham, Hesham H Ali e Ted Lewis: *Task scheduling in multiprocessing systems*. Computer, 28(12):27–37, 1995. 3

[17] Augonnet, Cédric, Samuel Thibault, Raymond Namyst e Pierre André Wacrenier: *Starpu: a unified platform for task scheduling on heterogeneous multicore architectures*. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011. 3, 22, 26, 27

[18] Lopes, Rafael A. S., Samuel Thibault e Alba CAM Melo: *Masa-starpu: Parallel sequence comparison with multiple scheduling policies and pruning*. Em *SBAC-PAD 2020-IEEE 32nd International Symposium on Computer Architecture and High Performance Computing*, 2020. 4

[19] Needleman, Saul B e Christian D Wunsch: *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, 1970. 7, 8, 9

[20] Gotoh, Osamu: *An improved algorithm for matching biological sequences*. Journal of molecular biology, 162(3):705–708, 1982. 10, 11

[21] Hirschberg, Daniel S.: *A linear space algorithm for computing maximal common subsequences*. Communications of the ACM, 18(6):341–343, 1975. 11

[22] Maier, David: *The complexity of some problems on subsequences and supersequences*. Journal of the ACM (JACM), 25(2):322–336, 1978. 11

[23] Dagum, Leonardo e Ramesh Menon: *Openmp: an industry standard api for shared-memory programming*. IEEE computational science and engineering, 5(1):46–55, 1998. 16

[24] Conway, Melvin E: *A multiprocessor system design*. Em *Proceedings of the November 12-14, 1963, fall joint computer conference*, páginas 139–146. ACM, 1963. 16

[25] Stone, John E, David Gohara e Guochun Shi: *Opencl: A parallel programming standard for heterogeneous computing systems*. Computing in science & engineering, 12(3):66–73, 2010. 18

[26] Duran, Alejandro, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell e Judit Planas: *Ompss: a proposal for programming heterogeneous multi-core architectures*. Parallel Processing Letters, 21(02):173–193, 2011. 20

[27] Ayguadé, Eduard, Rosa M Badia, Francisco D Igual, Jesús Labarta, Rafael Mayo e Enrique S Quintana-Ortí: *An extension of the starss programming model for platforms with multiple gpus*. Em *European Conference on Parallel Processing*, páginas 851–862. Springer, 2009. 20

[28] P., Tang. e P. C. Yew: *Processor self-scheduling for multiple nested parallel loops*. Em *Int. Conf. on Parallel Processing (ICPP'86)*, páginas 528–535, 1986. 27

[29] Blumofe, R. e C. Leiserson: *Scheduling multithreaded computations by work stealing*. Journal of ACM, 46(05), 1999. 27

[30] Topcuoglu, Haluk, Salim Hariri e Min you Wu: *Performance-effective and low-complexity task scheduling for heterogeneous computing*. IEEE transactions on parallel and distributed systems, 13(3):260–274, 2002. 27

[31] Topcuoglu, H. e S. Hariri: *Performance-effective and low-complexity task scheduling for heterogeneous computing*. IEEE Transactions on Parallel and Distributed Systems, 13(03):260–274, 2002. 27

[32] Sandes, Edans Flavius De Oliveira, Azzedine Boukerche e Alba Cristina Magalhaes Alves De Melo: *Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification*. ACM Computing Surveys (CSUR), 48(4):63, 2016. 29, 31

[33] Zou, Huihui, Shanjiang Tang, Ce Yu, Hao Fu, Yusen Li e Wenjie Tang: *Asw: Accelerating smith–waterman algorithm on coupled cpu–gpu architecture*. International Journal of Parallel Programming, páginas 1–15, 2018. 30, 31, 32

[34] Zhao, Guoguang, Cheng Ling e Donghong Sun: *Sparksw: scalable distributed computing system for large-scale biological sequence alignment*. Em *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, páginas 845–852. IEEE, 2015. 30, 31, 32

[35] Rucci, Enzo, Carlos Garcia, Guillermo Botella, Armando E De Giusti, Marcelo Naiouf e Manuel Prieto-Matias: *Oswald: O pencl s mith–w aterman on a ltera's fpga for l arge protein d atabases*. The International Journal of High Performance Computing Applications, 32(3):337–350, 2018. 30, 31, 32

[36] Xu, Bo, Changlong Li, Hang Zhuang, Jiali Wang, Qingfeng Wang, Jinhong Zhou e Xuehai Zhou: *Dsa: scalable distributed sequence alignment system using simd instructions*. Em *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, páginas 758–761. IEEE Press, 2017. 30, 31, 32

[37] Feng, Xiaowen, Hai Jin, Ran Zheng, Lei Zhu e Weiqi Dai: *Accelerating smith-waterman alignment of species-based protein sequences on gpu*. International Journal of Parallel Programming, 43(3):359–380, 2015. 30, 31, 32

[38] Sandes, Edans FO, George LM Teodoro, Maria Emilia MT Walter, Xavier Martorell, Eduard Ayguade e Alba CMA Melo: *Formalization of block pruning: Reducing the number of cells computed in exact biological sequence comparison algorithms*. The Computer Journal, 61(5):687–713, 2018. 43

[39] *Starpu*. `http://starpu.gforge.inria.fr/`. Acessado em: 09/12/2019. 46

[40] *Sobre ubuntu*. `https://ubuntu.com/about`. Acessado em: 09/12/2019. 46

[41] *Sobre ubuntu*. `https://ubuntu.com/community/debian`. Acessado em: 09/12/2019. 46

[42] *Ncbi*. `https://www.ncbi.nlm.nih.gov/`. Acessado em: 05/08/2020. 52

Anexo I

2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)

MASA-StarPU: Parallel Sequence Comparison with Multiple Scheduling Policies and Pruning

1st Rafael A. Lopes
Department of Computer Science
University of Brasilia (UnB)
Brasilia, Brazil
rafael.lopes@gmail.com

2nd Samuel Thibault
Lab. Bordeleais de Recherche en Informatique
INRIA Bordeaux
Bordeaux, France
samuel.thibault@labri.fr

3rd Alba C. M. A. Melo
Department of Computer Science
University of Brasilia (UnB)
Brasilia, Brazil
alves@unb.br

***Abstract*—Sequence comparison tools based on the Smith-Waterman (SW) algorithm provide the optimal result but have high execution times when the sequences compared are long, since a huge dynamic programming (DP) matrix is computed. Block pruning is an optimization that does not compute some parts of the DP matrix and can reduce considerably the execution time when the sequences compared are similar. However, block pruning's resulting task graph is dynamic and irregular. Since different pruning scenarios lead to different pruning shapes, we advocate that no single scheduling policy will behave the best for all scenarios. This paper proposes MASA-StarPU, a sequence aligner that integrates the domain specific framework MASA to the generic programming environment StarPU, creating a tool which has the benefits of StarPU (i.e., multiple task scheduling policies) and MASA (i.e., fast sequence alignment). MASA-StarPU was executed in two different multicore platforms and the results show that a bad choice of the scheduling policy may have a great impact on the performance. For instance, using 24 cores, the 5M x 5M comparison took 1484s with the *dmdas* policy whereas the same comparison took 3601s with *lws*. We also show that no scheduling policy behaves the best for all scenarios.**

***Index Terms*—Parallel sequence comparison, parallel programming environment, dynamic programming**

I. Introduction

In the last decades, we have observed an astonishing evolution of the sequencing methods, which allowed the rapid assembly of genetic sequences by a huge number of laboratories around the world. Even though this is clearly a great benefit, it created the so-called *data deluge* and, thus, genetic sequences are being produced in a rate that is much higher than the rate of their analysis [1].

One of the first steps in biological sequence analysis is pairwise sequence comparison, where a newly obtained sequence is compared to sequences which have been catalogued, in search of similarities. Smith-Waterman (SW) [2] is a well-known algorithm that provides the optimal result. It uses dynamic programming and has quadratic time and space complexities. This leads to high execution times when the sequences compared are long (Megabase comparison).

In order to accelerate sequence comparison algorithms, parallel platforms composed of multicores or multicores and accelerators such as GPUs (Graphics Processing Units), Intel Xeon Phis and FPGAs (Field Programmable Gate Arrays) have been used. CUDAlign 4.0 [3] is a tool that uses a variant of the SW algorithm to compare huge DNA sequences in GPUs. Its code has been re-structured into the MASA [4] architecture and now it runs in multicores, GPUs and Intel Xeon Phis with various programming environments (CUDA, OpenCL, OmpSs and OpenMP). In addition, MASA incorporates the block pruning capability, which accelerates considerably the execution when the sequences compared have a high degree of similarity. One drawback of block pruning is that the pruning behaviour is determined during execution and, for this reason, scheduling issues may occur [5].

StarPU [6] is a general-purpose task-based parallel programming environment which provides multiple task scheduling policies and runs in several parallel platforms. In StarPU, the programmer composes a graph of task dependencies and StarPU keeps track of the tasks that become ready. Then, the ready tasks are executed according to the selected task scheduling policy. Currently, StarPU offers more than 10 different scheduling policies [7].

Related work in the area of Megabase DNA sequence comparison show that impressive performance can be attained with accelerators [3] [8] [9]. The performance results for multicores (CPUs) are less impressive but this is the platform used at most Bioinformatics laboratories in developing countries and, for this reason, this is the target platform used in this paper. In the literature, very few CPU tools based on Smith-Waterman are able to align Megabase sequences. The popular Water tool (*www.ebi.ac.uk/Tools/psa/emboss_water*) restricts the sizes of the sequences to less than 1M. MASA-OpenMP and MASA-OmpSs are part of the MASA framework [4] and are able to align Megabase sequences, achieving the best performance of about 4.80 and 5.88 GCUPS (Billions of Cells Updated per Second), respectively, in a multicore platform with 12 cores when comparing two Megabase DNA sequences. ASW [10] is another tool able to align Megabase sequences, which obtained 7.2 GCUPS in a platform containing an APU (Accelerator Processing Unit) composed of 512 GPU cores and 4 CPU cores. In the CPU-only execution, ASW attained 0.46 GCUPS [10]. As far as we know, there is no Megabase DNA sequence comparison tool in the literature that supports multiple allocation policies.

This paper proposes and evaluates MASA-StarPU, a bio-

2643-3001/20/$31.00 ©2020 IEEE
DOI 10.1109/SBAC-PAD49847.2020.00039