Universidade Federal de Mato Grosso do Sul
Programa de Doutorado em Ciência da Computação da
Faculdade de Computação

SOLUÇÕES PARA OS PROBLEMAS DA SOMA MÁXIMA E DO K-ÉSIMO MENOR ELEMENTO DE UMA SEQUÊNCIA USANDO O MODELO BSP/CGM

Anderson Corrêa de Lima

Orientador: Prof. Dr. Edson Norberto Cáceres

Campo Grande, dezembro de 2015.

SOLUÇÕES PARA OS PROBLEMAS DA SOMA MÁXIMA E DO K-ÉSIMO MENOR ELEMENTO DE UMA SEQUÊNCIA USANDO O MODELO BSP/CGM

Anderson Corrêa de Lima

Tese de Doutorado apresentada à
Faculdade de Computação da
Universidade Federal de Mato Grosso do Sul
como parte dos requisitos necessários para obtenção
do título de Doutor em Ciência da Computação

Orientador: Prof. Dr. Edson Norberto Cáceres

Campo Grande, dezembro de 2015.

À minha mãe e irmãs, os pilares da minha vida.

Agradecimentos

Agradeço em primeiro lugar a Deus que iluminou o meu caminho durante esta caminhada. O Deus que esteve ao meu lado em todas as idas e vindas entre Campo Grande e Ponta Porã, nos meus primeiros anos do Doutorado, me acalmando em todas as intempéries climáticas e perigos da estrada. O Deus que guiou as mãos da equipe médica do Hospital Brasília, que recentemente salvou minha vida. O Deus que sempre me deu forças para continuar nos momentos mais difíceis e que em pensamento compartilhou comigo cada pequena vitória e alegria. O Deus que eu visualizei nos primeiros sorrisos do meu querido sobrinho.

Agradeço também à minha família por todo apoio e incentivo nesse longo processo, compreendendo a minha ausência e oferecendo acolhimento e suporte para nunca desanimar. À minha querida mãe Maria pelo carinho e pelos inúmeros momentos de cafés, onde relaxamos e conversamos sobre o dia e as pequenas coisas da vida. Momentos simples e únicos. As minhas duas irmãs, de personalidades tão distintas e que me completam. Laura e sua garra para a vida com inabalável fé e Ana Paula, o sinônimo de singeleza e alegria. Ao meu sobrinho Leonardo, que tem o dom de me fazer sorrir sempre.

Quero agradecer também ao meu orientador, Edson Norberto Cáceres, meu maior exemplo de professor. Um profissional curioso, que está sempre buscando novos conhecimentos. Agradeço a confiança depositada em meu trabalho. Sinto-me orgulhoso por ser seu primeiro orientando neste programa. Nunca esquecerei os diversos conselhos e os bons caminhos apontados em todas as situações que precisei. Obrigado Professor!

Aos meus amigos de hoje e de sempre, por terem me ouvido, me acolhido e sido pacientes nas etapas cruciais deste meu percurso. Deixo aqui o meu carinho e a minha admiração por cada um de vocês. Agradecimentos especiais para a Jucele, uma das melhores pessoas com as quais já convivi, o Cláudio, meu melhor amigo, corumbaense inabalável e professor exemplar. Ao amigo Will, pelos puxões de orelha e conselhos, que nos são sempre necessários. Ao Fernando, pelo incentivo e companheirismo, sempre me guiando para uma plena saúde física. Ao Roussian, meu amigo goiano e professor das primeiras lições técnicas do meu trabalho. Ao Samuel e ao Rodrigo, por todos os trabalhos e pela construção de conhecimento que juntos buscamos. Aos meus colegas de Doutorado, meu agradecimento pela convivência neste percurso cheio de desafios.

Aos meus amigos da Faculdade de Computação (FACOM) da UFMS pela receptividade e pelos divertidos momentos de amizade, dentro e fora do trabalho. Em especial para a Graziela, a Valéria e o Eraldo.

Aos meus colega de trabalho do Câmpus de Ponta Porã da UFMS, em especial ao diretor e amigo Amaury. Agradeço por terem aceitado o meu pedido de afastamento para o Doutorado. Agora, eu retorno ao trabalho tranquilo, com a sensação de dever cumprido e espero contribuir muito com o futuro de nossa instituição.

Obrigado.

The Climb

The struggles I'm facing. The chances I'm taking. Sometimes might knock me down, But no, I'm not breaking. I may not know it, But these are the moment that I'm gonna remember most and Just gotta keep going, And I, I got to be strong. Just keep pushing on, cause

There's always gonna be another mountain. I'm always gonna want to make it move. Always gonna be an uphill battle. Sometimes you're going to have to lose. Ain't about how fast I get there. Ain't about what's waiting on the other side. It's the climb.

Keep on moving. Keep running. Keep the faith. Baby. It's all about. It's all about the climb. Keep the faith. Keep your faith.

Jessi Alexander and Jon Mabe

Resumo

Neste trabalho são propostos algoritmos paralelos para os seguintes problemas: a subsequência de soma máxima, a submatriz de soma máxima, o hiper-retângulo de soma máxima e a seleção do k-ésimo menor elemento de um sequência não ordenada. Todos os problemas tratados possuem aplicações em diversas áreas da ciência, com destaque para biologia computacional, visão computacional, análise de volumes rochosos e de ordem. No projeto de nossos algoritmos adotamos uma extensão do modelo BSP/CGM de computação paralela e mostramos que, além do ambiente de memória distribuída, o modelo BSP/CGM também pode ser utilizado em arquiteturas com memória compartilhada e com múltiplos núcleos, tais como as GPUs. Diferentemente de soluções anteriores, nossos algoritmos e implementações utilizam novas estratégias na solução de cada problema. Apresentamos algoritmos paralelos para subproblemas relacionados ao problema da soma máxima, para os quais, de acordo com o nosso melhor conhecimento, a literatura não apresenta soluções no modelo BSP/CGM. As implementações foram construídas utilizando CUDA, MPI e OpenMP. Por fim, destacamos que nossos algoritmos são competitivos, quando comparados com as respectivas soluções sequenciais e paralelas já existentes.

Abstract

In this work we propose parallel algorithms for the following problems: the maximum subsequence sum, the maximum subarray sum, the maximum hyperrectangle sum and the selection of the k-th smallest element of an unsorted sequence. The problems treated have applications in many areas of science, such as bioinformatics, computer vision, rock analysis and order. In the design of our algorithms we adopted an extension of the BSP/CGM parallel computing model, showing that it can be used not only for distributed memory environments but also in architectures with shared memory and multiple cores, such as GPUs. Differently from previous solutions, our algorithms and implementations use new strategies for solving each problem. Besides, we also present parallel algorithms for maximum sum related problems, and to the best of our knowledge, there are no BSP/CGM solutions for this subproblems in the literature. All implementations were built using MPI, OpenMP and CUDA. Finally, we emphasize that our algorithms have achieved competitive performance speedups, compared to sequential and parallel solutions described in the literature.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

BSP/CGM - Bulk Synchronous Parallel/Coarse Grained Multicomputer
CRCW - Concurrent Read, Concurrente Write
CREW - Concurrent Read, Exclusive Write
CUDA - Compute Unified Device Architecture
CWD - Compute Work Distribution
DevIL- Developer's Image Library
DRAM - Dynamic Random-Access Memory
EREW - Exclusive Read, Exclusive Write
GPGPU - General Purpose Computation on Graphics Processing Units
GPU - Graphics Processing Unit
HPCVL - High Performance Computing Virtual Laboratory
HPC - High Performance Computing
ID - Identifier
MPI - Message Passing Interface
PRAM - Parallel Random-Access Machine
RAM - Random Access Memory
SIMD - Single Instruction Multiple Data
SM - Streaming Multiprocessor
SP - Streaming Processor
SPMD - Single Program, Multiple Data
VLSI - Very Large Scale Integration

CAPÍTULO 1

INTRODUÇÃO

Na última década, a criação de microprocessadores com múltiplos núcleos tornou-se uma tendência, principalmente quando a indústria percebeu que o aumento da frequência de processamento estava criando uma série de obstáculos, devidos entre outros fatores aos problemas de superaquecimento dos *chips* [28]. Seguindo esta tendência computadores em um contexto geral passaram a implementar processamento em múltiplas *threads* (*multithreading*).

Muitos dos sistemas operacionais modernos implementam o conceito de múltiplas *threads*. Nesses sistemas uma aplicação é descrita como um processo composto de várias *threads*. Existem inúmeras maneiras de incorporar *multithreading* em programas e muitas delas são de fácil compreensão. Processadores *multicore* são tipicamente compostos por dois, quatro, seis ou oito núcleos de processadores independentes conectados entre si por um barramento. Processadores *multicore* facilitam a execução de múltiplas *threads* simultaneamente, geralmente melhorando o desempenho nos processos de computação intensiva. Por sua vez, processadores *manycore* são aqueles em que o número de núcleos (*cores*) é tão grande que as técnicas para execução de programas tradicionais em ambiente *multicore* podem não ser eficientes. Geralmente estes processadores são voltados a aplicações mais específicas, que exigem abordagens diferenciadas para construção, execução e depuração de programas paralelos [28].

Hardwares paralelos compostos de múltiplos processadores já estão presentes nos principais equipamentos ou plataformas computacionais. Estações de trabalho, servidores, supercomputadores ou até mesmo sistemas embarcados já fazem uso desta arquitetura. A presença de mais processadores aumentou drasticamente o poder computacional destes dispositivos e isto possibilitou o tratamento de problemas, que há bem pouco tempo eram impraticáveis, tanto pelo tamanho da entrada, quanto pela complexidade envolvida. Novas abordagens paralelas, proporcionadas pelas novas arquiteturas de múltiplos núcleos, ofereceram então novas propostas para solução de antigos problemas. O desafio de explorar estas novas abordagens é muito pertinente, visto a possibilidade de ganho de tempo. Porém, antes de qualquer jornada neste caminho, é necessário compreender e utilizar as novas arquiteturas de multiprocessamento da melhor maneira possível. Pensar em paralelo, por si só exige maior esforço do programador. É preciso tratar de problemas de concorrência e sincronização, para os quais, a depuração é um processo bastante custoso e as vezes manual. Logo, percebe-se que mesmo com as arquiteturas *multi/manycore* sendo uma realidade, a programação paralela continua a ser uma tarefa complexa. Além da dependência e disponibilidade de ferramentas e ambientes de programação adequados para memória compartilhada ou

distribuída, enfrenta-se uma série de problemas não existentes em programação sequencial: código paralelo, concorrência, comunicação, sincronização, granulosidade e balanceamento de carga, estão entre eles. Ademais, podemos enumerar uma série de outros questionamentos acerca da migração de uma solução sequencial ou paralela distribuída para uma solução em arquitetura compartilhada *multi/manycore*. Por exemplo, pode-se questionar como se dará a escalabilidade, a eficiência, a reestruturação de código, a portabilidade e a depuração do sistema. Além disto, diferentes arquiteturas estabelecem diferentes formas e abordagens de paralelismo, desde o controle de acesso a dados em ambientes de memória compartilhada até a comunicação entre processos em ambiente de memória distribuída [28].

Como visto, são diversos os obstáculos, mas se não forem criados programas que façam uso dos recursos da computação paralela *multi/manycore*, o desperdício de energia continuará [28]. Uma vez que se opte em migrar a solução de um problema sequencial para uma solução em arquitetura paralela, seja ela compartilhada ou distribuída, um dos primeiros passos a seguir é a escolha do modelo de computação adequado. Modelos de computação são ferramentas muito importantes para o bom desenvolvimento de algoritmos. Em geral eles visam facilitar o trabalho de projetistas abstraindo diversas características das máquinas reais. Devido à grande variedade de arquiteturas paralelas, a utilização de um modelo de computação paralela é importante para consolidar a área de algoritmos paralelos. Da mesma forma que o modelo *Random Access Memory* (RAM) motivou o projeto e a análise de algoritmos sequenciais, nesta nova era é imprescindível um modelo que possibilite o projeto e a análise de algoritmos paralelos eficicientes [47]. A adoção de um modelo explícito de computação paralela para arquiteturas *multi/manycore* é fundamental para garantir o crescimento da área, que já é bastante acentuado.

É importante destacar que um modelo de computação paralela depende de um grande número de parâmetros, tais como: o número de processadores, as capacidades de memórias locais, esquemas de comunicação e os protocolos de sincronização. Isto faz com que o projeto, a avaliação e a análise de algoritmos paralelos sejam bem mais complexos do que no modelo sequencial. Vários modelos para o projeto de algoritmos paralelos já foram propostos, desde modelos teóricos como o modelo o *Parallel Random-Access Machine* (PRAM), até modelos chamados realísticos, como o *Bulk Synchronous Parallel/Coarse Grained Multicomputer* (BSP/CGM). Cada modelo apresenta vantagens dependendo da situação. O modelo de computação paralela BSP/CGM se mostrou bem adequado para o projeto de algoritmos paralelos que utilizam memória distribuída, entretanto a nova arquitetura *multi/manycore* resgata uma proximidade muito grande com o modelo PRAM, devido entre outros fatores, à utilização de memória compartilhada, à facilidade de comunicação e à forma de acesso e controle de dados da arquitetura.

Com isso, não só a solução de problemas em ambientes *multi/manycore* é fundamental, mas se a solução estiver associada a um modelo computacional e se o *speed-up* da implementação for compatível com a análise prevista no modelo, teremos um arcabouço teórico importante para o projeto e desenvolvimento de algoritmos paralelos.

1.1 Motivação e Descrição dos Problemas

O caminho para construir aplicações paralelas em arquitetura *multi/manycore* é desafiador. Muitas das soluções paralelas são de problemas facilmente paralelizáveis, problemas onde há pouca dependência entre as subtarefas a serem executadas. Um dos principais desafios da computação paralela é o de obter soluções para problemas onde há uma grande dependência

entre as subtarefas.

Valiant [46] propôs o modelo BSP, que juntamente ao modelo CGM [15] proporcionaram o desenvolvimento de algoritmos paralelos para ambientes de memória distribuída cujos *speed-ups* refletiam a análise e correção dos algoritmos nos respectivos modelos. Como esses modelos são muito parecidos, os algoritmos passaram a utilizar a terminologia BSP/CGM. Com a popularização das arquiteturas *multi/manycore*, Valiant propôs um novo modelo, o multi-BSP [47]. Neste nosso trabalho, para o projeto de algoritmos, utilizamos uma extensão do modelo BSP/CGM, considerando o fato de que no modelo CGM as rodadas de comunicação podem ser mapeadas com sincronizações ou movimentação entre as memórias na arquitetura *multi/manycore*. No capítulo 2, apresentaremos a extensão do modelo BSP/CGM que será utilizada no projeto e na implementação dos algoritmos deste trabalho.

Neste trabalho propomos soluções paralelas e implementações em um ambiente *multi/manycore* para problemas em que é muito forte a dependência entre as subtarefas e em que é necessário o projeto de um novo algoritmo. Nestes problemas, a simples divisão das tarefas entre os processadores e a utilização de algoritmos sequenciais conhecidos, geralmente não garantem a obtenção de bons *speed-ups*. Os problemas estudados são: (a) A subsequência unidimensional, bidimensional e d-dimensional de soma máxima e (b) a seleção do k-ésimo menor elemento. Abaixo descrevemos cada um dos problemas:

1. O problema da subsequência de soma máxima (1D) consiste em encontrar a subsequência contígua com maior soma em uma sequência de números reais. Caso a sequência seja composta somente de números não negativos, a solução é óbvia, mas o problema se torna mais interessante quando também aparecem números negativos. Abordamos o caso quando a sequência possui números negativos. O problema tem aplicações científicas, como em biologia computacional e em diversas situações corriqueiras, por exemplo, encontrar padrões de apostas consecutivas em jogos de azar ou encontrar uma sequência de assentos disponíveis em salas de cinema;

2. O problema da submatriz de soma máxima (2D) consiste em localizar a submatriz com maior soma dentre todas as possíveis submatrizes derivadas da matriz original. Também nesse caso, consideramos a existência de números negativos na matriz. O problema tem aplicações científicas em diversas áreas computacionais, com destaque para a visão computacional. Assim como na versão 1D, o histórico do problema descreve soluções sequenciais e paralelas eficientes [4, 35]. As últimas, tanto em memória compartilhada [35], utilizando o modelo PRAM, quanto em memória distribuída, com implementações sobre o modelo realístico BSP/CGM [2];

3. O problema do hiper-retângulo de soma máxima é uma extensão natural do problema da submatriz de soma máxima. Inicialmente, consideramos a inserção de uma nova dimensão (3D). Este problema também possui aplicações em diversas área da ciência, com destaque para aquelas que envolvem a visualização tridimensional de ambientes geográficos, tais como solos ou regiões oceânicas. Destacamos que as estratégias de solução deste problema podem ser replicadas na generalização com d-dimensões;

4. Para os problemas 1D e 2D, ainda ampliamos nosso objetivo de forma a solucionar os seguintes problemas relacionados: i) soma máxima com o maior número de elementos, ii) soma máxima com o menor número de elementos e iii) número de somas máximas. Particularmente, para a versão 2D, também buscamos soluções para o problema de somas máximas com maior ou menor densidade;

5. Considerando a importância de em alguns casos obter as k maiores somas máximas, também estudamos o problema de determinar o k-ésimo menor elemento de um conjunto de números reais (não ordenados) de tamanho n. Constituem-se em casos especiais deste problema a determinação do elemento mínimo ($k = 1$), do elemento máximo ($k = n$) e da mediana ($k = \lceil n/2 \rceil$) de um conjunto. Esse problema tem uma importância que transcende as somas máximas e pode ser utilizado na solução de vários outros problemas.

Os algoritmos paralelos que projetamos com a utilização da extensão do modelo BSP/CGM foram implementados usando a arquitetura *multi/manycore*. Para compararmos o desempenho, desenvolvemos também implementações usando memória distribuída.

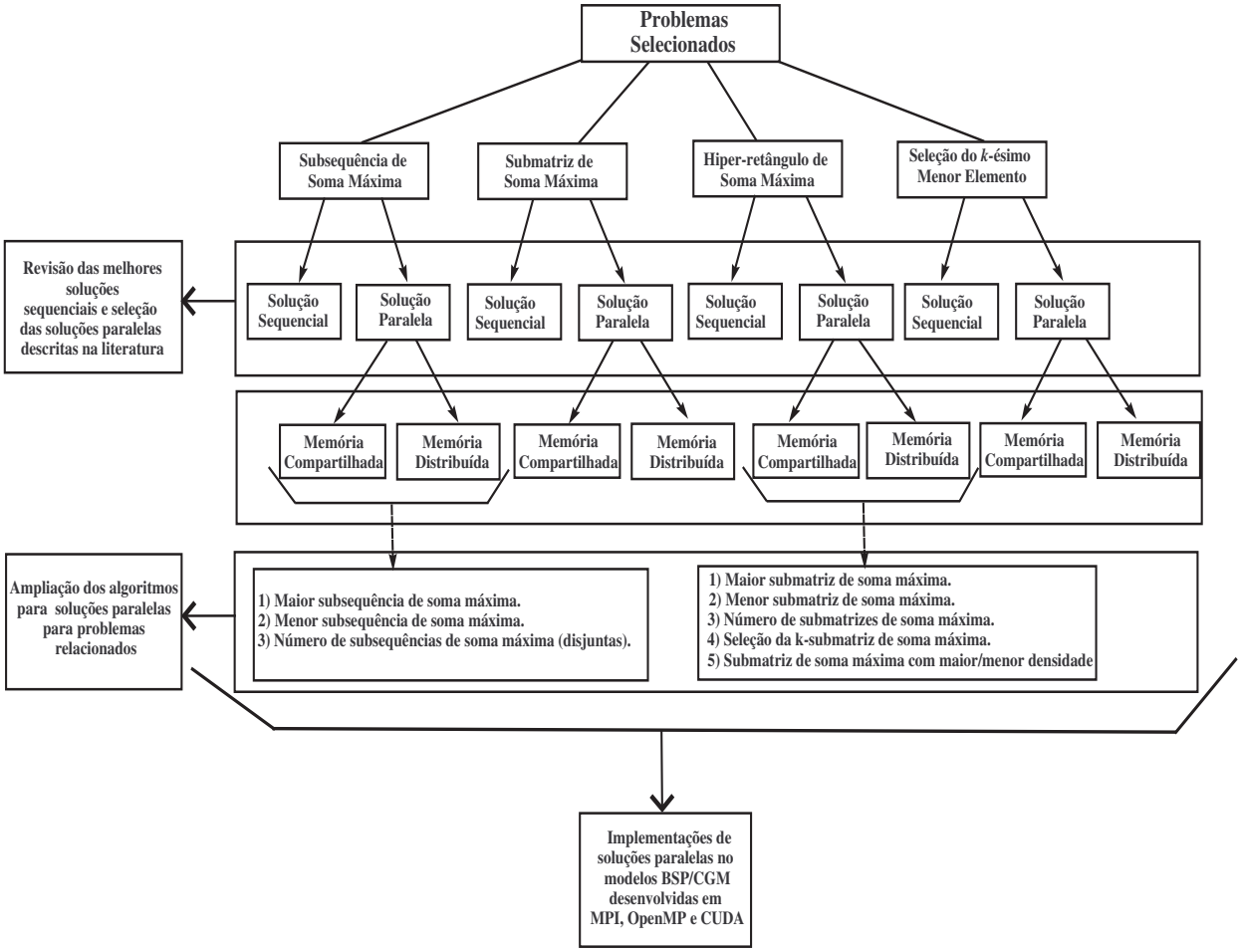


Figura 1.1: Linha de pesquisa e o objetivo geral.

A Figura 1.1 ilustra a linha de pesquisa deste trabalho e organiza os problemas abordados, alguns dos problemas relacionados e os objetivos principais.

1.2 Contribuições do Trabalho

Destacamos as principais contribuições advindas deste trabalho:

- A extensão do modelo BSP/CGM de computação paralela, como forma de projetar algoritmos em arquiteturas com muitos núcleos e memória compartilhada, tais como as GPUs;

- Algoritmos paralelos eficientes para os problemas de soma máxima com d-dimensões e problemas relacionados (soma máxima com o maior número de elementos, soma máxima com o menor número de elementos e número de somas máximas);
- Uma aplicação para a detecção de regiões mais/menos brilhantes em uma imagem. Essa aplicação pode ser utilizada no diagnóstico por imagens;
- Algoritmos paralelos para o problema da seleção do k-ésimo elemento de uma sequência;
- Implementações com desempenho competitivo usando uma arquitetura *multi/manycore*. Para podermos comparar de forma mais efetiva o desempenho, os algoritmos também foram implementados no modelo de memória distribuída;
- Destacamos que os algoritmos paralelos para os problemas relacionados com soma máxima, ao nosso conhecimento, são inéditos. Além disto, as soluções propostas no ambiente *multi/manycore* para os problemas tratados se mostraram competitivas, em comparação com as soluções apresentadas até o momento;

1.3 Estrutura do Trabalho

O restante desta tese está organizado da seguinte forma: No **Capítulo 2** são apresentados os conceitos e fundamentos de computação paralela necessários para compreensão dos próximos capítulos. Destacamos a proposta da extensão do modelo BSP/CGM para o projeto de algoritmos paralelos em arquiteturas *multi/manycore*. Ainda neste capítulo são descritas algumas das principais linguagens para implementação de programas em ambientes com arquiteturas de memória compartilhada e distribuída. No **Capítulo 3** é apresentado o problema da subsequência de soma máxima, com foco em sua definição, aplicações e no histórico de soluções sequenciais e paralelas presentes na literatura. Ainda neste capítulo são apresentados os algoritmos que projetamos, para o problema geral e para os problemas relacionados, acompanhados de seções de resultados e conclusões locais. Os **Capítulos 4** e **5** seguem a mesma estrutura do capítulo **3**, abordando, respectivamente, dois outros problemas, que são extensões do problema da subsequência de soma máxima: a submatriz de soma máxima e o hiper-retângulo de soma máxima. O **Capítulo 6** apresenta três soluções para o problema da seleção do k-ésimo elemento de uma sequência. No **Capítulo 7** apresentamos as conclusões do trabalho e apresentamos possibilidades de trabalhos futuros. Por fim, são apresentadas as referências bibliográficas e o apêndice com os principais códigos dos algoritmos desenvolvidos.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo apresentamos a base teórica e conceitual necessária para compreensão do trabalho. Discutimos inicialmente os princípios de computação paralela. Em seguida, apresentamos resumidamente os mais conhecidos modelos de computação paralela, os respectivos ambientes de desenvolvimento e algumas das linguagens mais utilizadas para programação paralela. Por fim, discutimos as razões de escolha do modelo selecionado.

2.1 Computação Paralela e de Alto Desempenho

Segundo Ian Foster, um computador paralelo é um conjunto de processadores interconectados, capazes de trabalhar de forma simultânea e cooperativa a fim de resolver um mesmo problema computacional [20]. Essa definição é suficientemente ampla para incluir supercomputadores paralelos, que englobam centenas ou milhares de computadores, redes locais de computadores e estações de trabalho com múltiplos processadores. A ideia principal da computação paralela consiste em executar tarefas simultaneamente. O número de tarefas pode variar durante a execução do programa. Uma tarefa encapsula um programa sequencial em memória local. Além disso, um conjunto de canais de entrada e de saída delimitam a interface com o ambiente. Uma tarefa pode realizar quatro ações básicas, além de ler e escrever em sua memória local: enviar mensagens, receber mensagens, criar novas tarefas e terminar. Canais podem ser criados e excluídos, e as referências aos canais (portas) podem ser incluídos em mensagens, de modo que a conectividade pode variar dinamicamente [20].

Mais amplamente, um sistema de computação paralela depende de um grande conjunto de parâmetros, tais como a quantidade de processadores, o tamanho das memórias locais, o esquema de comunicação e os protocolos de sincronização, tornando o projeto, a avaliação e a análise de algoritmos paralelos mais complexos do que no modelo sequencial. O tempo de execução de um algoritmo paralelo não depende apenas do tamanho da entrada de dados, mas também da arquitetura utilizada e da quantidade de processadores utilizados.

O processamento paralelo proporciona um aumento de capacidade e velocidade de processamento. Para quantificar esse aumento, utilizam-se diferentes parâmetros, entre eles desempenho (*speedup*) e eficiência: seja T_1 o tempo gasto pelo melhor algoritmo sequencial para resolver um determinado problema e T_p o tempo gasto por um algoritmo paralelo utilizando p processadores para resolver o mesmo problema. A **performance** (*speedup*) (S_p) representa

quão mais rápido é o algoritmo paralelo em relação ao sequencial:

$$S_p = T_1/T_p.$$

O caso ótimo ocorre quando $S_p = p$, ou seja na medida que aumentamos o número de processadores, aumenta-se diretamente a velocidade de processamento. Algumas vezes o *speedup* fica acima de p, é o chamado *speedup* superlinear. Essa situação pode ocorrer em função de características do ambiente paralelo que podem não estar presentes no ambiente sequencial.

A **eficiência** do algoritmo mede a utilização dos p processadores e é definida por:

$$E_p = S_p/p.$$

Como normalmente o *speedup* é menor que p, a eficiência, nestes casos, fica entre 0 e 1. Um valor de E_p aproximadamente igual a 1, para algum p, indica que o algoritmo paralelo executa aproximadamente p vezes mais rápido usando p processadores, do que o faria usando apenas 1 processador. Dificilmente obtém-se eficiência igual a 1, pois há perdas na paralelização do algoritmo, com sobrecarga de comunicação e sincronização entre os processadores.

Como vimos anteriormente, um objetivo da computação paralela é aumentar a velocidade de processamento ou até mesmo viabilizar as simulações numéricas de problemas físicos e com elevado grau de discretização, que são muito custosas, ou impossíveis de serem executadas em máquinas monoprocessadas. Ao longo dos anos, o desenvolvimento científico tem exigido das simulações numéricas resultados cada vez mais confiáveis e próximos da realidade. Dessa forma, para acompanhar esses avanços, a computação científica se depara com o desafio de superar as diversas barreiras que surgem em virtude da limitação física dos computadores, principalmente relacionadas ao super consumo e a emissão de calor. É alta a demanda por processamento numérico, armazenamento de dados, visualização, entre outros. Nesse contexto, a computação de alto desempenho (*High Performance Computing* - HPC), termo amplamente utilizado na computação científica, tem se apresentado como uma importante e atual frente de pesquisa. O termo pode ser definido como qualquer conjunto de técnicas que visem otimizar ou até mesmo viabilizar o processamento de simulações numéricas [43]. Podemos citar como uma dessas técnicas o uso de processamento paralelo, em *clusters* e supercomputadores, que consiste na divisão de uma tarefa em tarefas menores e na execução de cada uma destas tarefas em diferentes processadores. A comunidade de computação de alto desempenho tem desenvolvido programas paralelos há décadas. No entanto estes só funcionavam em computadores de grande escala, muito caros, nos quais apenas algumas aplicações de elite podiam justificar o seu uso, limitando, desse modo, a prática de programação paralela a um pequeno número de desenvolvedores de aplicações. Além disto, desenvolver *software* em programação paralela para uma ambiente de processamento paralelo é uma tarefa de programação árdua e bastante desafiadora, pois é necessário um considerável esforço intelectual.

Uma outra fonte de problemas que exige a utilização de computação de alto desempenho é a solução de problemas combinatórios, onde o tamanho real da entrada e a complexidade inviabilizam a solução por meio de algoritmos sequenciais. Em alguns casos, a solução sequencial é viável, mas o tempo utilizado extrapola o que é exigido pela aplicação, como por exemplo o cálculo do menor caminho numa aplicação de auxílio a um motorista numa grande cidade.

2.1.1 Projeto de Algoritmos Paralelos

Ao projetar uma aplicação paralela, é preciso analisar muito bem a arquitetura a ser utilizada. Restrições de tamanho e compartilhamento de memória, além do custo de troca de mensagens têm grande influência. Três aspectos devem ser considerados: granulosidade das tarefas, comunicação e sincronismo necessários para realizar as tarefas e balanceamento de carga entre os processadores [20]. Para paralelizar a solução de um problema, deve-se encontrar a melhor forma de dividir suas tarefas entre os processadores, atentando para o desempenho final. Ian Foster sugere uma metodologia para a paralelização de problemas. Esta metodologia é estruturada em quatro estágios [20]:

1. Particionamento: busca verificar possibilidades para a execução paralela da solução através de sua decomposição em termos de dados e de computação;
2. Comunicação: define como se dará a comunicação entre as partes definidas no particionamento, visto que a decomposição realizada pode gerar partes dependentes entre si, que necessitam de comunicação para coordenar as tarefas;
3. Agregação: avalia os requisitos de desempenho e os custos de implementação da proposta de tarefas gerada através da definição de particionamento e comunicação; Caso seja necessário, as tarefas podem ser combinadas em tarefas maiores a fim de melhorar os resultados;
4. Mapeamento: especifica que tarefa será executada em cada processador de maneira a satisfazer os objetivos de utilização máxima de processadores e minimização de custos com comunicação.

2.1.2 Desempenho de Algoritmo Paralelos

O desempenho de um algoritmo paralelo em termos de custo, consiste de uma análise de seu pior caso: Seja Q um problema para o qual temos um algoritmo paralelo que executa em tempo $T(n)$ usando $P(n)$ processadores, para uma instância de tamanho n. O produto $C(n) = T(n) \times P(n)$ representa o custo do algoritmo paralelo. Nota-se que um algoritmo paralelo pode ser convertido em um algoritmo sequencial de custo $\mathrm{O}\big(C(n)\big)$, neste caso dispomos de um único processador capaz de simular os $P(n)$ processadores em $\mathrm{O}\big(P(n)\big)$ para cada um dos $T(n)$ passos paralelos. Existem diversas formas de se medir e representar o custo final de um algoritmo paralelo. As duas mais comuns são:

1. $P(n)$ processadores em tempo $T(n)$;
2. $C(n) =$ custo $P(n) \times T(n)$ em tempo $T(n)$.

2.2 Linguagens de Programação Paralela

Existe um grande número de linguagens para programação de sistemas paralelos e distribuídos. Estas linguagens são baseadas em uma grande variedade de paradigmas de programação, tais como troca de mensagens, objetos concorrentes, lógica e programação funcional. Existe também muita discussão sobre qual paradigma é o melhor. Geralmente

são acrescentadas às linguagens de programação (C, C++ e Fortran) algumas bibliotecas que permitem a execução concorrente de processos e, principalmente, a troca de mensagens (dados) e a sincronização entre os processos em execução no ambiente.

2.2.1 A Linguagem MPI

A linguagem MPI (*Message Passing Interface*) surgiu como uma tentativa de padronização, independente do sistema paralelo, para ambientes de troca de mensagens. Ele baseia-se nas melhores características de todas as bibliotecas de trocas de mensagens, levando-se em consideração as características gerais dos sistemas paralelos, tentando explorar as vantagens de cada um deles. Os programas em MPI possuem o que é chamado de estilo SPMD *(Single Program, Multiple Data)*, ou seja, cada processador executa uma cópia do mesmo programa. Cada instância do programa pode determinar a sua própria identidade e, dessa forma, executar operações distintas. As instâncias interagem através das funções da biblioteca MPI. MPI define um conjunto de rotinas, que oferecem serviços de criação, execução, comunicação e sincronização de processos. A forma de execução de um programa MPI depende da implementação que está sendo construída. A execução é realizada através de linhas de comando, por via de comandos específicos e além disto MPI inclui uma variedade de funções para comunicação e sincronização. Estas funções permitem a interação entre todos os membros de um grupo de processos iniciados [28].

2.2.2 A Linguagem OpenMP

O padrão OpenMP é desenvolvido e mantido pelo grupo OpenMP ARB *Architecture Review Board*, formado pelos maiores fabricantes de *software* e *hardware* do mundo, tais como SUN *Microsystems*, SGI, IBM, Intel, dentre outros, que, no final de 1997, reuniram esforços para criar um padrão de programação paralela para arquiteturas de memória compartilhada. O Open significa que é padrão e está definido por uma especificação de domínio público e MP são as siglas de *Multi Processing*. O OpenMP consiste em uma API e um conjunto de diretivas que permite a criação de programas paralelos com compartilhamento de memória por meio da implementação automática e otimizada de um conjunto de *threads*. Suas funcionalidade podem atualmente ser utilizadas nas linguagens Fortran 77, Fortran 90, C e C++. A primeira versão desse padrão, o OpenMP 1.0, foi introduzida ao público no final de 1997. Em 2000, para Fortran, e em 2002, para C/C++, foi publicada a versão 2.0. Atualmente, o OpenMP encontra-se na versão 2.5 [43].

O OpenMP não é uma linguagem de programação, ele representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação, nos programas sequenciais, de diretivas que indicam como o trabalho será dividido entre os processadores ou *cores*. Dessa forma, muitas aplicações podem tirar proveito desse padrão com pequenas modificações no código. Mesmo podendo ser usado em máquinas monoprocessadas, o OpenMP foi planejado para satisfazer a implementação em larga escala das arquiteturas SMPs. O sucesso do OpenMP é propagado pelas arquiteturas *multicore* e *multithread* e pode ser atribuído a fatores como a estrutura de apoio robusta para programação paralela e a facilidade do seu uso. No OpenMP, a paralelização é explicitamente realizada com múltiplas *threads* dentro de um mesmo processo. A criação de *threads* é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas simultaneamente. Cada *thread* possui sua própria pilha de execução, porém compartilha o mesmo endereço de memória com as outras *threads* do mesmo processo (enquanto que cada processo possui seu próprio espaço de memória). O

modelo de programação inicia com uma única *thread* que executa sozinha as instruções até encontrar uma região paralela (identificada pela diretiva), ao encontrar essa região ela cria um grupo de *threads*, que juntas executam o código dentro dessa região. Quando as *threads* completam a execução do código na região paralela, elas sincronizam-se e somente a *thread* inicial segue na execução do código até que uma nova região paralela seja encontrada ou que o programador decida encerrar essa *thread*. Esse modelo de programação é conhecido na literatura como *fork-join*. Seguem algumas vantagens do uso do OpenMP [43]:

- Normalmente são feitas poucas alterações no código serial existente;
- Possui uma robusta estrutura para apoio à programação paralela;
- Fácil compreensão e uso das diretivas;
- Apoio a paralelismo aninhado;
- Possibilita o ajuste dinâmico do número de *threads*.

2.2.3 Modelo de Programação GPGPU/CUDA

As *Graphics Processing Units* (GPUs) são dispositivos criados originalmente para o tratamento de imagens e vídeos, que requerem o processamento de um grande volume de dados. As GPUs geralmente executam um grande número de instruções em vários núcleos, ou *cores*, ultrapassando a vazão de um único processador principal em várias vezes, mesmo geralmente possuindo uma taxa de *clock* muito menor e seus *cores* sendo muito mais simples. Instruções matemáticas são predominantes em execuções em GPUs, nestes casos a maioria das operações são instruções aritméticas e os dados (processados por diferentes núcleos) são independentes. O termo GPGPU (*General Purpose Computation on Graphics Processing Units*) foi cunhado para que se pudesse aproveitar a natureza de uma GPU para resolver problemas gerais, e não somente problemas de geração de imagens. Para que a execução de um problema tenha um bom desempenho em uma GPGPU os seguintes itens devem ser satisfeitos:

- A existência de um grande volume de dados, visto que as GPUs trabalham com bilhões de *pixels* por segundo e cada *pixel* é submetido a centenas de operações aritméticas. Problemas cujo cálculo da solução é lento devido à alta complexidade de execução de seus algoritmos, e não à quantidade de dados, não são bons candidatos a serem executados em uma GPGPU já que possivelmente não serão capazes de utilizar a alta capacidade de paralelização deste *hardware*;
- A independência de dados, pois na geração de uma cena, por exemplo, os vértices podem ser calculados independentemente, evitando dependências de execução, ou seja, esperar que um certo cálculo seja realizado para outro poder executar;
- A escalabilidade, visto que se mais núcleos forem adicionados ao *hardware*, melhor desempenho o programa terá;
- A vazão em uma GPGPU, visto que nelas os processadores utilizam *pipelines*. Como consequência, a resolução de um conjunto de cálculos pode levar um tempo considerável. Porém vários conjuntos de cálculos são realizados em um determinado período de tempo, devido ao processamento paralelo. Problemas que serão executados em uma GPGPU devem priorizar uma alta vazão de resultados, e não uma baixa latência.

Para que um problema seja executado em uma GPGPU, o programador deve separar pequenos trechos de código, que podem ser executados paralelamente e escrevê-los em uma linguagem própria. Para este fim a empresa NVIDIA criou a linguagem CUDA. CUDA é similar à linguagem C, nela ocorre a separação de trechos de códigos que serão executados na CPU ou GPU. As GPUs da NVIDIA com capacidade de executar código CUDA estão divididas em várias versões, conforme a arquitetura das GPUs foi evoluindo. Estas versões são chamadas de *Compute Capability* ou capacidade de computação. As primeiras GPUs com CUDA tinham *Compute Capability* (ou C.C) 1.0 e eram bem limitadas. Hoje temos placas com CC 5.0, com características muito mais sofisticadas do que as anteriores e as últimas com CC 5.3.

O CUDA (*Compute Unified Device Architecture*) mais que uma linguagem de computação paralela, trata-se de uma arquitetura de *software* e *hardware* criada pela NVIDIA, capaz de executar computações em GPUs NVIDIA sem a necessidade de utilizar APIs gráficas [10]. CUDA é uma extensão das linguagens de programação C e C++. CUDA utiliza o conceito de *manycore*, aproveitando-se de um número elevado de núcleos para realizar a computação. Compiladores especializados transformam o algoritmo paralelo em instruções específicas da GPU, que trabalham sobre o *pipeline* gráfico.

Os aspectos de CUDA são semelhantes a MPI e OpenMP no sentido de que o programador gerencia as construções de código paralelo. Embora os compiladores OpenMP façam mais automação no gerenciamento da execução paralela, CUDA estende a sintaxe de declaração de função em C para dar apoio à computação paralela heterogênea. Por meio das funções qualificadores $\_\_global\_\_$, $\_\_device\_\_$ ou $\_\_host\_\_$, um programador CUDA pode instruir o compilador para gerar uma função de *kernel*, uma função de *device* ou uma função de *host* [28]:

- Qualificador $\_\_global\_\_$: As funções com este qualificador são executadas no dispositivo, mas elas podem ser chamadas somente a partir do *host*;
- Qualificador $\_\_device\_\_$: As funções com este qualificador são executadas no dispositivo. Estas funções podem ser chamadas apenas a partir do dispositivo;
- Qualificador $\_\_host\_\_$: As funções com este qualificador são executadas no *host* e podem ser chamadas somente a partir do *host*.

Se a função não tiver qualquer palavra-chave de extensão CUDA, a função é considerada uma função de *host* como padrão. A execução de um programa típico CUDA se inicia no *host* (CPU). A CPU prepara os dados. Quando uma função do *kernel* é invocada, ou disparada, a execução é passada para um *device* (GPU), no qual um número maior de *threads* é gerado para tirar proveito do abundante paralelismo de dados. Isto é realizado por meio de um conjunto de instruções específicas. Durante este processo é realizada a transferência dos dados da memória principal para a memória da GPU. Em seguida são inicializados os programas de computação paralela (*kernel*), que são executados de forma assíncrona. Ao final da computação, os dados são trazidos da memória da GPU para a memória principal para geração do resultado [28]. Como visto, um *kernel* é executado em paralelo por meio de um conjunto de *threads* paralelas. As *threads* são organizadas em uma hierarquia de *grids* de blocos de *threads*, como ilustrado na Figura 2.1.

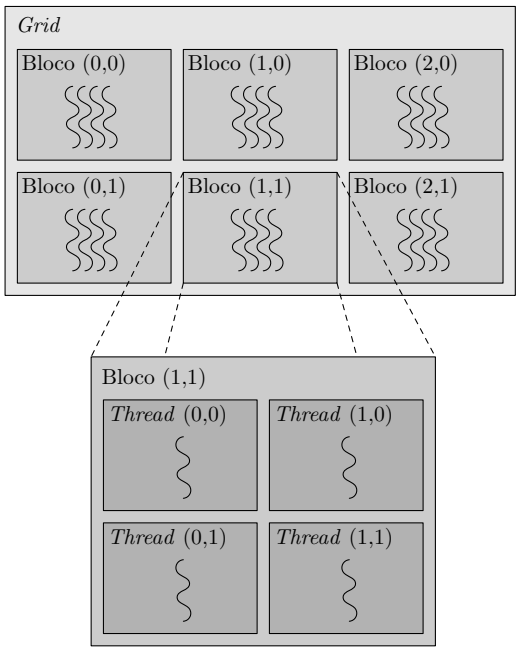


Figura 2.1: Organização das *threads* no modelo de programação CUDA [28].

As seguintes definições são importantes:

- Bloco de *threads*: Conjunto de *threads* concorrentes, que podem cooperar entre si por meio de uma barreira de sincronização e de uma memória compartilhada pertencente ao bloco;
- *Grid*: Um conjunto de blocos de *threads* que podem ser executados concorrentemente. Não há dependência entre blocos.

A independência entre os blocos de *threads* ajuda a evitar possíveis *deadlocks* e permite que os mesmos possam ser agendados em qualquer ordem, com qualquer quantidade de *cores*, isto assegura a escalabilidade do modelo e mantém um apoio eficiente ao paralelismo de granulosidade fina. CUDA também não estabelece restrições sobre a localização de cada *core*, assim programas CUDA podem se tornar escaláveis de maneira transparente e por meio de múltiplos dispositivos [16]. O código apresentado a seguir, nesta subseção, exemplifica em CUDA a soma de dois vetores, com 102.400 elementos cada um. O código inicia executando no sistema *host*. Em seguida o *host* transmite os dados e invoca um *kernel* para a execução do soma dos dois vetores em paralelo na GPGPU CUDA. Após concluir o processamento a GPGPU escreve o resultado em sua memória global. O *host* pode então acessar este resultado. Este exemplo é modelado por um *grid* unidimensional com 100 blocos de *threads*, que por sua vez possuem 1024 *threads* cada um. Em outras palavras, o problema é dividido em 100 partições independentes que podem ser executadas concorrentemente. As *threads* de um mesmo bloco calculam os 1024 elementos de cada partição de forma concorrente, visto que são operações independentes.

```
int main(int argc, char* argv[])
{
/* Tamanho dos vetores */
int n = 102400;
```

```
/* Aloca memória para os vetores no host */
double *h_a = (double *) malloc(n * sizeof(double));
double *h_b = (double *) malloc(n * sizeof(double));
double *h_c = (double *) malloc(n * sizeof(double));
/* Aloca memória para os vetores na GPGPU */
double *d_a, double *d_b, *d_c;
cudaMalloc(&d_a, n * sizeof(double));
cudaMalloc(&d_b, n * sizeof(double));
cudaMalloc(&d_c, n * sizeof(double));
/* Inicializa os vetores no host */
...
/* Copia os vetores para a GPGPU */
cudaMemcpy(d_a, h_a, n * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, n * sizeof(double), cudaMemcpyHostToDevice);
/* Invoca um kernel com 100 blocos de 1024 threads cada um */
somaVetores<<<100, 1024>>>(d_a, d_b, d_c, n);
/* Copia resultados da GPGPU para o host */
cudaMemcpy(h_c, d_c, n * sizeof(double), cudaMemcpyDeviceToHost);
/* Imprime resultados */
...
/* Libera a memória alocada para os vetores na GPGPU */
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
/* Libera a memória alocada para os vetores no host */
free(h_a); free(h_b); free(h_c);
return(0);
}


/* Kernel CUDA - Cada thread calcula um elemento do vetor 'c' */
  global   void somaVetores(double *a, double *b, double *c, int n)
{
/* Calcula o ID da thread */
int id = blockIdx.x * blockDim.x + threadIdx.x;
/* Verifica se o limite do vetor não foi ultrapassado */
if (id < n)
c[id] = a[id] + b[id];
}
```

Sincronização

Um programa CUDA pode executar vários *grids* de forma independente ou dependente. *Grids* independentes podem executar concorrentemente se houver recursos de *hardware* suficientes. Por sua vez, *grids* dependentes executam sequencialmente, com uma barreira implícita inserida entre eles, garantindo assim que todos os blocos do primeiro *grid* terminarão de executar, antes que qualquer bloco do próximo *grid* seja executado. A execução em paralelo e o gerenciamento das *threads* é feito de forma automática. Toda criação, escalonamento e finalização de *threads*, geralmente é realizado pelo *hardware* e não pelo programador. As *threads* de um bloco são executadas concorrentemente e podem sincronizar em uma barreira por meio do comando *__syncthreads()*. Garante-se assim que nenhuma das *threads*, participantes da barreira, pode prosseguir até que todas as outras também atinjam a barreira. Assegura-se

assim, a todas as *threads*, a visualização e o acesso de tudo que é escrito na memória. *Threads* em um bloco podem se comunicar através da escrita e leitura na memória compartilhada e com a ajuda da barreira de sincronização são evitadas condições de corrida [10].

Arquitetura das GPGPUs CUDA

A arquitetura de dispositivos massivamente paralelos como as GPGPUs é caracterizada fundamentalmente por blocos físicos denominados SMs (*Streaming Multiprocessors*). Cada SM possui vários *cores* denominados SPs (*Streaming Processors*). SMs são blocos físicos *multithreads* [10]. A Figura 2.2 ilustra a arquitetura de uma GPGPU com n SMs e 8 SPs em cada um, assim cada SM possui 8 *cores* SIMD (*Single Instruction Multiple Data*) interconectados por 4 porções externas de memória DRAM (*Dynamic Random-Access Memory*). O uso do modelo SIMD significa que cada *core* executa a mesma instrução, contudo, cada *core* pode usar dados diferentes [16].

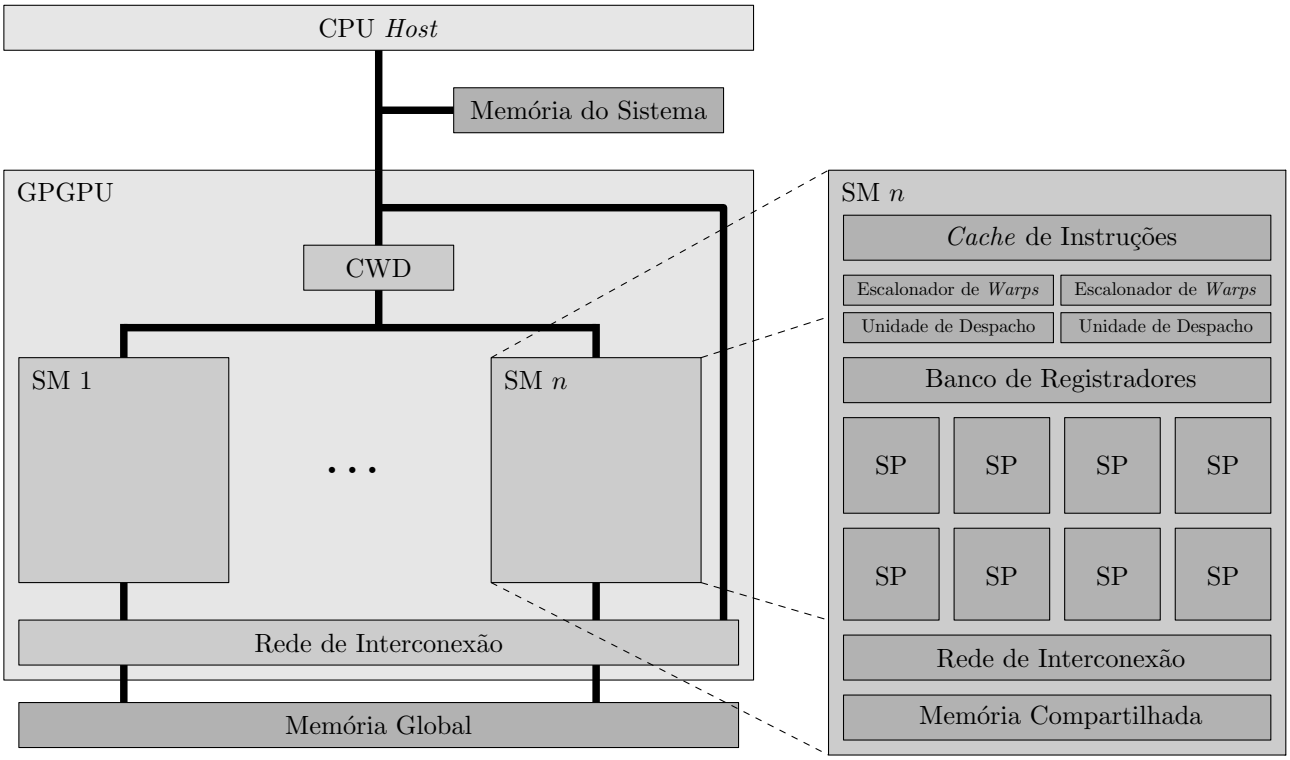


Figura 2.2: Arquitetura de uma GPGPU CUDA [10].

Um *warp* é a unidade básica para o escalonamento de trabalho dentro de um SM e na maioria das GPGPUs é um bloco de 32 *cores* SIMD. Uma vez que cada *warp* é por definição um bloco de *cores* SIMD, o escalonador não precisa verificar por dependências no fluxo de instruções. Na Figura 2.2 é possível identificar dois escalonadores de *warps* e duas *dispatch units*, que permitem que dois *warps* possam ser despachados e executados concorrentemente. Cada *warp* iniciará executando uma instrução, não necessariamente a mesma. Cada SM é responsável pelo escalonamento dos seus recursos internos, a cada *tick* de *clock* o escalonador de *warp* do SM decide qual *warp* executar em seguida [16]. Quando um programa CUDA no sistema *host* invoca um *kernel*, a unidade CWD (*Compute Work Distribution*) enumera os blocos de *threads* do grid e inicia a distribuição deles aos SMs com capacidade de execução disponíveis. As *threads* de um bloco executam concorrentemente em um SM. Ao fim da execução de um bloco, a unidade CWD atribui um novo bloco ao SM vago.

Como visto, em cada bloco existe um conjunto de *threads* que cooperam entre si e apenas *threads* de um mesmo bloco podem compartilhar dados. Esses fatos permitem uma abstração natural, de como é realizado o mapeamento do paralelismo de um *kernel* em uma GPGPU: Efetua-se o particionamento de um problema em múltiplas partições independentes, que podem ser executadas concorrentemente. O escalonador oferece a cada SM um ou mais blocos para serem executados, de acordo com a limitação do *hardware* da GPGPU [16].

A arquitetura das GPGPUs é fator chave na escalabilidade desses dispositivos. Adicionando-se mais SMs em uma GPGPU pode-se executar mais tarefas ao mesmo tempo ou executar uma mesma tarefa mais rapidamente, se ainda houver paralelismo a ser explorado [10]. Logo, aumentando ou diminuindo a quantidade de SMs em uma GPGPU, os fabricantes conseguem atender as necessidades do mercado [16].

Capacidade de Computação

A capacidade de computação de uma GPU NVIDIA é representada por um número com o formato $X.Y$, onde X indica a versão da arquitetura e Y a versão das melhorias realizadas nesta arquitetura [28]. As diferenças entre as capacidades de computação são evidentes principalmente nos seguintes aspectos:

- Quantidade de SPs;
- Quantidade e capacidade das unidades de operações aritméticas;
- Quantidade de escalonadores de *warp*;
- Tamanho da memória local;
- Quantidade de bancos na memória local;
- Requisitos necessários para a coalescência de memória global;
- Existência de cache de DRAM.

Como a capacidade de computação é definida pela arquitetura do dispositivo, ela também determina os requisitos necessários para que algumas otimizações possam ser aplicadas.

Hierarquia de Memórias GPGPU CUDA

A hieraquia de memórias em uma GPGPU CUDA se apresenta da seguinte forma:

- Em cada SM há um barramento próprio de acesso à memória global. As memórias de textura e a memória constante são apenas abstrações dentro da memória global;
- Cada bloco tem uma memória compartilhada e visível por todas as *threads* do bloco. As *threads* tem o mesmo tempo de vida que o bloco;
- As *threads* podem acessar dados de vários locais de memória durante o seu tempo de execução. Cada *thread* tem uma memória local privada. CUDA utiliza essa memória na declaração de variáveis privadas da *thread* que não se encaixam nos registradores da mesma. Todas as *threads* têm acesso à mesma memória global. Atualmente a memória global está na ordem de *gigabytes*.

A Figura 2.3 esboça a hierarquia de memórias de uma GPGPU CUDA.

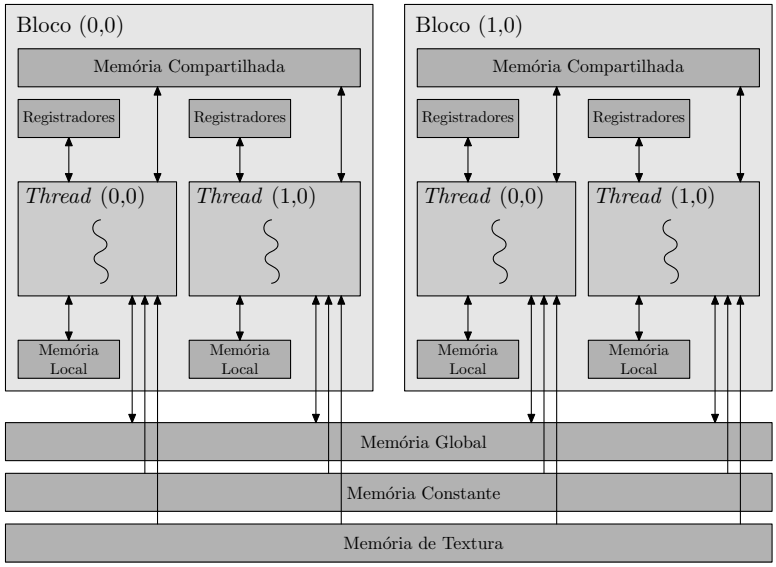


Figura 2.3: Hierarquia de memória em uma GPGPU CUDA [10].

Como visto, programas em CUDA declaram variáveis na memória compartilhada e na memória global por meio dos qualificadores *__shared__*e *__device__*, respectivamente. Esses locais correspondem a memórias fisicamente separadas: a memória compartilhada por bloco possui baixa latência, enquanto a memória global possui uma latência maior e está acessível a todas as *threads* de todos os blocos [34]. Para programas gráficos, a memória global armazena imagens de vídeos e informações de texturas para renderizações 3D. Já para aplicações computacionais não gráficas, a memória global pode ser utilizada como memória de alta largura de banda, embora com uma maior latência que a memória do sistema *host*. Para aplicações massivamente paralelas a alta largura de banda sobressai-se sobre a longa latência [29]. De maneira mais simples, a memória compartilhada desempenha um papel semelhante a *cache* L1 de uma CPU, dada sua baixa latência e por estar próxima às *threads*. Ela pode prover uma comunicação de alta velocidade e compartilhamento de dados entre as *threads* de um mesmo bloco. Visto que esta memória tem o mesmo tempo de vida do seu bloco correspondente, o código do *kernel* normalmente inicializa dados em variáveis compartilhadas, faz cálculos usando variáveis compartilhadas e copia os resultados para a memória global. Blocos de *threads* pertencentes a grids dependentes também utilizam a memória global, neste caso, para se comunicarem usando-a para leitura e escrita de resultados [34].

Escalabilidade do Modelo de Programação CUDA

A simplicidade do particionamento do trabalho de computação realizado por um programa em blocos de tamanho fixo, durante a modelagem do problema, garante o do desempenho e da escalabilidade do modelo de programação CUDA.

Os blocos realizam o mapeamento entre o paralelismo do problema em questão e o *hardware* da GPGPU [16]. Como as *threads* de um bloco podem compartilhar memória local e se sincronizam através de barreiras, elas devem residir no mesmo SM. Contudo, o número de blocos pode exceder a capacidade dos SMs de uma GPGPU. Dessa forma, os elementos de processamento podem ser virtualizados, o que permite ao programador flexibilidade para paralelizar a granulosidade da forma que lhe parecer mais conveniente. Isso permite a decomposição de problemas de forma intuitiva, uma vez que o número de blocos pode ser ditado pelo tamanho dos dados a serem processados e não mais pela quantidade de processadores no

sistema. Portanto, um mesmo programa CUDA é escalável em GPGPUs com uma quantidade arbitrária de SMs, com isso propicia-se o ganho de desempenho na execução desse programa, sem modificações, em gerações de GPGPUs mais novas e poderosas [34].

Para gerenciar essa virtualização de processamento e fornecer escalabilidade, CUDA requer que os blocos executem de forma independente. Deve ser possível a execução dos blocos em qualquer ordem, em paralelo ou em série. Diferentes blocos não têm nenhum meio de comunicação direta, embora possam coordenar suas atividades com operações atômicas na memória global, visíveis a todas as *threads* [34]. Em resumo, a abstração de blocos e a replicação de SMs no *hardware* trabalham em conjunto para fornecer de forma transparente escalabilidade ilimitada e eficiente [16].

Limitações do Modelo de Programação CUDA

Quando desenvolvemos programas CUDA é bastante importante conhecer possíveis limitações do modelo, em grande parte por razões de eficiência. Para apoiar uma arquitetura de sistema heterogêneo combinando uma CPU e uma GPU, cada um com seu próprio sistema de memória, os programas CUDA devem copiar os dados e resultados entre memória do *host* e a memória do dispositivo. Existe então um *overhead* gerado por esta comunicação. Para minimizar este *overhead*, a arquitetura provê uma DMA para transferências de blocos e meios de interconexão mais velozes. É claro que problemas grandes o suficiente para precisarem de uma GPU podem aumentar o desempenho e amortizar um possível *overhead*.

2.3 Modelos de Computação Paralela

Em teoria da computabilidade, um modelo de computação é a definição de um conjunto de operações que podem ser usadas em uma computação e seus respectivos custos. Utilizando um modelo específico de computação é possível analisar os recursos computacionais requeridos, como o tempo de execução e espaço de armazenamento, ou discutir as limitações dos algoritmos ou computadores. Na área de análise de algoritmos é comum especificar um modelo de computação em termos de operações primitivas, cada uma com um custo unitário associado. Na computação paralela não existe um modelo de computação amplamente aceito. Isto faz com que cada algoritmo seja analisado, em termos de complexidade de tempo e uso de recursos, de forma diferenciada de acordo com o modelo escolhido. Em alguns casos a análise do desempenho é feita de forma empírica com base nas execuções do algoritmo em um dado ambiente computacional ou com a utilização de *benchmarking*.

Existem alguns modelos que servem de paradigma para simplificar a análise de algoritmos paralelos. Nesta seção são apresentados alguns dos que são mais utilizados.

2.3.1 O Modelo PRAM

O modelo teórico mais utilizado em computação paralela é conhecido como Máquina Paralela de Acesso Aleatório (*Parallel Random Access Machine – PRAM*) e pode ser considerado uma extensão natural do modelo básico sequencial (*Random Access Machine – RAM*). Neste modelo os processadores têm acesso a uma única unidade de memória compartilhada. Mais precisamente, este modelo de memória compartilhada é constituído por um certo número de processadores, cada um dos quais tem a sua própria memória local e pode executar o seu próprio

programa local. Neste modelo os processadores se comunicam por meio de troca de dados via a unidade de memória compartilhada. Cada processador é unicamente identificado por um índice, chamado número do processador ou o *ID* (*Identifier*) do processador, o qual está disponível localmente. Para um determinado algoritmo neste modelo, o tamanho dos dados transferidos entre a memória compartilhada e as memórias locais dos diferentes processadores representa a quantidade de comunicação necessária pelo algoritmo. A Figura 2.4 ilustra uma visão geral de um modelo de memória compartilhada com p processadores. Estes processadores são indexados de 1 a p. A memória compartilhada também é conhecida como memória global [27].

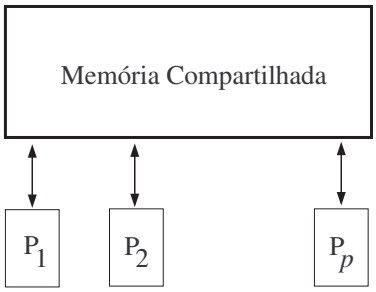


Figura 2.4: O modelo de memória compartilhada.

O modelo PRAM padrão funciona de modo síncrono, onde os processadores operam de forma síncrona sob o controle de um relógio comum. Há também o modelo assíncrono, onde é responsabilidade do programador definir pontos de sincronização apropriados sempre que necessário. Mais precisamente, se os dados precisam ser acessados por um processador, cabe ao programador garantir que os valores corretos sejam obtidos, uma vez que o valor de uma variável compartilhada é determinado dinamicamente durante a execução dos programas dos diferentes processadores [27].

Algoritmos projetados para o modelo PRAM não levam em conta a comunicação e assumem que o número de processadores, p, disponíveis é da mesma ordem do tamanho do problema, n, ou seja são considerados de "granulosidade fina". Quando algoritmos baseados em PRAM são implementados nas máquinas paralelas existentes, geralmente os *speedups* obtidos são muitas vezes desapontadores. Em muitos casos, o custo (tempo multiplicado pelo número de processadores) obtido pelos algoritmos paralelos é bastante superior ao do algoritmo sequencial. O modelo PRAM possui variações baseadas na forma como a manipulação concorrente a uma mesma posição de memória é efetuada [27]:

- *Exclusive Read, Exclusive Write* (EREW): leitura e escrita exclusiva, ou seja, não permite qualquer acesso simultâneo a uma mesma posição de memória;
- *Concurrente Read, Exclusive Write* (CREW): permite a leitura simultânea de uma mesma posição de memória por mais de um processador, mas não a escrita simultânea;
- *Concurrent Read, Concurrente Write* (CRCW): permite simultaneidade na leitura e na escrita na mesma posição de memória por diversos processadores, porém são necessários critérios para solucionar o problema da escrita concorrente. Entre as políticas adotadas para decidir qual dos processadores efetará a escrita temos:

  - escrita comum: todos os processadores devem escrever o mesmo valor.

- escrita arbitrária: permite que qualquer processador obtenha sucesso na escrita.

- escrita proprietária: define uma prioridade entre os processadores e aquele com maior prioridade terá sucesso na escrita.

Com o passar do tempo, o modelo PRAM mostrou-se incapaz de capturar todos os detalhes da computação paralela. Parcialmente, devido a isso, a área de algoritmos paralelos atravessou uma séria crise no final dos anos 80. Vários resultados teóricos obtiveram *speedups* desapontadores. Neste período, em função das dificuldades de implementação do modelo PRAM, surgiram trabalhos alternativos sobre modelos que levavam em conta a limitação real dos processares e a comunicação entre eles, esses modelos foram denominados como modelos ponte ou realísticos (*bridging models*). Os principais exemplos são o *Bulk Synchronous Parallel - BSP* [46] e o *Coarse Grained Multicomputer - CGM* [15]. Estes modelos tentavam capturar e tratar as dificuldades não contempladas pelo modelo PRAM e ficaram conhecidos como modelos de computação de granulosidade grossa. O termo "granulosidade grossa" (*coarse grained*) vem do fato de que o tamanho do problema, n, é consideravelmente maior que o número de processadores., ou seja, $n/p > p$ [27].

2.3.2 O Modelo BSP

O modelo BSP (*Bulk Synchronous Parallel*) foi proposto por Leslie G. Valiant, em 1990 [46]. Além de ser um dos modelos realísticos mais importantes, foi um dos primeiros a considerar os custos de comunicação e a abstrair as características de uma máquina paralela para um pequeno número de parâmetros. O principal objetivo deste modelo é servir de modelo ponte entre as necessidades de *hardware* e *software* da computação paralela. O modelo BSP consiste de um conjunto de p processadores com memória local, comunicando-se através de algum meio de interconexão, gerenciados por um roteador e com facilidades de sincronização global. Um algoritmo BSP consiste em uma sequência de superpassos separados por barreiras de sincronização. Um superpasso consiste de uma combinação de passos de computação, usando dados disponibilizados localmente no início do superpasso, e passos de comunicação, através de instruções de envio e recebimento de mensagens.

A Figura 2.5 ilustra os superpassos de um algoritmo BSP. Os superpassos são separados por uma barreira de sincronização, que ocorre a cada L unidades de tempo. A barreira de sincronização é realizada para determinar se o superpasso foi completado por todos os processadores, assegurando que os dados recebidos pelos processadores estarão disponíveis para o próximo superpasso. Os parâmetros do modelo BSP são os seguintes [46]:

- n: tamanho do problema;
- p: número de processadores disponíveis, cada qual com sua memória local;
- L: tempo mínimo entre dois passos de sincronização, chamado de parâmetro de periodicidade ou latência de um superpasso;
- g: capacidade computacional dividida pela capacidade de comunicação de todo o sistema. Este parâmetro descreve a taxa de eficiência de computação e comunicação do sistema.

Neste modelo uma h-relação em um superpasso corresponde ao envio e/ou recebimento de, no máximo, h mensagens em cada processador. A resposta a uma mensagem enviada em um superpasso somente será utilizada no próximo superpasso.

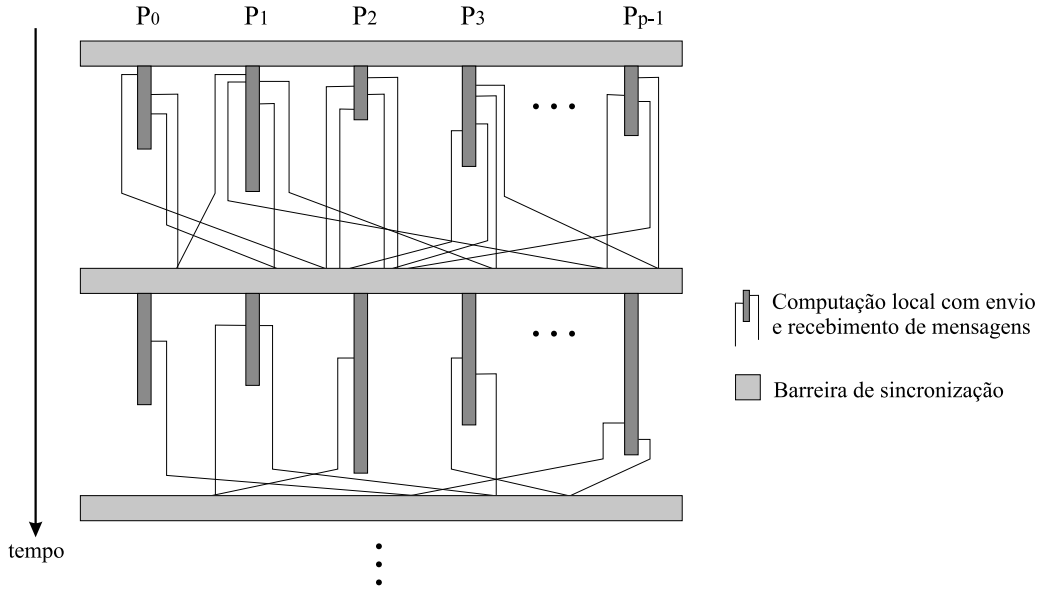


Figura 2.5: O modelo BSP [22].

2.3.3 O Modelo CGM

O modelo CGM (*Coarse Grained Multicomputer*) proposto por Frank Dehne *et al.* em 1994, é parecido com o modelo BSP, no entanto utiliza apenas dois parâmetros: o número de processadores p e o tamanho da entrada n [15]. O propósito era obter um modelo próximo às máquinas paralelas com memória distribuída existentes. Uma máquina CGM consiste de um conjunto de p processadores, cada um com memória local de tamanho $O(n/p)$. Os processadores podem estar conectados por qualquer meio de interconexão. Um algoritmo CGM consiste de uma sequência de rodadas, alternando fases bem definidas de computação local e comunicação global, separadas por uma barreira de sincronização, como é representado na Figura 2.6. Normalmente, durante uma rodada de computação é utilizado o melhor algoritmo sequencial para o processamento dos dados disponibilizados localmente. Em cada rodada de comunicação, cada processador troca no máximo $O(n/p)$ dados com outros processadores. Um algoritmo CGM é um caso especial de um algoritmo BSP onde todas as operações de comunicação de um superpasso são feitas na h-relação. Conforme observado por Frank Dehne *et al.*, os algoritmos CGM, quando implementados, se comportam bem e requerem *speedups* similares aqueles previstos em suas análises. Para estes algoritmos, o maior objetivo é minimizar o número de superpassos e a quantidade de computação local [15].

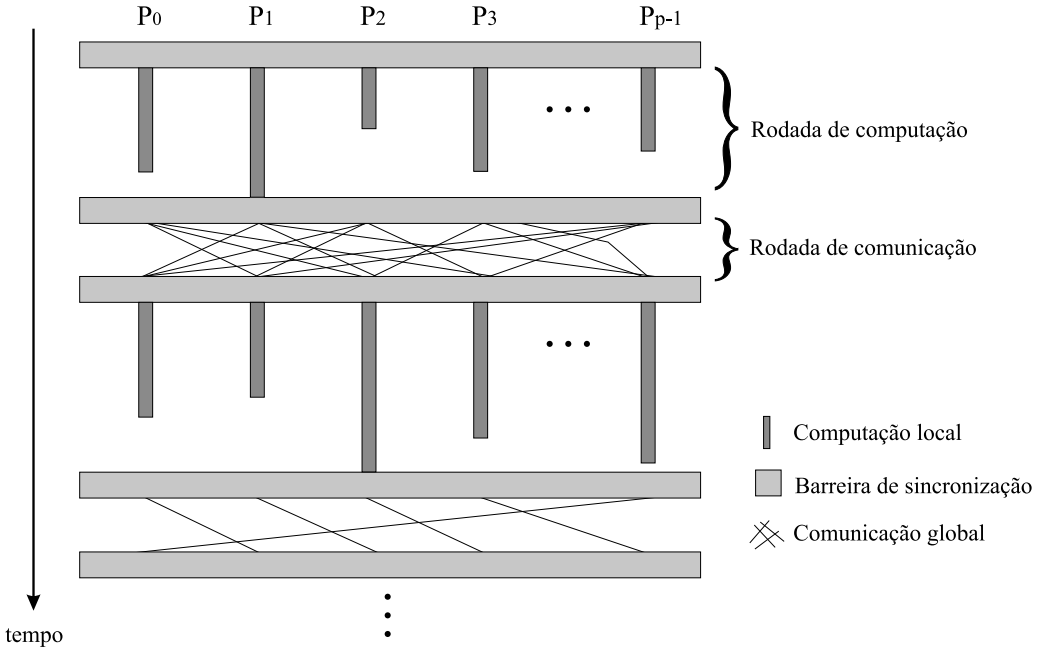


Figura 2.6: O modelo CGM [15].

2.3.4 O Modelo LogP

Em 1993 foi proposto o modelo LogP [12]. O objetivo deste modelo era representar de forma mais detalhada as características implicítas durante a transmissão de mensagens. A proposta consistia em caracterizar adequadamente os principais fatores de desempenho das máquinas paralelas de memória distribuída, que realizam muita comunicação. O modelo foi estimulado pela tendência de crescimento no uso destas máquinas, podendo ser utilizado na modelagem e no estudo de aplicações baseadas em troca de mensagens. O modelo LogP enfatiza os aspectos de desempenho relacionados à rede de interconexão e procura sintetizá-los nos seguintes parâmetros [12]:

- L: Latência envolvida na transmissão de mensagens através da rede;
- O: *Overhead* relacionado às tarefas de envio e recepção da mensagem (durante este intervalo de tempo o processador encontra-se dedicado, não podendo executar outras operações);
- G: denominado *Gap*, este parâmetro trata do tempo mínimo que deve ser respeitado entre transmissões ou recepções consecutivas;
- P: número de nós de processamento.

Através destes parâmetros o modelo LogP procura auxiliar a construção de algoritmos paralelos eficientes e portáveis.

2.4 Modelos de Computação Paralela para Ambientes *Multi/Manycore*

Com o surgimento de novas arquiteturas de computadores paralelos, principalmente as *multi/manycore*, é incessante a busca por modelos de computação paralela que sejam

satisfatórios e que atendam às novas restrições consideradas. Nesta seção descrevemos brevemente a proposta de Valiant de um novo modelo para este cenário [47]. Em função da complexidade da proposta, ao final propomos uma extensão do modelo BSP/CGM para projetar algoritmos paralelos na arquitetura *multi/manycore*.

2.4.1 O Desafio *Multicore*

Em 2003, surgiu um novo paradigma para computação paralela que se baseia na utilização de mais de uma unidade de processamento, conhecidas como núcleos (*cores*), em cada *chip*, com o propósito de aumentar o poder de processamento [28]. A tecnologia presente em computadores *multi/manycore* (múltiplos núcleos) consiste em colocar duas ou mais unidades de execução (*cores*) no interior de um único "pacote de processador" (um único *chip*). O sistema operacional trata esses núcleos como se cada um fosse um processador diferente, com seus próprios recursos de execução. Na maioria dos casos, cada unidade possui seu próprio *cache* e pode processar várias instruções simultaneamente. Adicionar novos núcleos de processamento a um processador possibilita que as instruções de aplicações sejam executadas em paralelo em vez de serialmente, como ocorre em um núcleo único. É importante notar que, para uma total utilização do poder de processamento oferecido pela tecnologia *multi/manycore*, as aplicações devem ser escritas de modo a usar intensivamente o conceito de *threads*. A menos quando temos várias aplicações distintas rodando cada uma em um núcleo. Uma arquitetura *multi/manycore* é geralmente composta por um multiprocessador simétrico (SMP) implementado em um único circuito *Very Large Scale Integration* (VLSI). O objetivo da arquitetura é melhorar o paralelismo no nível de *threads*, ajudando especialmente as aplicações que não conseguem se beneficiar dos processadores superescalares atuais, por não possuírem um bom paralelismo no nível de instruções.

A busca por modelos que executem em arquitetura de múltiplos núcleos tem se tornado objeto de estudo de diversas pesquisas. Entretanto, neste ambiente a programação para problemas gerais de alto desempenho, frequentemente não tem obtido sucesso. Em contrapartida, grande parte dos novos microprocessadores são computadores paralelos, com isto o número de aplicações que precisam ser desenvolvidas como programas paralelos tem aumentado rapidamente. Com isso, as novas aplicações deversão usar novas arquiteturas e linguagens de programação com foco em computadores *multi/manycore*.

Temos então que a arquitetura *multi/manycore*, baseada em muitos processadores e em memórias caches locais é um dispositivo atraente dado as atuais possibilidades tecnológicas e as limitações físicas já conhecidas. No entanto, a construção de algoritmos paralelos para tais máquinas tem de enfrentar tantos desafios que o caminho para a sua exploração eficiente não está claramente definido. Entre estes desafios destacam-se [43]:

1. Os algoritmos resultantes precisam competir e superar os algoritmos sequenciais já existentes, que são mais facilmente compreendidos e altamente otimizados;

2. Existe uma limitação, na melhor das hipóteses, de um aumento de *speedup*, isto ocorre essencialmente devido ao número de processadores;

3. Por fim, temos a existência e a utilização de máquinas diferentes, com isso *speedups* obtidos para uma máquina podem não ser os mesmos *speedups* em outras, de modo que todo o esforço do projeto pode ser substancialmente desperdiçado.

2.4.2 O Modelo *Multi*-BSP

Vimos que é difícil prever de forma eficiente como algoritmos *multi/manycore* podem ser criados e explorados. No futuro é possível que, mesmo com a proliferação destas máquinas, o seu potencial computacional seja grandemente subutilizado. Recentemente, Leslie G. Valiant propôs um modelo de transição que visa capturar os parâmetros dos elementos mais básicos de arquiteturas *multi/manycore* [47]. O modelo proposto, denominado Multi-BSP, é um modelo de múltiplos níveis que tem parâmetros explícitos para os números de processadores, tamanhos de memória *cache*, custos de comunicação e custos de sincronização. O nível mais baixo do modelo Multi-BSP simula o comportamento do modelo PRAM de memória compartilhada. De acordo com Leslie G. Valiant, os parâmetros de um algoritmo Multi-BSP devem funcionar eficientemente em todas as arquiteturas relevantes para qualquer número de níveis e com qualquer combinação de parâmetros. Ele também mostrou que estes parâmetros podem ser utilizados em algoritmos para uma série de problemas fundamentais em computação, entre eles multiplicação de matrizes, transformação rápida de *Fourier* e Ordenação [47]. O modelo Multi-BSP pode ser visto sob duas perspectivas:

1. A primeira ilustra um modelo hierárquico, com um número arbitrário de níveis. Este modelo se adapta às realidades físicas de múltiplas memórias e vários níveis de *caches*, tanto em arquitetura de *chips* únicos, bem como em arquiteturas de múltiplos *chips*. O objetivo é modelar todos os níveis de uma arquitetura juntos, até mesmo para uma estrutura como um *Data Center*;

2. A segunda abrange cada nível. Em cada nível o modelo incorpora o tamanho da memória como um parâmetro adicional. Afinal, é a limitação física da quantidade de memória, que pode ser acessada em um determinado período de tempo a partir da localização física de um processador, que cria a necessidade de vários níveis.

O modelo Multi-BSP, para uma profundidade de d níveis é especificado por $4d$ parâmetros numéricos: $(p_1,\ g_1,\ L_1,\ m_1)$, $(p_2,\ g_2,\ L_2,\ m_2)$, $(p_3,\ g_3,\ L_3,\ m_3)\ \ldots (p_d,\ g_d,\ L_d,\ m_d)$. O modelo simula uma árvore com profundidade d com memórias *caches* como nós internos e processadores como folhas. Para cada nível, os quatro parâmetros significam respectivamente:

1. O número de subcomponentes;

2. A largura de banda de comunicação;

3. O custo de sincronização;

4. O tamanho da memória cache.

Uma instância do modelo Multi-BSP é uma estrutura de árvore de componentes aninhados onde o nível mais baixo ou folhas são processadores e todos os outros níveis contém alguma capacidade de armazenamento. Para definir uma instância de Multi-BSP fixa-se d como a profundidade ou o número de níveis, e $4d$ outros parâmetros: $(p_1,\ g_1,\ L_1,\ m_1)$, $(p_2,\ g_2,\ L_2,\ m_2)$, $(p_3,\ g_3,\ L_3,\ m_3)\ \ldots (p_d,\ g_d,\ L_d, m_d)$. As Figuras 2.7 e 2.8 ilustram, respectivamente, este modelo em termos de seus componentes de nível i-1 e de nível 1 [47]. Segue-se uma descrição dos componentes:

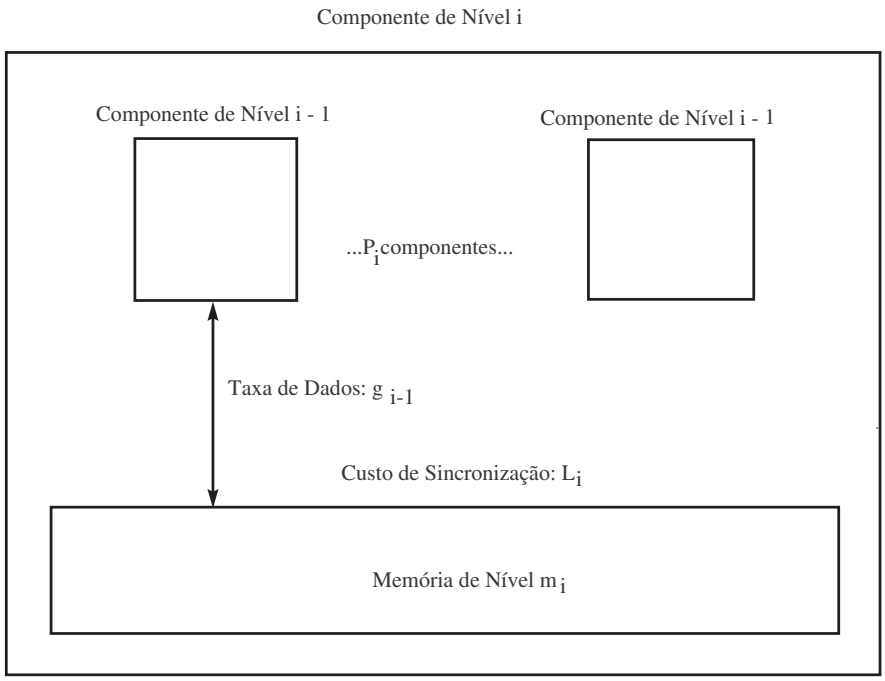


Figura 2.7: Modelo Multi-BSP em termos de seus componentes em um nível i-1.

- p_i descreve o número de componentes do nível $i-1$ dentro de um componente de nível i.
- g_i é o parâmetro de largura de banda de comunicação. É a relação entre o número de operações que um processador pode executar em um segundo e o número de palavras que podem ser transmitidas, em um segundo, entre as memórias de um componente de nível i, e seu componente superior, no nível $i+1$. Uma palavra é a quantidade de dados que uma operação do processador é capaz de executar.
- Um superpasso de nível i é um passo de execução de um componente de nível i que permite que p_i processadores do nível $i-1$ possam executar de forma independente até atingirem a barreira de comunicação. Quando todos os processadores dos componentes de nível i-1 atingirem a barreira, eles podem trocar toda a informação com a memória m_i dos componentes de nível i, com custo de comunicação especificado por g_{i-1}. Neste caso, o custo de comunicação será estimado por $m.g_{i-1}$, onde m é o número máximo de palavras comunicadas entre a memória do componente de nível i e um de seus subcomponentes de nível $i-1$.
- L_i é o custo cobrado para sincronização da barreira em um super passo de nível i;
- O componente m_i é o número de palavras de memória dentro de um componente de nível i, que não estão em um componente do nível $i-1$.

Finalmente, é preciso especificar a natureza da comunicação entre um componente do nível i e do nível $i+1$. Os algoritmos desenvolvidos para Multi-BSP são, em geral, de Leitura-Exclusiva e Escrita-Exclusiva ($EREW$).

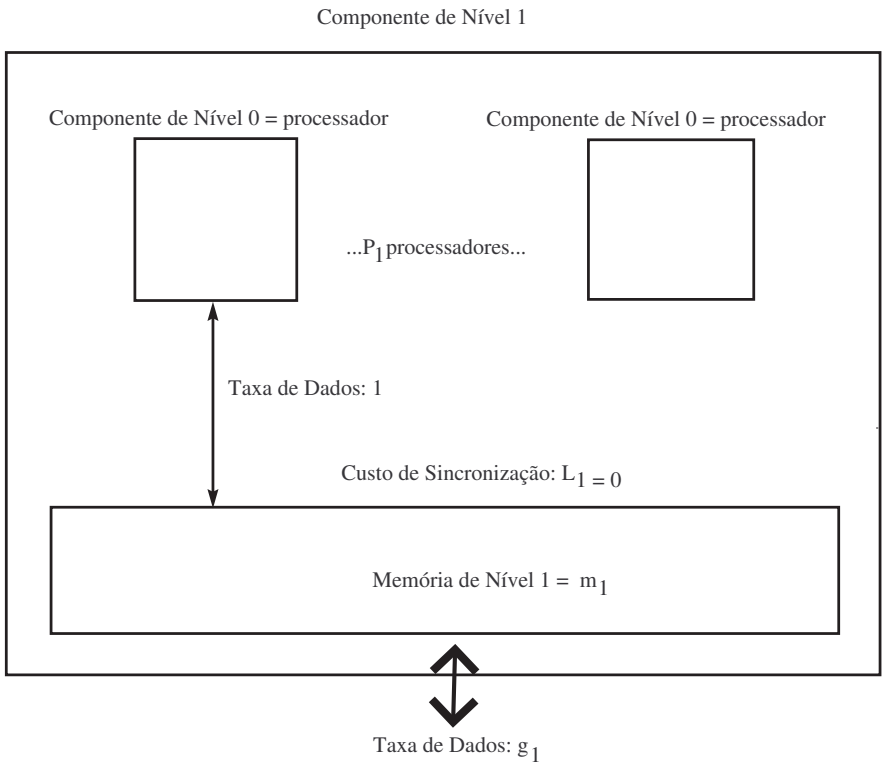


Figura 2.8: Esquema do modelo Multi-BSP em termos de seus componentes de nível 1.

A seguir, apresentamos um exemplo de como relacionar os parâmetros do modelo Multi-BSP em projetos de arquitetura *multi/manycore* já existentes: considere uma máquina paralela que consiste em p *chips multicore Sun Niagara UltraSparc* conectados a um dispositivo de armazenamento externo que é grande o suficiente para armazenar a entrada para o problema dado. Os níveis e os parâmetros do modelo para esta arquitetura são ilustrados pela Tabela 2.1 [47]

Nível 1: 1 *core* com 1 processador com 4 *threads* mais uma cache L_1.	
p_1	4
g_1	1
L_1	3
m_1	8KB
Nível 2: 1 chip com 8 *cores* mais uma cache L_2.	
p_2	8
g_2	3
L_2	23
m_2	3MB
Nível 3: p *chips multicore* mais uma memória externa acessível via rede.	
p_3	p
g_3	∞
L_3	108
$m_3 \leq$	128GB

Tabela 2.1: Níveis Multi-BSP para uma arquitetura real.

Neste esquema hierárquico, os parâmetros L (de latência) são limitados pelas especificações dos *chips* ao invés dos reais custos de sincronização. Em segundo lugar, os *caches* no *chip* são controlados por protocolos implícitos, em vez de explicitamente pelos programas, como é usual para memórias. Em terceiro lugar, os *chips* de cada processador físico executam quatro

threads, e os grupos de processadores compartilham uma unidade aritmética comum. Devido a estes fatores os valores reais de g e L geralmente são difíceis de definir. São três as principais considerações sobre as atuais arquiteturas e o modelo Multi-BSP [47]:

1. A sincronização em uma barreira necessita ser realizada de forma eficiente nas arquiteturas atuais;
2. O modelo Multi-BSP controla o armazenamento de dados explicitamente. Não é claro como os vários protocolos de *cache* utilizados atualmente são favoráveis a este modelo;
3. É necessária a existência de máquinas que apoiem as características deste modelo.

2.4.3 Uma Proposta de Extensão do Modelo BSP/CGM

Como vimos acima, existem vários modelos, linguagens e arquiteturas para a computação paralela. O nosso principal objetivo neste trabalho é a obtenção de algoritmos paralelos eficientes e que tenham bom desempenho tanto em ambientes de memória compartilhada como memória distribuída. Nosso enfoque foi na arquitetura *multi/manycore*. Considerando as disponibilidades, optamos pela utilização da GPGPU com a linguagem CUDA. Considerando que o modelo BSP/CGM vem sendo utilizado com sucesso no projeto e implementação de algoritmos paralelos utilizando memória distribuída, analisamos inicialmente a utilização do modelo Multi-BSP [47]. Entretanto em função da complexidade desse modelo, optamos por estender o modelo BSP/CGM para projetar os nossos algoritmos.

A implementação de algoritmos BSP/CGM geralmente apresenta resultados similares aqueles preditos nas respectivas análises teóricas [14]. Uma vez que a biblioteca MPI é destinada para ambientes de memória distribuída, um algoritmo BSP/CGM pode ser mapeado para uma implementação MPI utilizando os recursos de envio e recebimento de mensagens desta biblioteca.

Para o projeto e análise de algoritmos paralelos na arquitetura *multi/manycore*, acrescentamos ao modelo BSP/CGM as seguintes definições:

- Um par de processadores no modelo BSP/CGM pode ser relacionado a um SM e sua respectiva memória compartilhada na GPU;
- Um *round* de comunicação no modelo BSP/CGM pode ser representado pela comunicação entre CPU e GPU;
- Os superpassos do modelo BSP/CGM podem ser representados por funções de *kernel* da GPU;

É importante destacar que estas relações não são fixas e pela experiência das implementações cada algoritmo apresentou particularidades, que podem levar a pequenas alterações. A Figura 2.9 ilustra nossa sugestão para este processo, onde o modelo BSP/CGM é representado à direita [22].

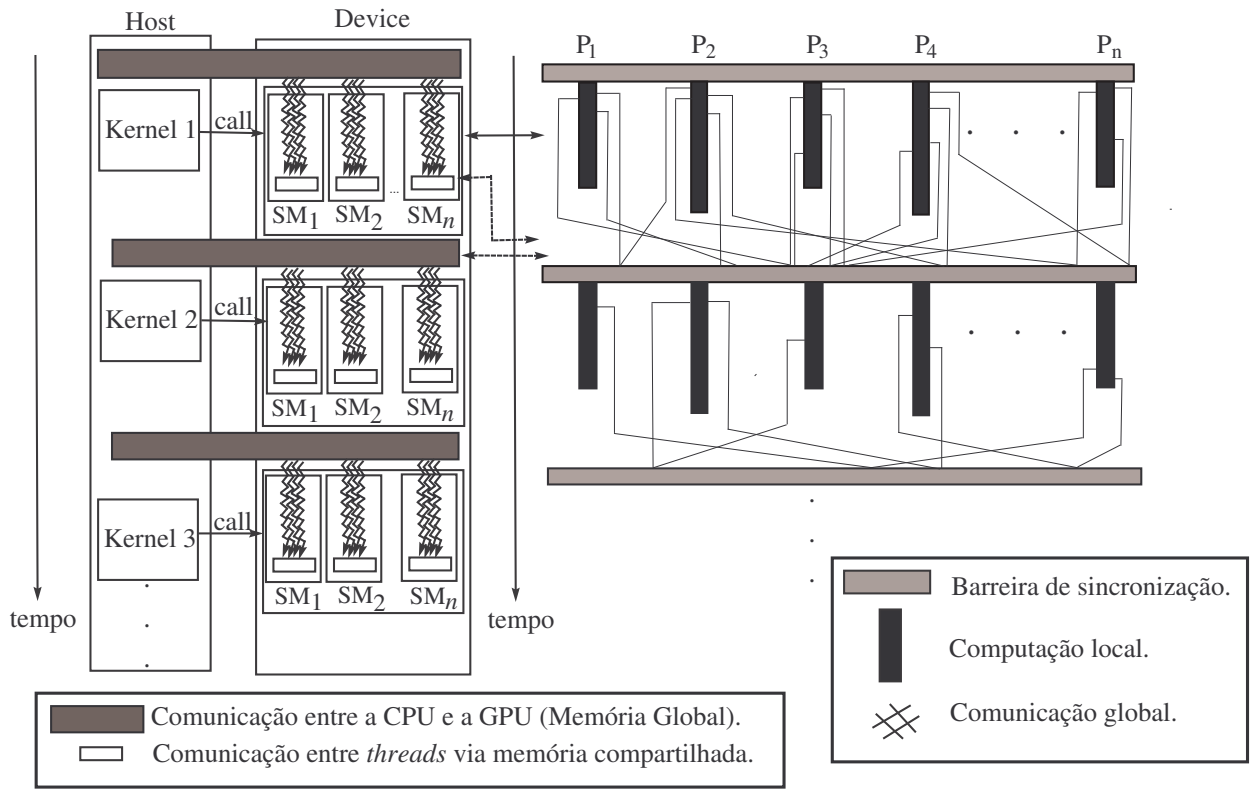


Figura 2.9: Abstração entre uma GPGPU e o modelo BSP/CGM.

Nos capítulos de 3 a 6, veremos que a nossa proposta de extensão do modelo BSP/CGM possibilitou o projeto de algoritmos de cada um dos problemas abordados. Conforme será apresentado, as implementações apresentaram bons desempenhos, onde uma única GPU, por vezes, obteve desempenho melhor do que o desempenho obtido por um *cluster* de estações de trabalho.

2.5 Ambiente Computacional Utilizado

Neste trabalho executamos nossos algoritmos em dois sistemas computacionais diferentes. Os algoritmos CUDA foram executados em uma máquina com 8 GB de RAM e Sistema Operacional Linux (Ubuntu 14:04), (R) Intel Core (TM) processador i5-2430M @ 2.40GHz CPU e uma GTX NVIDIA GeForce 680 com 8 SMX's, onde cada SMX possui 192 *cores* físicos, o que resulta em 1536 CUDA *cores*. Os algoritmos MPI foram executados em um *cluster* com 64 nós, cada nó consistindo de 4 processadores Dual-Core AMD Opteron 2.2 GHz com 1024 KB de cache e 8 GB of memória. Todos os nós deste ambiente são interconectados por meio de um *switch Foundry SuperX* utilizando *Gigabit ethernet*. Este *cluster* pertence ao Laboratório de Alta Performance de Computação Virtual (*High Performance Computing Virtual Laboratory* (HPCVL)) da Universidade de Carleton. Para efeito de comparações, com os algoritmos em CUDA e em MPI, os algoritmos sequenciais equivalentes foram executados no ambiente do *cluster* e na GPU com CUDA. Experimentos com diferentes tamanhos de sequências de entrada foram realizados. Para cada sequência de entrada, os algoritmos foram executados 10 vezes e a média aritmética dos tempos de execução foi computada. Calculamos também o desvio padrão para cada uma das médias.

2.6 Considerações Finais do Capítulo

Neste capítulo apresentamos brevemente alguns dos principais modelos, linguagens e arquiteturas de computação paralela. Considerando que o nosso trabalho é sobre o projeto e implementação de algoritmos paralelos utilizando arquiteturas *multi/manycore*, e as nossas disponibilidades, também descrevemos em mais detalhes as GPGPU's e a linguagem CUDA. Para projetar e analisar os algoritmos desse trabalho apresentamos uma proposta de extensão do modelo de computação BSP/CGM, como veremos posteriormente. Apesar de simples, essa extensão se mostrou robusta para os algoritmos abordados nesse trabalho, e acreditamos que com mais estudos essa extensão possa ser um arcabouço de um modelo de computação paralela para ambientes *multi/manycore*.

CAPÍTULO 3

A SUBSEQUÊNCIA DE SOMA MÁXIMA

Neste capítulo revisitamos o problema da subsequência de soma máxima e propomos, além da solução geral, soluções paralelas BSP/CGM para três problemas relacionados: a maior subsequência de soma máxima, a menor subsequência de soma máxima e o número de subsequências disjuntas de soma máxima. Não é de nosso conhecimento a existência de soluções BSP/CGM para estes últimos problemas. Descobrir os tamanhos e os números de subsequências de soma máximas são tarefas importantes. Particularmente, na análise de DNA, se encontramos todas as subsequências de somas máximas, também encontraremos todas as possíveis ilhas de patogenicidade, que são trechos do DNA com alta possibilidade de ocasionarem doenças.

A fim de demonstrar a eficiência de nossas soluções não somente em teoria, mas também na prática, implementamos nossos algoritmos em ambientes computacionais de memória distribuída (um cluster) e compartilhada (uma máquina *manycore*). Resultados competitivos foram obtidos em ambos os ambientes computacionais.

Inicialmente discutimos o problema geral da subsequência de soma máxima e soluções sequenciais e paralelas conhecidas. Algumas destas soluções serviram como referência para a elaboração e o desenvolvimento de nossos algoritmos. Em seguida, descrevemos como ampliar nossas soluções para também resolver os problemas relacionados. Por fim, apresentamos nossos resultados utilizando tabelas e gŕaficos que ilustram comparações de tempo de execução e *speedup*.

3.1 Definição

Dada uma sequência de números reais, o problema de identificar a subsequência contígua com a maior soma de seus elementos é chamado de **problema da subsequência de soma máxima**. Se os números são todos positivos a solução é obviamente a sequência de entrada. Entretanto, é particularmente mais interessante quando também existem números negativos na sequência de entrada. O problema da subsequência de soma máxima também pode ser descrito de outra forma: Considere uma sequência de n números reais $(x_1, x_2, \ldots, x_n)$. Uma subsequência contígua é qualquer intervalo $(x_i, \ldots, x_j)$ da sequência de entrada, tal que $1 \leq i \leq j \leq n$. Por simplicidade consideraremos subsequência como subsequência contígua. Para o problema da subsequência de soma máxima é preciso determinar a subsequência $M = (x_i, \ldots, x_j)$ que possua o maior somatório $T_M = \sum_{k=i}^{j} x_k$. Sem perda de generalidade

assume-se que existe ao menos um x_i positivo. Como exemplo, considere a seguinte sequência $(3, 5, 10, -5, -30, 5, 7, 2, -3, 10, -7, 5)$. Neste caso uma subsequência de soma máxima é $M = (5, 7, 2, -3, 10)$ com soma total $T_M = 21$ [2].

Este problema aparece em aplicações de biologia computacional, tais como: a identificação de domínios transmembranos em proteínas, análise de proteínas, análise de sequências de DNA, e na identificação de genes [2]. Uma aplicação prática é vista quando se deseja encontrar em uma sequência de DNA a região mais rica em nucleotídeos G e C, neste caso, basta atribuir à sequência de DNA valores positivos para G e C e valores negativos para A e T. Encontrar esta região é primordial para o cálculo do valor de *GC-content*, cuja fórmula é dada por: $\frac{G+C}{A+G+C+T}$. O valor de *GC-content* é especialmente importante na operação de busca de ilhas de patogenicidade [33]. Como exemplo, a Tabela 3.1 ilustra a aplicação de um algoritmo sequencial de subsequência de soma máxima para encontrar os valores de alta porcentagem de *GC-content*, destacados em negrito, de partes contíguas do genoma de um tipo de bactéria do gênero salmonella, que é causadora de diversas doenças [19].

Pontuação para nucleotídeos	n=131072	n=262144	n=524288	n=1048576	n=2097152	n=4194304
$(A = -2, T = -2, G = 2, C = 2)$	**90546**	**184592**	**360450**	**713188**	**1408718**	**2809990**
$(A = -2, T = -2, G = 1, C = 1)$	**4299**	**9167**	**10024**	**14519**	**20682**	**25356**

Tabela 3.1: Regiões mais ricas em nucleotídeos G e C dos n primeiros nucleotídeos do genoma da bactéria *Pseudomonas aeruginosa* B136-33.

Várias soluções sequenciais foram propostas para esse problema, e o melhor algoritmo sequencial de tempo linear para o problema da subsequência de soma máxima em vetores (1D) é atribuído a Jay Kadane. A descrição formal do algoritmo e a correção de seu custo ótimo $\mathrm{O}(n)$ podem ser vistas no artigo original [4]. No caso de algoritmos paralelos, como veremos a seguir, várias soluções com bom desempenho foram apresentadas.

3.2 Histórico de soluções

Nesta seção destacamos as principais soluções sequenciais e algumas soluções paralelas eficientes para o problema da subsequência de soma máxima. Algumas vezes no texto nos referimos a sequências como vetores.

3.2.1 Algoritmo Cúbico

O algoritmo ingênuo para encontrar a subsequência de soma máxima itera sobre todos os pares de números inteiros L e U de uma sequência de inteiros X de comprimento n, onde $1 \leq L \leq U \leq n$. Para cada par calcula-se a soma contígua dos elementos presentes em $X[L, \ldots, U]$. Considera-se o primeiro par como uma subsequência de soma máxima, verifica-se então se a soma do próximo par é maior que a soma máxima e assim por diante. O Algoritmo 1 ilustra o pseudocódigo deste algoritmo [5]. O código é curto, simples e fácil de entender. Infelizmente, tem a grave desvantagem de ser muito lento. Em um computador típico de 1986, por exemplo, a execução demorava cerca de uma hora se n fosse igual a 1000 e 39 dias, se n fosse igual a 10.000 [5]. O número de passos que este algoritmo executa é da ordem de $\mathrm{O}(n^3)$.

Algoritmo 1: Algoritmo cúbico para a subsequência de soma máxima.

Entrada: (1) Sequência de inteiros X.
Saída: (1) Subsequência de soma máxima de X.

```
1: MaxSoFar := 0
2: for L = 1 to n do
3:    for U = L to n do
4:       Sum := 0
5:       for I = L to U do
6:          Sum := Sum + X[I]
7:       end for
8:       MaxSoFar := max(MaxsoFar, Sum)
9:    end for
10: end for
```

3.2.2 Algoritmo Quadrático

O segundo algoritmo é quadrático, sua estratégia é baseada em uma observação simples: Percebeu-se que a soma de $X[L..U]$ tem uma relação íntima com a soma previamente calculada $X[L, \ldots, U-1]$. O Algoritmo 2 explora esta relação [5]. As instruções dentro do primeiro laço são executadas n vezes, e aquelas dentro do segundo laço são executadas no máximo n vezes a cada execução do laço mais externo, de modo que o tempo total de execução do algoritmo é $\mathrm{O}(n^2)$.

Algoritmo 2: Algoritmo quadrático para a subsequência de soma máxima.

Entrada: (1) Sequência de inteiros X.
Saída: (1) Subsequência de soma máxima de X.

```
1: MaxSoFar := 0
2: for L = 1 to n do
3:    Sum := 0
4:    for U = L to n do
5:       Sum := Sum + X[U]
6:       MaxSoFar := max(MaxSoFar, Sum)
7:    end for
8: end for
```

3.2.3 Os Algoritmos Subquadráticos

Os próximos algoritmos executam em menos tempo. São duas as abordagens apresentadas [5]:

- A primeira é mais complexa e utiliza a técnica de divisão e conquista. Para resolver um problema de tamanho n, recursivamente são resolvidos dois subproblemas de tamanho aproximado igual a $n/2$, e em seguida combinam-se as soluções destes, para obter enfim a solução para o problema original. O vetor original é dividido em dois subvetores de aproximadamente igual comprimento, chamados de A e B. Recursivamente é preciso

encontrar os subvetores de soma máxima em A e B, definidos como M_A e M_B. O algoritmo resolve o problema em tempo sequencial $\mathrm{O}(n \log n)$, $\mathrm{O}(n)$ para cada um dos $\mathrm{O}(\log n)$ níveis da recursão como descrito em [5].

- A segunda abordagem descreve um algoritmo de programação dinâmica, simples e elegante, que consome tempo $\mathrm{O}(n)$. O pseudocódigo é ilustrado pelo Algoritmo 3. Uma descrição formal deste algoritmo é encontrada em [2, 5]. O algoritmo é baseado na estratégia de que se houver como determinar a subsequência de soma máxima M de pontuação T_M de uma sequência $(x_1, x_2, ..., x_k)$, então pode-se facilmente estender o resultado para determinar a subsequência de soma máxima de uma sequência $(x_1, x_2, ..., x_k, x_{k+1})$. A estratégia do algoritmo sequencial consiste em criar um acumulador. Toda vez que este acumulador se torna negativo ele é reiniciado, além disso, a cada rodada de computação é realizada uma comparação com a última estimativa de máximo. O algoritmo se comporta analisando dois casos acerca do elemento x_k:

  1. No primeiro caso x_k é o último elemento da subsequência de soma máxima. Sendo assim, se $x_{k+1} > 0$, basta inserir o elemento x_{k+1} em M e adicionar o valor a T_M. Caso contrário a subsequência continua inalterada, M e T_M continuam inalterados.
  2. O valor x_k não pertence a subsequência de soma máxima M. Defininimos o sufixo máximo da sequência $(x_1, x_2, ..., x_k)$ como sendo $S = (x_s, x_{s+1}, ..., x_k)$ com pontuação de soma máxima T_S. Neste caso os passos 6 a 14 do Algoritmo 3 demonstram como estender a subsequência M e sua pontuação de soma máxima correspondente T_M.

Os seguintes parâmetros são considerados na entrada/saída do algoritmo:

- M é a subsequência de soma máxima de uma sequência $(x_1, x_2, \ldots, x_k)$.
- T_M é a soma dos elementos de M (pontuação de soma máxima de M).
- S é o sufixo máximo da sequência $(x_s, x_{s+1}, \ldots, x_k)$.
- T_S é a soma dos elementos de S (pontuação de soma máxima de S).

Algoritmo 3: Algoritmo sequencial para a subsequência de soma máxima.

Entrada: (1) M (sequência de entrada), S (sequência sufixo), T_M e T_S (inteiros nulos).
Saída: (1) Variável T_M que armazena o valor de soma máxima de M.

1: **if** x_k é o último número de M **then**
2: **if** $x_{k+1} > 0$ **then**
3: Acrescente x_{k+1} a M e obtenha $T_M := T_M + x_{k+1}$
4: **end if**
5: **else**
6: **if** $T_S + x_{k+1} > T_M$ **then**
7: Acrescente x_{k+1} a S e obtenha $T_S := T_S + x_{k+1}$, $M := S$ e $T_M := T_S$
8: **else**
9: **if** $T_S + x_{k+1} > 0$ **then**
10: Acrescente x_{k+1} a S e obtenha $T_S := T_S + x_{k+1}$
11: **else**
12: S será vazio.
13: **end if**
14: **end if**
15: **end if**

No Algoritmo 3, se considerarmos a sequência $(x_1, x_2, ..., x_k) = (3, 5, 10, -5, -30, 5, 7, 2, -3, 10, -7, 5)$ como entrada, após a execução teremos $M = (5, 7, 2, -3, 10)$ com pontuação $T_M = 21$ e $S = (5, 7, 2, -3, 10, -7, 5)$ com pontuação $T_S = 19$. Vamos supor então, que gostaríamos de estender o resultado acrescentando um novo elemento à sequência original, digamos $x_{k+1} = 40$. Neste caso, após a execução dos passos 6 e 7, temos $S = (5, 7, 2, -3, 10, -7, 5, 40)$ com uma nova pontuação $T_S = 59$. Neste caso M será igual a S e a nova pontuação de T_M também será 59.

Exemplo de Execução

Para melhor compreensão vamos executar o Algoritmo 3, iteração a iteração, para uma entrada $X = (x_1, x_2, ..., x_k)$, onde X = (3, 5, 10, -5, -30, 5, 7, 2, -3, 10, -7, 5). A cada iteração um elemento x_k da sequência X é avaliado. A Tabela 3.2 ilustra a execução do algoritmo.

Tabela 3.2: Executando Algoritmo 3 (Linear).

Passo	x_k	M	T_M	S	T_S
1	3	{3}	3	{ ∅ }	0
2	5	{3, 5}	8	{ ∅ }	0
3	10	{3, 5, 10}	18	{ ∅ }	0
4	-5	{3, 5, 10}	18	{ ∅ }	0
5	-30	{3, 5, 10}	18	{ ∅ }	0
6	5	{3, 5, 10}	18	{5 }	5
7	7	{3, 5, 10}	18	{5, 7}	12
8	2	{3, 5, 10}	18	{5, 7,2 }	14
9	-3	{3, 5, 10}	18	{5, 7, 2, -3}	11
10	10	{5, 7, 2, -3, 10}	21	{5, 7, 2, -3, 10}	21
11	-7	{5, 7, 2, -3, 10}	21	{5, 7, 2, -3, 10, -7}	14
12	5	{5, 7, 2, -3, 10}	21	{5, 7, 2, -3, 10, -7, 5}	19

O algoritmo 3 apresenta a melhor solução sequencial para o problema da subsequência de soma máxima, possuindo uma complexidade de tempo $O(n)$. Contudo, apesar de ser um algoritmo simples e rápido ele retorna pouca informação acerca da subsequência de soma máxima. A saída retorna apenas o valor numérico correspondente a soma máxima.

3.3 Os Algoritmos Paralelos

Trabalhos anteriores destacam algoritmos paralelos eficientes para o problema da subsequência de soma máxima.

Em 1987, Smith [44] apresentou algoritmos sequenciais para os problemas 1D e 2D com tempo $O(n)$ e $O(n^3)$. Os algoritmos são descritos como passíveis de paralelização de uma forma natural. Teoricamente, estes algoritmos consumiriam tempo $O(\log n)$ e $O(\log^2 n)$. Por sua vez, em 1995, Perumalla e Deo [35] apresentaram algoritmos para versão em vetor (1D) e em matriz (2D) para o modelo PRAM. Os autores afirmam que as soluções apresentadas são substancialmente diferentes das soluções propostas anteriormente. O algoritmo é executado em tempo paralelo $O(\log n)$ usando $n/\log n$ processadores. Ainda em 1995, Zhaofang Wen [48] apresentou algoritmos paralelos para as versões em vetor (1D) e em matriz (2D), ambas foram projetadas no modelo EREW-PRAM. As versões consomem tempo $O(\log n)$, utilizando respectivamente, $n/\log n$ e $n^3/\log n$ processadores. Já em 1999, Qiu e Akl [36] apresentaram algoritmos para versões 1D e 2D. As implementações descrevem a utilização dos algoritmos em redes de interconexão como o hipercubo e a estrela, ambas de tamanho p. Elas consomem tempo $O(n/p + \log p)$ com p processadores na versão 1D e tempo $O(\log n)$ com $n^3/\log n$ processadores na versão 2D. Por fim, em 2004, Alves *et al.* [2] apresentaram algoritmos para a versão 1D e 2D, utilizando o modelo realístico BSP/CGM (*Coarse-Grained Multicomputer*). O primeiro algoritmo consome tempo de $O(n/p)$ com p processadores em um número constante de rodadas de comunicação, nas quais $O(p)$ números são transmitidos. O segundo algoritmo consome tempo de $O(n^3/p)$ com p processadores também com um número constante de rodadas de comunicação. Os resultados destes algoritmos foram descritos como bastante promissores.

Devido as nossas questões de projeto de algoritmos em arquitetura *multi/manycore* e da utilização de modelos realísticos de computação paralela, para o problema da subsequência de soma máxima nos baseamos nos algoritmos paralelos apresentados em dois trabalhos

anteriores [35, 2], conforme ilustrado pela Figura 3.1. Descrevemos a seguir as estratégias, correção e resultados alcançados por estes algoritmos paralelos.

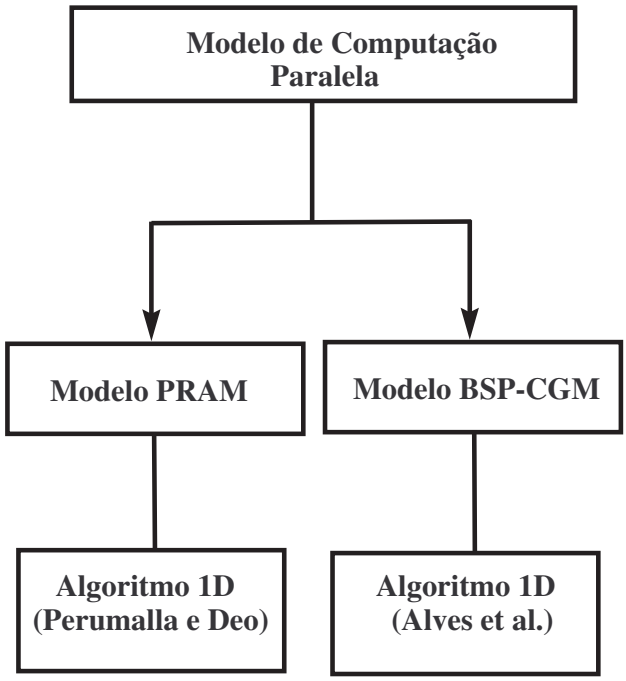


Figura 3.1: Os algoritmos paralelos analisados para a subsequência de soma máxima.

3.3.1 Algoritmo Paralelo PRAM

O Algoritmo 4 apresenta uma solução baseada no modelo PRAM, para o problema da subsequência de soma máxima. Este algoritmo foi proposto por Perumalla e Deo [35]. Para uma melhor compreensão do algoritmo, apresentamos as principais definições e lemas desse trabalho: dada uma sequência $Q = [q_1, q_2, \ldots q_n]$, considere Q_{ij} como sendo a subsequência $[q_i, q_{i+1}, \ldots, q_j]$ e $Range(q_k)$ como sendo o conjunto de todas as subsequências de Q que incluem q_k. Se todos os valores de q_i são negativos, então a subsequência de soma máxima de Q é definida como o menor número negativo encontrado. Pode-se redefinir este valor para zero, caso desejado. Considere agora um elemento da sequência Q, denominado q_k. Considere $l(q_k) = [q_1, q_2, \ldots q_{k-1}, q_k]$ a subsequência composta pelos elementos a esquerda de q_k incluindo q_k. Considere $r(q_k) = [q_k, q_{k+1}, \ldots, q_n]$ a subsequência composta pelos elementos a direita de q_k incluindo q_k. Seja $S_l(q_k) = [s_1, s_2, \ldots s_{k-1}, s_k]$ a **soma de sufixos** de $l_{(q_k)}$, e $P_r(q_k) = [p_k, p_{k+1}, \ldots, p_n]$ a **soma de prefixos** de $r_{(q_k)}$. Considere M_s^k o valor máximo da soma de sufixos de $S_l(qk)$ e M_p^k o valor máximo da soma de prefixos de $P_r(q_k)$. Estas observações conduzem ao Lema 3.3.1, também ilustrado pela Figura 3.2.

Lema 3.3.1. *O valor máximo entre as somas de todas as subsequências que incluem q_k é dado por $Max(q_k) = M_s^k + M_p^k - q_k$.*

Demonstração. Considere SQ_{ij} como sendo a soma dos elementos em Q_{ij}. Considere agora a subsequência definida pela equação Q_{ab}^k, desta forma $M_s^k = SQ_{ak}$ e $M_p^k = SQ_{kb}$, para $1 \leq a \leq k \leq b \leq n$. Considere agora que existe uma outra subsequência $Q_{a'b'}^k$ que inclui q_k, tal que a soma $SQ_{a'b'}^k$ é maior que a soma dada por SQ_{ab}^k. Visto que a subsequência $Q_{a'b'}^k$ pode ser vista

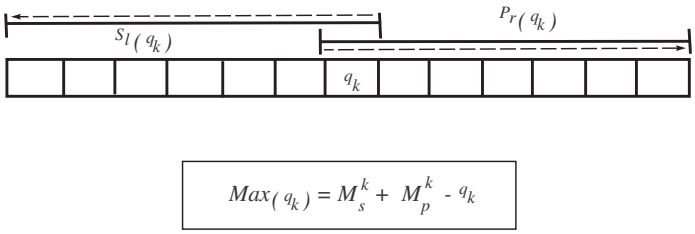


Figura 3.2: O valor máximo entre as somas de todas as subsequências que incluem q_k.

como duas subsequências $Q_{a'k}$ e $Q_{kb'}$, ambas incluindo q_k, o valor da soma $SQ^k_{a'b'}$, pode ser escrito como $SQ^k_{a'b'} = SQ_{a'k} + SQ_{kb'} - q_k$. Entretanto, por definição $M^k_s = SQ_{ak} \geq SQ_{ik}$ para todo $1 \leq i \leq k$, segue assim que $SQ_{ab'} \geq SQ_{a'b'}$. Um argumento similar pode ser aplicado para b e b', dado que $SQ_{ab} \geq SQ_{ab'}$, então $SQ^k_{ab} \geq SQ^k_{a'b'}$ para todo $SQ^k_{a'b'} \in Range(q_k)$. □

Vejamos o exemplo ilustrado pela Figura 3.3.

$Q =$	3	2	-7	11	10	-6	4	9	-6	1	-2	-3	4	-3	0	2
$l(q_7) =$	3	2	-7	11	10	-6	4									
$S_l(q_7) =$	17	14	12	19	8	-2	4									
$r(q_7) =$							4	9	-6	1	-2	-3	4	-3	0	2
$P_r(q_7) =$							4	13	7	8	6	3	7	4	4	6

Figura 3.3: Os cálculos para M^7_s.

Para M^7_s temos os seguintes valores:

- M^7_s = Máximo $(S_l) = 19$,
- M^7_p = Máximo $(P_r) = 13$,
- $M^7_s + M^7_p - q_7 = 19 + 13 - 4 = 28 = Max_{(q7)}$

Suponha que o valor $Max(q_k)$ é calculado para todo valor q_k, isto é, para cada elemento q_k da sequência Q, a soma máxima (soma de maior valor) entre todas as subsequências que incluem q_k é computada. Então claramente, a subsequência de soma máxima da sequência Q é composta pelos máximos destes máximos, isto é: MaxSeqSum = Máximo $(Max(q_k), 1 \leq k \leq n)$.

Como a soma de sufixos de q_{k-1} está relacionada com a soma de sufixos de q_k, é suficiente computar a soma de sufixos de toda sequência apenas uma vez. Em outras palavras, se $S_l(q_k) = [s_1, s_2, \ldots, s_k]$, então $S_l(q_{k-1}) = [s_1 - q_k, s_2 - q_k, \ldots, s_{k-1} - q_k,]$. Relações similares podem ser construídas para soma de prefixos. Para encontrar o valor de soma máxima de $S_l(q_k)$ para todo k, é necessário calcular o prefixo máximo das somas de sufixo de Q. Similarmente, para obter o valor de soma máxima de $P_r(q_k)$ para todo k, é necessário computar o sufixo máximo das somas de prefixos de Q.

O Algoritmo 4 de Perumalla e Deo [35] descreve a solução PRAM para o problema.

Algoritmo 4: Algoritmo PRAM para a subsequência de soma máxima.

Entrada: (1) Sequência $Q[1...n]$ de números positivos e negativos.
Saída: (1) Subsequência de soma máxima de Q.

1: Compute **em paralelo** as somas de prefixos de Q em um vetor *PSUM*.
2: Compute **em paralelo** as somas de sufixos de Q em um vetor *SSUM*.
3: Compute **em paralelo** o sufixo máximo de PSUM em um vetor *SMAX*.
4: Compute **em paralelo** o prefixo máximo de SSUM em um vetor *PMAX*.
5: **for** $i = 1$ to n **do**
6: **in paralell**
7: $(a) M_s[i] := PMAX[i] - SSUM[i] + Q[i]$.
8: $(b) M_p[i] := SMAX[i] - PSUM[i] + Q[i]$.
9: $(c) M[i] := M_s[i] + M_p[i] - Q[i]$.
10: **end for**
11: Localize a sequência contígua de máximos de M e armazene os valores correspondentes de Q em *MSQ*.
12: Imprima a subsequência de soma máxima *MSQ*.

Custo do Algoritmo

O custo do Algoritmo 4 é expresso em termos de seus passos: os passos das linhas 1 e 2 podem ser executados em tempo $\mathrm{O}(\log n)$ usando algoritmos conhecidos para soma de prefixos e soma de sufixos. Passos das linhas 3 e 4 podem ser resolvidos por $n/\log n$ processadores em uma máquina EREW-PRAM em tempo $\mathrm{O}(\log n)$ utilizando variações de algoritmos paralelos de soma de prefixos e soma de sufixos. Similarmente, os passos de 5 e 6 podem ser determinados em tempo ótimo igual a $\mathrm{O}(\log n)$. O tempo total do algoritmo é igual a $\mathrm{O}(\log n)$ [35]. Visto que o melhor algoritmo sequencial requer tempo $\mathrm{O}(n)$ [2], temos que Algoritmo 4 tem custo ótimo.

Um Exemplo

Para uma entrada $Q = \{3, 2, -7, 11, 10, -6, 4, 9, -6, 1, -2, -3, 4, -3, 0, 2\}$. Os vetores *PSUM*, *SSUM*, *SMAX*, *PMAX* e Mresultantes da aplicação do Algoritmo 4 são os ilustrados pela Tabela 3.3 [35]. Observando M, descobrimos que a soma máxima em uma subsequência contígua é igual a 28 (Soma Máxima = **28**). A subsequência de soma máxima se estende por todos os elementos de M iguais à soma máxima, no caso a subsequência de soma máxima compreeende os elementos do intervalo [4,8] da sequência de entrada Q. São eles: $\{11, 10, -6, 4, 9\}$.

$Q =$	3	2	-7	11	10	-6	4	9	-6	1	-2	-3	4	-3	0	2
$PSUM =$	3	5	-2	9	19	13	17	26	20	21	19	16	20	17	17	19
$SSUM =$	19	16	14	21	10	0	6	2	-7	-1	-2	0,	3	-1	2	2
$SMAX =$	26	26	26	26	26	26	26	26	21	21	20	20	20	19	19	19
$PMAX =$	19	19	19	21	21	21	21	21	21	21	21	21	21	21	21	21
$M =$	26	26	26	**28**	**28**	**28**	**28**	**28**	23	23	22	22	22	21	21	21

Tabela 3.3: Os cálculos dos vetores do Algoritmo 4.

3.3.2 Algoritmo Paralelo BSP/CGM

Apresentamos agora um outro algoritmo paralelo para o problema da subsequência de soma máxima composta de n números. O algoritmo foi proposto por Alves *et al.*, no modelo BSP/CGM, com p processadores e com memória local $O(n/p)$ [2]. Este algoritmo requer um número constante de rodadas de comunicação.

A seguir apresentamos as principais definições e lemas desse trabalho: considere a sequência de entrada $Q = (x_1, x_2, \ldots, x_n)$ de tamanho n. Sem perda de generalidade, assuma que n é divisível por p. Em seguida, particione n em intervalos iguais de tamanho p, de forma que cada intervalo seja armazenado em um processador p diferente: o intervalo $(x_1, x_2, \ldots, x_{n/p})$ de Q é armazenado no processador 1, o intervalo $(x_{n/p+1}, \ldots, x_{2n/p})$ de Q é armazenado no processador 2, e assim por diante. Demonstra-se que cada intervalo I de tamanho n/p pode ser representado por cinco vetores (subsequências) concatenados: P, N_1, M, N_2 e S. Considerando I como $(y_1, y_2, \ldots, y_{n/p})$ temos [2] :

1. $M = (y_a, \ldots, y_b)$ é a subsequência de soma máxima de I, com pontuação $T_M \geq 0$.
2. $P = (y_1, \ldots, y_r)$ é o **prefixo máximo** de $(y_1, \ldots, y_{a-1})$, com pontuação $T_p \geq 0$. Trata-se da maior soma de prefixos positiva localizada.
3. $S = (y_s, \ldots, y_{n/p})$ é o **sufixo máximo** de $(y_{b+1}, \ldots, y_{n/p})$, com pontuação $T_S \geq 0$. Trata-se da maior soma de sufixos positiva localizada.
4. N_1 é o intervalo entre P e M, com pontuação $T_{N_1} \leq 0$
5. N_2 é o intervalo entre M e S, com pontuação $T_{N_2} \leq 0$.

Observação: Para cada processador o algoritmo encontra a subsequência M de I, o prefixo máximo P e o sufixo máximo S (conforme definidos acima), além das possíveis sequências N_1 e N_2. No caso de todos os y_i serem números negativos, então tem-se que M, P, e S são vazias com $T_M = T_P = T_S = 0$, N_1 será a entrada I e N_2 será vazio com $T_{N_2} = 0$ [2].

Variações de Valores de um Intervalo I

Percebe-se que existem possibilidades de variações para os valores compreendidos em um intervalo I. Os Lemas 3.3.2 e 3.3.3 explicitam estas possibilidades [2].

Lema 3.3.2. *Se M é não vazia, então um dos três casos deve acontecer:*

1. *P está á esquerda de M, com $r < a$ e com N_1 entre eles;*
2. *M é igual a P, com $a = 1$ e $b = r$. Neste caso, não existe N_1;*
3. *M é uma subsequência própria de P, com $a > 1$ e $b = r$. Neste caso não existe N_1.*

Demonstração. Se $r < a$, temos o caso 1. Vamos supor agora que $r \geq a$, que acontece nos casos 2 e 3: com $r \geq a$, se $r < b$ então a pontuação de $(y_a, \ldots, y_r)$ é menor que T_M, de modo que a pontuação de $(y_{r+1}, \ldots, y_b)$ é positiva. Então o prefixo $(y_1, \ldots, y_b)$ teria uma pontuação maior que T_P, uma contradição. Similarmente, com $r \geq a$ e $b < r$, $(y_{b+1}, \ldots, y_r)$ teria uma pontuação positiva e $(y_a, \ldots, y_r)$ teria uma pontuação maior que T_M, novamente uma contradição. Assim, $r \geq a$ deve conduzir a $r = b$ (casos 2 e 3). □

A demonstração do Lema 3.3.3, que trata do sufixo máximo é similar.

Lema 3.3.3. *Se M é não vazia, então um dos seguintes casos deve acontecer:*

- *S está à direita de M, com $s > b$ e com N_2 entre eles;*
- *M é igual a S, com $a = s$ e $b = n/p$. Neste caso, não existe N_2;*
- *M é uma subsequência própria de S, com $a = s$ e $b < n/p$. Neste caso não existe N_2.*

Os cinco valores T_P, T_{N_1}, T_M, T_{N_2} e T_S, para cada intervalo de I, são utilizados no algoritmo paralelo. Quando M e P não são disjuntos, isto é, M é uma subsequência de P, conforme ilustrado na Figura 3.4, redefine-se T_P para 0 e T_{N_1} como a pontuação não positiva do prefixo que imediatamente precede M. Uma adaptação semelhante também é realizada com S e T_{N_2} quando M e S não são disjuntos.

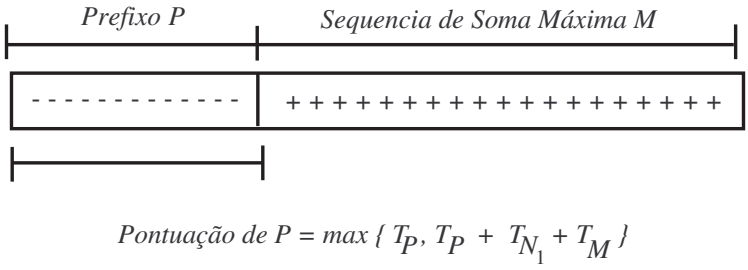


Figura 3.4: M como subsequência de P.

Após estas adaptações, temos:

- Pontuação de $P = max\{T_P, T_P + T_{N_1} + T_M\}$,
- Pontuação de $S = max\{T_M + T_{N_2} + T_S, T_S\}$,
- $T_P + T_{N_1} + T_M + T_{N_2} + T_S = \sum_{i=1}^{n/p} y_i$.

Desta forma é possível construir uma sequência composta por cinco números, que representa a mesma sequência I da entrada, mantendo a mesma pontuação. Os zeros, aparentemente inúteis, são mantidos para simplificar a computação do último passo do algoritmo paralelo. Após o cálculo dos cinco valores, cada processador os enviam para o processador 1. O processador 1 soluciona o problema da subsequência de soma máxima com os 5 valores recebidos de cada processador, sequencialmente, em tempo $O(p)$. O pseudocódigo é apresentado pelo Algoritmo 5.

Algoritmo 5: Algoritmo BSP/CGM para a subsequência de soma máxima.

Entrada: (1) A sequência de entrada particionada entre p processadores.
Saída: (1) A subsequência de soma máxima da sequência da entrada.

1: Considere a sequência armazenada em cada processador dada por $I = (y_1, y_2, ..., y_{n/p})$. Cada processador deve obter a subsequência máxima M de I com pontuação T_M.
2: Cada processador deve obter o prefixo máximo P com soma T_P, o sufixo máximo S com soma T_S, o intervalo entre P e M definido como N_1 com soma T_{N_1} e o intervalo entre M e S definido como N_2 com soma T_{N_2}.
3: Considerando o Lema 3.3.3, se for necessário, redefine-se os valores apropriados para T_P, T_{N_1}, T_M, T_{N_2} e T_S
4: Cada processador deve enviar os cinco valores T_P, T_{N_1}, T_M, T_{N_2} e T_S para o processador P_1.
5: O processador P_1 recebe os 5 valores de cada processador e calcula a subsequência de soma máxima com os valores recebidos.
6: Considere a subsequência de soma máxima como $m_1, \ldots, m_k$; o processador que armazena m_1 pode facilmente localizar o índice de ínicio da subsequência de soma máxima correspondente a sequência original, enquanto que o processador que armaneza m_k pode localizar o índice de fim.

Um exemplo

É fácil perceber que a subsequência que considera os cinco valores (T_P, T_{N_1}, T_M, T_{N_2} e T_S) ainda corresponde a mesma subsequência da sequência original. Se a subsequência de soma máxima estiver completamente contida em um dos intervalos mantidos pelos p processadores, a correspondência é direta. Caso contrário, ela começa dentro de um intervalo (no sufixo máximo), se estende por zero ou mais intervalos de inteiros, e termina dentro de outro intervalo (no prefixo máximo). Os cinco valores de cada processador contêm todas as informações necessárias para retornar a subsequência original. Como exemplo, considere a sequência de entrada Q, composta pelos seguintes elementos: $Q = \{1, 1, -5, 20, 20, -5, 1, 1, 1, 1, -5, 20, 20, -5, 1, 1, 1, 1, -5, 20, 20, -5, 1, 1, 1, 1, -5, 20, 20, -5, 1, 1\}$. Considere que o número de processadores disponíveis é igual a 4(quatro). A distribuição dos elementos de Q nos 4 processadores é representada pela Tabela 3.4. Cada processador armaneza n/p valores de Q. No exemplo, $\frac{32}{4}$=8.

P_1	[$1, 1, -5, 20, 20, -5, 1, 1$]
P_2	[$1, 1, -5, 20, 20, -5, 1, 1$]
P_3	[$1, 1, -5, 20, 20, -5, 1, 1$]
P_4	[$1, 1, -5, 20, 20, -5, 1, 1$]

Tabela 3.4: Divisão de Q entre quatro processadores: P_1, P_2, P_3 e P_4.

O próximo passo consiste em calcular a subsequência de soma máxima em cada um dos processadores localmente, reduzindo n/p para 5 valores, de acordo com os Lemas 3.3.2 e 3.3.3. Os cinco valores encontrados em cada processador são ilustrados pela Tabela 3.5. Em seguida, cada processador deve enviar seu vetor de 5 valores para o processador 1.

P_1	[$2, -5, 40, -5, 2$]
P_2	[$2, -5, 40, -5, 2$]
P_3	[$2, -5, 40, -5, 2$]
P_4	[$2, -5, 40, -5, 2$]

Tabela 3.5: Os cinco valores encontrados em cada processador.

Por fim, o processador 1 (P_1) calcula a subsequência de soma máxima resultante da concatenação dos 5 valores recebidos de cada processador (incluso os seus), conforme ilustrado pela Tabela 3.6. Neste exemplo em P_1 e P_4 estarão os índices de início e fim da subsequência de soma máxima, visto que o primeiro e o último valor igual a 40 irão estar na subsequência de soma máxima.

P_1	[$2, -5,$ **40, -5, 2, 2, -5, 40, -5, 2, 2, -5, 40, -5, 2, 2, -5, 40**$, -5, 2$]

Tabela 3.6: Encontrando a subsequência de soma máxima.

Custo do Algoritmo

O teorema a seguir é decorrente do Algoritmo 5 e discute o custo do algoritmo. A demonstração pode ser conferida no trabalho original [2]:

Teorema 3.1. *O Algoritmo 5 localiza a subsequência de soma máxima após um número constante de rodadas de comunicação, com transmissão de* $\mathrm{O}(p)$ *números e consumindo* $\mathrm{O}(n/p)$ *de tempo de computação local.*

3.4 Problemas Relacionados

Discutiremos a seguir três problemas derivados do problema da subsequência de soma máxima. Algoritmos para estes problemas foram objetivo de nosso trabalho. Como vimos na seção 3.1, o problema da subsequência de soma máxima consiste em determinar dentre todas as subsequências, a subsequência $M = (x_i, \ldots, x_j)$ que possui maior soma (acumulada) $(\sum_{k=i}^{j} x_k)$ [4]. Na sequência representada pela Figura 3.5 existem três subsequências disjuntas de soma máxima: a primeira é composta pelos elementos (x_1, x_2) e possui comprimento 2. A segunda consiste apenas do elemento (x_4) e possui comprimento 1. A terceira e última é composta pelos elementos $(x_6, x_7,\ x_8,\ x_9)$ e possui comprimento 4. As três subsequências possuem o mesmo valor de soma máxima (15). Por simplicidade, e sem perda de generalidade, assumimos que os números da sequência original são sempre inteiros.

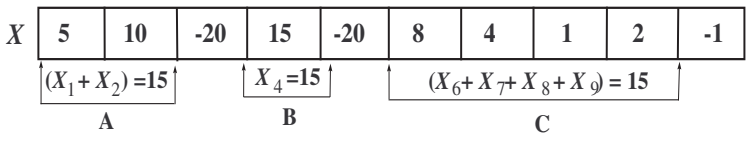


Figura 3.5: As três subsequências de soma máxima de X com a mesma soma.

Neste contexto, ao menos três novos problemas aparecem: a maior subsequência de soma máxima, a menor subsequência de soma máxima e o número de subsequências disjuntas de soma máxima. Nos primeiros dois problemas, o comprimento (maior/menor) está relacionado com o número de elementos que compõem as subsequências.

3.5 Algoritmos Propostos

Apesar de já existir um algoritmo paralelo eficiente BSP/CGM (Algoritmo 5) para o problema geral da subsequência de soma máxima [2], não conseguimos utilizá-lo para encontrar, diretamente, a maior ou menor subsequência de soma máxima. O principal obstáculo é a atividade de compressões que o algoritmo 5 realiza durante sua execução. Por outro lado, observamos que utilizando as ideias do algoritmo PRAM proposto por Perumalla e Deo [35], seria possível desenvolver um novo algoritmo paralelo BSP/CGM para o problema, de forma que, utilizando a saída final, teríamos como encontrar a maior ou menor subsequência de soma máxima e o número de subsequências disjuntas de soma máxima.

A Figura 3.6 ilustra um exemplo do vetor saída do algoritmo de Perumalla e Deo [35] (Algoritmo 4). Neste exemplo existem três subsequências disjuntas de soma máxima, entretanto elas possuem diferentes tamanhos. A partir dos conceitos deste algoritmo, desenvolvemos soluções paralelas e comprovadamente eficientes para solucionar quatro problemas:

- A subsequência de soma máxima (problema geral).
- A maior subsequência de soma máxima.
- A menor subsequência de soma máxima.
- O número de subsequências disjuntas de soma máxima.

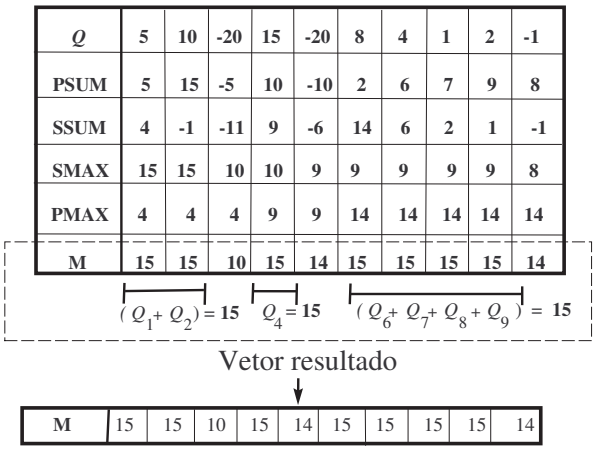


Figura 3.6: O valor 15 é o maior e representa o valor de soma máxima de Q.

3.5.1 O Algoritmo BSP/CGM Proposto

Incialmente projetamos a solução BSP/CGM que soluciona o problema geral da subsequência de soma máxima. A solução é aqui representada pelo Algoritmo 6.

Algoritmo 6: Algoritmo BSP/CGM proposto - problema geral

Entrada: (1) Um conjunto de P processadores; (2) O número i que rotula cada processador $p_i \in P$, onde $1 \leq i \leq P$; (3) Uma sequência Q de n inteiros.
Saída: (1) O vetor $\mathbf{M}[1 \ldots n]$ de inteiros com todas as subsequências disjuntas de soma máxima; (2) O valor soma máxima de $\mathbf{M}$.

1: Utilize o conjunto de processadores P e o vetor original Q para obter o vetor de soma de prefixos *PSUM*.
2: Utilize o conjunto de processadores P e o vetor original Q para obter o vetor de soma de sufixos *SSUM*.
3: $SMAX \leftarrow$ **Sufixo_Máxima**(P, $PSUM$).
4: $PMAX \leftarrow$ *Prefixo_Máxima*(P, $SSUM$).
5: Processador p_1 envia n/p elementos de cada vetor Q, *PSUM*, *SSUM*, *SMAX* e *PMAX* para cada processador $p_i \in P$.
6: Cada processador p_i obtém os vetores locais
$LocalMS \leftarrow PMAX(n/p)$-$SSUM(n/p)$+$Q(n/p)$
$LocalMP \leftarrow SMAX(n/p)$-$PSUM(n/p)$+$Q(n/p)$.
$LocalM \leftarrow LocalMS(n/p)$+$LocalMP(n/p)$-$Q(n/p)$.
7: Cada processador p_i envia o vetor *LocalM* para o processador p_1, que computa fo array $M = [LocalM_{p_{1_1}}, \ldots, LocalM_{p_{1_{n/p}}}, \ldots, LocalM_{p_{p_1}}, \ldots, LocalM_{p_{p_{n/p}}}]$.
8: Utilize o conjunto de processadores P para encontrar o maior valor númérico em M (soma máxima).

Algoritmo 7: Algoritmo BSP/CGM de sufixos máximos.

Entrada: (1) Vetor $PSUM[1 \ldots n]$ de inteiros.
Saída: (1) Vetor $SMAX[1 \ldots n]$ de inteiros.

1: Processador p_1 envia n/p elementos de *PSUM* para cada processador $p_i \in P$.
2: Em cada processador p_i, uma operação de propagação de máximos deve ser executada em cada vetor $PSUM(n/p)$. A operação é iniciada no elemento de índice n/p e executa até o elemento de índice 1. Ao final o maior elemento está na primeira posição.
3: Por fim, um operação de propagação de máximos também é executada entre os processadores, sendo que cada processor p_i irá trabalhar com os processadores p_k, onde $k > i$.

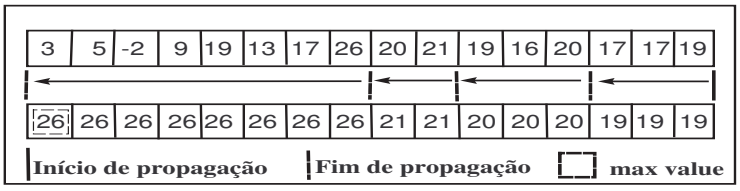
Algoritmo 8: Algoritmo BSP/CGM de prefixos máximos.

Entrada: (1) Vetor $SSUM[1 \ldots n]$ de inteiros.
Saída: (1) Vetor $PMAX[1 \ldots n]$ de inteiros.

1: Processador p_1 envia n/p elementos de $SSUM$ para cada processador $p_i \in P$.
2: Em cada processador p_i, uma operação de propagação de máximos deve ser executada em cada vetor $SSUM(n/p)$. A operação é iniciada no elemento de índice 1 e executa até o elemento de índice n/p. Ao final o maior elemento está na posição de índice n/p.
3: Por fim, uma operação de propagação de máximos também é executada entre os processadores, sendo que cada processador p_i irá trabalhar com os processadores p_k, onde $k < i$.

Os Algoritmos 7 (**Sufixo_Maxima**) e 8 (**Prefixo_Maxima**) possuem os mesmo nomes descritos no trabalho original de Perumalla e Deo [35]. Nestes algoritmos duas operações diferentes são executadas, entretanto ambas consistem basicamente da propagação de valores máximos [35].

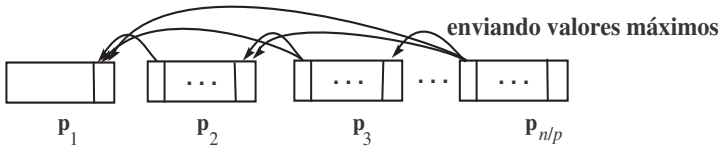
A primeira operação ocorre nos vetores locais de cada processador. Neste caso, valores maiores são propagados substituindo valores menores. A substituição é executada até que novos valores maiores sejam encontrados, o que inicia uma nova propagação de máximos, como ilustrado na Figura 3.7 (a). A segunda operação ocorre entre cada processador e os valores máximos de cada processador subsequente ou precedente, como ilustrado na Figura 3.7 (b).



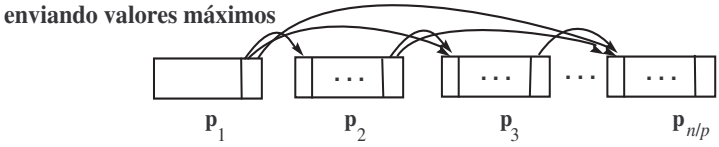
a) Algoritmo **Sufixo_Máxima** em cada processador

b) Algoritmo **Prefixo_Máxima** em cada processador

(a) Operações locais.



a) Algoritmo **Sufixo_Máxima** entre processadores



b) Algoritmo **Prefixo_Máxima** entre processadores

(b) Operações globais.

Figura 3.7: Operações de propagação de máximos.

As operações de propagação de valores máximos são especializações da operação básica de soma de prefixos [35]. Para implementá-las é preciso então adequá-las ao contexto, conforme descrito no algoritmo. Existem diversas técnicas de soluções paralelas para soma de prefixos, algumas bastante eficientes, seja em ambiente de memória compartilhada [6] ou distribuída [13].

Complexidade

Utilizando algoritmos paralelos BSP/CGM para soma de prefixos [13], os passos 1 e 2 do Algoritmo 6 podem ser computados utilizando p processadores em tempo $O(n/p)$ e com um número constante de rodadas de comunicação. Nos passos 3 e 4, a invocação dos algoritmos 7 (**Sufixo_Maxima**) e 8 (**Prefixo_Maxima**), utiliza, respectivamente, p processadores e consome tempo $O(n/p)$ com um número constante de rodadas de comunicação. Os passos de 5 a 7 também podem ser computados utilizando p processadores em tempo $O(n/p)$ e com um número constante de rodadas de computação. Finalmente, um algoritmo paralelo BSP/CGM de redução para máximo é aplicado no passo 8, ele utiliza p processadores em tempo $O(n/p)$ e com um número constante de rodadas de comunicação. Concluímos então que o Algoritmo 6 computa a subsequência de soma máxima utilizando p processadores em tempo $O(n/p)$ e com um número constante de rodadas de comunicação. A correção do algoritmo decorre diretamente do algoritmo proposto por Perumalla e Deo [35].

3.5.2 O Algoritmo BSP/CGM para os Problemas Relacionados

Neste trabalho também desenvolvemos um algoritmo paralelo capaz de solucionar os problemas relacionados com o problema geral da subsequência de soma máxima, são eles: (1) a maior subsequência de soma máxima, (2) a menor subsequência de soma máxima e o (3) número de subsequências disjuntas de soma máxima. O vetor saída gerado pelo algoritmo 6 é o ponto de partida destes algoritmos. Por simplicidade, as estratégias de solução para os três problemas foram compiladas em apenas um algoritmo, representado aqui pelo Algoritmo 9. Discutimos a seguir as estratégias para cada um dos três problemas:

Algoritmo 9: Algoritmo BSP/CGM proposto - problemas relacionados.

Entrada: (1) Vetor **M**$[1 \ldots n]$ de inteiros (Saída do Algoritmo 6);
(2) O valor de soma máxima (**maxsum**) (Saída do Algoritmo 6).
Saída: (1) O **maior valor** = comprimento da maior subsequência de soma máxima
(2) O **menor valor** = comprimento da menor subsequência de soma máxima;
(3) O **valor de contagem** = Número de subsequências disjuntas de soma máxima.

1: Em paralelo utilize o número de processadores P para obter o vetor binário Binary$[1 \ldots n]$, da seguinte forma:
2: **if** ($M[i]$ = maxsum) **then**
3: Binary$[i] \leftarrow 1$.
4: **else**
5: Binary$[i] \leftarrow 0$.
6: **endif**
7: Em paralelo utilize o número de processadores P para obter o vetor binário BinaryLK$[1 \ldots n]$, da seguinte forma: BinaryLK$[1 \ldots n] \leftarrow$ Vetor de distâncias calculado via operações de *list ranking* sobre todas as subsequências disjuntas de valores iguais a 1 em Binary$[1 \ldots n]$.
8: maior valor $\leftarrow$ **max**(BinaryLK$[1 \ldots n]$).
9: Em paralelo utilize o número de processadores P para modificar o vetor binário Binary$[1 \ldots n]$, da seguinte forma:
10: **if**((i != 0 and BinaryLK$[i-1]$!=0) or (BinaryLK$[i]$=0)) **then**
11: BinaryLK[i] $\leftarrow$ maior valor.
12: **else**
13: valor de contagem $\leftarrow$ valor de contagem + 1.
14: **endif**
15: menor valor $\leftarrow$ **min**(BinaryLK$[1 \ldots n]$).

1. A solução para o primeiro dos três problemas é composta por dois passos: no primeiro passo, uma operação de conversão em mapa de bits é aplicada no vetor M. Neste caso, em todas as posições de M que são iguais ao valor máximo **maxsum**, armazenamos o valor 1 ou o valor 0, caso contrário. O novo vetor gerado é chamado de *Binary*. No segundo passo, um algoritmo paralelo de *list ranking*, ligeiramente modificado, é aplicado em todas as subsequências disjuntas e contíguas de 1's de *Binary*. A modificação preve que distâncias com um único valor 1 possuem tamanho um. O novo vetor gerado é chamado *BinaryLK*. Por fim, para encontrar o comprimento da maior subsequência de soma máxima **maior valor**, precisamos aplicar um algoritmo paralelo de redução para máximo sobre o array *BinaryLK*;

2. Na solução do segundo problema, o vetor *BinaryLK* é utilizado novamente. Entretanto, a aplicação direta de um algoritmo paralelo de redução sobre o vetor *BinaryLK* não resolve o problema, devido a presença de valores iguais a 0. Por isto, em nossa estratégia, o vetor *BinaryLK* precisou ser modificado. Em todas as posições que não são o início de subsequências de distâncias, o maior valor é armazenado (comprimento da maior subsequência de soma máxima). Após esta mudança, um algoritmo paralelo de redução é aplicado sobre o vetor *BinaryLK*, para encontrar o comprimento da menor subsequência de soma máxima (menor valor);

3. A solução para o problema 3 é similar à solução do problema 2. Em nossa estratégia o vetor original *BinaryLK* é novamente utilizado, entretanto o valor que será armazenado agora será igual a 0. No fim, um algoritmo de contagem é aplicado sobre o vetor *BinaryLK* para calcular o número de elementos que são diferentes de 0. A resposta corresponde ao número de subsequências disjuntas de soma máxima.

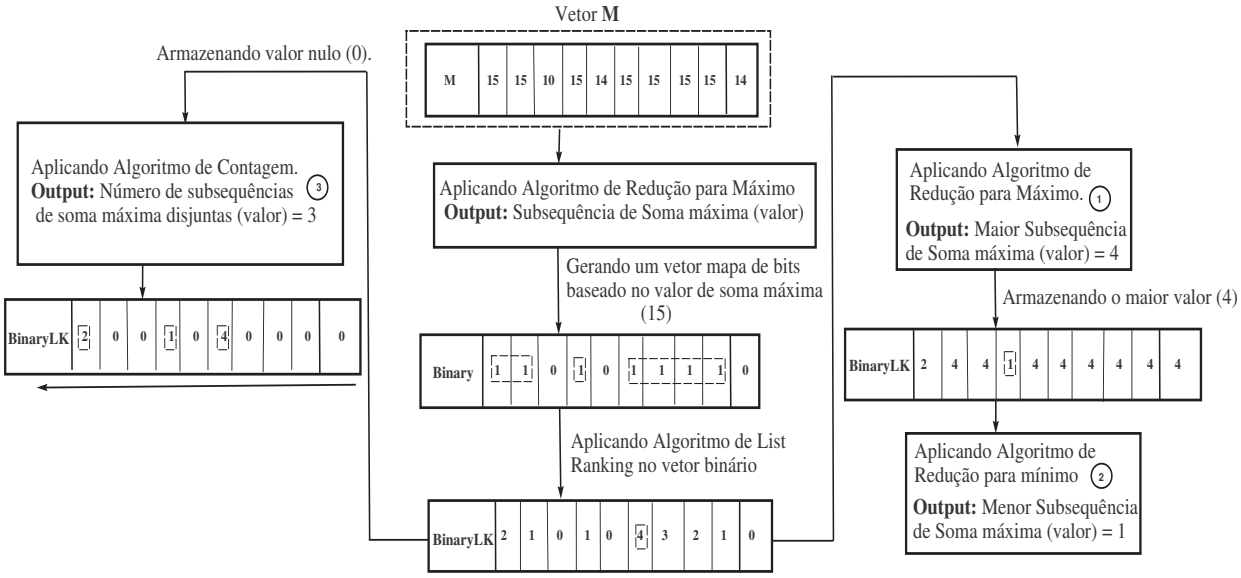


Figura 3.8: Problemas que podem ser solucionados a partir do vetor saída M do Algoritmo 6.

A computação da soma de prefixos/sufixos, list ranking e reduções podem ser feitas com a utilização de algoritmos paralelos BSP/CGM eficientes descritos em [14, 13]. A Figura 3.8 ilustra a relação entre as rotinas internas listadas pelo algoritmo 9 e as soluções que podem ser obtidas. O ponto de partida é o vetor saída do Algoritmo 6. O Algoritmo 9 computa corretamente e retorna os três valores esperados.

Complexidade

Utilizando algoritmos paralelos BSP/CGM para redução, os passos do Algoritmo 9 associados com os comprimentos da subsequência de soma máxima podem ser computados utilizando p processadores em tempo $O(n/p)$ e com um número constante de rodadas de comunicação. Particularmente, o passo que utiliza *list ranking* pode ser computado utilizando p processadores, com $O(\log p)$ rodadas de comunicação e consumindo $O(n/p)$ computação local em cada rodada.

3.6 Implementações e Resultados

Os algoritmos foram implementados usando os modelos de memória compartilhada e distribuída:

1. Inicialmente desenvolvemos a implementação para o problema geral da subsequência de soma máxima. Neste caso, para fins de comparação de desempenho trabalhamos com duas versões: a primeira foi projetada para ser executada em um cluster de computadores com memória distribuída, utilizando MPI [2]. A segunda foi projetada para executar em uma máquina com GPU e memória compartilhada, utilizando CUDA. As duas versões possuem diferentes estratégias: a versão MPI é baseada em um algoritmo paralelo BSP/CGM desenvolvida por Alves *et al.* [2] e envolve compressão de dados. A versão CUDA foi construída a partir do nosso algoritmo BSP/CGM (Algoritmo 6). A diferença fundamental entre as duas versões está relacionada com a sequência de entrada. Na versão MPI, a sequência de entrada é compactada uma série de vezes. Já na versão CUDA, a sequência original persiste até o final do algoritmo. Como é ilustrado pela Figura 3.8, nossos algoritmos para os problemas relacionados foram solucionados utilizando como ponto de partida uma sequência com o mesmo comprimento da sequência de entrada;

2. As implementações para os três problemas relacionados foram contruídas apenas em CUDA. Para isto utilizamos o Algoritmo 6 que não comprime a sequência de entrada. Para estas implementações utilizamos algumas soluções eficientes disponibilizadas em CUDA, principalmente para as operações de reduções para máximo/mínimo [11] e de *list ranking* [37].

3.6.1 Resultados

A seguir apresentamos os resultados alcançados pelo algoritmos. Quatro casos de comparação são apresentados. Em todos os casos a sequência de entrada foi gerada aleatoriamente.

O primeiro caso ilustra a comparação de tempos de execução dos algoritmos para o problema geral da subsequência de soma máxima. Ainda que a melhor solução sequencial para o problema seja rápida e possua complexidade de tempo de $O(n)$ [4], nossa versão paralela em CUDA para o Algoritmo 6 obteve um desempenho até nove vezes melhor, como pode ser observado na Figura 3.9 e descrito na Tabela 3.7.

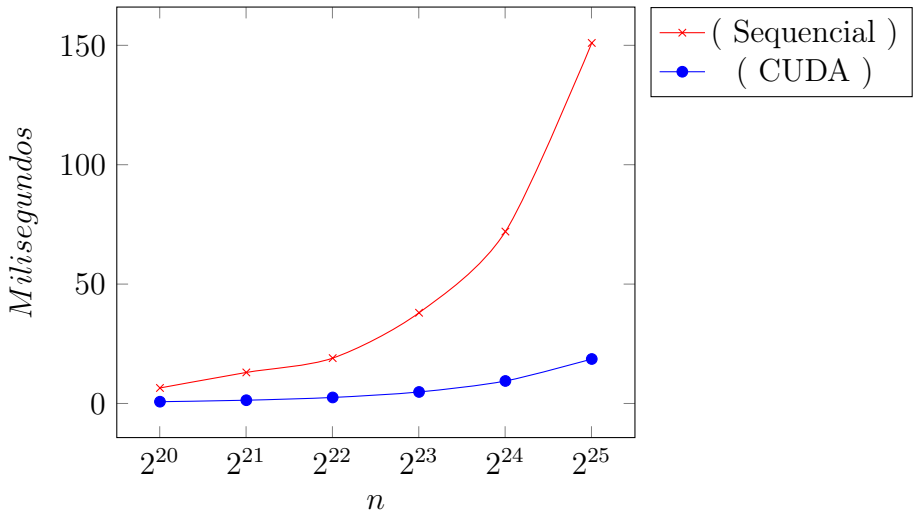


Figura 3.9: Sequencial × Paralelo em CUDA.

n milhões	Tempo Sequencial	Desvio Padrão	Tempo CUDA	Desvio Padrão	*Speedup*
1,048.576	6.50000	2.29128	0.74437	0.00216	8.73221
2,097.152	13.00000	4.58257	1.36492	0.00357	9.52437
4,194.304	19.00000	3.00000	2.52194	0.00919	7.53388
8,388.608	38.00000	4.00000	4.82443	0.01941	7.87658
16,777.216	72.00000	4.00000	9.42475	0.01517	7.63946
33,554.432	151.00000	3.00000	18.62913	0.00997	8.10558

Tabela 3.7: Tempos de execução (**Milisegundos**): Sequencial × CUDA.

O segundo caso também ilustra uma comparação entre os tempos de execução de dois algoritmos para o problema geral da subsequência de soma máxima, um estritamente sequencial [4] e o outro paralelo em MPI [2]. Os algoritmos foram executados em ambiente de *cluster*. A melhor execução para o algoritmo MPI foi alcançada utilizando 64 processadores. Neste caso, a versão paralela obteve um desempenho quatorze vezes melhor, como pode ser observado na Figura 3.10 e descrito na Tabela 3.8.

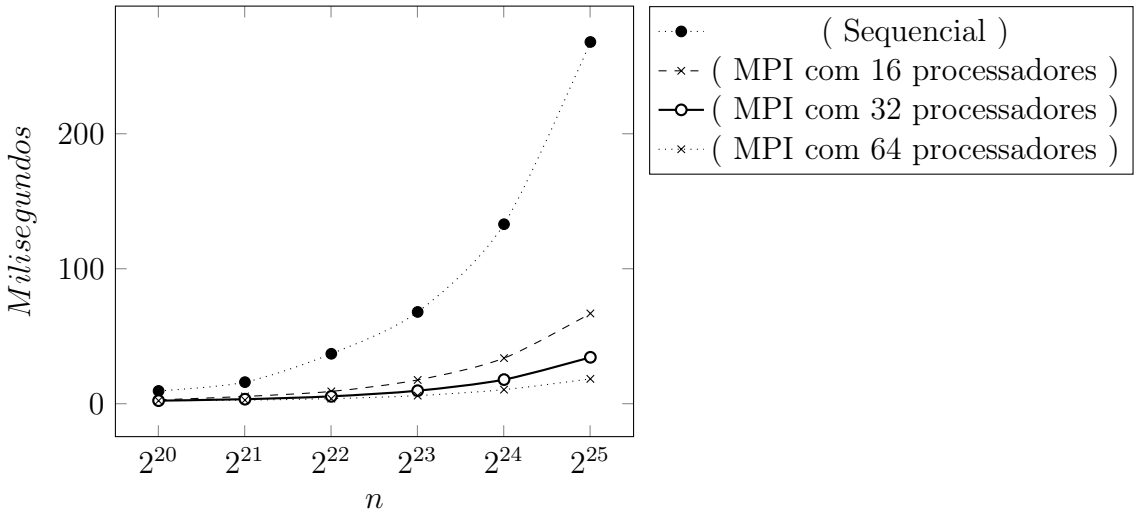


Figura 3.10: Sequencial × MPI Paralelo (três execuções: 16, 32, 64 (processadores)).

n mihões(s)	Tempo Sequencial	Desvio Padrão	Tempo MPI (16 p)	Desvio Padrão	Tempo MPI (32 p)	Desvio Padrão	Tempo MPI (64 p)	Desvio Padrão	*Speedup* (MPI/64 p)
1,048.576	9.50000	1.58114	2.87492	0.04287	2.28464	0.02575	2.20983	0.10873	4.29897
2,097.152	16.00000	5.16398	5.39832	1.34498	3.36065	0.12312	2.69845	0.11492	5.92933
4,194.304	37.00000	4.83046	9.15089	0.11829	5.47266	0.10671	3.78101	0.14212	9.78575
8,388.608	68.00000	4.21638	17.62521	0.12682	9.66182	0.14289	6.00693	0.13388	11.32026
16,777.216	133.00000	4.83046	33.78429	0.13014	17.8839	0.06888	10.44182	0.49717	12.73724
33,554.432	268.00000	4.21638	66.89167	0.41443	34.4008	0.40952	18.41991	0.17940	14.54947

Tabela 3.8: Tempos de execução (**Milissegundos**): Seq. × MPI.

No terceiro caso comparamos a versão MPI, de melhor tempo (64 processadores), com a versão CUDA para o problema geral. Neste caso, observamos que ambas as versões se mostraram competitivas, caminhando para equivalência, como é ilustrado na Figura 3.11 e descrito pela Tabela 3.9.

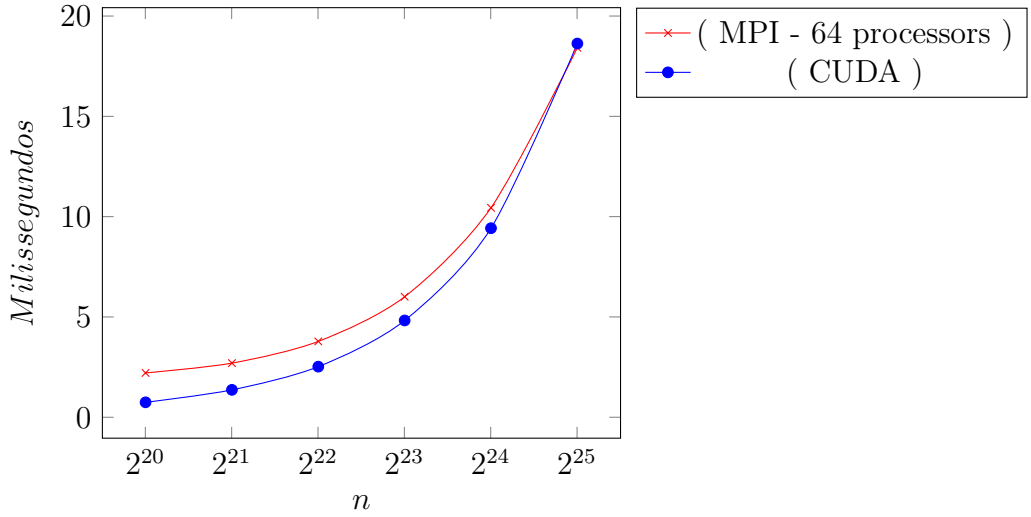


Figura 3.11: MPI × CUDA.

n milhões	Tempo MPI	Desvio Padrão	Tempo CUDA	Desvio Padrão	*Speedup*
1,048.576	2.20983	0.10873	0.74437	0.00218	2.96873
2,097.152	2.69845	0.11492	1.36492	0.00357	1.97700
4,194.304	3.78101	0.14212	2.52194	0.00919	1.49924
8,388.608	6.00693	0.13388	4.82443	0.01941	1.24511
16,777.216	10.44182	0.49717	9.42475	0.01517	1.10791
33,554.432	18.41991	0.17940	18.62913	0.00997	0.98876

Tabela 3.9: Tempos de execução (**Milissegundos**): MPI × CUDA.

No último caso, comparamos os tempos dos algoritmos para os problemas relacionados. Foram realizadas comparações entre uma versão sequencial e nossa versão CUDA (Algoritmo 9). Neste caso, modificamos o algoritmo sequencial para o problema geral, para também solucionar os problemas relacionados. A comparação entre o algoritmo sequencial modificado e o algoritmo CUDA é apresentada pela Figura 3.12 e pela Tabela 3.10.

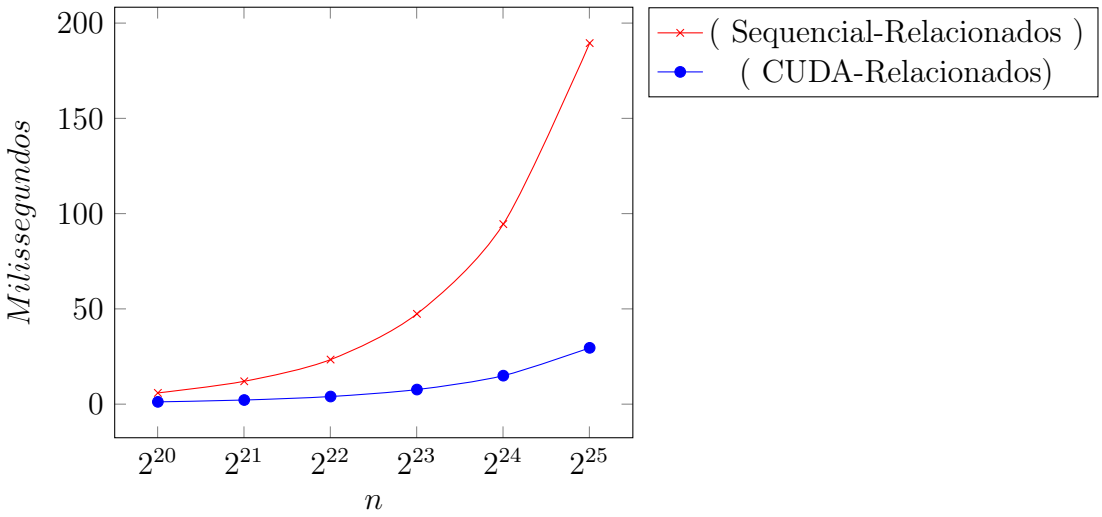


Figura 3.12: Sequencial-Problemas Relacionados × CUDA-Problemas Relacionados.

n milhões	Tempo Sequencial	Desvio Padrão	Tempo CUDA	Desvio Padrão	*Speedup*
1,048.576	5.90000	0.31623	1.20034	0.003853	4.91527
2,097.152	12.00000	0.47140	2.16783	0.003887	5.53549
4,194.304	23.40000	0.51639	4.00009	0.009847	5.84987
8,388.608	47.40000	0.51639	7.65928	0.022247	6.18857
16,777.216	94.50000	0.52705	14.9523	0.015626	6.32009
32,554.432	189.50000	0.52705	29.5581	0.010664	6.41110

Tabela 3.10: Tempos de execução (**Milissegundos**): Sequencial × CUDA.

3.7 Considerações Finais do Capítulo

Neste capítulo apresentamos soluções paralelas para o problema geral da subsequência de soma máxima e para três problemas relacionados: a maior subsequência de soma máxima, a menor subsequência de soma máxima e o número de subsequências disjuntas de soma máxima. Não é de nosso conhecimento a existência de algoritmos paralelos para estes problemas relacionados. Nossos algoritmos utilizaram p processadores e consumiram tempo paralelo $O(n/p)$, com um número constante de rodadas de comunicação para o problema geral da subsequência de soma máxima e $O(\log p)$ rodadas de comunicação, com $O(n/p)$ de computação local por rodada, para os algoritmos relacionados. O bom desempenho dos algoritmos paralelos foi confirmado por meio de resultados experimentais, tanto nos modelos de memória compartilhada como distribuída.

A solução proposta para o problema geral, além de ter a mesma complexidade do algoritmo BSP/CGM anterior possibilita a solução de novos problemas. Além disso, a implementação do algoritmo em CUDA reforça a robustez da extensão do modelo BSP/CGM no projeto e análise de algoritmos paralelos.

CAPÍTULO 4

A SUBMATRIZ DE SOMA MÁXIMA

Neste capítulo ampliamos nosso objetivo para uma extensão natural e importante do problema da subsequência de soma máxima: a submatriz de soma máxima. Propomos aqui, além de uma solução geral, soluções paralelas BSP/CGM para problemas relacionados. Não é de nosso conhecimento a existência de soluções BSP/CGM para esses problemas. Como será apresentado obtivemos bons resultados com as implementações dos algoritmos, tanto em um cluster com memória distribuída, quanto em uma GPU.

4.1 Definição

O problema da submatriz de soma máxima consiste em encontrar a submatriz com a maior soma entre todas as submatrizes retangulares de uma matriz quadrática A, de dimensões $n \times n$ de inteiros, que possui ao menos um número positivo [4]. Esse problema pode ser definido mais formalmente considerando A como uma matriz quadrática de inteiros, com pelo menos um valor positivo e (g, h) como um par de inteiros, onde $1 \leq g, h \leq n$. Denotamos $R^{(g,h)}$ como o conjunto que representa todas as submatrizes $A[i_1 \ldots i_2, g \ldots h]$, onde $1 \leq i_1 \leq i_2 \leq n$. Similarmente, também denotamos a sequência $C_j^{(g,h)}$ de tamanho n como a coluna resultante da soma de todos os elementos de cada linha de $A[1 \ldots n, g \ldots h]$, que estão entre as colunas g e h, incluindo g e h, isto é: $C_j^{(g,h)} = \sum_{k=g}^{h} a_{ik}$.

Definimos então o problema da submatriz de soma máxima como a tarefa de obter a submatriz com a maior soma (acumulada) entre então todas as submatrizes $R^{1,n}$ de A [4]. Como exemplo, no arranjo representado pela Figura 4.1 existem ao menos três submatrizes de soma máxima, todas com soma igual a 16.

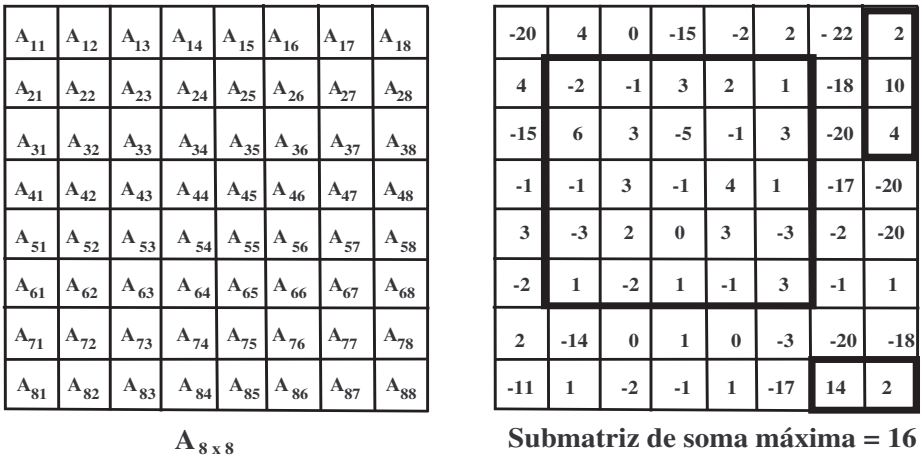


Figura 4.1: Submatriz $A_{8\times 8}$ com $A[2,6,2,6]$, $A[1,3,8,8]$ e, $A[8,8,7,8]$: as três submatrizes de soma máxima.

O problema da submatriz de soma máxima aparece em diversas áreas da ciência, que utilizam a solução desse problema como sub-rotina. Uma destas áreas é a visão computacional, onde aplicações, por exemplo, podem auxiliar na detecção de regiões de interesse em diagnósticos por imagem.

4.2 Problemas Relacionados

Da mesma forma que no caso unidimensional, dado um arranjo quadrático A de inteiros, como ilustrado na Figura 4.1, é possível a existência de mais de uma submatriz de soma máxima. No caso da matriz A existem três submatrizes de soma máxima. Todas possuem soma igual a 16, mas com diferentes tamanhos (número de elementos): 3, 25 e 2, respectivamente.

Neste trabalho, além de revisitar o problema da submatriz de soma máxima também propomos soluções para cinco problemas relacionados:

1. A maior submatriz de soma máxima;
2. A menor submatriz de soma máxima;
3. O número de submatrizes de soma máxima;
4. A seleção da k-submatriz de soma máxima;
5. A submatriz de soma máxima com maior densidade relativa.

Como visto, a Figura 4.1 contextualiza os três primeiros problemas. O quarto é um problema clássico em computação, que como veremos pode ser resolvido a partir de nossa nossa solução. O quinto problema é particularmente útil quando a concentração de elementos é mais interessante do que a soma máxima global. Não é de nosso conhecimento a existência de algoritmos BSP/CGM para esses problemas.

4.3 Histórico de Soluções

O melhor algoritmo sequencial conhecido para o problema da submatriz de soma máxima possui complexidade de tempo $O(n^3)$ [4]. Trabalhos anteriores também reportaram boas soluções paralelas para o problema da submatriz de soma máxima. Qiu and Akl apresentaram um algoritmo que trabalha sobre redes de interconexão (hipercubo e estrela) de tamanho p, utilizando $O(\log n)$ de tempo paralelo, com $O(n^3/\log n)$ processadores [36]. Zhaofang Wen apresentou um algoritmo paralelo PRAM utilizando $O(\log n)$ de tempo paralelo, com $n^3/\log n$ processadores [48]. Perumalla and Deo também apresentaram um algoritmo paralelo PRAM com a mesma complexidade de tempo e o mesmo número de processadores [35]. Um algoritmo paralelo BSP/CGM para o problema foi apresentado por Alves *et al.* utilizando $O(n^3/p)$ de tempo paralelo com p processadores e um número constante de rodadas de comunicação [2]. Bae apresentou soluções paralelas para o problema utilizando a estrutura de um *mesh* [3]. Muitos destes algoritmos não exploram as características de suas soluções e retornam apenas o valor unitário correspondente à soma da submatriz de soma máxima.

4.4 Os Algoritmos Paralelos

Motivados por nosso propósito de projeto de algoritmos em arquitetura *multi/manycore* e da utilização de um modelo adequado e realístico de computação paralela, para este capítulo baseamos nosso trabalho nos algoritmos paralelos para submatriz de soma máxima apresentados em dois trabalhos anteriores [35, 2]. São dois os algoritmos, conforme ilustrado pela Figura 4.2. A seguir descrevemos mais detalhadamente esses resultados.

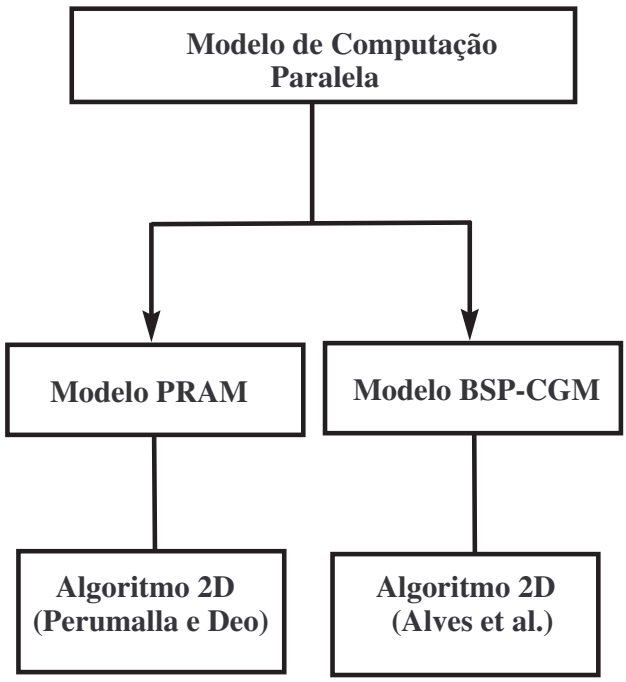


Figura 4.2: Os algoritmos paralelos para subsequência de soma máxima.

4.4.1 Algoritmo Paralelo PRAM

Em 1995 Perumalla e Deo [35] apresentaram uma solução baseada no modelo PRAM para o problema da subsequência de soma máxima em matrizes bidimensionais (2D). A seguir apresentamos as principais definições e lemas desse trabalho: dada uma matriz de ordem $n \times n$, de números inteiros, um par (g, h)-par, $1 \leq g \leq h \leq n$, é definido como sendo a coluna $C^{(g,h)}$, de tamanho n, resultante da compressão (soma) dos elementos de todas as colunas de A que estejam entre as colunas g e h, inclusive g e h, isto é, $C_i^{(g,h)} = \sum_{j=g}^{h} a_{ij}$. Por sua vez, um conjunto R^{gh} é definido como o conjunto de todas as submatrizes de cada $C^{(g,h)}$ que se iniciam na g-ésima coluna e que terminam na h-ésima coluna de A, isto é, os valores $SA_{i_1 g i_2 h}, 1 \leq i_1 \leq i_2 \leq n$. A subsequência de soma máxima de cada $C^{(g,h)}$ é definida como aquela que retorna a maior soma $MSQ^{(g,h)}$ dentre todas as possibilidade de seu conjunto R^{gh}. Esta maior soma é encontrada aplicando-se o algoritmo sequencial (Algoritmo 3) em cada $C^{(g,h)}$. Por fim, basta comparar todos os (g,h)-pares entre si, encontrar aquele de maior soma e retornar a submatriz de soma máxima correspondente, denominada *SSM* [35].

No exemplo dado pela Figura 4.3, dada a matriz A e considerando um de seus possíveis (g, h)-pares, neste caso $C^{(3,6)}$. Existem várias possibilidades em $R^{(3,6)}$ de $C^{(3,6)}$, por exemplo a submatriz $SA_{2,3,5,6}$ cujo valor de soma resulta em 7, uma outra submatriz $SA_{3,3,7,6}$ cujo valor de soma resulta em 14, entretanto nos interessa aquela cujo valor de soma é o maior possível dentre todas as possibilidades, buscamos o valor $MSQ^{3,6}$, no caso a submatriz de soma máxima de $C^{(3,6)}$ é correspondente a $SSM_{1,3,6,6}$. A subsequência de soma máxima do par $C^{(3,6)}$ de A tem valor igual a 35.

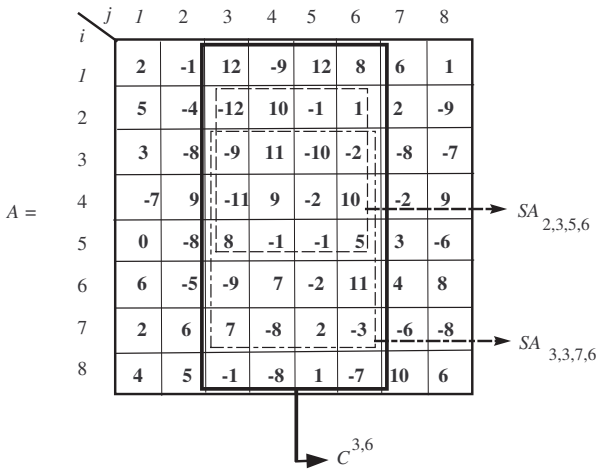


Figura 4.3: Possibilidades de $SA_{i_1 3 i_2 6}, 1 \leq i_1 \leq i_2 \leq n$, de $C^{(3,6)}$.

Observação: Para computar as sequências $C^{(g,h)}$ eficientemente, as linhas podem ser pré-processadas, para tal deve-se substituir cada linha pela soma prefixo correspondente entre as colunas g e h. Assim, as somas que compõem um par $C^{(g,h)}$ podem ser consultadas em tempo $\mathrm{O}(1)$ [35]. O pseudocódigo do algoritmo paralelo PRAM para submatriz de soma máxima é ilustrado pelo Algoritmo 10. O terceiro teste

Algoritmo 10: Algoritmo PRAM para a submatriz de soma máxima.

Entrada: Matriz A de números inteiros de ordem $n \times n$.
Saída: Submatriz de soma máxima de A.

1: Substitua cada linha de A pela linha de soma de prefixos correspondente.
2: Adicione uma coluna de zeros como primeira coluna de A.
3: Para todo $i \leq g \leq h \leq n$ **em paralelo faça**
4: (a) Compute a sequência $C^{(g,h)}$:
5: **for** $i = 1$ to n **faça**
6: $C^{(g,h)}[i] = A[i][h] - A[i][g-1]$.
7: **end for**
8: (b) Compute a subsequência de soma máxima de $C^{(g,h)}$ e insira em $M_{(g,h)}$.
9: Encontre o valor máximo em $M_{(g,h)}$, $1 \leq g \leq h$ e insira em *MSA*.
10: Escreva a saída *MSA*.

Exemplo do Cálculo de $C^{(g,h)}$

A seguir apresentamos como é efetuado o cálculo dos pares $C^{(g,h)}$, passo fundamental do Algoritmo 10. Considerando a matriz de entrada A representada pela Figura 4.4 serão necessários então $n(n+1)/2$ cálculos de pares $C^{(g,h)}$, ou seja, 36 pares.

$A_{8x8} =$

$C^{1,1}$	$C^{1,2}$	$C^{1,3}$	$C^{1,4}$	$C^{1,5}$	$C^{1,6}$	$C^{1,7}$	$C^{1,8}$
	$C^{2,2}$	$C^{2,3}$	$C^{2,4}$	$C^{2,5}$	$C^{2,6}$	$C^{2,7}$	$C^{2,8}$
		$C^{3,3}$	$C^{3,4}$	$C^{3,5}$	$C^{3,6}$	$C^{3,7}$	$C^{3,8}$
			$C^{4,4}$	$C^{4,5}$	$C^{4,6}$	$C^{4,7}$	$C^{4,8}$
				$C^{5,5}$	$C^{5,6}$	$C^{5,7}$	$C^{5,8}$
					$C^{6,6}$	$C^{6,7}$	$C^{6,8}$
						$C^{7,7}$	$C^{7,8}$
							$C^{8,8}$

Figura 4.4: Cálculos de pares $C^{(g,h)}$ necessários para $A_{8\times 8}$.

Por exemplo, o cálculo de $C^{(3,6)}$ é delimitado pela região retangular expressa na Figura 4.5 (a). Para calcular $C^{(3,6)}$ deve-se comprimir as linhas da região delimitada, efetuando o somatório de seus elementos. Após a compressão é gerada a sequência linear que representa $C^{(3,6)}$, ilustrada pela Figura 4.5 (b).O terceiro teste

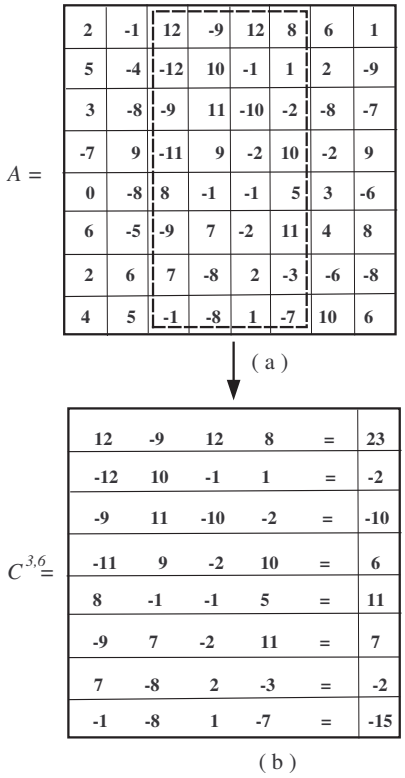


Figura 4.5: O cálculo de $C^{(3,6)}$.

Após obter a sequência linear, deve-se aplicar o Algoritmo 3, que irá delimitar e encontrar a subsequência de soma máxima de $C^{(3,6)}$, conforme ilustrado pela Figura 4.6.

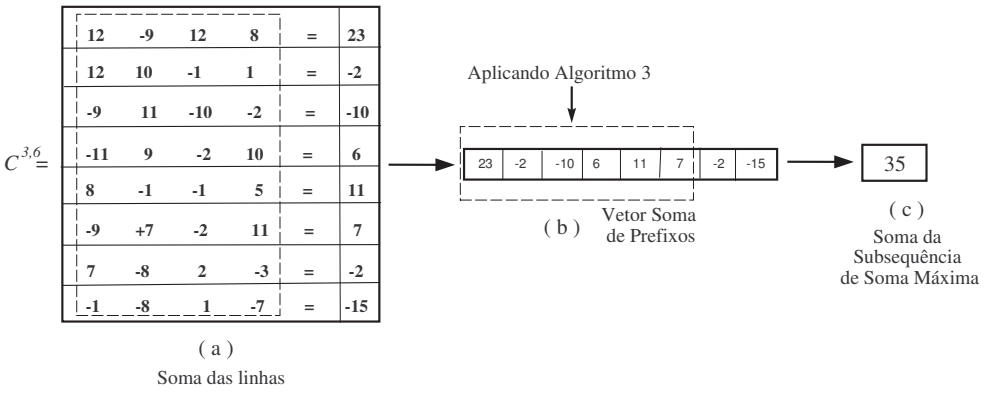


Figura 4.6: A subsequência de soma máxima de $C^{(3,6)}$.

Inicialmente o algoritmo considera os 36 valores de $C^{(g,h)}$ possíveis de A. Depois, separadamente encontra a subsequência de soma máxima de cada um deles. Em seguida, realiza uma comparação entre todas as 36 subsequências de soma máxima encontradas e retorna aquela cuja soma possui maior valor. Por fim, basta selecionar a região retangular em A, que representa a subsequência com maior valor, ela será a submatriz de soma máxima de A.

Reduzindo os Cálculos de Compressão

Como forma de reduzir o número de cálculos de compressões, o algoritmo utiliza uma estratégia criativa, ele converte a matriz de entrada A na sua matriz de soma de prefixos correspondente. Através da matriz de soma de prefixos, os cálculos da etapa de compressão de qualquer $C^{(g,h)}$ podem ser realizados em tempo constante para tal, para encontrar a sequência $C^{(g,h)}$ é necessário o seguinte passo: Subtraia da última coluna delimitada por $C^{(g,h)}$ a coluna imediatamente anterior a primeira coluna da região delimitada por $C^{(g,h)}$. A Figura 4.7 ilustra os cálculos para $C^{(3,6)}$ de A, melhor exemplificados pela Tabela 4.1.

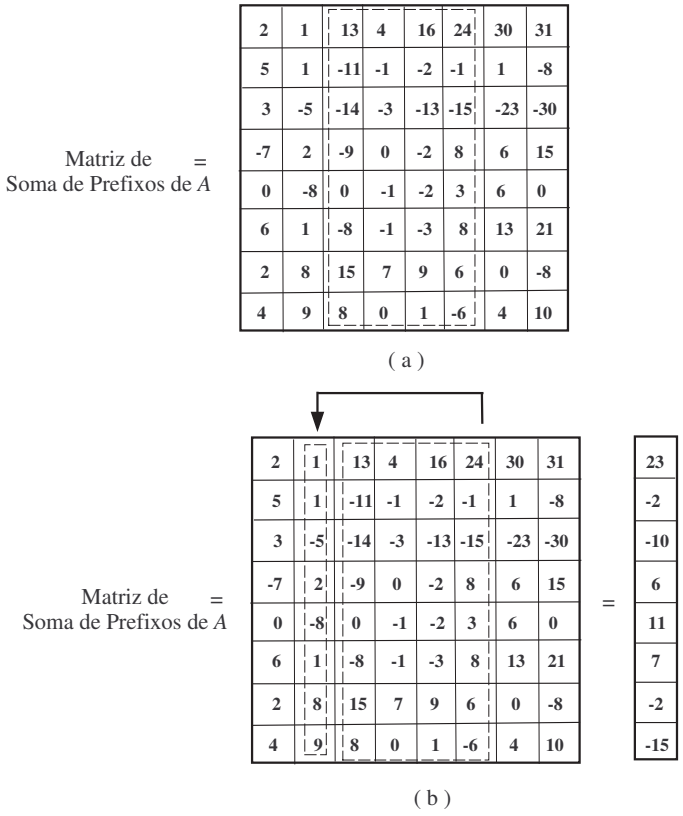


Figura 4.7: O cálculo de $C^{(3,6)}$ utilizando a matriz de soma de prefixos de A.

Linha i	$C^{(g,h)}[i] =$	$A[i][h] - A[i][g-1]$	Sequência de $C^{3,6}$
Linha 1	$C^{(3,6)}[1] =$	$A[1][6] - A[1][3-1] = 24 - 1$	$= 23$
Linha 2	$C^{(3,6)}[2] =$	$A[2][6] - A[2][3-1] = -1(-1)$	$= -2$
Linha 3	$C^{(3,6)}[3] =$	$A[3][6] - A[3][3-1] = -15 + 5$	$= -10$
Linha 4	$C^{(3,6)}[4] =$	$A[4][6] - A[4][3-1] = 8 - 2$	$= 6$
Linha 5	$C^{(3,6)}[5] =$	$A[5][6] - A[5][3-1] = 3 + 8$	$=11$
Linha 6	$C^{(3,6)}[6] =$	$A[6][6] - A[6][3-1] = 8 - 1$	$= 7$
Linha 7	$C^{(3,6)}[7] =$	$A[7][6] - A[7][3-1] = 6 - 8$	$= -2$
Linha 8	$C^{(3,6)}[8] =$	$A[8][6] - A[8][3-1] = -6 - 9$	$= -15$

Tabela 4.1: Cálculos de $C^{(3,6)}$.

Custo do Algoritmo

O custo do Algoritmo 10 é detalhado de acordo com seus passos [35]: o passo 1 pode ser computado em tempo $O(\log n)$. O passo 2 pode ser computado em tempo $O(1)$. O passo 3 (a) pode ser computado em tempo $O(\log n)$ utilizando $O(n/\log n)$ processadores no modelo CREW-PRAM e n processadores no modelo EREW-PRAM, respectivamente, para cada (g,h)-par. O passo 3 (b) pode ser computado em tempo $O(\log n)$ utilizando $O(n/\log n)$ processadores para cada $C^{(g,h)}$, utilizando o algoritmo de subsequência de soma máxima versão 1D. O passo 4 pode ser computado em tempo máximo $O(\log n)$. Por fim, o tempo total do algoritmo é $O(\log n)$ com $n^3/(\log n)$ processadores no modelo CREW-PRAM e n^3 processadores no modelo EREW-PRAM [35] .

4.4.2 Algoritmo Paralelo BSP/CGM

O próximo algoritmo paralelo parte dos mesmos princípios de cálculos de $C^{(g,h)}$'s descritos no algoritmo anterior. Trata-se de um algoritmo BSP/CGM proposto por Alves *et al.* [2]. A diferença agora é que a matriz bidimensional de entrada é particionada explicitamente entre os processadores e além disto operações de compressão de dados são continuamente executadas. A seguir descrevemos com um exemplo a execução desse algoritmo: dada uma matriz $A_{8\times 8}$, uma possível divisão de dados pode ser ilustrada pela Figura 4.8. Cada processador recebe um número n/p de linhas e n de colunas. Em seguida, cada processador realiza os cálculos de $C^{(g,h)}$'s que lhe cabem. Entretanto, agora para calcular o valor integral de um $C^{(g,h)}$ qualquer é necessário um passo de comunicação entre todos processadores. Além disso, para reduzir o tamanho de um vetor $C^{(g,h)}$ o algoritmo utiliza a estratégia de compressão em cinco valores já descrita no Algoritmo 5. Após o passo de comunicação, o processador 1 é capaz de encontrar a submatriz de soma máxima.

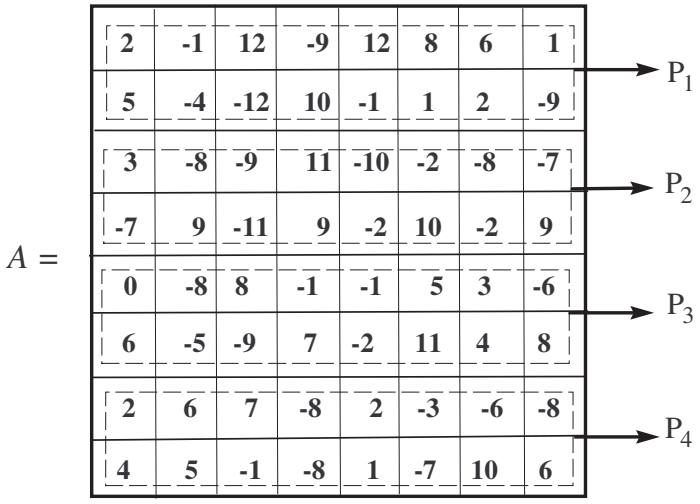


Figura 4.8: Partição da matriz de entrada entre os processadores.

O pseudocódigo deste algoritmo é ilustrado pelo Algoritmo 11.

Algoritmo 11: Algoritmo BSP/CGM para a submatriz de soma máxima.

Entrada: Cada processador i, onde $1 \leq i \leq p$, recebe o subvetor $A[(i-1)n/p..in/p][1 \ldots n]$.
Saída: A submatriz de soma máxima de A.

1: Cada processador obtém a soma de prefixos das linhas que possui.
2: Cada processador calcula todos os valores de $C^{(g,h)}$'s possíveis com os dados das linhas que possui.
3: Para cada $C^{(g,h)}$ é realizada a compressão de dados para um vetor de 5 valores como descrito no Algoritmo 5. Cada processador realiza $n^2/2$ destes cálculos de compressão.
4: Cada processador i envia para cada processador j a j-ésima parte do vetor calculado no passo anterior e recebe a i-ésima parte do vetor de cada processador j.
5: Cada processador i calcula a submatriz de soma máxima com os dados armazenados e recebidos.
6: Processador 1 obtém a submatriz de soma máxima dentre todos os processadores.

Custo do Algoritmo

O custo do Algoritmo 11 é determinado através da análise dos passos 3 e 4. No passo 4, todos os processadores enviam e recebem um vetor de tamanho n^2/p (uma h-relação de tamanho n/p). O Passo 3 é executado para cada uma das alternativas $\mathrm{O}\big(n^2\big) \times C^{(g,h)}$. Uma vez que existem n/p elementos em cada coluna, temos a complexidade de tempo final $\mathrm{O}\big(n^3/p\big)$ [2].

4.5 Algoritmos Propostos

Os Algoritmos 10 (PRAM) e 11 (BSP/CGM) utilizam as mesmas premissas e executam basicamente os mesmos cálculos computacionais. Particularmente, o algoritmo BSP/CGM possui um diferencial importante. Ocorre que as constantes divisões de dados e sucessivas compressões realizadas desconfiguram as submatrizes originais e dificultam a busca por dados que envolvam tamanho e número de submatrizes de soma máxima.

Como vimos, nosso objetivo aqui consiste em ampliar a solução geral do problema da submatriz de soma máxima, para também cobrir soluções de alguns problemas relacionados. Desta forma, baseamos nossas soluções nas ideias do algoritmo de Perumalla e Deo, que mantém a integridade dos dados. A partir dele desenvolvemos um novo algoritmo BSP/CGM para solucionar o problema geral da submatriz de soma máxima, que foi estendido para também resolver cinco problemas relacionados.

4.5.1 O Algoritmo BSP/CGM para o Problema Geral

Inicialmente projetamos um novo algoritmo paralelo BSP/CGM que soluciona o problema geral da submatriz de soma máxima. A entrada do algoritmo, aqui representado pelo Algoritmo 12, é uma matriz quadrática A de inteiros, tal como aquela ilustrada pela Figura 4.1. O primeiro passo do algoritmo consiste na computação de uma matriz denominada *PS*. Cada linha da matriz *PS* armazena a soma de prefixos da linha correspondente na matriz de entrada A, ou seja, $PS[i,j] = \sum_{k=1}^{j} A[i,k]$. A Figura 4.9 ilustra a computação da matriz *PS*.

Algoritmo 12: Algoritmo BSP/CGM proposto - problema geral.

Entrada: (1) Um conjunto de P processadores; (2) A identificação de cada processador p_i, onde $1 \leq i \leq P$; (3) Matriz $A[1 \ldots n][1 \ldots n]$ de inteiros.
Saída: (1) Vetor M com os valores de soma máxima de cada submatriz de A. (2) valor max, que é o maior valor em M.

1: Para cada processador p_i **em paralelo faça** $PS[1 \ldots n][1 \ldots n] \leftarrow$ as somas de prefixos, linha a linha, de cada linha da submatriz A;
2: Para cada processador p_i **em paralelo faça** $Temp_j[1 \ldots n] \leftarrow C^{(g,h)}(PS)$;
3: Para cada processador p_i **em paralelo faça** $max\_local_j \leftarrow$ Algoritmo 3($Temp_j[1 \ldots n]$)
4: Para cada processador $p_i \neq 1$ **em paralelo faça** send($max\_local_j, p_i = 1$);
5: **if** $p_i = 1$ **then**
6: **for** $k = 2$ **to** P **do**
7: receive(max_local_j, p_k)
8: **end for**
9: **for** $j = 1$ **to** $n(n+1)/2$ **do**
10: $M[j] \leftarrow max\_local_j$
11: **end for**
12: **end if**
13: Para cada processador p_i **em paralelo faça** $max \leftarrow M[1, \ldots, n(n+1)/2]$.
14: **return** (M, max)

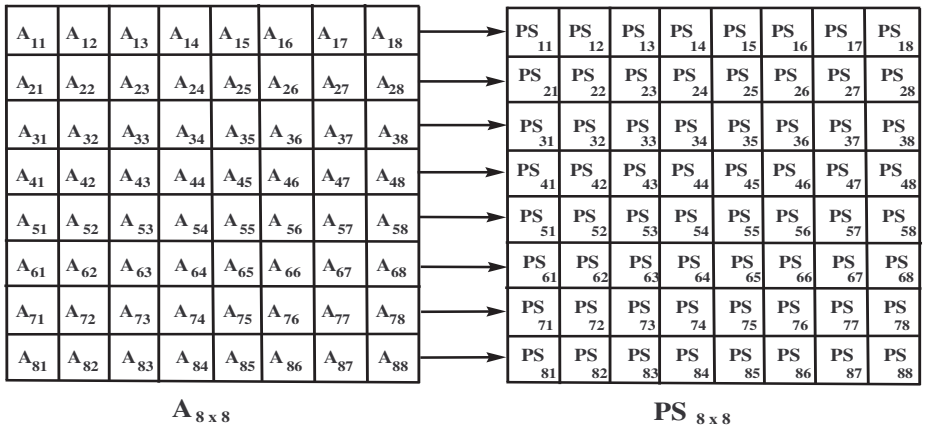


Figura 4.9: Computação da matriz PS.

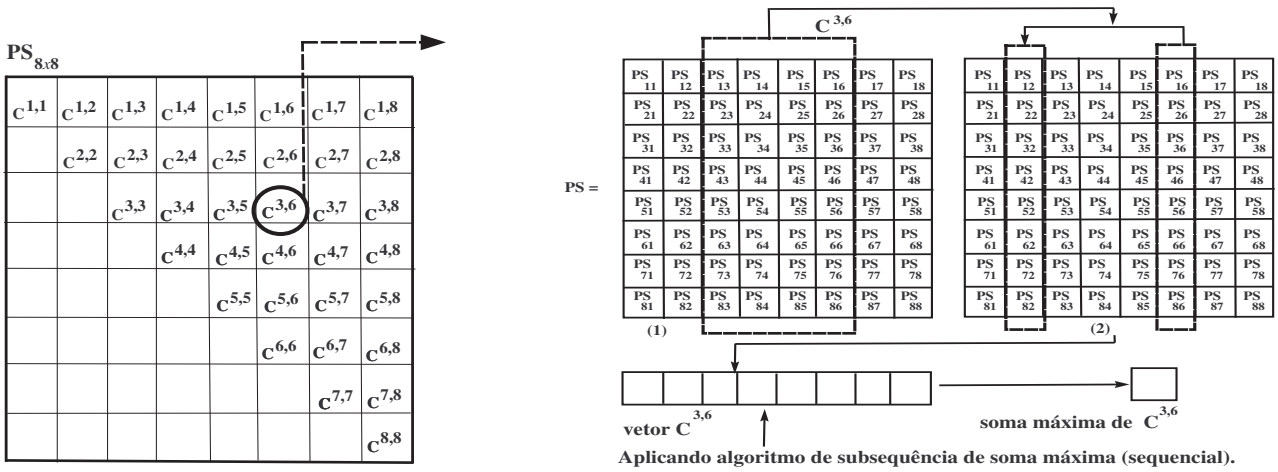
Na próxima etapa aplicamos o Algoritmo 13 sobre a submatriz PS para computar um conjunto de submatrizes $C^{(g,h)}$'s, onde $1 \leq g \leq h \leq n$ [35]. Uma vez que cada linha de PS corresponde a soma de prefixos da respectiva linha na submatriz A, as submatrizes $C^{(g,h)}$'s podem ser facilmente computadas (em tempo constante) através da fórmula $C_j^{gh}[i] = PS[i][h] - A[i][g-1]$. A computação e o número de submatrizes $C^{(g,h)}$'s derivados da submatrizes A podem ser visualizados na Figura 4.10 (a).

Algoritmo 13: Algoritmo BSP/CGM para o cálculo de $C^{(g,h)}$ de PS.

Entrada: (1) Um conjunto de P processadores; (2) A identificação de cada processador p_i, onde $1 \leq i \leq P$; (3) Matriz $PS[1 \ldots n][1 \ldots n]$ de inteiros.
Saída: Conjunto de vetores $Temp_j[1 \ldots n]$.

1: **if** $p_i = 1$ **then**
2: $\quad Temp_j[1 \ldots n] \leftarrow PS[i][h]$.
3: **end if**
4: Each processor $p_i \neq 1$ **em paralelo faça**:
$\quad\quad Temp_j[1 \ldots n] \leftarrow PS[i][h] - PS[i][g-1]$.
5: **return** $Temp_j[1 \ldots n]$

No próximo passo, para cada submatriz $C_j^{(g,h)}$ (comprimida em uma sequência) o algoritmo localiza a subsequência de soma máxima e o respectivo valor de soma máxima utilizando o melhor algoritmo para subsequência de soma máxima (Algoritmo 3), cuja complexidade de tempo é $O(n)$. Estes passos são melhor descritos pela Figura 4.10 (b).



(a) Número de $C^{(g,h)}$'s$= n(n+1)/2$ (b) Computação de $C^{(g,h)}$

Figura 4.10: Os passos para computação das submatrizes $C^{(g,h)}$.

No último passo, através de uma operação de redução para máximo (em paralelo), o algoritmo localiza a submatriz $C_j^{(g,h)}$ com a soma máxima.

4.5.2 O Algoritmo BSP/CGM para os Problemas Relacionados

Nosso próximo algoritmo basicamente consiste de alterações do Algoritmo 12. As modificações foram realizadas, para que o novo algoritmo pudesse obter mais informação acerca das submatrizes de soma máxima e assim provesse soluções para o problema geral e para cinco problemas relacionados. Nossas estratégias são descritas pelos Algoritmos 14 e 15.

O Algoritmo 14 e sua versão anterior (Algoritmo 12) possuem estruturas similares. Entretanto, no Algoritmo 14 um novo vetor foi criado. Este vetor denominado $E[1 \ldots n(n+1)/2]]$, armazena o número de elementos de cada posição do vetor $M[1 \ldots n(n+1)/2]]$.

Algoritmo 14: Algoritmo BSP/CGM proposto - problemas relacionados - 01.

Entrada: (1) Um conjunto de P processadores; (2) A identificação de cada processador p_i, onde $1 \leq i \leq P$; (3) Matriz $A[1 \ldots n][1 \ldots n]$ de inteiros.
Saída: (1) Vetor M com os valores de somas máximas de cada submatriz de A;
(2) Vetor E com os números de elementos de cada submatriz de soma máxima de A.

1: Para cada processador p_i **em paralelo faça**
$PS[1 \ldots n][1 \ldots n] \leftarrow$ as somas de prefixos de cada linha da matriz $A[1 \ldots n][1 \ldots n]$;
2: $Cgh\_quantity \leftarrow n(n+1)/2$;
3: $k \leftarrow Cgh\_quantity/P$;
4: **for** $j = 1$ **to** k **em paralelo faça**
5: $Temp_j[1 \ldots n] \leftarrow C^{(g,h)}(PS)$;
6: $LocalM[1 \ldots j], LocalE[1 \ldots j] \leftarrow$Algoritmo 3($Temp_j[1 \ldots n]$);
7: **end for**
8: Para cada processador p_i **em paralelo faça** send($LocalM[1 \ldots k]$,$p_i = 1$);
9: Para cada processador p_i **em paralelo faça** send($LocalE[1 \ldots k]$,$p_i = 1$);
10: $j \leftarrow 1$;
11: **if** $p_i = 1$ **then**
12: **for** $i = 2$ **to** P **do**
13: receive($LocalMi[1 \ldots k], p_i$);
14: receive($LocalEi[1 \ldots k], p_i$);
15: **end for**
16: Realize a computação dos vetores $M[1 \ldots n(n+1)/2]$ e $E[1 \ldots n(n+1)/2]$, onde:
$M[1 \ldots n(n+1)/2]] = [LocalM_{p_{1_1}}, \ldots, LocalM_{p_{1_k}}, \ldots, LocalM_{p_{p_1}}, \ldots, LocalM_{p_{p_k}}]$.
$E[1 \ldots n(n+1)/2]] = [LocalE_{p_{1_1}}, \ldots, LocalE_{p_{1_k}}, \ldots, LocalEp_{p_1}, \ldots, LocalE_{p_{p_k}}]$.
17: **end if**

Os dois vetores gerados como saída do Algoritmo 14 serão utilizados pelo Algoritmo 15 para solucionar o problema geral da submatriz de soma máxima e os cinco problemas relacionados. Para obter estas soluções o Algoritmo 15 aplica um algoritmo paralelo de ordenação por chave-valor sobre os vetores $M[1 \ldots n(n+1)/2]]$ e $E[1 \ldots n(n+1)/2]]$. Particularmente, para solucionar o problema da localização da submatriz de soma máxima com maior densidade relativa, efetuamos divisões entre os elementos válidos de $M[1 \ldots n(n+1)/2]]$ e $E[1 \ldots n(n+1)/2]]$.

Algoritmo 15: Algoritmo BSP/CGM proposto - problemas relacionados - 02.

Entrada: (1) Um conjunto de P processadores; (2) A identificação de cada processador p_i, onde $1 \leq i \leq P$; (3) Vetores $M[1 \ldots n(n+1)/2]$ e $E[1 \ldots n(n+1)/2]$ do Algoritmo 14.
Saída: (1) Soluções para o problema da submatriz de soma máxima (problema geral) e os cinco problemas relacionados.

1: $(M[1 \ldots n(n+1)/2]$,$E[1 \ldots n(n+1)/2]) \leftarrow$ Algoritmo Paralelo de Ordenação por chave valor(M,E);
2: **submatriz_soma_máxima** $\leftarrow M[n(n+1)/2]$; {problema base}
3: $(NewM[1 \ldots (n(n+1)/2) - i]$, $NewE[1 \ldots (n(n+1)/2) - i]) \leftarrow$ Descarte todos os elementos de índice i, tal que $M[i] <$ **valor de submatriz de soma máxima**
4: **menor_submatriz_soma_máxima** $\leftarrow NewE[0]$; {problema relacionado 1}
5: **maior_submatriz_soma_máxima** $\leftarrow NewE[$tamanho($NewE$)$]$; {problema relacionado 2}
6: k**-máxima_submatriz_soma** $\leftarrow NewE[k]$; {problema relacionado 3}
7: **número_de_submatrizes_de_soma_máxima** $\leftarrow$ tamanho($NewE$); {problema relacionado 4}
8: **máxima_densidade_relativa** $\leftarrow$ Algoritmo Paralelo de Redução para Máximo(M,E), utilizando $d_i = M[i]/E[i]$; {problema relacionado 5}

A Figura 4.11 ilustra um exemplo do processo de solução de todos os problemas. Após a ordenação chave-valor os passos que solucionam os problemas relacionados, com exceção do oitavo (Passo 8), podem ser executados em tempo constante.

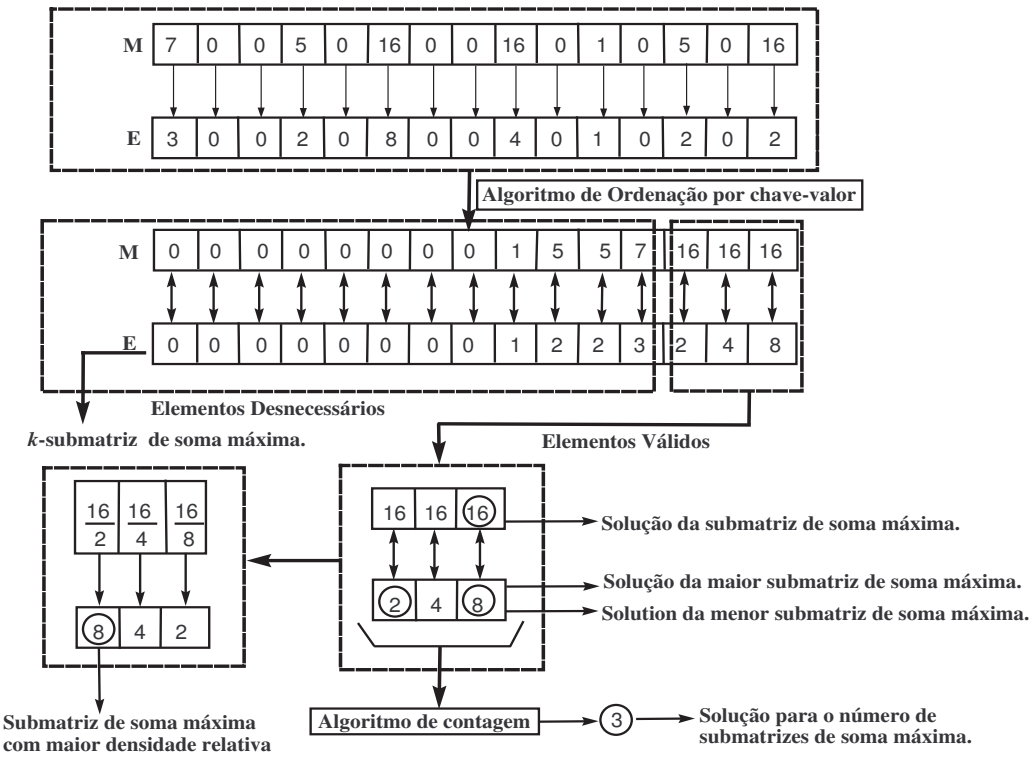


Figura 4.11: Soluções para o problema geral e problemas relacionados.

Observando a Figura 4.11 a partir dos vetores de elementos válidos e em tempo constante, obtemos soluções para os problemas. A **submatriz de soma máxima** possui soma igual a 16. A **maior submatriz de soma máxima** possui valor igual a 8. A **menor submatriz de**

soma máxima possui valor igual a 2. O número **de submatrizes de soma máxima** é 3. A submatriz de soma máxima com maior **maior densidade relativa** possui valor densidade igual a 8.

4.5.3 Complexidade

A complexidade de nossos algoritmos é obtida através da análises de cada passo:

- **Algoritmo 12**: no Passo 1 a soma de prefixos pode ser computada através de um algoritmo paralelo BSP/GM utilizando p processadores e em tempo $O(n/p)$, com um número constante de rodadas de comunicação. Visto que existem n linhas, a complexidade de tempo final deste passo é $O(n^2/p)$. No passo 2, um conjunto de n^2 vetores é obtido pelos processadores. Subsequentemente, o algoritmo sequencial ($O(n)$) executa em cada vetor, assim a complexidade final destes passos até aqui é $O(n^3/p)$. O Passo 4 é apenas de comunicação. Os passos finais podem ser computados com o auxílio de algoritmo paralelo de redução para máximo, utilizando p processadores e em tempo $O(n/p)$, com um número constante de rodadas de comunicação. Assim sendo, concluímos que a complexidade de tempo final para o Algoritmo 12 utilizando p processadores é $O(n^3/p)$ com um número constante de rodadas de computação. A correção decorre diretamente dos algoritmo proposto por Perumulla e Deo [35];
- O **Algoritmo 14** possui a mesma complexidade final do **Algoritmo 12**, ou seja, utilizando p processadores, a complexidade de tempo final é $O(n^3/p)$ com um número constante de rodadas de computação. Particularmente, o **Algoritmo 15** também possui a mesma complexidade final.

4.6 Implementações e Resultados

Nesta seção discutimos as implementações de nossos algoritmos para submatriz de soma máxima e problemas relacionados. Em relação ao problema da submatriz de soma máxima (Algoritmo 12) duas implementações foram construídas. A primeira em MPI e a segunda em OpenMP. Em nossa implementação MPI exploramos o ambiente de memória distribuída e em nossa implementação em OpenMP exploramos o ambiente de memória compartilhada. Também construímos implementações que abrangem o problema geral e os problemas relacionados (Algoritmos 14 e 15), mas apenas em CUDA. É importante destacar que a partir do Algoritmo 14 também é possível solucionar o problema geral da submatriz de soma máxima.

Em nossas implementações buscamos, sempre que possível, otimizar a maior parte do processamento local, executando operações de forma independente e assim evitando passos de comunicação extras. Para subsequente divisão de dados utilizamos e adaptamos um processo de mapeamento, como é ilustrado pela Figura 4.12. No mapeamento, as posições de uma matriz triangular superior são mapeadas para um vetor com o mesmo número de entradas. Em nossa adaptação, a matriz diagonal também precisa ser comtemplada.

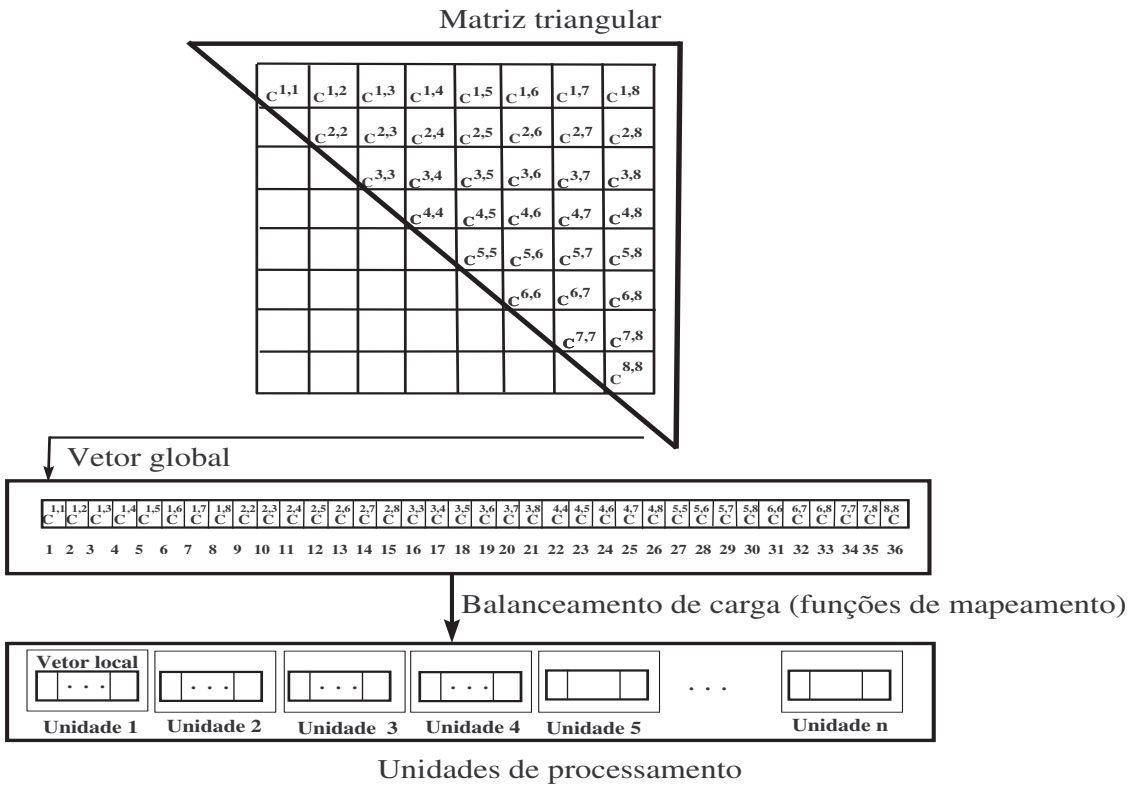


Figura 4.12: O processo de mapeamento: cada processador é responsável pelo mesmo número de submatrizes $C^{(g,h)}$'s.

4.6.1 Resultados

A seguir apresentamos os resultados obtidos por nossos algoritmos paralelos BSP/CGM para submatriz de soma máxima e problemas relacionados. Três casos de comparação entre os algoritmos são apresentados. A entrada dos algoritmos consiste de matrizes de inteiros geradas aleatoriamente. Em todos os casos a unidade de tempo utilizada foi o milissegundo. Para os cálculos de desempenho (*speedup*) trabalhamos com as execuções de melhor tempo de execução. Para comparações equivalentes, o algoritmo sequencial foi executado nos ambientes de memória compartilhada e distribuída, por isto existem tempos diferentes para o mesmo tamanho de entrada.

4.6.2 Primeiro Caso

O primeiro caso ilustra a comparação dos tempos de execução de algoritmos para o problema geral da submatriz de soma máxima. Conduzimos três testes que utilizaram três algoritmos: dois em MPI e um em OpenMP.

O primeiro teste utiliza um algoritmo MPI descrito e apresentado em um trabalho anterior de Alves *et al.* [2]. Este algoritmo envolve compressão de dados e uma considerável comunicação entre os processadores. A Figura 4.13 e a Tabela 4.2 ilustram os resultados de comparação entre este algoritmo e a solução sequencial. Os resultados demonstram a eficiência desta solução MPI, com um aumento contínuo dos valores de desempenho. Contudo inicialmente ocorre um pico

no desempenho com 64 processadores e acreditamos tratar-se de um *overhead*.

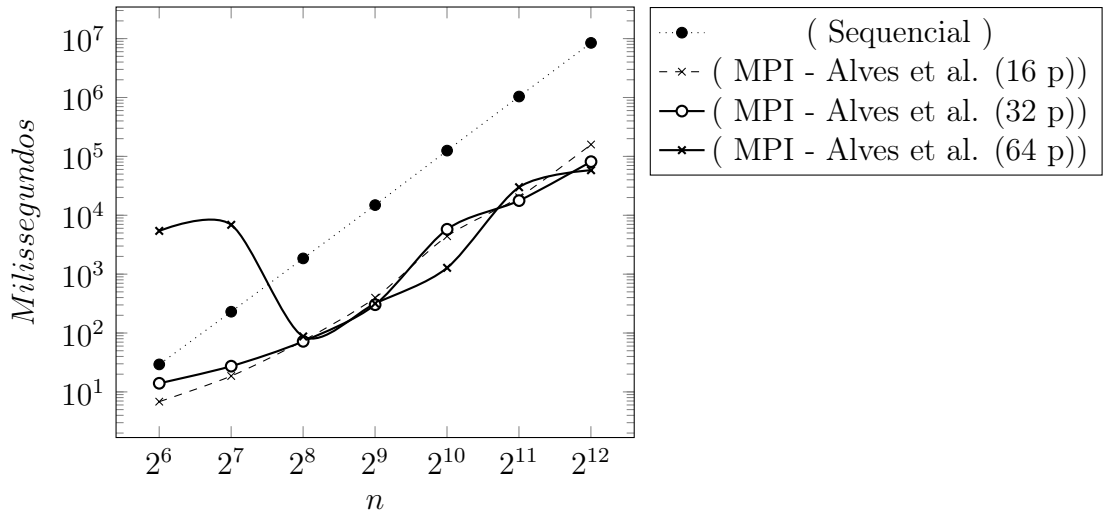


Figura 4.13: Sequencial × Versão MPI (Alves *et. al*).

$n \times n$	Tempo sequencial	Tempo-MPI(16p.)	Tempo-MPI(32 p.)	Tempo-MPI(64p.)	*Speedup*(Seq./MPI-32p.)
64×64	29.2992	6.8273	14.0064	5422.5232	2.0918
128×128	229.9130	18.6285	27.3484	6877.8257	8.4068
256×256	1.849.6825	74.1179	72.2008	87.3313	25.6186
512×512	14.879.4916	397.6235	302.5005	316.9507	49.1883
1024×1024	125.239.8713	4.403.7971	5.783.0959	1.279.9464	21.6562
2048×2048	1.038.533.5501	20.241.3991	17.722.9556	29.944.1617	58.5982
4096×4096	8.460.234.4575	158.989.7957	81.369.5981	58.257.9493	103.9729

Tabela 4.2: Tempos de execução (**Milissegundos**): Sequencial × Versão MPI (16, 32 e 64 p.).

O segundo teste utiliza nossa solução MPI para o Algoritmo 12. Diferentemente do Algoritmo de Alves *et al.* [2], em nossa solução cada processador é responsável pelo cálculo de um conjunto de submatrizes $C^{(g,h)}$'s. Além disto, não aplicamos compressão de dados. Este é um fato importante, pois não realizar a compressão nos permitiu estender de maneira mais simples nossa solução geral para também cobrir os problemas relacionados. A Figura 4.14 e a Tabela 4.3 ilustram os resultados de comparação entre o Algoritmo 12 e o Algoritmo sequencial [4]. Neste teste obtivemos bons resultados de desempenho, contudo os valores diminuíram para entradas maiores. Acreditamos que esse *overhead* ocorreu devido à manutenção de matrizes cada vez maiores na memória de cada processador, entretanto é esta manutenção que nos permitiu estender nossa solução.

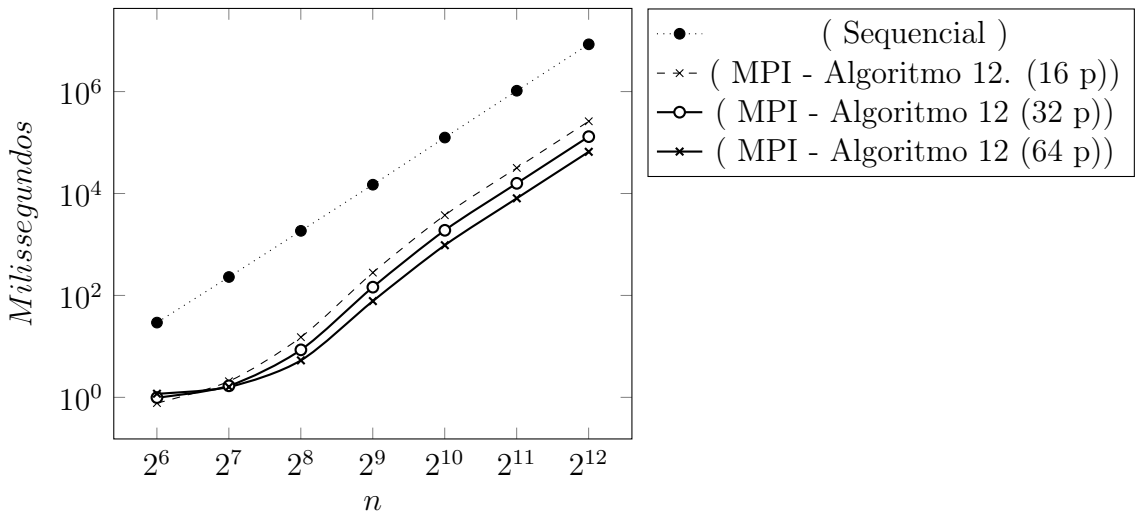


Figura 4.14: Sequencial × MPI versões (Algoritmo 12).

Tabela 4.3: Tempos de execução (**Milissegundos**): Seq. × MPI (16, 32 e 64 proc.).

$n \times n$	Sequencial	Tempo MPI (16 p.)	Tempo MPI (32 p)	Tempo MPI (64 p.)	*Speedup*(Seq./MPI-32p.)
64×64	29.2992	0.7725	0.9899	1.1785	29.5981
128×128	229.9130	2.0684	1.6854	1.6009	136.4145
256×256	1849.6825	15.1679	8.5535	5.3154	216.2486
512×512	14.879.4916	280.3312	145.2910	77.4639	102.4117
1024×1024	125.239.8713	3.748.8917	1.896.4386	966.3708	66.0395
2048×2048	1.038.533.5501	31.783.7091	15.835.1481	8.046.7096	65.5841
4096×4096	8.460.234.4575	261.292.5225	130.540.967	65.800.121	64.8090

O terceiro teste utiliza nossa implementação OpenMP para o Algoritmo 12. Nesse teste trabalhamos com 8 *threads*, pois a máquina disponível possuía um processador com dois núcleos (*cores*), cada um com 4 *threads*. A Figura 4.15 e a Tabela 4.4 ilustram os resultados de comparação deste algoritmo com a versão sequencial [4].

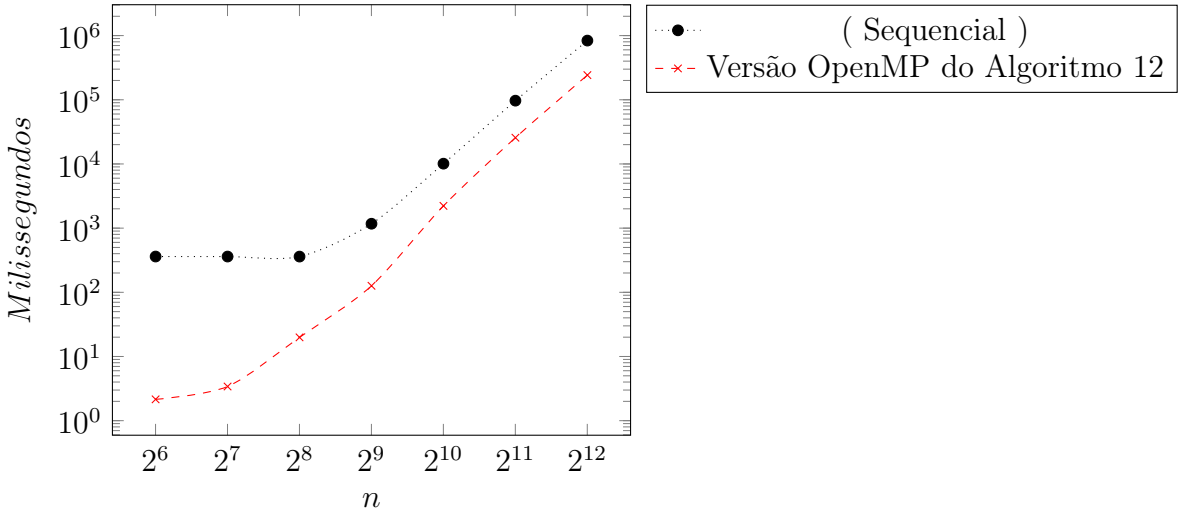


Figura 4.15: Sequencial × OpenMP (Algoritmo 12).

$n \times n$	Sequencial	OpenMP	*Speedup*
64×64	360.9985	2.1495	167.9453
128×128	360.8055	3.4071	105.8981
256×256	360.3314	19.8622	18.1416
512×512	1.171.1028	126.3485	9.2688
1024×1024	10.099.6082	2.221.9307	4.5454
2048×2048	96.861.6133	25.647.4277	3.7767
4096×4096	835.974.2750	242.314.7492	3.4500

Tabela 4.4: Tempos de Execução: (**Milissegundos**) Seq. × OpenMP.

O Segundo Caso de Comparação

O segundo caso de comparação utiliza nossas implementações CUDA para o Algoritmo 14. Neste caso duas versões foram implementadas, com e sem o balanceamento triangular, já descrito na Seção 4.6. A Figura 4.16 e a Tabela 4.5 ilustram os resultados de comparação com

a versão sequencial. É possível observar que a versão com balanceamento triangular mostrou-se um pouco mais eficiente.

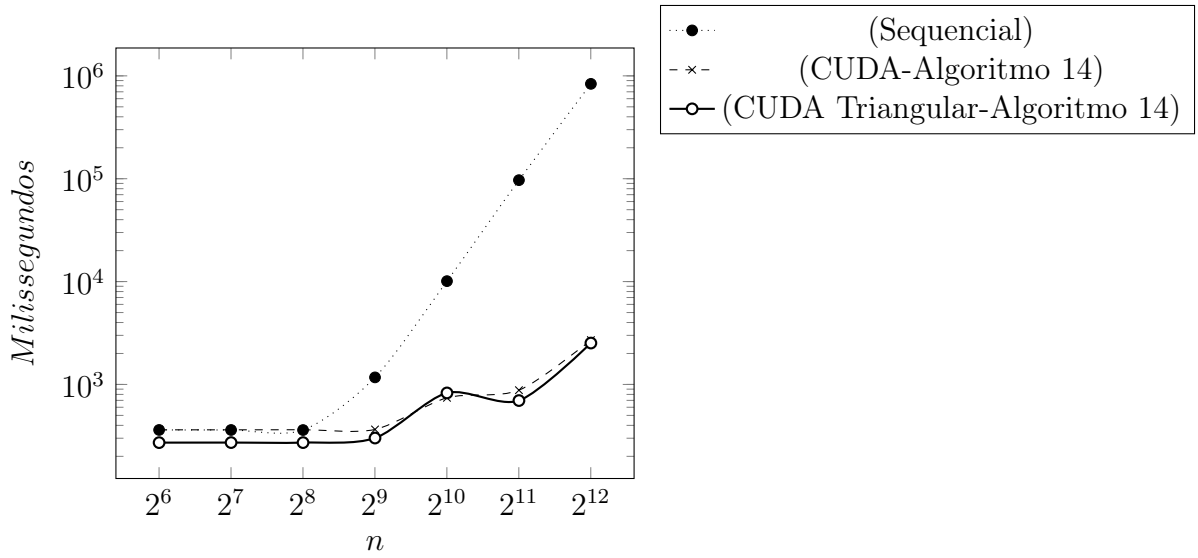


Figura 4.16: Sequencial × versões CUDA.

n	Sequencial	CUDA	CUDA Triangular	*Speedup* (Seq./CUDA)
64×64	360.9985	362.1512	271.7961	1.3282
128×128	360.8055	362.0914	272.1506	1.3258
256×256	360.3314	362.9239	271.9573	1.3250
512×512	1171.1028	365.0831	301.2226	3.8878
1024×1024	10099.6082	741.5493	826.9745	12.2127
2048×2048	96861.6133	879.3046	697.1044	138.9485
4096×4096	835974.2750	2699.4783	2525.3061	331.0388

Tabela 4.5: Tempos de Execução: Sequencial × CUDA.

O Terceiro Caso de Comparação

O terceiro e último caso ilustra a comparação de desempenho da nossa implementação CUDA em GPU (Algoritmo 14), com os resultados de um recente algoritmo CUDA em GPU desenvolvido por Cleber *et al.* [17], também para o problema da submatriz de soma máxima. É importante destacar que foram utilizados recursos computacionais diferentes no desenvolvimento de cada algoritmo. Além disso, nossa implementação CUDA realiza um trabalho computacional extra, para prover informações para as soluções dos problemas relacionados. A Tabela 4.6 ilustra a comparação entre as duas versões.

n	Speedup (Cleber *et al.*)	*Speedup* (Algoritmo 14)
512 × 512	80.2260	3.8878
1024 × 1024	121.9900	12.2127
2048 × 2048	215.2870	138.9485
4096 × 2048	235.9650	331.0388

Tabela 4.6: Comparações de desempenho.

4.7 Um Exemplo de Aplicação

Em diagnósticos baseados em imagens de raios-X, as áreas de interesse podem ser as mais brilhantes ou com maior intensidade luminosa (com mais pixels brancos). Neste contexto, uma

aplicação real para o problema da submatriz de soma máxima foi apresentada por Raouf *et al.*, cujo objetivo era localizar as regiões mais brilhantes em imagens de mamografia, com o intuito de detectar macrocalcificações [39].

Particularmente, as imagens *bitmap* são um bom exemplo de entrada para algoritmos que solucionam o problema de submatriz de soma máxima. Para isto os *pixels* RGB de uma imagem *bitmap* precisam ser convertidos em uma matriz de número reais ou inteiros.

O termo **luminância** expressa a intensidade de cor e em se tratando do sistema da visão humana, corresponde à percepção de brilho das imagens [45]. A luminância de cada pixel pode ser mensurada através da equação: $Luminância = 0.30R + 0.59G + 0.11B$. Os correspondentes numéricos de luminância de cada pixel formam a submatriz numérica da imagem original [3]. Para o cálculo de luminância um intervalo deve ser definido, geralmente é utilizado o intervalo $[0, 1]$.

4.8 A Nossa Aplicação

Utilizamos imagens reais como entrada de nosso algoritmo para a submatriz de soma máxima. A Figura 4.18 ilustra o resultado obtido por nosso algoritmo para uma entrada representada por uma imagem *bitmap* do raio-X de um cólon humano. A imagem pertence a um banco de dados público, que contém diversas imagens de tipos de câncer [26]. A Figura 4.17 ilustra nosso processo para o mapeamento da imagem inicial.

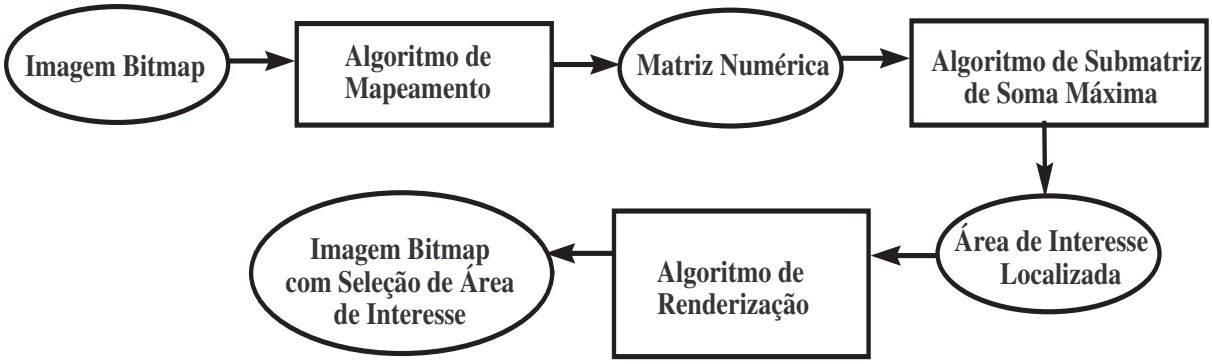


Figura 4.17: Os passos para encontrar a região mais brilhante em uma imagem de raio-X.

4.8.1 Passos da Aplicação

Vejamos a descrição dos passos executados durante a aplicação proposta:

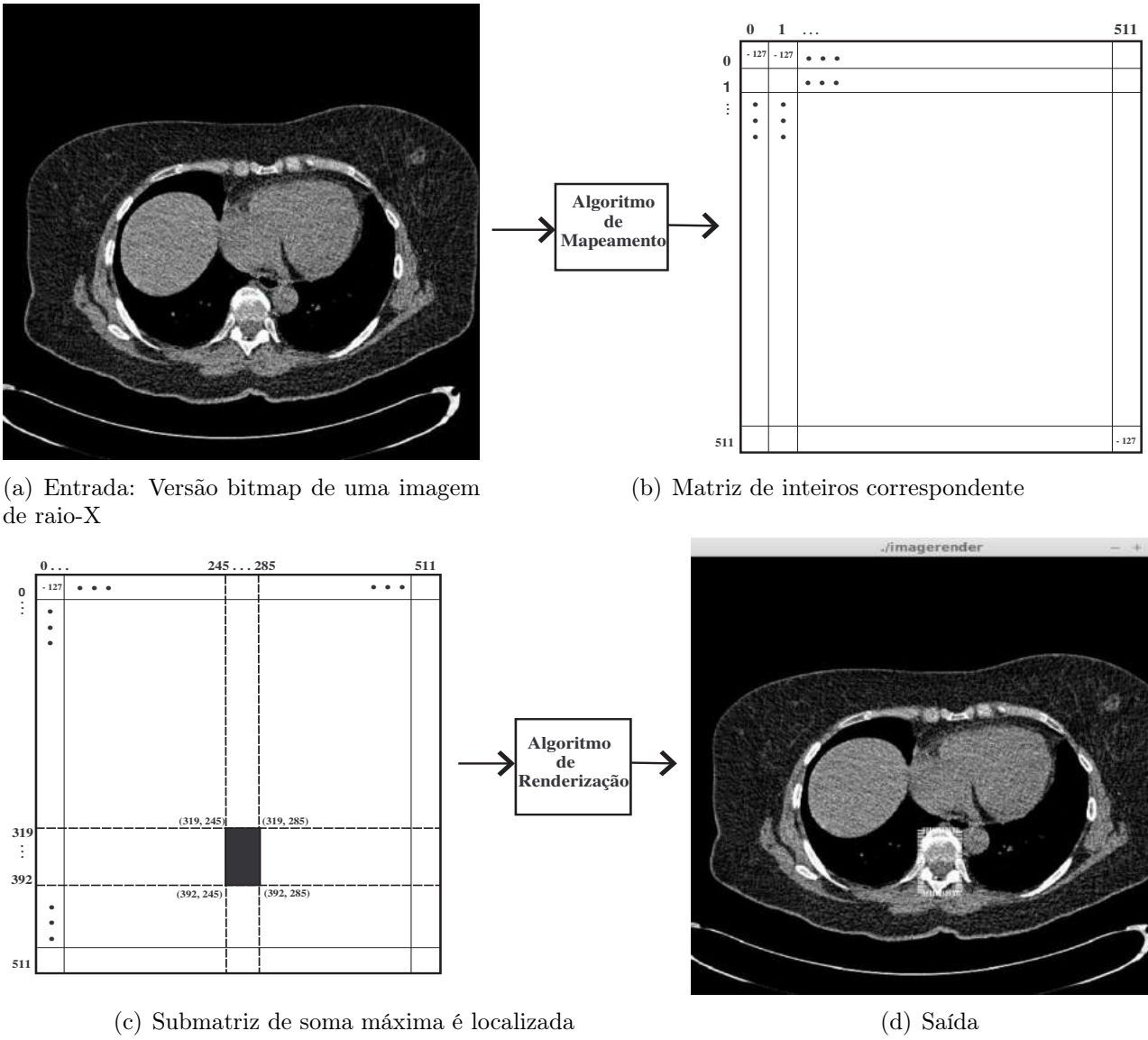
- Na Figura 4.18 temos que cada pixel da imagem *bitmap* original é convertido em um número inteiro utilizando um algoritmo de mapeamento. Este algoritmo utiliza a fórmula de luminância e um fator de correção. Nosso fator de correção é igual a -0.5. O intervalo padrão $[0, 1]$ foi convertido para $[-0.5, +0.5]$, assim obtivemos matrizes compostas de números positivos e negativos. Os pixels da imagem *bitmap* são substituídos por valores numéricos inteiros gerando a matriz numérica;

- A matriz numérica é a entrada para o próximo passo, que consiste da localização da área de interesse. Para tal aplica-se o algoritmo para localizar a submatriz de soma máxima.

No final deste passo, a saída gerada retorna quatro pontos, que delimitam a área de interesse (retângulo);

- Em seguida, os quatro pontos encontrados são utilizados na entrada de nosso algoritmo de renderização, que por fim ilustra novamente a imagem original, delimitando a área mais brilhante.

No exemplo da Figura 4.18 os quatro pontos (x, y) que definem a matriz retangular de soma máxima são: $(319, 245)$, $(319, 285)$, $(392, 245)$ e $(319, 285)$. A submatriz de soma máxima possui valor soma igual a 132382, como observado na Figura 4.18 (c). Em nosso algoritmo de renderização utilizamos soluções fornecidas pela biblioteca **DevIL** (*Developer's Image Library*) e o pacote ***GraphicsMagick***, ambas ferramentas são de código aberto e são dedicadas ao processamento de imagens.



(a) Entrada: Versão bitmap de uma imagem de raio-X

(b) Matriz de inteiros correspondente

(c) Submatriz de soma máxima é localizada

(d) Saída

Figura 4.18: (a) Imagem *bitmap* do raio-X de um cólon. (b) Matriz quadrática (512×512) gerada após o mapeamento. (c) Localização da submatriz de soma máxima. (d) A imagem saída com a região mais brilhante delimitada após a renderização.

4.9 Considerações Finais do Capítulo

Neste capítulo apresentamos soluções paralelas BSP/CGM eficientes para o problema da submatriz de soma máxima e para cinco problemas relacionados. Não é de nosso conhecimento a existência de algoritmos BSP/CGM para os problemas relacionados.

O bom desempenho dos nossos algoritmos BSP/CGM foi confirmado por meio de resultados experimentais. Diferentemente da solução para a submatriz de soma máxima, apresentada por Alves *et al.* [2], em nossos algoritmos não efetuamos a compressão de dados, e sempre que possível mantivemos as unidades de processamento executando de forma independente umas das outras. Nossos resultados confirmaram, que mesmo sem a compressão, as soluções obtidas foram efetivas, com *speedups* de até centenas de vezes melhor que a melhor solução sequencial!

Além disto, como parte de nossos resultados, também conduzimos um experimento que utilizou uma imagem real. Mostramos que nosso algoritmo de submatriz de soma máxima pode ser aplicado com sucesso para localização de regiões retangulares mais brilhantes em imagens de raio-X.

CAPÍTULO 5

O HIPER-RETÂNGULO DE SOMA MÁXIMA

Podemos definir um hiper-retângulo (também chamado de caixa) como um objeto geométrico de n-dimensões. As dimensões não necessariamente possuem tamanhos iguais. Um hiper-retângulo tridimensional é chamando de prisma ou de paralelepípedo. Neste capítulo discutimos uma generalização do problema da submatriz de soma máxima. Agora, consideramos a inserção de um terceiro eixo (dimensão de profundidade). A entrada deste novo problema consiste de um objeto tridimensional de dimensões $n \times n \times n$ (um cubo). A saída, entretanto, apesar de também ser um objeto tridimensional, não necessariamente possui dimensões de mesmo tamanho, tal como um paralelepípedo. Em nosso caso preferimos manter apenas o termo geral hiper-retângulo.

Propomos aqui um algoritmo para o problema do hiper-retângulo de soma máxima. Não é de nosso conhecimento a existência de soluções BSP/CGM para este problema. Como será apresentado, obtivemos resultados promissores. O algoritmo possui aplicações em diversas áreas da ciência, com destaque para aquelas que envolvem a visualização tridimensional de imagens médicas ou de ambientes geográficos, como solos ou regiões oceânicas. Particularmente, o algoritmo pode ser utilizado no mapeamento de volumes rochosos, para rápida análise de suas propriedades e também para a interpretação de amplitudes sísmicas e suas anomalias.

5.1 Histórico e Definição

Em nossos estudos encontramos diversas referências de soluções sequenciais e paralelas para o problema bidimensional de soma máxima, como já foi descrito no capítulo anterior. Entretanto, não é de nosso conhecimento a existência de algoritmos BSP/CGM para a versão tridimensional do problema.

Perumalla e Deo [35] apresentaram uma descrição teórica deste problema, relatando que ele pode ser generalizado no modelo PRAM. Estes autores descrevem que a entrada do problema é composta por um conjunto de subplanos ou subfaces, que por sua vez são matrizes bidimensionais. A estratégia consiste então em realizar a compressão de tubos tridimensionais, computando a subsequência de soma máxima de cada plano bidimensional do tubo. Perumalla e Deo [35] afirmaram que é possível encontrar o sub-d-cubo de soma máxima de um d-cubo, com lados de comprimento n, em tempo $\mathrm{O}\big(\log n\big)$, utilizando $n \times \binom{n}{2}^{d-1}$ processadores no modelo PRAM [35]. Uma ilustração de entrada e saída do problema é representada pela Figura 5.1.

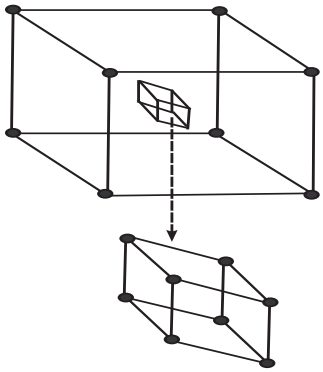


Figura 5.1: Hiper-retângulo tridimensional de soma máxima obtido a partir de um 3-cubo.

5.2 Algoritmo Proposto

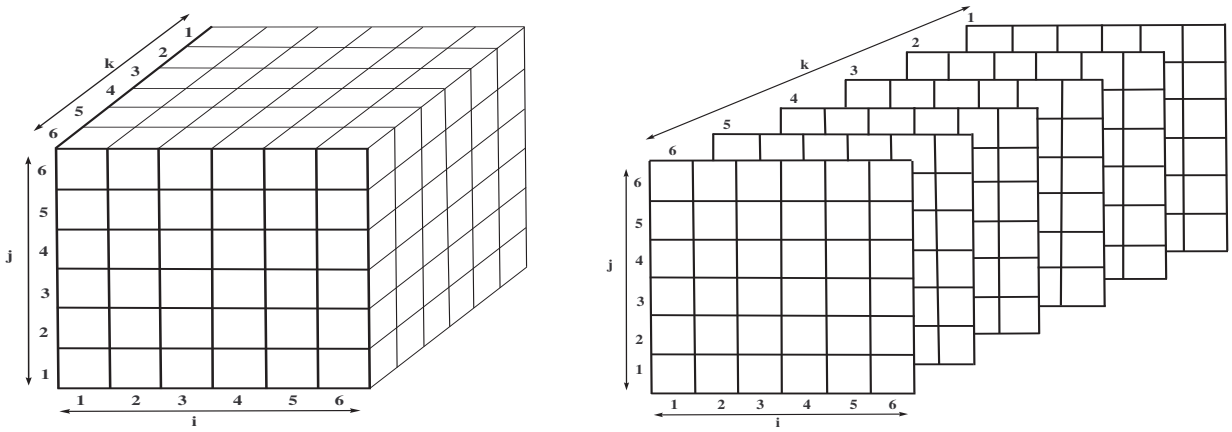
Nosso algoritmo paralelo é baseado nas ideias descritas por Perumalla and Deo [35]. Trata-se de um algoritmo paralelo para o problema do hiper-retângulo de soma máxima. O algoritmo utiliza as estratégias do algoritmo para submatriz de soma máxima, mas considerando agora a existência de um eixo de profundidade vinculado com cada antiga submatriz $C^{(g,h)}$. Em cada submatriz adiciona-se então um par de valores (r, t) que representa o intervalo de profundidade. Definimos o novo conjunto de trabalho como submatrizes $C^{(g,h,r,t)}$'s.

Durante sua execução o algoritmo se desdobra computando as submatrizes bidimensionais considerando a compressão dos elementos perperdiculares do eixo de profundidade. Por fim, assim como na versão 2D, o algoritmo invoca o melhor algoritmo sequencial para localizar a subsequência de soma máxima da submatriz de soma máxima $C^{(g,h,r,t)}$, que por sua vez contém o hiper-retângulo de soma máxima.

5.2.1 Um Exemplo Passo a Passo

Apresentamos a seguir as estratégias, que formulamos para solucionar o problema do hiper-retângulo tridimensional de soma máxima. Dividimos o nosso algoritmo em quatro passos principais e sempre que possível buscamos estratégias de paralelização.

Para melhor compreensão ilustramos a solução do algoritmo através de imagens. A entrada do algoritmo é um cubo, conforme ilustrado pela Figura 5.2 (a). Obviamente, o cubo é delimitado por três eixos: i (linhas), j (colunas) e k (profundidade), como é ilustrado pela Figura 5.2 (b).



(a) Os três eixos de um cubo.

(b) Os n conjuntos de submatrizes quadráticas.

Figura 5.2: O cubo e suas n submatrizes $n \times n$.

1. O primeiro passo do algoritmo consiste em aplicar um algoritmo de soma de prefixos, linha por linha, em cada uma das n subfaces bidimensionais do cubo original, conforme ilustrado pela Figura 5.3. O resultado deste passo é um cubo de soma de prefixos (face a face);

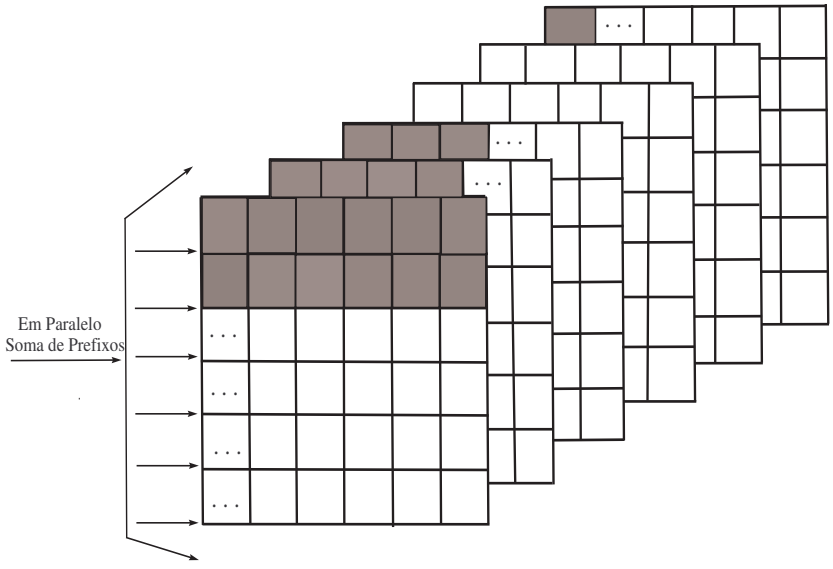


Figura 5.3: A obtenção de soma de prefixos (linha a linha).

2. No segundo passo o cubo de prefixos é rotacionado à esquerda. Este passo é o cerne do algoritmo, visto que agora é preciso considerar a inserção dos elementos do eixo de profundidade e as respectivas computações de soma de prefixos. Após a rotação, as n subfaces são novamente selecionadas. Entretanto, serão consideradas agora, apenas as subfaces que são perperdiculares ao eixo j e paralelas ao eixo k (profundidade). Feito isto, novamente operações de soma de prefixos são aplicadas, linha a linha, em cada uma das subfaces, conforme ilustrado pela Figura 5.4. A saída deste passo ilustra o cubo de soma de prefixos completo;

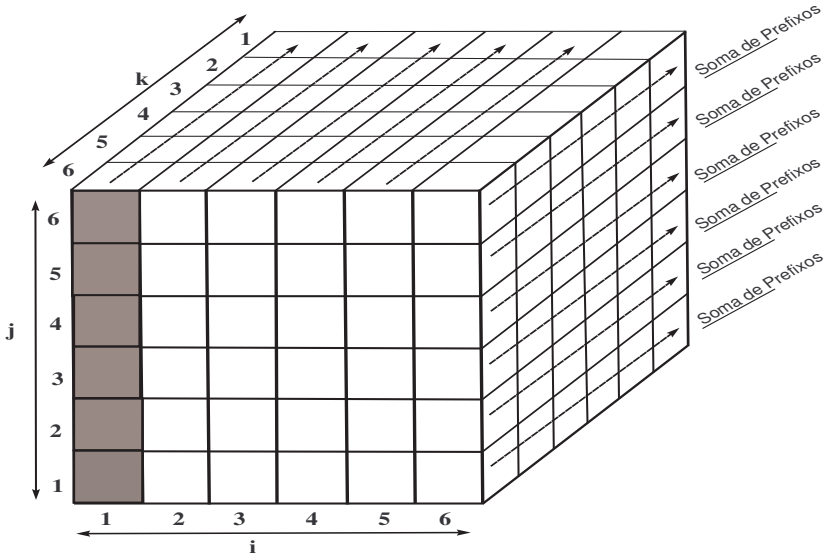


Figura 5.4: A obtenção de soma de prefixos (eixo perpendicular).

3. O próximo passo consiste em localizar o valor de soma máxima de cada hiper-retângulo do cubo de soma de prefixos. A determinação do conjunto de prismas retangulares é similar a dos conjuntos bidimensionais $C^{(g,h)}$ do problema do *subarray* de soma máxima. Agora, além dos índices g e h que compõem o intervalo de colunas da subface bidimensional do hiper-retângulo, criamos dois novos índices: r e t, que descrevem o intervalo de profundidades possíveis para cada subface $C^{(g,h)}$ de cada hiper-retângulo. Estendemos então a notação $C^{(g,h)}$ para $C^{(g,h,r,t)}$. Por exemplo, a Figura 5.5 ilustra os cálculos de $C^{(1,1,1,1)}$ até $C^{(1,1,1,n)}$;

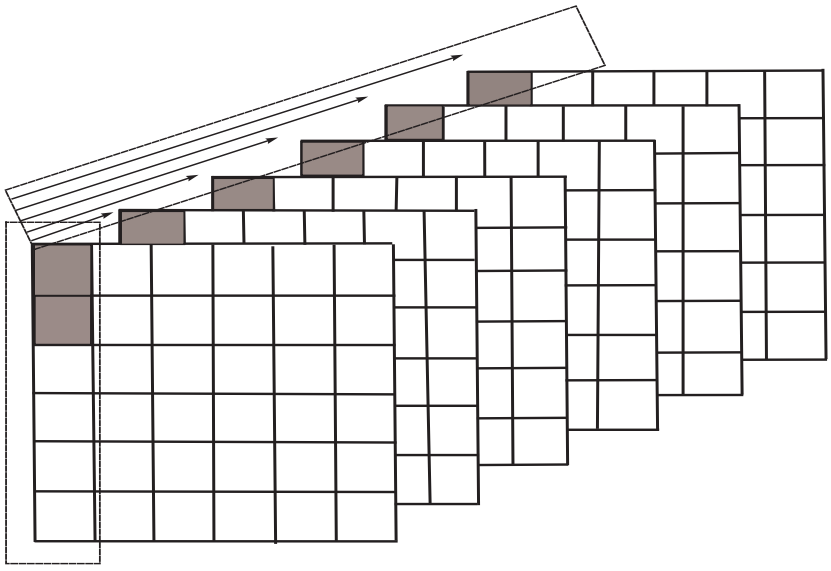


Figura 5.5: Varredura das profundidades de um hiper-retângulo $C^{(g,h,r,t)}$.

4. No último passo aplicamos um algoritmo paralelo de redução para máximo. O objetivo é localizar o maior valor de soma dentre as somas máximas de todos os hiper-retângulos do cubo.

Nossa versão sequencial do problema consiste da aplicação sequencial dos passos descritos

anteriormente. Nossa solução paralela é similar (com paralelização sempre que possível). O pseudocódigo do algoritmo paralelo é representado pelo Algoritmo 16.

Algoritmo 16: Algoritmo BSP/CGM - problema geral.

Entrada: (1) Um conjunto de P processadores; (2) O número i de cada processador $p_i \in P$, onde $1 \leq i \leq P$; (3) Um inteiro n; (4) Uma matriz tridimensional M com n^3 elementos inteiros.
Saída: (1) O inteiro $global\_max$ que representa o maior valor de soma máxima entre todos os possíveis prismas retangulares de M.

1: **Em paralelo faça:** $PSUM \leftarrow$ Algoritmo de Soma de Prefixos (P , M) em cada linha do eixo vertical (j). {Temos aqui n^2 linhas, com n elementos cada}
2: **Em paralelo faça:** $PSUM \leftarrow$ Algoritmo de Soma de Prefixos (P, $PSUM$) em cada linha no eixo perpendicular (k). {Temos aqui n^2 linhas, com n elementos cada}
3: **Em paralelo faça:** $local\_max \leftarrow C^{(g,h,r,t)}(i, n, PSUM)$. {Algoritmo de seleção do conjunto de trabalho de cada processador p_i (Algoritmo 17)}.
4: **Para cada processador** p_i **em paralelo faça** send($local\_max$,$p_i = 0$)
5: **if** $p_i = 0$ **then**
6: $global\_max \leftarrow$ INT_MIN
7: **for** $j = 0$ **to** P **do**
8: $global\_max \leftarrow max$(receive($max\_local\_j$),$global\_max$)
9: **end for**
10: **end if**
11: **return** $global\_max$

Durante a execução do Algoritmo 16 são realizadas chamadas para dois outros algoritmos. O primeiro algoritmo é responsável pela seleção de cada hiper-retângulo (Algoritmo 17). Neste algoritmo, o mapeamento da carga de trabalho de cada processador é similar ao mapeamento triangular utilizado no problema da submatriz de soma máxima. O segundo algoritmo é responsável pela localização da respectiva soma máxima. Este último é uma especialização do melhor algoritmo sequencial (Algoritmo 3), que agora também deve comtemplar os elementos do eixo de profundidade.

Algoritmo 17: Algoritmo BSP/CGM para o cálculo de $C^{(g,h,r,t)}$.

Entrada: (1) A identificação de cada processador p_i, onde $0 \leq i \leq P$; (2) Um inteiro n; (3) Uma matriz tridimensional M com n^3 elementos inteiros com somas de prefixos previamente calculados.
Saída: (1) Um inteiro *local_max* que representa o valor máximo local do conjunto de trabalho do processador p_i.

1: $local\_max \leftarrow$ INT_MIN
2: $C^{(g,h,r,t)}$'s $\leftarrow$ O conjunto de trabalho gerado pelos valores g, h, r e t vinculados com cada processador p_i. {Mapeamento paralelo da carga de trabalho de cada processador.}
3: **for** Cada elemento $C^{(g,h,r,t)}$ do conjunto $C^{(g,h,r,t)}$'s do processador p_i **do**
4: $\quad local\_max \leftarrow max(local\_max,$ Algoritmo_3_Especializado$(n, M, g, h, r, t))$
5: **end for**
6: **return** *local_max*

5.2.2 Complexidade

A complexidade de nosso algoritmo paralelo (Algoritmo 16) é obtida através da análise dos passos de computação e comunicação. Vejamos os detalhes em cada etapa:

- **Computação**. **Passo 1:** Inicialmente o Algoritmo 16 executa uma operação de soma de prefixos em cada linha do cubo de entrada. Existem $O(n^2)$ linhas no total. Cada processador calcula a soma de prefixos de cada linha em tempo $O(n)$. Assim, a complexidade do passo 1 é $O(\frac{n^3}{P})$. **Passo 2:** O procedimento executado neste passo é o mesmo do passo 1, entretanto as operações são efetuadas em um eixo diferente. A complexidade do passo continua a mesma, em tempo $O(\frac{n^3}{P})$. **Passo 3**: Neste passo uma chamada para o Algoritmo 17 é realizada. Cada processador é responsável por calcular um conjunto de $C^{(g,r,h,t)}$'s. Cada subface do cubo possui um conjunto de $C^{(g,h)}$'s de tamanho $O(n^2)$. Entretanto, considerando o novo eixo de profundidade, o número de $C^{(g,h,r,t)}$'s é igual a $O(n^4)$. Além disto, em cada $C^{(g,h,r,t)}$ uma variação do algoritmo sequencial para subsequência de soma máxima é executada (Algoritmo 3). O Algoritmo 3 possui complexidade $O(n)$, similar ao algoritmo sequencial original [4], com isto o passo 3 executa em tempo $O(\frac{n^4}{P} * O(n))$, considerando o número de processadores a complexidade gerada é $O(\frac{n^5}{P})$. **Steps 4-11:** Os passos de 4 a 11 executam em $O(1)$ para $p_i \neq 0$ e $O(P)$ para $p_i = 0$. Ao final, a complexidade total do algoritmo é $O(\frac{n^3}{P}) + O(\frac{n^5}{P}) + O(P) = O(\frac{n^5}{P})$. Particularmente, em nossa implementação em GPU, a comunicação pode ser considerada em tempo $O(1)$.

5.3 Implementaçõe e Resultados

A seguir apresentamos os resultados obtidos por nosso algoritmo paralelo BSP/CGM para o hiper-retângulo de soma máxima. Para este algoritmo desenvolvemos versões paralelas em OpenMP e em CUDA. Acreditamos que uma versão MPI também pode elaborada, mas por hora investimos nosso esforço na implementação em GPU. O alto desempenho do algoritmo em CUDA foi comprovado por experimentos, como será descrito. Realizamos testes, que

ilustram comparações entre as versões paralelas do algoritmo e a respectiva versão sequencial. É importante destacar que a versão sequencial também foi executada na mesma máquina que possui o ambiente de memória compartilhada (GPU).

O Caso de Comparação

Realizamos um teste que ilustra os resultados comparativos de três versões do algoritmo para o hiper-retângulo de soma máxima: a versão sequencial, uma versão OpenMP e a última em CUDA com milhares de *threads*.

A Figura 5.6 e a Tabela 5.1 ilustram os resultados de tempos de execução da solução sequencial e dos algoritmos paralelos.

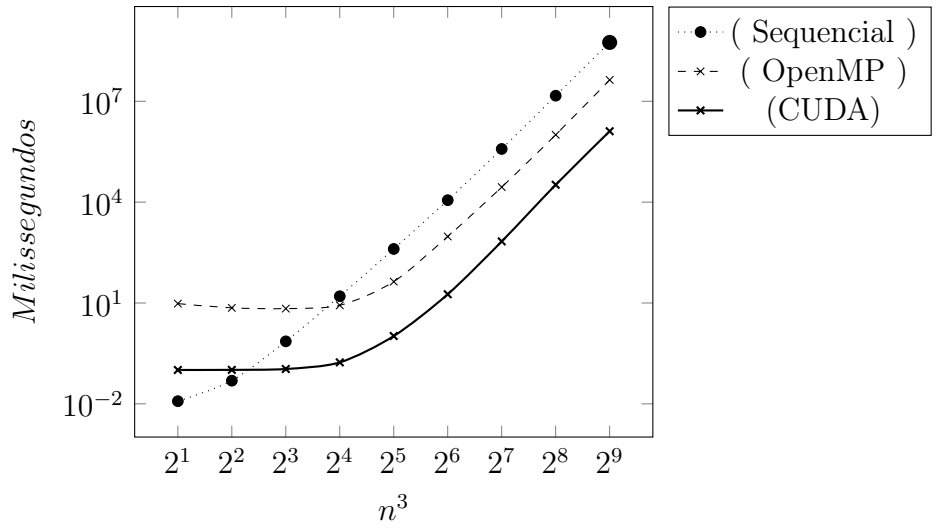


Figura 5.6: Sequencial × Versões Paralelas.

n^3	Sequencial	OpenMP	CUDA
$2x2x2$	0.0120	9.5857	0.1018
$4x4x4$	0.0488	7.1616	0.1025
$8x8x8$	0.7230	6.8034	0.1088
$16x16x16$	15.8895	8.5247	0.1712
$32x32x32$	404.4507	43.1587	1.0389
$64x64x64$	11.422.7108	950.2646	18.2016
$128x128x128$	381.724.4531	28.012.7312	678.2361
$256x256x256$	14.670.318.9000	1.007.455.7015	32.863.23183
$512x512x512$	563.805.265.462576*	43.105.127.6108	1.285.293.7000

Tabela 5.1: Tempos de Execução (**Milissegundos**): Sequencial. × Versões Paralelas (OpenMP and CUDA).

A Figura 5.7 e a Tabela 5.2 ilustram os resultados de comparação de desempenho entre a solução sequencial e os algoritmos paralelos. Os resultados comprovam a eficiência da versão CUDA com milhares de *threads*, que quando comparada com a versão sequencial obteve um *speedup* de até 627 melhor, para $n = 64 \times 64 \times 64$. Na Figura 5.7 é possível observar que a curva de *speedup* geralmente é crescente. Uma exceção está na coluna de desempenho entre a versão sequencial e a versão CUDA com milhares de *threads*. Neste caso, a curva é crescente até n igual a $64 \times 64 \times 64$. Após este pico, a curva decresce um pouco, isto pode ser explicado por um aumento de *overhead* durante a execução.

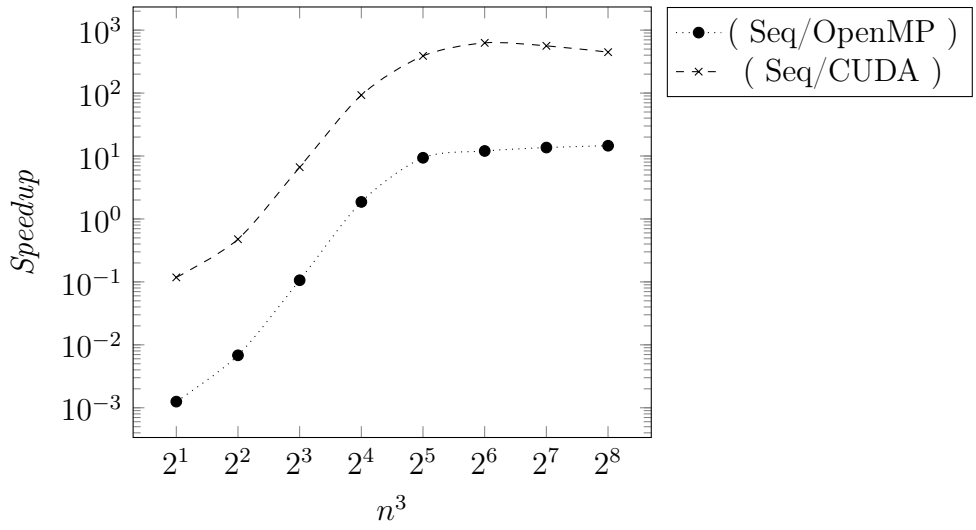


Figura 5.7: *Speedup*: Sequencial × Versões Paralelas (OpenMP e CUDA).

n^3	Seq/OpenMP	Seq/CUDA
$2x2x2$	0.0012	0.1179
$4x4x4$	0.0068	0.4763
$8x8x8$	0.1063	6.6457
$16x16x16$	1.8639	92.7880
$32x32x32$	9.3712	389.2962
$64x64x64$	12.0206	627.5675
$128x128x128$	13.6268	562.8194
$256x256x256$	14.5618	446.4052

Tabela 5.2: *Speedup*: Sequencial / Versões Paralelas.

5.4 Considerações Finais do Capítulo

Neste capítulo apresentamos uma solução paralela BSP/CGM para o problema do hiper-retângulo tridimensional de soma máxima. Não é de nosso conhecimento a existência de algoritmos BSP/CGM para este problema. A solução paralela se mostrou bastante eficiente, alcançando valores de *speedup* de até centenas de vezes melhor, do que a respectiva solução sequencial. Nosso algoritmo utilizou p processadores em tempo paralelo $O\left(\frac{n^5}{P}\right)$. Da mesma forma que para os casos uni e bidimensional, podemos também obter soluções BSP/CGM para os problemas relacionados. Essas soluções podem ser utilizadas para análise de sólidos tridimensionais, como por exemplos os *voxels*. Um voxel representa uma única amostra, ou dado pontual, em um *grid* regular tridimensional [1].

CAPÍTULO 6

O PROBLEMA DA SELEÇÃO

Neste capítulo abordamos o problema da seleção. O problema consiste em determinar o k-ésimo menor elemento de um conjunto (não ordenado) de tamanho n de números geralmente distintos. Constituem-se em casos especiais deste problema a determinação do elemento mínimo ($k = 1$), do elemento máximo ($k = n$) e da mediana ($k = \lceil n/2 \rceil$)) de um conjunto de n de números. Um exemplo do problema é ilustrado pela Figura 6.1. Particularmente, a seleção de medianas tem sido amplamente utilizada pela ciência da computação em problemas de ordem ou de processamento de sinais [21].

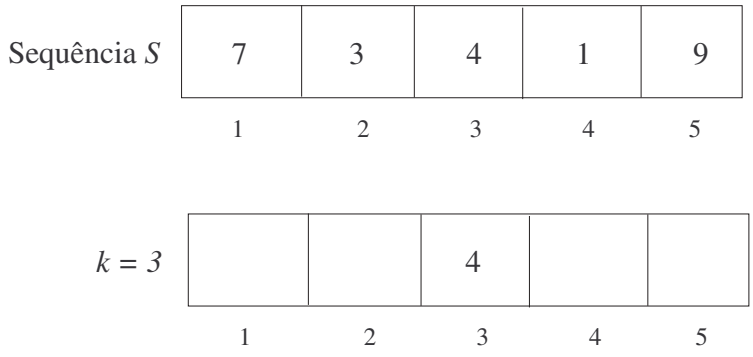


Figura 6.1: Terceiro menor elemento de uma sequência S não ordenada.

Na primeira seção deste capítulo (Seção 6.1) apresentamos o histórico do problema e destacamos conhecidas e eficientes soluções sequenciais. Já na Seção 6.2, elencamos e discutimos algumas das principais soluções paralelas. É importante destacar que as estratégias de algumas soluções sequenciais são por vezes utilizadas pelas soluções paralelas.

Em seguida, na Seção 6.3, apresentamos e discutimos duas soluções paralelas para uma especialização do problema da seleção: a seleção de mediana. Essas soluções foram implementadas para o modelo de memória compartilhada de uma GPGPU. A primeira solução utiliza em parte a estratégia do algoritmo sequencial bitônico para ordenação. A segunda solução é baseada em um trabalho recente, que discute três novas e diferentes soluções sequenciais para o problema de seleção de mediana. Os autores deste trabalho indicam que uma solução em ambiente com muitos núcleos (milhares) e em memória compartilhada, poderia ser muito eficiente [21].

Por fim, na Seção 6.4, terminamos discutindo e apresentando uma solução paralela em memória compartilhada para o problema geral de seleção. Particularmente, descreveremos em detalhes o algoritmo BSP/CGM proposto em 1997 por Saukas e Song [42]. Trata-se de um algoritmo eficiente, que faz uso da melhor solução sequencial e que além disto promove a diminuição de elementos e um justo balanceamento de carga entre as unidades de processamento, a cada nova rodada de computação. Não é de nosso conhecimento a implementação paralela deste algoritmo em um ambiente de múltiplos núcleos, tal como uma GPU. Neste sentido, apresentamos a inédita adaptação deste algoritmo para o modelo de memória compartilhada de uma GPGPU.

6.1 Histórico

O problema de seleção é talvez melhor exemplificado pelo cálculo de medianas. Além de ser um dos problemas fundamentais e clássicos da ciência da computação, o problema da seleção é utilizado na solução de outros problemas básicos, tais como a ordenação. O estudo do problema da seleção é bastante antigo. Em 1883, Lewis Carrol publicou um artigo denunciando o método injusto pelo qual o segundo melhor jogador geralmente é determinado em um torneio eliminatório de chaves. Ele afirmou que o perdedor do jogo final muitas vezes não é o segundo melhor e que qualquer um dos jogadores que perderam apenas para o melhor jogador pode ser o segundo melhor. Por volta de 1930, Hugo Steinhaus trouxe o problema para a área da complexidade algorítmica, apresentando em seu trabalho o número mínimo de jogos necessários para selecionar o primeiro e o segundo melhores jogadores de um grupo de n concorrentes. A partir da década de 30 diversos outros artigos foram publicados a respeito do ponto de vista algorítmico do problema [30].

Formalmente, o problema da seleção é definido da seguinte forma [27]: dados um conjunto de elementos (distintos) $A = a_1, a_2, \ldots, a_n$ e um inteiro k, com $1 \leq k \leq n$, determinar o elemento a_i de A tal que $rank(a_i : A) = k$. Por sua vez, *rank* é definido da seguinte forma: Seja $X = x_1, x_2, \ldots, x_t$ uma sequência cujos elementos pertencem ao conjunto S. Seja $x \in S$, o *rank* de x em X, denotado por $rank(x : X)$, é o número de elementos de X que são menores ou iguais a x.

A definição anterior é adequada somente para conjuntos com elementos distintos, uma restrição adotada (ao menos na formalização do problema) nas referências utilizadas. No caso de conjuntos com elementos não necessariamente distintos, a aplicação direta desta definição implicaria que, por exemplo, o problema de seleção da mediana da sequência $A = 5, 6, 7, 7, 8$ não teria solução, pois esta sequência não possui nenhum elemento com *rank* igual a 3 (note que $rank(7{:}A) = 4$). Para contornar esta restrição, o problema da seleção pode ser mais genericamente definido desta forma: Dados um conjunto de elementos $A = a_1, a_2, \ldots, a_n$ e um inteiro k, com $1 \leq k \leq n$, determinar um elemento a_i de A tal que $rank(a_i : A) = k$ e para todo elemento a_j de A, se $a_j < a_i$ então $rank(a_j : A) \leq k$ [41]. Deve-se observar que esta definição corresponde ao conceito de considerar a seleção como o problema de determinar qual é o elemento de índice k na sequência A após a ordenação de seus elementos (independentemente desta sequência ser ou não composta somente de elementos distintos) [41, 42].

A seguir apresentamos alguns dos principais algoritmos disponíveis na literatura para o problema da seleção. Em geral, os algoritmos paralelos para memória distribuída consistem em adaptações de algoritmos paralelos para memória compartilhada, que por sua vez foram elaborados a partir de algoritmos sequenciais. Iniciaremos nosso estudo pelos algoritmos sequenciais.

6.1.1 Principais Algoritmos Sequenciais para Seleção

São três os mais conhecidos algoritmos sequenciais para seleção. Nesta seção descrevemos cada um deles.

O Algoritmo *Find*: A Estratégia do Quicksort

O problema da seleção pode ser resolvido sequencialmente em tempo linear $O(n)$ através da estratégia de particionamentos sucessivos. Note-se que $\Omega(n)$ é um limite inferior trivial para este problema [41, 42].

O algoritmo *Find*, proposto por Hoare em 1961 [25], resolve este problema em tempo linear no caso médio. O objetivo do algoritmo é selecionar o k-ésimo menor elemento de um arquivo de n elementos (sequência de entrada S). A solução proposta por Hoare consiste em inicialmente selecionar aleatoriamente um elemento x da sequência S e em seguida particionar a sequência com base neste separador(utilizando o mesmo procedimento de particionamento do algoritmo *Quicksort*(*Partition*) Não por acaso, o algoritmo **Find** também é conhecido como *Quickselect*. A Figura 6.2 ilustra exemplos de saídas para este algoritmo.

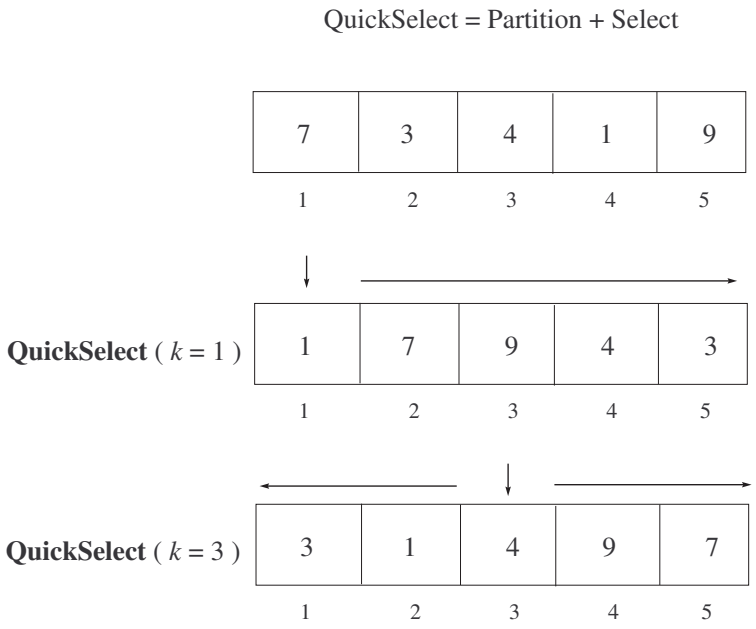


Figura 6.2: Saída de *Quickselect* para o primeiro e o quinto menores elementos.

Relembrando, o algoritmo *QuickSort* adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves menores precedam as chaves maiores. Em seguida, o *Quicksort* ordena as duas sublistas de chaves menores e maiores recursivamente, até que a lista completa se encontre ordenada. Os passos do *Quicksort* são:

1. Escolha um elemento da lista, denominado pivô;
2. Rearranje a lista de forma que todos os elementos à esquerda do pivô sejam menores e à direita sejam maiores;
3. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada partição.

O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte. Assim como **QuickSort**, o algoritmo *Quickselect* é baseado em um paradigma composto de duas etapas: **Partição** e **Busca**:

1. **Partição**: Consiste em escolher aleatoriamente um elemento x (chamando pivô) e particionar a sequência de entrada S em:
   - L elementos menores do que x. (A esquerda de x).
   - E elementos iguais a x.
   - G elementos maiores que x. (A direita de x).
2. **Busca**: Dependendo de k, o k-ésimo elemento está em E, senão precisamos utilizar recursividade no conjunto L ou em G.

De acordo com o paradigma, durante a execução da etapa de Partição os seguintes casos podem ocorrer [25]:

1. $k \leq |L|$. Neste caso o k-ésimo menor elemento está a esquerda do pivô.
2. $k > |L| + |E|$. Neste caso o k-ésimo menor elemento está a direita do pivô e o valor de k é alterado: $k = k$ - $|L|$ - $|E|$.
3. $|L| < k \leq |L| + |E|$. Neste caso já encontramos o k-ésimo menor elemento, isto é, existem exatamente k-1 elementos que são menores que o k-ésimo elemento.

O pseudocódigo do algoritmo *Quickselect* é ilustrado pelo Algoritmo 18.

Algoritmo 18: Algoritmo *Quickselect*.

Entrada: Sequência S. Índice k (k-ésimo menor elemento)
Saída: k-ésimo menor elemento = $S[k]$

```
1: Partition(S,k,pivo)
2: if k < |L| then
3:     QuickSelect(L,k,pivo)
4: else if k > |L| + |E| then
5:     k = k − |L| − |E|
6:     QuickSelect(G,k,pivo)
7: else
8:     return pivo
9: end if
```

Um exemplo da execução passo a passo do Algoritmo **Quickselect** é ilustrado pela Figura 6.3. Após a execução de **QuickSelect** o vetor de entrada estará parcialmente ordenado e seleções (buscas) subsequentes podem ser realizadas de forma mais rápida.

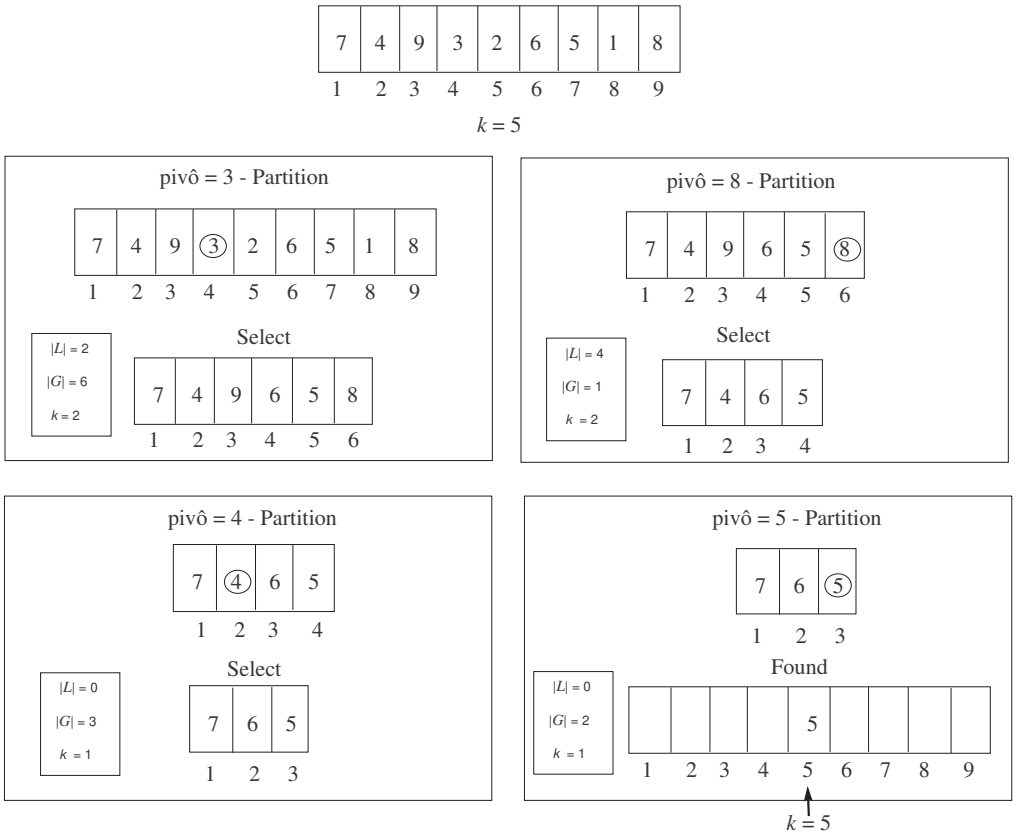


Figura 6.3: Execução de Quickselect na busca do quinto menor elemento.

Desde a apresentação do algoritmo *Quickselect* até o presente, muitas outras soluções foram propostas para o problema da seleção. A maioria das soluções propostas, ainda que em base teórica pareçam melhor que o algoritmo de Hoare, não são práticas, porque são difíceis de implementar. Os fatores constantes e termos de ordem inferior ocultos são muito grandes. Por sua vez, a probabilidade de grandes desvios de desempenho em *Quickselect* é provavelmente muito pequena e os termos de ordem inferior e fatores constantes ocultos em qualquer implementação razoável do algoritmo provavelmente são muito pequenos. Assim, a simplicidade, a elegância e robustez do algoritmo *Quickselect* até hoje persistem o tornando um dos melhores melhores algoritmo de seleção para a maioria das situações práticas. Assim como em *QuickSort*, estratégias diferentes para a escolha do pivô são consideradas. Uma estratrégia para escolha do pivô é conhecida como a mediana-de-três versões, onde o elemento pivô é escolhido como a mediana de uma amostra aleatória de três elementos, para tal, os autores estabelecem algumas equações diferenciais hipergeométricas, e encontram fórmulas explícitas, para o número médio de passos e comparações [41].

a escolha do pivô é bastante relevante para o término do algoritmo. Pode-se ter boas e más chamadas recursivas, dependendo do pivô escolhido. Considere uma chamada recursiva de Quickselect sobre uma sequência S de tamanho s. Pode-se ter boas e más escolhas do pivô definidas da seguinte forma:

- Boas Escolhas: tamanhos de L e G são menores que $\frac{3}{4}$ de s;
- Más Escolhas: L ou G tem tamanho maior que $\frac{3}{4}$ de s.

A Figura 6.4 ilustra um exemplo de escolha de pivô com chamadas recursivas boas e ruins.

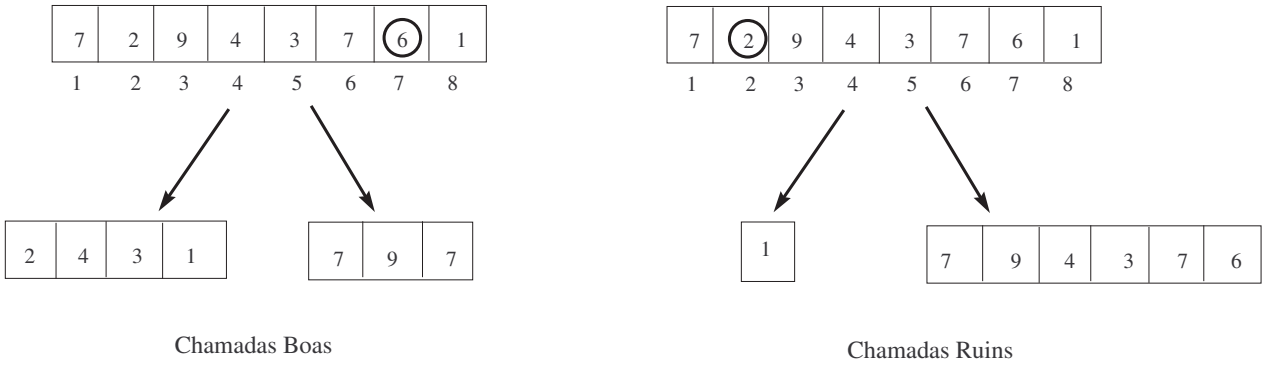


Figura 6.4: Escolhas de pivô.

Uma boa chamada tem probabilidade igual a $\frac{1}{2}$ (50 por cento), ou seja, metade dos possíveis pivôs podem levar a boas chamadas, conforme o exemplo ilustrado na Figura 6.5.

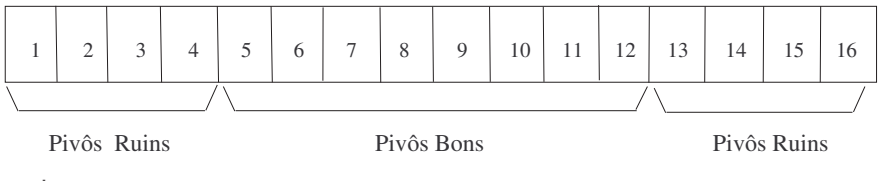


Figura 6.5: Boas probabilidades de escolha de pivô.

O Algoritmo Pick: A Estratégia da Mediana das Medianas

O primeiro algoritmo sequencial em tempo pior caso linear $\mathrm{O}(n)$, para o problema da seleção foi desenvolvido somente em 1973 por Blum *et al.* [7]. Este algoritmo, denominado *Pick*, divide inicialmente a sequência de entrada de n elementos em grupos de tamanho constante $c \geq 5$. Depois, ele calcula a mediana de cada um destes n/c grupos, determina a mediana das n/c medianas (empregando recursivamente o próprio algoritmo) e utiliza esta mediana das medianas como separador para o particionamento da sequência. Assim como no algoritmo **Find**, particionamentos sucessivos são empregados para reduzir o tamanho do problema. Entretanto agora a utilização da mediana das medianas m garante que a cada particionamento sejam eliminados pelo menos $1/4$ dos elementos analisados.

A Figura 6.6 ilustra o particionamento sucessivo do algoritmo **Pick**. Linhas verticais representam grupos de c elementos e m a mediana das medianas. O conjunto L expressa elementos menores ou iguais a m ($L \leq m$) e o conjunto G os elementos maiores ou iguais a m. A linha tracejada representa as medianas de cada conjunto c.

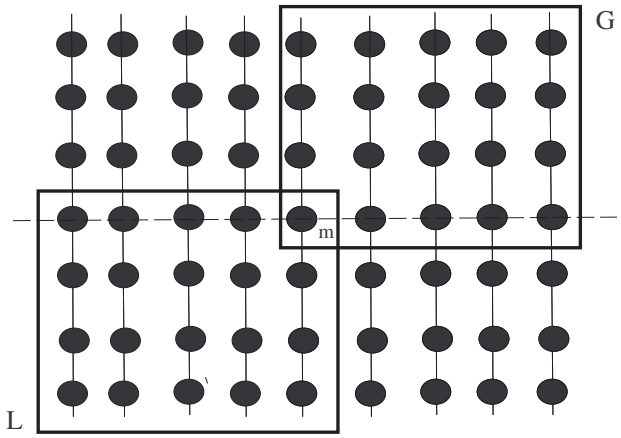


Figura 6.6: A mediana das medianas = m.

A Figura 6.7 ilustra o exemplo de uma possível retirada de elementos após uma execução do algoritmo **Pick**.

12	15	11	02	09	05	00	07	03	21	44	40	01	18	20	32	19	35	37	39
13	16	14	08	10	26	06	33	04	27	49	46	52	25	51	34	43	56	72	79
17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	83
22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91

										44	40	01	18	20	32	19	35	37	39
										49	46	52	25	51	34	43	56	72	79
									47	50	55	58	60	63	65	66	67	81	83
22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91

Figura 6.7: Configuração após a primeira execução do algoritmo *Pick*, onde elementos desnecessários são desconsiderados.

O Algoritmo Select: A Estratégia dos Dois Separadores

Em 1975, Flyod e Rivest [18] propuseram o algoritmo probabilístico (não determinístico) **Select**, assintoticamente equivalente ao algoritmo **Find**, mas exigindo um número menor de comparações. Este algoritmo trabalha com uma amostragem aleatória da sequência de n elementos e determina, com base nesta amostra, dois separadores a serem utilizados na divisão da sequência em três partições. A partir da comparação de k com o tamanho das partições obtidas, mantém-se apenas uma das partições e aplica-se o algoritmo recursivamente apenas

nesta partição. Provavelmente a partição contida entre dois separadores é mantida, ela possui um tamanho significativamente menor do que o número total de elementos analisados, o que garante um bom desempenho do algoritmo.

O Algoritmo **Select** particiona o conjunto de entrada S, utilizando dois pivôs: u e v tal que $u \leq x_k \leq v$, com alta probabilidade de x_k ser o k-ésimo menor elemento de S. Particiona-se S em elementos menores que u, iguais a u, entre u e v, iguais a v e maiores que v. O algoritmo então verifica se u ou v são iguais a x_k, ou determina um subconjunto S' de S e um inteiro k', tal que o algoritmo **Select** é chamado recursivamente para encontrar o k-ésimo menor elemento em S' [18].

6.2 Algoritmos Paralelos

Apresentamos a seguir uma cronologia de alguns dos principais algoritmos paralelos já descritos para o problema da seleção. Terminamos esta seção com uma explanação sobre os algoritmos paralelos em memória distribuída. Particularmente, o último deles é de interesse em nosso trabalho.

6.2.1 Algoritmos Determinísticos – Memória Compartilhada

Em 1983 a partir do algoritmo sequencial **Pick** (Mediana das medianas), Vishkin [41] desenvolveu um algoritmo paralelo ótimo (total de $O(n)$ operações) no modelo EREW-PRAM empregando a técnica de aceleração em cascata. A cada passo este algoritmo executa as seguintes tarefas:

1. Divide a sequência de entrada em grupos de tamanho $\log\ n$ elementos;
2. Calcula a mediana de cada grupo;
3. Determina a mediana das medianas através da ordenação do conjunto de $n/\log\ n$ medianas;
4. Determina o número de elementos menores, iguais e maiores que a mediana das medianas e compara estes números com o valor de k, o que permite eliminar $1/4$ dos elementos analisados (ou interromper a execução do algoritmo caso k já corresponda à mediana das medianas).

O processo é repetido até que o número de elementos restantes seja reduzido a $n/\log\ n$, o que permite que estes elementos sejam então ordenados em $O(n)$ operações. Como são necessários $O(\log\ \log\ n)$ passos e cada um gasta $O(\log\ n)$, este algoritmo requer um tempo total de $O(\log\ n\ \log\ \log\ n)$ [41].

Em 1988, Cole [9] adapta uma versão do algoritmo **Select** (dois separadores) para o modelo PRAM. Ele obtém um algoritmo paralelo ótimo. A cada passo do algoritmo a determinação de dois separadores (para particionar a sequência de elementos) é feita em tempo $O(\log\ n)$, e após $O(\log\ n)$ passos o número de elementos é reduzido a $n\ /\log^2\ n$. Utiliza-se então o algoritmo **MergeSort** para ordenar estes elementos. O algoritmo requer tempo $O(\log\ n\ \log\ n)$ no modelo EREW PRAM, podendo ainda ser modificado para tempo $O(\log\ n\ \log\ n\ /\log\ \log\ n)$ no modelo CRCW PRAM, em ambos os casos executando apenas $O(n)$ operações.

Em 1993, Chadhuri *et al.* [8] desenvolveram um algoritmo ótimo de tempo $O(\log\ n/\log\ \log\ n)$ para o modelo CRCW PRAM a partir do conceito de λ-seleção aproximada, que consiste em selecionar um elemento de *rank* entre $k-\lambda.n$ e $k+\lambda.n$, o que requer um trabalho consideravelmente menor do que a seleção exata do k-ésimo elemento. Utiliza-se a seleção aproximada para obter uma amostra de tamanho inferior a $\sqrt{n}$ e determinar dois separadores a partir desta amostra, reduzindo-se para $O(n/2^{\sqrt{\log\ n}})$ o número de elementos do conjunto de entrada. Por fim, aplica-se o algoritmo **MergeSort** nos elementos restantes.

Já em 1994, Dietz e Raman [41] apresentam um outro algoritmo ótimo que resolve o problema da seleção no modelo CRCW-PRAM em tempo $O(\log\ \log\ n + \log\ k\ /\log\ \log\ n)$.

6.2.2 Algoritmos Paralelos Não Determinísticos – Memória Compartilhada

Descrevemos a seguir trabalhos que descrevem algumas das soluções paralelas e não determinísticas em memória compartilhada para o problema.

O Algoritmo de Reischuk

Em relação a soluções probabilísticas, destaca-se o trabalho de Reischuk [38], que propôs em 1985 um algoritmo probabilístico para o modelo PCT com n processadores que executa em tempo esperado constante $O(1)$. Este algoritmo seleciona aleatoriamente uma amostra de tamanho $\sqrt{n}$ da sequência de n elementos e a ordena em tempo constante, utilizando-a em seguida para obter dois separadores, possivelmente próximos ao k-ésimo elemento, que são então empregados no particionamento da sequência. Com alta probabilidade, o número de elementos restantes é reduzido a $\sqrt{n}$ após a execução de um número constante destas rodadas, sendo possível comparar todos os elementos em tempo constante [38].

O Algoritmo de Gerbessiotis e Siniolakis

Em 1996, Gerbessiotis e Siniolakis [23] basearam-se no algoritmo de Reischuk e propuseram um novo algoritmo probabilístico para seleção da mediana de n elementos. Este algoritmo emprega um número constante de rodadas de comunicação. Realiza uma amostragem aleatória dos elementos. Distribui a amostra igualmente entre os processadores. Ordena apenas a amostra (utilizando um algoritmo paralelo para memória distribuída). Seleciona dois separadores na amostra e realiza um particionamento com base nestes separadores. Com alta probabilidade, a partição contida entre os dois separadores contém a mediana e possui um tamanho reduzido o suficiente para que um único processador complete a resolução do problema. Portanto, o número esperado de rodadas de comunicação é $O(1)$.

6.2.3 Algoritmos Paralelos – Memória Distribuída

Existem alguns trabalhos publicados para o problema da seleção em máquinas paralelas com memória distribuída, em sua maioria o problema é abordado do ponto de vista de estrutura de interconexão específicas. Descrevemos brevemente a cronologia de alguns destes algoritmos:

- Em 1987 Santoro *et al.* [40] desenvolveram um algoritmo probabilístico que supõe uma distribuição aleatória uniforme dos elementos entre os processadores. A cada rodada,

considera-se uma amostra aleatória, o (maior) conjunto de elementos armazenados na memória local de um dos processadores e escolhe-se, nesta amostra, um elemento que com alta probabilidade, possua um *rank* próximo a k. Este elemento é então propagado aos demais processadores para a realização do particionamento, eliminando-se uma das partições e corrigindo-se o valor de k em relação aos elementos restantes. Com alta probabilidade após $O(\log\ \log\ k)$ rodadas o valor corrigido de k será reduzido a $O(p)$, o que permite que cada processador elimine todos exceto seus k menores (ou maiores) elementos. Realiza-se então uma nova série de rodadas de particionamento a partir de elementos selecionados aleatoriamente, resolvendo-se o problema após mais $O(\log\ p)$ rodadas. Este algoritmo requer um total de $O(\log\ \log\ k + \log\ p)$ rodadas de comunicação [41];

- Em 1996, Bader e JáJá [41] apresentaram um novo algoritmo determinístico para computadores paralelos com memória distribuída, desenvolvido a partir de algoritmos paralelos e de memória compartilhada anteriores. A cada rodada, realiza-se inicialmente uma etapa de comunicação para redistribuir igualmente, entre os processadores, o conjunto de elementos a serem analisados. Em seguida, cada processador calcula a mediana dos elementos armazenados em sua memória local, determina-se a mediana dessas medianas, particiona-se o conjunto de elementos com base nesta mediana e elimina-se pelo menos $1/4$ destes elementos. Após $O(\log(n\ /p^2))$ rodadas de comunicação, o número de elementos é reduzido de n para p^2 e os elementos restantes são então reunidos em um único processador, que aplica um algoritmo sequencial para completar a resolução do problema;

- Em 1997, Saukas e Song [41] apresentaram um novo algoritmo determinístico e paralelo para solucionar o problema da seleção.

6.3 A Seleção de Medianas

Inicialmente nos dedicamos ao estudo de um caso particular do problema geral de seleção: a seleção de medianas. Para este problema, o nosso trabalho seguiu duas abordagens:

1. A primeira utiliza a teoria sobre sequências bitônicas e *splits* bitônicos;

2. A segunda é baseada em um trabalho recente [21], que descreve soluções sequenciais para o problema utilizando uma estrutura de dados específica: árvores idênticas e diametralmente opostas.

6.3.1 A Seleção de Mediana por Estratégia Bitônica

Nossa primeira solução para o problema da seleção de mediana utiliza, em parte, a mesma estratégia do algoritmo paralelo de ordenação bitônica. Este último é baseado na ideia de unir pares de subsequências, com fases intermediárias envolvendo ordenações locais. A entrada do algoritmo de ordenação bitônica é uma sequência de n números divididos entre p processadores, onde n e p são potências de dois e $n/p \geq p$. seguir apresentamos algumas definições necessárias para o entendimento do problema:

- Uma sequência de números (a_1,a_2, ..., a_n) é dita **bitônica**, se existe um inteiro $1 \leq j \leq n$, tal que $a_1 \leq a_2 \leq \cdots \leq a_j \geq a_{j+1} \geq \ldots a_n$. Uma sequência de números (a_1,a_2,

$\ldots$, a_n) também é denominada bitônica se pudermos deslocá-la ciclicamente, tal que a sequência resultante S seja bitônica, ou seja, existe um número inteiro $1 \leq k \leq n$, tal que $S_{(1)} \leq S_{(2)} \leq \cdots \leq S_{(k)} \geq S_{(k+1)} \geq \cdots \geq S_{(n)}$ [24]. Por exemplo, a sequência $(1, 3, 5, 6, 7, 4, 2)$ é bitônica com $k = 5$. A sequência $(9, 6, 3, 2, 5, 7, 10)$ é bitônica, pois a sequência $(2, 5, 7, 10, 9, 6, 3)$ é bitônica com k=4;

- Seja $S = (a_1, a_2, \ldots, a_{2n})$ uma sequência bitônica. Podemos obter duas sequências bitônicas aplicando a operação de divisão bitônica **(*split* bitônico)** da seguinte forma:

  - $S_{min} = (\min\{a_1, a_{n+1}\}, \ldots, \min\{a_n, a_{2n}\})$;
  - $S_{max} = (\max\{a_1, a_{n+1}\}, \ldots, \max\{a_n, a_{2n}\})$.

  Além disso, temos que $\max(S_{min}) \leq \min(S_{max})$, ou seja, todos os elementos de S_{min} são menores ou iguais aos elementos de S_{max} [24]. Por exemplo, dada a sequência $S = (2, 3, 6, 7, 8, 5, 4, 1)$, obtemos as seguintes sequências bitônicas: $S_{min} = (2, 3, 4, 1)$ e $S_{max} = (8, 5, 6, 7)$.

Em cada rodada do algoritmo paralelo de ordenação bitônica, cada processador possui exatamente n/p elementos da sequência global. A estratégia do algoritmo é baseada na operação de divisões bitônicas sucessivas e ordenações locais, até que toda sequência esteja ordenada. É importante observar que apesar de utilizar operações de divisão bitônica, a entrada deste algoritmo não precisa ser necessariamente uma sequência bitônica, pois o mesmo transforma a sequência de entrada em uma sequência bitônica [24]. Descreve-se que o algoritmo ordena corretamente n inteiros armazenados em uma máquina CGM de p processadores, com n/p inteiros por processador, utilizando $O(\log p)$ rodadas de comunicação e tempo de computação local $O((n \times \log n)/p)$ [24]. O pseudocódigo deste algoritmo é detalhado e muito bem descrito [24]. A partir deste algoritmo, implementamos particularmente uma versão para seleção de mediana em GPU. Porém, em nossa versão não é necessária a ordenação total da sequência original, visto que buscamos apenas o valor da mediana. De forma geral, o nosso algoritmo segues os passos descritos no Algoritmo 19.

Algoritmo 19: Algoritmo BSP/CGM para a seleção de mediana - estratégia Bitônica.

Entrada: (1) Um conjunto de P processadores; (2) O número i que rotula cada processador $p_i \in P$, onde $1 \leq i \leq P$; (3) Uma sequência Q de n inteiros; (4) três valores inteiros: m, $max\_direita$ e $max\_esquerda$.
Saída: (1) O valor m correspondente a mediana.

1: **Em paralelo** aplique o algoritmo de construção bitônica [24] sobre a sequência Q, utilizando o conjunto de processadores P, de forma que ao final a metade esquerda de Q ($Q[1, \ldots, n/2]$) esteja em ordem crescente e a metade direita de Q ($Q[(n/2)+1, \ldots, n]$) esteja em ordem decrescente.
2: **Em paralelo** aplique um passo de *split* bitônico sobre a sequência Q [24], utilizando o conjunto de processadores P, de forma que ao final os menores elementos de Q estejam na metade esquerda ($Q[1, \ldots, n/2]$) e os maiores elementos de Q estejam na metade direita ($Q[(n/2)+1, \ldots, n]$) .
3: $max\_esquerda \leftarrow$ **Em paralelo** aplique em $Q[1, \ldots, n/2]$, utilizando o conjunto de processadores P, um algoritmo de redução para máximo.
4: $max\_direita \leftarrow$ **Em paralelo** aplique em $Q[(n/2)+1, \ldots, n]$, utilizando o conjunto de processadores P, um algoritmo de redução para mínimo.
5: $m \leftarrow$ **max**(max_esquerda,max_direita).
6: Retorne m.

A Figura 6.8 ilustra os passos de nossa implementação.

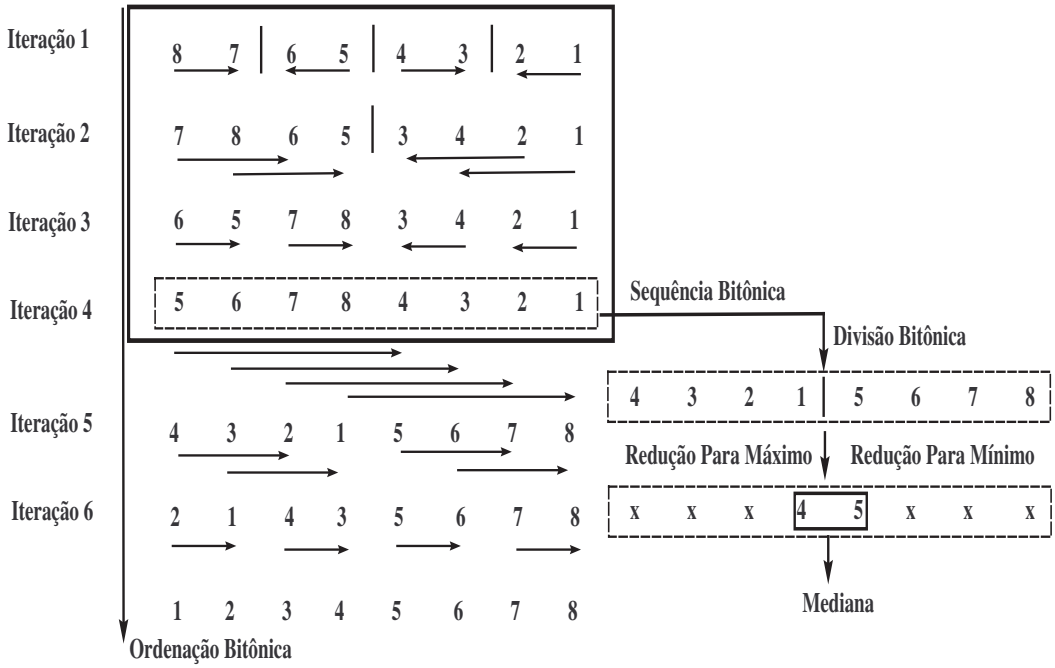


Figura 6.8: Ordenação bitônica e seleção de mediana.

Implementações e Resultados

A Figura 6.9 e a Tabela 6.1 ilustram os resultados obtidos por nossa implementação em GPU, para o problema de seleção de mediana utilizando estratégia bitônica. Comparativamente, com a melhor versão sequencial, nossa implementação apresentou desempenho até sete vezes melhor.

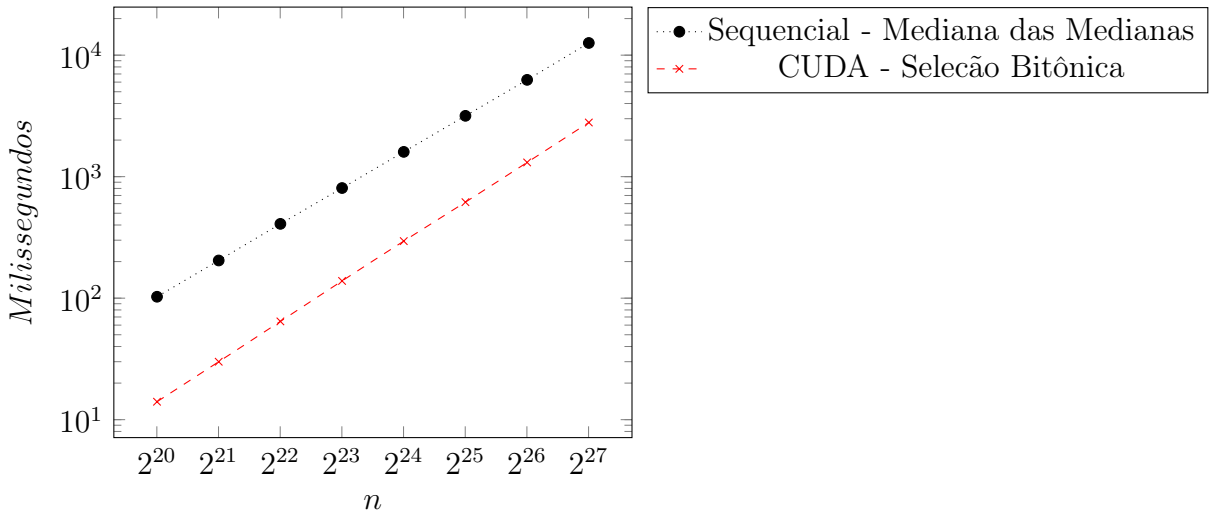


Figura 6.9: Sequencial (Mediana das Medianas) × CUDA Seleção Bitônica.

n milhões	Sequencial	CUDA - Seleção Bitônica	*Speedup*
1,048.576	102.7200	14.0493792	7.3113550811
2,097.152	204.25581	29.9821984	6.8125694879
4,194.304	408.78598	64.383892	6.3491964729
8,388.608	806.08101	138.6211808	5.8149916582
16,777.216	1.599.39032	294.8401366	5.4246017467
33,554.432	3.166.95723	616.9543702	5.1332114383
67,108.864	6.261.39448	1.309.2481813	4.7824351177
134,217.728	12.597.26318	2.790.826245	4.5138113498

Tabela 6.1: Tempos de Execução (**Milisegundos**): Sequencial. × CUDA - Seleção Bitônica.

6.3.2 A Seleção de Mediana por Árvores Opostas

Nossa próxima implementação para o problema de seleção de medianas é baseada em um algoritmo apresentado em 2012 por Garrigues e Manzanera [21]. Trata-se de um um novo algoritmo de granulosidade fina, que insere corretamente a mediana no índice central de um vetor de entrada de tamanho n. O algoritmo utiliza p processadores, sendo que sua versão aleatória converge em um tempo médio de $\mathrm{O}\big((n/p) \times (\log n + \mu)\big)$, onde $0 < p < (n/4)$ e μ representa o tempo para trocar um par de elementos entre dois nós de processamento. A cada iteração, os nós processam apenas os seus elementos locais e se necessário for, realizam a troca de uma parte deles com outro processador de rede. Os autores descrevem que o algoritmo pode ser executado em ambientes *manycore* com resultados promissores. Foi a partir desta constatação, que estudamos o trabalho de Garrigues e Manzanera, movidos pela curiosidade de mensurar o tempo do algoritmo probabilístico em uma GPU.

Para a compreeensão do algoritmo é preciso apresentar notações e alguns algoritmos preliminares descritos no trabalho original [21]. O trabalho apresenta três algoritmos para seleção de mediana em um vetor de tamanho n. Em cada iteração, cada algoritmo faz uso de memórias distribuídas em redes de computação paralela, realizando troca de elementos entre os processadores. São algoritmos de granulosidade fina. O vetor de entrada dos algoritmos é mapeado para duas ávores diametralmente opostas, como ilustrado pela Figura 6.10 [21].

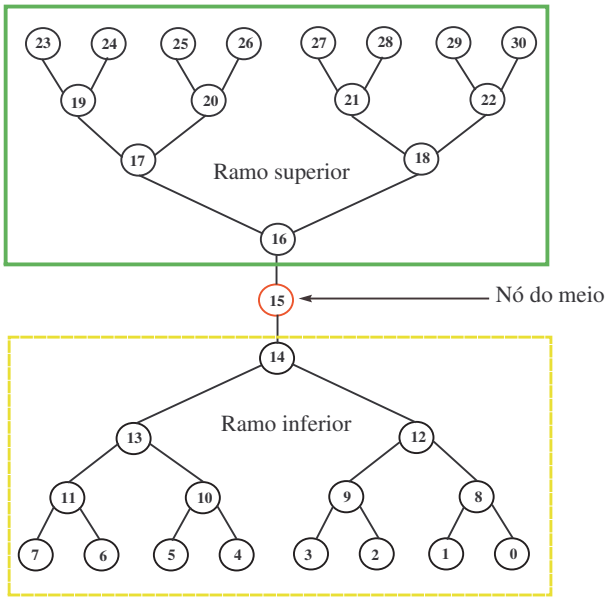


Figura 6.10: Mapeamento de um vetor de entrada A para as árvores. Elementos maiores acima e menores abaixo da mediana.

Os algoritmos invocam rotinas de troca de elementos em diferentes momentos de execução.

São elas:

1. Rotina $reorder(A, i, j)$: reordena o i-ésimo e o j-ésimo elemento de A tal que $A[i] \leq A[j]$;
2. Rotina $select_{min}(A, i, j, k)$: realiza uma troca entre $A[i]$ (raiz) e $min(A[j], A[k])$. Rotina invocada por cada subárvore do ramo superior;
3. Rotina $select_{max}(A, i, j, k)$: realiza uma troca entre $A[i]$ (raiz) e $max(A[j], A[k])$. Rotina invocada por cada subárvore do ramo inferior.

A Figura 6.11 ilustra um exemplo com entrada e saída para cada uma das três rotinas.

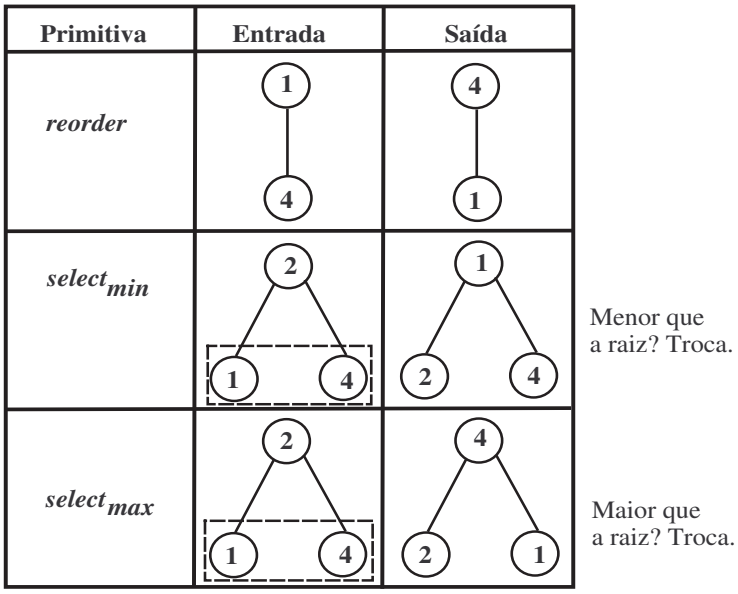


Figura 6.11: As três rotinas e as operações condicionais de troca de elementos.

Os Três Algoritmos

Discutimos a seguir a estratégia de cada um dos três algoritmos presentes no trabalho original:

Algoritmo 1 – *(Sand Clock)*: este algoritmo mapeia o vetor de entrada A, de modo que existam duas subárvores. Uma subárvore à esquerda (ramo inferior) e outra à direita (ramo superior) com a mediana no meio (raiz). O tamanho de A é dado por $n = 2^k - 1, k > 1$. Por exemplo, para $k = 5$, $n = 31$. Cada nó é rotulado com sua posição no *array*. Em cada subárvore binária as operações de $select_{min}$ e $select_{max}$ são executadas. A cada iteração, o algoritmo atualiza o vetor de entrada A utilizando as operações:

- $select_{min}$ = Aplicada no ramo superior;
- $select_{max}$ = Aplicada no ramo inferior;
- reorder = Aplicada na raiz que é o nó do meio.

Desta forma, a cada cada iteração uma troca do maior elemento do ramo inferior com o menor elemento do ramo superior é realizada. O pseudocódigo de *Sand Clock* é ilustrado pelo Algoritmo 20.

Algoritmo 20: Algoritmo sequencial de seleção de mediana: Sandy Clock.

Entrada: (1) Sequência A com n inteiros.
Saída: (1) Valor mediana na posição central.

```
1: middle = (n − 1)/2;
2: repeat
3:    for i = ((middle − 1)/2) to 1 do
4:       select_max(A, middle−i, middle−2i, middle−2i − 1);
5:       select_min(A, middle+i, middle+2i, middle+2i + 1);
6:    end for
7:    eff = reorder(A, middle−1, middle, middle+1);
8: until not eff
```

Considerações do Algoritmo 20:

- **Baixa eficiência**: todos os elementos que estão na metade incorreta da árvore ($n/2$ em cada metade) deverão passar pelo meio (raiz) ;
- **Gargalo**: apenas um elemento pode viajar do ramo superior para o inferior em cada iteração.

O próximo algoritmo busca amenizar este gargalo.

Algoritmo 2 – *Troca Entre Ramos*: Este algoritmo melhora a conectividade entre os ramos inferior e superior, fornecendo atalhos para viajar de um ramo para o outro. As instruções do Algoritmo *Sand Clock* ainda são mantidas, entretanto, agora, a cada iteração um nó do ramo inferior é reordenado com o seu correspondente no ramo superior, como ilustrado pela Figura 6.12.

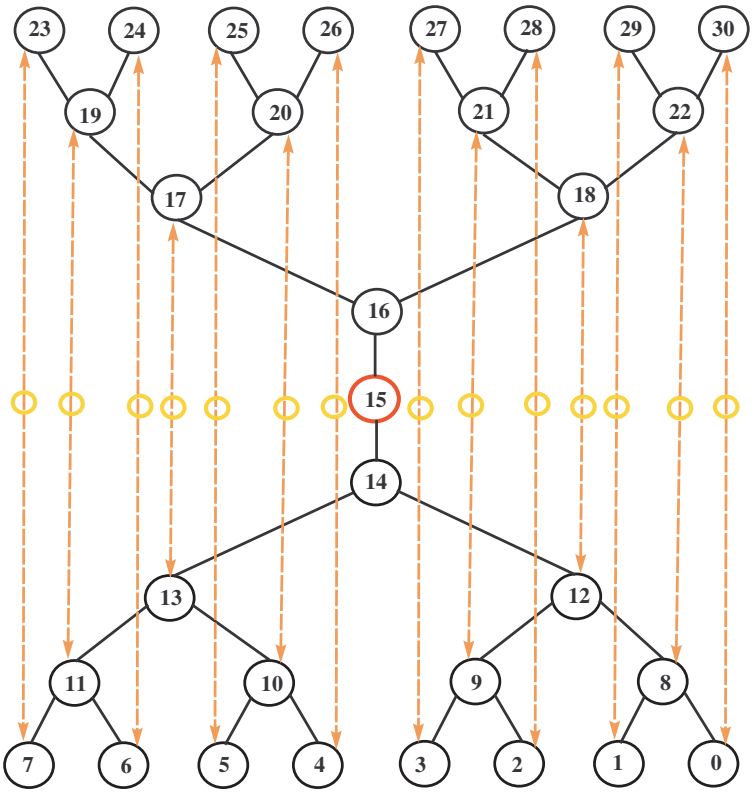


Figura 6.12: *Reorder*: a operação de troca entre ramos.

O pseudocódigo do algoritmo que realiza troca entre ramos é ilustrado pelo Algoritmo 21.

Algoritmo 21: Algoritmo sequencial de seleção de mediana: Troca entre Ramos

Entrada: (1) Sequência A com n inteiros.
Saída: (1) Valor mediana na posição central.

```
1:  middle = (n-1)/2;
2:  repeat
3:     for  i = 0 to middle-1  do
4:        reorder(A,i,n − i − 1)
5:     end for
6:     for  i = ((middle - 1)/2) to 1  do
7:        select_max(A, middle−i, middle−2i, middle−2i − 1); {Operações no ramo inferior}
8:        select_min(A, middle+i, middle+2i, middle+2i + 1); {Operações no ramo superior}
9:     end for
10:    eff = reorder(A, middle−1, middle, middle+1); {Operações na raiz (nó do meio)}
11: until not eff
```

Considerações do Algoritmo 21:

- Uma operação de *reorder(n,m)* é executada entre cada nó n do ramo inferior e m do ramo superior. Esta operação diminui o gargalo de troca de um elemento por vez;
- As demais instruções são idênticas aquelas do Algoritmo 20.

Algoritmo 3 – Probabilístico: Este algoritmo diminui ainda mais o gargalo existente. Em média agora a solução é encontrada em $log_2(n)$ iterações. O algoritmo realiza um embaralhamento horizontal e contínuo, que troca as raízes de cada subárvore de um ramo. Um exemplo deste movimento de troca é ilustrado pela Figura 6.13.

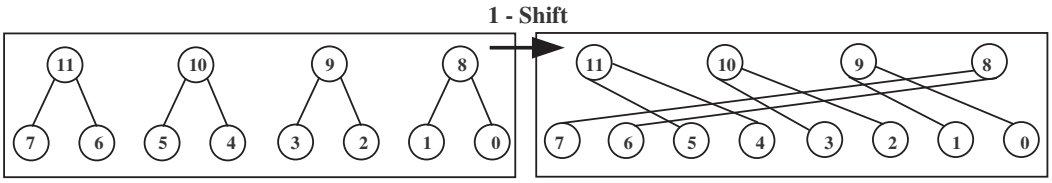


Figura 6.13: Movimento diagonal de troca de raízes.

Garrigues e Manzanera descrevem que independente do tamanho de entrada, 10 passos de *pipeline* são suficientes para colocar 99 % dos elementos no lugar, em média [21]. Se o número retornado pela funções aleatórias de *offset* for zero, o algoritmo probabilístico é equivalente ao algoritmo anterior (Algoritmo 20) e neste caso, o número de iterações no pior caso será igual a $n/4$. De forma geral, em média, o algoritmo probabilístico leva $2 \times log_2(n)$ passos de pipeline para convergir. Utilizando p processadores, um passo de *pipeline* executa em tempo $\mathrm{O}\left(\frac{n}{p}\right)$. Assim, em média, a complexidade de tempo para convergência é $\mathrm{O}\left(\log n \times \frac{n}{p}\right)$ [21].

Neste trabalho implementamos a versão probabilística do algoritmo de Garrigues e Manzanera. O pseudocódigo é representado pelo Algoritmo 22. Realizamos uma implementação em memória compartilhada de uma GPU utilizando milhares de *threads*, contudo é possível em memória distribuída atribuir n/p valores de A para cada cada processador p_i.

Algoritmo 22: Algoritmo probabilístico de seleção de mediana: Troca entre Ramos.

Entrada: (1) Um conjunto de P processadores; (2) O número i que rotula cada processador $p_i \in P$, onde $1 \leq i \leq P$; (3) Uma sequência A de n inteiros.
Saída: (1) Valor mediana na posição central.

1: Rotule e atribua para processador $p_i \in P$ o valor i correspondente da sequência A de n inteiros.
2: $middle = (n-1)/2$;
3: **repeat**
4: $eff = false$
5: $Lsize = (n+1)/4$
6: $Plow = Lsize\text{-}1$
7: $Pupp = n$-Lsize
8: $Olow_$son = RANDOM($Lsize$)
9: $Oupp_$son = RANDOM($Lsize$)
10: **repeat**
11: $Olow_$dad = RANDOM($Lsize/2$)
12: $Oupp_$dad = RANDOM($Lsize/2$)
13: **for** $k = 0$ **to** (($Lsize/2$)-1) **do**
14: Ilowz_dad = $Plow$ + $Lsize/2$ - MOD(k + Olow_dad, $Lsize/2$)
15: Iupp_dad = $Pupp$ - $Lsize/2$ + MOD(k + Oupp_dad, $Lsize/2$)
16: Ilow_child1 = $Plow$ - MOD(2k + Olow_son, $Lsize$)
17: Ilow_child2 = $Plow$ - MOD(2k + 1 + Olow_son, $Lsize$)
18: $Iupp_$child1 = $Pupp$ + MOD(2k + Oupp_son, $Lsize$)
19: $Iupp_$child2 = $Pupp$ + MOD(2k + 1 + Oupp_son, $Lsize$)
20: $select_{max}$(A, p**[Ilow_dad]**, p**[Ilow_child1]**, p**[Ilow_child2]**)
21: $select_{min}$(A, p**[Iupp_dad]**, p**[Iupp_child1]**, p**[Iupp_child2]**)
22: $eff = eff$ || $reorder$(A, p**[Ilow_dad]**, p**[Iupp_dad]**)
end for
$Plow = Plow + Lsize/2$
$Pupp = Pupp$ - $Lsize/2$
$Olow_$son = $Olow_$dad
$Oupp_$son = $Oupp_$dad
$Lsize = Lsize/2$
until $Lsize == 1$
reorder(A, p**[*middle* - 1]**, p**[*middle*, *middle* + 1]**)
until not eff

Estudo Probabilístico dos Algoritmos

Discutimos a seguir as configurações probabilísticas para cada uma das operação realizadas pelos três algoritmos. O estudo probabilístico e de convergência dos algoritmos podem ser conferidos em detalhes no trabaho original [21]. Vejamos inicialmente as configurações para as operações $select_{max}$ e $select_{min}$, aqui representadas pela Figura 6.14:

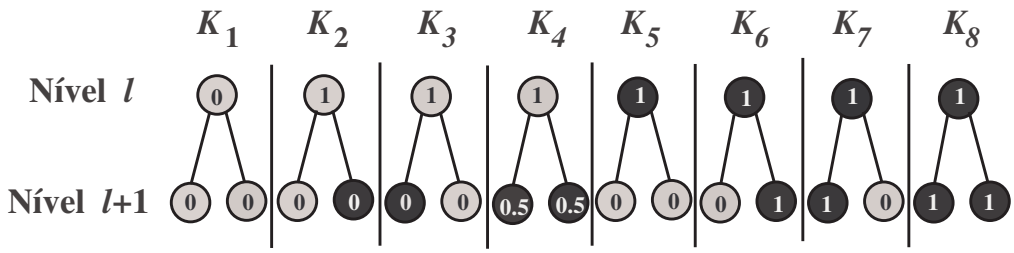
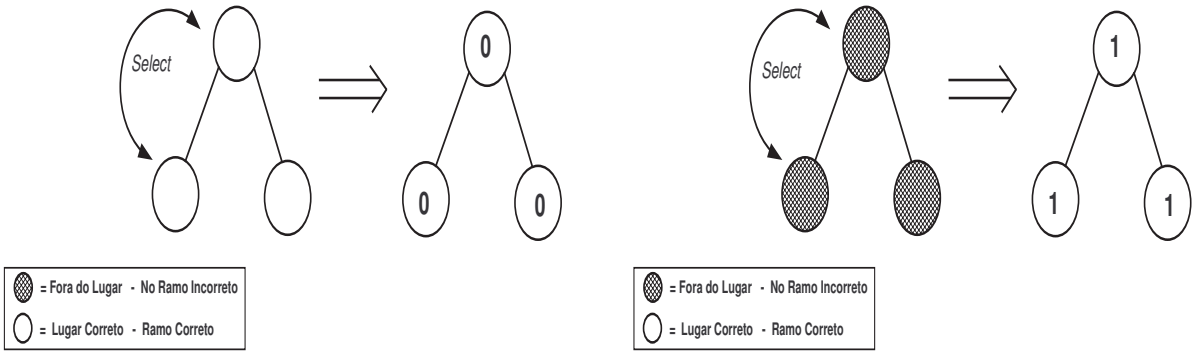


Figura 6.14: As oito configurações possíveis para as operações *select*.

- K_1: se todos os nós já estiverem no ramo correto, após *select*, todos continuarão no ramo correto, com probabilidade de 0% de estarem fora do lugar, como ilustrado na Figura 6.15 (a);
- K_8: se todos os nós estiverem fora do lugar, após *select*, todos continuarão no ramo incorreto, com probabilidade de 100%, como ilustrado na Figura 6.15 (b).

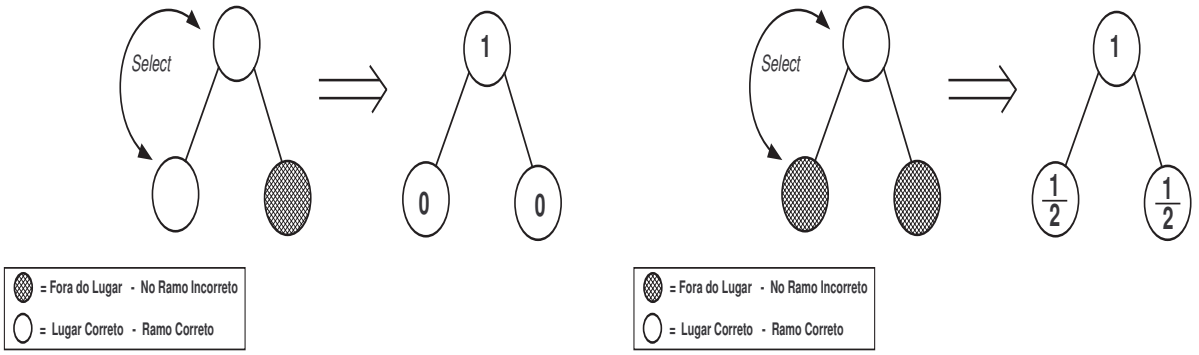


(a) Configuração K_1.

(b) Configuração K_8.

Figura 6.15: Configurações K_1 e K_8.

- K_2 e K_3: Se apenas um dos filhos estiver fora do lugar, então, após **select**, o filho incorreto troca de lugar com o pai, o que leva a configuração de pai com probabilidade de 100% fora do lugar (1) e dois filhos com probabilidade de 0% de estarem fora do lugar (0,0), como ilustrado na Figura 6.16 (a);
- K_4: se o pai estiver no lugar correto, com dois filhos fora do lugar, então um dos filhos deve subir (descer). Os filhos têm mesma probabilidade de subir 50%, então, após **select** temos que o pai está fora do lugar (100% de probabilidade), um dos filhos está no lugar e o outro fora do lugar incorreto, mas qual? Ambos têm 50% de chances (1/2), como ilustrado na Figura 6.16 (b).



(a) Configuração K_2. K_3 é similar.

(b) Configuração K_4.

Figura 6.16: Configurações K_2, K_3 e K_4.

- Se o pai estiver fora do lugar, com dois filhos no lugar certo, a operação *select* não altera a configuração, assim temos: pai ainda fora do lugar com probabilidade de 100% (1) e filhos com probabilidades de 0% de estarem fora do lugar, como é ilustrado pela Figura 6.17.

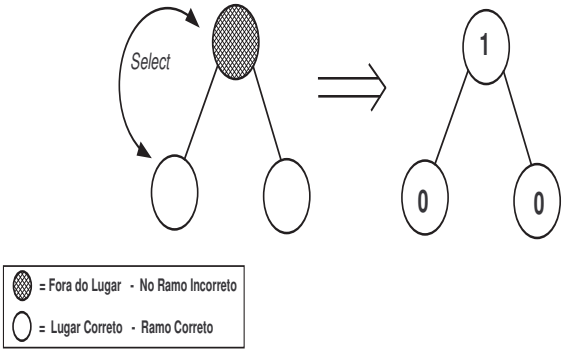


Figura 6.17: Configuração K_5.

- Se o pai estiver fora do lugar, com um dos dois filhos no lugar errado, a operação *select* não altera a configuração, assim temos: pai ainda fora do lugar com probabilidade de 100% (1) e um dos filhos com probabilidades de 100% (1) de estar fora do lugar. Este é o caso das configurações K_6 e K_7 ilustradas pela Figura 6.14.

Vejamos agora as configurações para a operação *reorder*. Trata-se de uma operação global que faz troca entre ramos. Os nós fora do lugar, após uma operação *reorder*, seguem para o ramo certo (sobem ou descem). As configurações possíveis são ilustradas pela Figura 6.18.

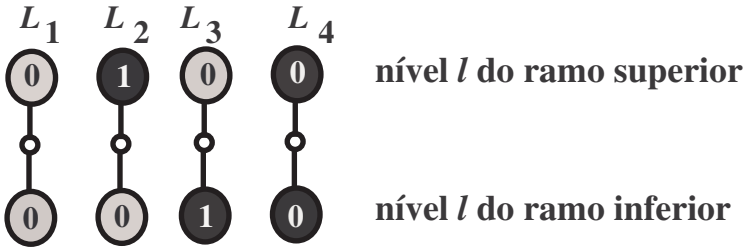
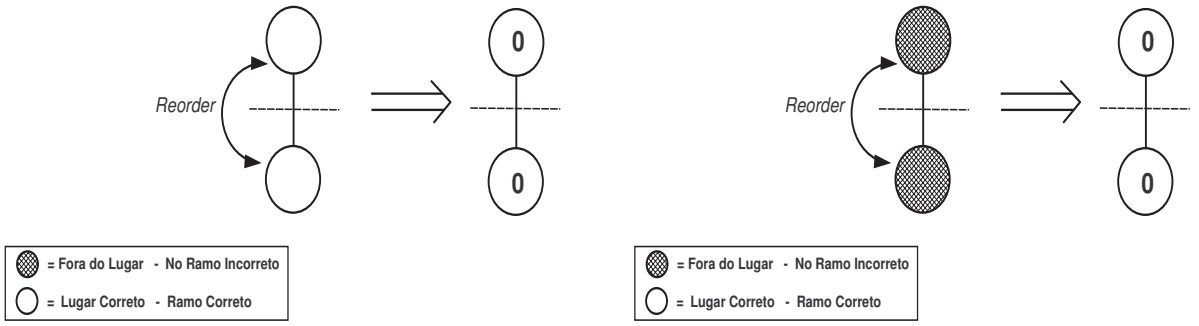


Figura 6.18: As quatro configurações para reorder.

- L_1: se os nós já estiverem no lugar certo, então *reorder* não efetua troca e após a operação, ambos permanecem com 0% de probabilidade de estarem fora do lugar, como é ilustrado pela Figura 6.19 (a);
- L_4: se ambos os nós estiverem em ramos incorretos, então, após *reorder* eles estarão no lugar certo, com 0% de probabilidade de estarem fora do lugar, como é ilustrado pela Figura 6.19 (b).

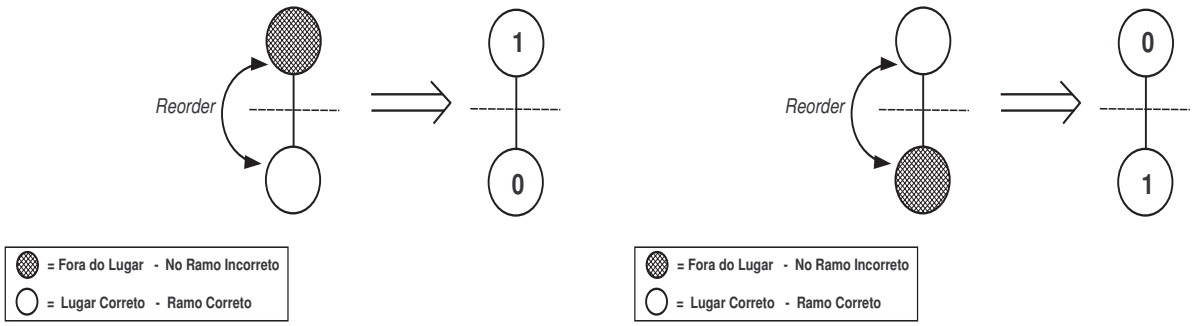


(a) Configuração L_1.

(b) Configuração L_4.

Figura 6.19: Configuração L_1 e L_4.

- L_2: se o nó superior estiver fora do lugar e o inferior no lugar, após *reorder*, o nó inferior estará no lugar certo, com 0% de probabilidade de estar fora do lugar. Por sua vez, o nó superior estará fora do lugar, com probabilidade de 100%, como é ilustrado pela Figura 6.20 (a);
- L_3: se o nó superior estiver no lugar certo e o inferior fora do lugar, após *reorder*, o nó inferior estará fora do lugar, com 100% de probabilidade de estar fora do lugar. Por sua vez, o nó superior estará no lugar certo, com probabilidade de 0% de estar fora do lugar, como é ilustrado pela Figura 6.20 (b).



(a) Configuração L_2.

(b) Configuração L_3.

Figura 6.20: Configuração L_2 e L_3.

Discussão de Resultados

Implementamos em GPU a versão probabilística do algoritmo de troca entre ramos (Algoritmo 22) do trabalho de Garrigues e Manzanera [21]. A implementação foi realizada para executar em memória global seguindo os passos do Algoritmo 22. Os resultados obtidos são representados pela Figura 6.21 e pela Tabela 6.2. Para comparação utilizamos o algoritmo sequencial mediana das medianas.

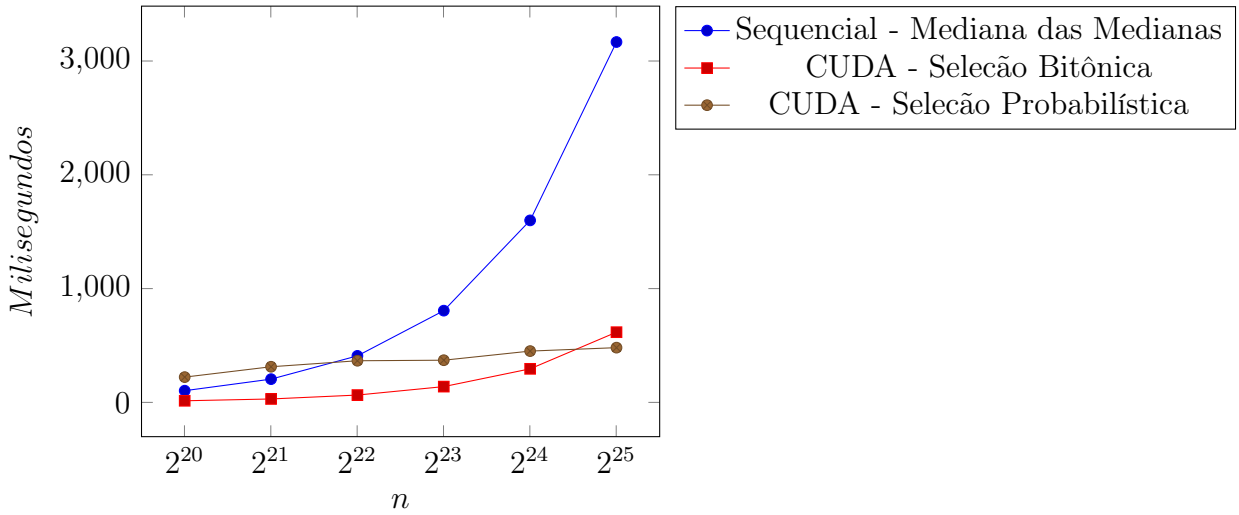


Figura 6.21: Sequencial (Mediana das Medianas) × CUDA Seleção Bitônica × CUDA Seleção Probalilística.

n milhões	Sequencial	Seleção Bitônica	Seleção Probabilística (Algoritmo 22)
1,048.576	102.7200	14.04937	222.0550
2,097.152	204.25581	29.98219	313.1284
4,194.304	408.78598	64.38389	365.8917
8,388.608	806.08101	138.6211	370.6086
16,777.216	1.599.39032	294.8401	450.7737
33,554.432	3.166.95723	616.9543	480.9904

Tabela 6.2: Tempos de Execução (**Milisegundos**): Sequencial. × CUDA - Seleção Bitônica × CUDA - Seleção Probabilística.

Os testes indicam que inicialmente o algoritmo probabilístico consumiu mais tempo, entretanto para entradas maiores a curva de tempo deste algoritmo cresceu de maneira menos acentuada que os demais. Como visto, o algoritmo probabilístico para a entrada de tamanho 33.554.432 apresentou um tempo de execução menor do que a versão sequencial e menor do que a versão por estratégia bitônica.

Correções e Complexidades

As correções dos algoritmos são provenientes dos trabalhos de referência [24, 21, 13]. Nosso algoritmo BSP/CGM de seleção bitônica executa metade dos passos do algoritmo BSP/CGM de ordenação bitônica [24] e em seguida executa já conhecidos algoritmos BSP/CGM para redução [13]. A correção do algoritmo de seleção por meio de árvores opostas pode ser vista em [21]. A complexidade da implementação BSP/CGM do algoritmo de seleção bitônica resulta da análise dos dois passos principais: 1) A ordenação bitônica é executada em tempo $O\big((n \times \log n)/p\big)$ [24], já nosso algoritmo executa metade dos passos, o que resulta em complexidade de tempo $\frac{O\left((n\times\log n)/p\right)}{2}$. 2) Em seguida, as reduções para máximo e para mínimo executam utilizando p processadores e em tempo $O\big(n/p\big)$ [13]. Por fim, a complexidade final do algoritmo é $\frac{O\left((n\times\log n)/p\right)}{2} + O\big(n/p\big)$. A complexidade da implementação BSP/CGM do algoritmo de seleção por árvores opostas é $O\big(\log n \times \frac{n}{p}\big)$, como é descrito em [21].

6.4 O Algoritmo BSP/CGM para Seleção de Saukas e Song

Os algoritmos para o problema geral de seleção em sua maioria baseiam-se na aplicação sucessiva de uma estratégia de particionamento. Algumas estratégias de particionamento foram discutidas:

- Uma delas (*Pick*) escolhe inicialmente um elemento M do conjunto A de tamanho n, particionando este conjunto nos subconjuntos A_l, A_e e A_g, respectivamente, com os elementos menores, iguais e maiores que M, conforme é ilustrado pela Figura 6.22. Em seguida, compara-se o valor de k como os tamanhos dos três subconjuntos. Se k corresponde à primeira partição A_l (ou seja, $k \leqslant \mid A_l \mid$), então o algoritmo de seleção prossegue apenas com o subconjunto A_l e descarta os demais elementos, reduzindo assim o tamanho do problema. Se k corresponde à partição A_e (ou seja, $\mid A_l \mid < k \leq \mid A_l \mid + \mid A_e \mid$), então M é a solução do problema. Se k corresponde à última partição (ou seja $k > A_l + A_e$) então o algoritmo de seleção prossegue com o subconjunto A_g para selecionar o elemento $k - (\mid A_l \mid + \mid A_e \mid)$, novamente reduzindo o tamanho do problema;

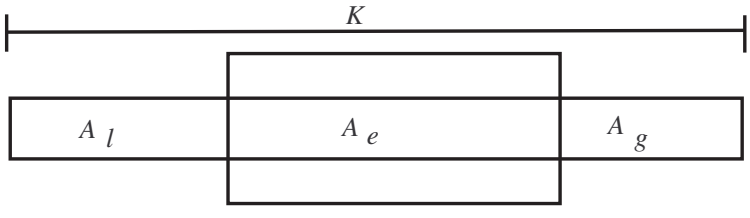


Figura 6.22: As partições do algoritmo *Find*.

- Em outra abordagem, os elmentos são repetidamente agrupados em blocos e utiliza-se uma mediana, escolhida entre as medianas de cada bloco, para realizar o particionamento, de forma a reduzir o número total de elementos a serem analisados em pelo menos 1/4 a cada rodada.

Utilizando o modelo CGM, Saukas e Song apresentaram um novo algoritmo determinístico e paralelo para solucionar o problema da seleção [41, 42]. Este novo algoritmo é baseado no sucessivo particionamento dos dados de entrada, para reduzir o tamanho da entrada de $O(n)$ para $O(\frac{n}{p})$. Por sua vez, os elementos divididos podem ser processados sequencialmente em tempo linear $O(\frac{n}{p})$ em processadores únicos. Em cada rodada, os elementos são divididos em blocos e medianas de cada bloco são obtidas.

O algoritmo destina-se a máquinas paralelas, com memória distribuída, de granulosidade grossa de modelo $CGM(n,p)$, portanto com p processadores, cada um com memória local $O(\frac{n}{p})$, interligados por uma rede arbitrária de comununicação. A granulosidade grossa indica que o tamanho n do problema é significativamente maior que o número p de processadores. Nestes algoritmos, considera-se que o conjunto de entrada A de n elementos encontra-se inicialmente distribuído entre p processadores, com cada processador armazenando em sua memória local $O(\frac{n}{p})$ elementos.

O algoritmo foi desenvolvido exclusivamente a partir dos algoritmos sequenciais e paralelos com memória compartilhada apresentados anteriormente (principalmente com base no algoritmo proposto por Vishkin em 1983 para o modelo PRAM). Para reduzir o número de rodadas de comunicação necessárias, os particionamentos sucessivos são repetidos apenas até que o número total de elementos a serem analisados seja reduzido de $O(n)$ para $O(\frac{n}{p})$, o que é obtido após $O(\log\ p)$ rodadas. Estes elementos restantes são então reunidos em um único processador, que resolve o problema sequencialmente em tempo $O(\frac{n}{p})$. Deve-se ressaltar que $\Omega(n/p)$ é claramente um limite inferior para o algoritmo paralelo, uma vez que o algoritmo sequencial requer tempo linear $\Theta(n)$ [41, 42].

Para minimizar a comunicação entre os processadores, em cada rodada de computação, o algoritmo assume que cada bloco é composto apenas pelo conjunto de elementos já armazenados em cada processador. Embora o modelo CGM enfoque apenas o número de rodadas, considerou-se necessário também reduzir a comunicação entre os processadores a cada rodada do algoritmo, pois em máquinas paralelas com memória distribuída, o custo de operações de comunicação é geralmente bem maior do que de processamento local [41, 42].

Para garantir a eliminação de pelo menos 1/4 dos elementos a cada rodada, o algoritmo poderia utilizar a estratégia da mediana das medianas de cada bloco como base para o particionamento, o que exigiria que estes blocos tivessem o mesmo tamanho em cada rodada. Entretanto, após a primeira rodada permanece, em geral, um número diferente de elementos restantes em cada processador, sendo que a redistribuição destes elementos para garantir novamente uma mesma quantidade de elementos em cada processador implicaria em um custo muito elevado em termos de comunicação. Para evitar a necessidade de redistribuição, a solução encontrada é que a mediana escolhida para o particionamento não seja simplesmente a mediana das medianas, mas sim a mediana ponderada das medianas, de forma que nesta escolha sejam considerados não só os valores das medianas dos blocos, mas também o número de elementos restantes em cada bloco. Isto permite que o algoritmo trabalhe nas rodadas seguintes com um número diferente de elementos em cada processador e mesmo assim atinja o objetivo de garantir a redução em pelo menos 1/4 do número total de elementos restantes a cada rodada [41, 42].

Ao evitar a redistribuição dos elementos a cada rodada, o algoritmo obtém uma redução considerável nos tempos de comunicação, suficiente para compensar o tempo adicional gasto com processamento local devido ao desbalanceamento de carga entre processadores.

A definição geral de **mediana ponderada** é a seguinte: dados p elementos (distintos) $m_1, m_2, \ldots, m_p$ com pesos positivos $w_1, w_2, \ldots, w_p$ tais que $\sum_{i=1}^{p} w_i = 1$, a mediana ponderada é o elemento m_k que satisfaz $\sum w_i \leq \frac{1}{2}$ $(m_i < m_k)$ e $\sum w_i \leq \frac{1}{2}$ $(m_i > m_k)$.

Embora a definição anterior refira-se apenas a elementos distintos, esta pode ser aplicada sem restrições (ao menos neste caso) em sequências que contenham elementos iguais. No caso do algoritmo descrito, a determinação da mediana ponderada é realizada considerando-se cada elemento m_i como a mediana dos elementos restantes no processador i e com $w_i = n_i/N$, onde n_i corresponde ao número de elementos restantes no processador i e $N = \sum_{i=1}^{p} n_i$ é o número total de elementos restantes [41, 42].

O pseudocódigo do algoritmo é ilustrado pelo Algoritmo 23. Descreve-se [41, 42], que este algoritmo obteve-se uma economia substancial no tempo de comunicação, que mais do que compensou o tempo de computação devido a um processamento local desequilibrado. Utilizando a mediana ponderada os autores não consideram apenas os valores das medianas em cada bloco, mas também a quantidade de elementos restantes em cada bloco. Isto permitiu que os ciclos subsequentes lidem com diferentes números de elementos em cada processador e, no entanto, ainda assim garante-se a redução de, pelo menos, $1/4$ do total de números de elementos restantes em cada rodada.

Algoritmo 23: Algoritmo BSP/CGM de seleção de Saukas e Song.

Entrada: Um conjunto A de n elementos distribuídos entre p processadores com cada processador i armazenado em sua memória local $n_i = \mathrm{O}\left(\frac{n}{p}\right)$ elementos, e um inteiro k tal que $1 \leq k \leq n$.
Saída: O k-ésimo menor elemento de A.

1: **for** $N := n$ to $(n/c \times p)$ **do**
2: Cada processador i calcula a mediana m_i de seus n_i elementos
3: Cada processador i transfere m_i e n_i para o processador 1
4: O processador 1 calcula a mediana ponderada M
5: O processador 1 irradia M para os demais processadores
6: Cada processador i calcula os números de elementos locais menores l_i, iguais e_i e maiores g_i que M
7: Cada processador i transfere l_i, e_i e g_i para o processador 1
8: O processador 1 calcula os números totais de elementos menores $L = \sum_{i=1}^{p} l_i$, iguais $E = \sum_{i=1}^{p} e_i$ e maiores $L = \sum_{i=1}^{p} g_i$ que M
9: Processador 1 irradia L, E e G para os demais processadores
10: **case**:
11: **se** $L < k \leq L + E$: retorna solução M e aborta a execução do algoritmo
12: **se** $k \leq L$: cada processador i compacta seus elementos, mantendo em A apenas os elementos menores que M, faz-se $N := L$
13: **se** $k > L + E$: cada processador i compacta seus elementos, mantendo em A apenas os elementos maiores que M, faz-se $N := G$ e $k = k - (L - E)$
14: **end for**
15: Todos os N elementos restantes são transferidos para o processador 1
16: O processador 1 termina sequencialmente a resolução do problema.

6.4.1 Passos de Execução do Algoritmo: Um Exemplo

Apresentamos a seguir um exemplo de execução (passo a passo) do Algoritmo 23. A entrada é um conjunto A de n elementos (n=100) ilustrada pela Figura 6.23. Os elementos são divididos entre os p processadores. Neste exemplo considera-se $p = 20$, com $\frac{n}{p}$ elementos em cada processador, ou seja 5, como é ilustrado na Figura 6.24. Além disto, cada processador também ordena localmente seus elementos, como é ilustrado pela Figura 6.25.

96	12	22	13	17	95	15	45	16	23
94	11	38	14	24	86	2	53	8	28
89	9	61	10	29	69	5	41	26	30
68	0	62	6	31	97	7	82	33	36
73	3	54	4	42	92	21	48	27	47
74	44	59	49	50	88	40	57	46	55
99	1	71	52	58	84	18	78	25	60
75	20	64	51	63	90	32	80	34	65
77	19	70	43	66	93	35	76	56	67
98	37	85	72	81	91	39	87	79	83

Figura 6.23: Conjunto de entrada.

96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91
12	15	11	2	9	5	0	7	3	21	44	40	1	18	20	32	19	35	37	39
22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
13	16	14	8	10	26	6	33	4	27	49	46	52	25	51	34	43	56	72	79
17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	83
p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}

Figura 6.24: A divisão de dados.

1	12	15	11	2	9	5	0	7	3	21	44	40	1	18	20	32	19	35	37	39
2	13	16	14	8	10	26	6	33	4	27	49	46	52	25	51	34	43	56	72	79
3	17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	83
4	22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
5	96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91
	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}

Figura 6.25: A ordenação local.

No primeiro passo (linha 2), cada processador i calcula a mediana m_i de seus n_i elementos, conforme ilustrado na Figura 6.26.

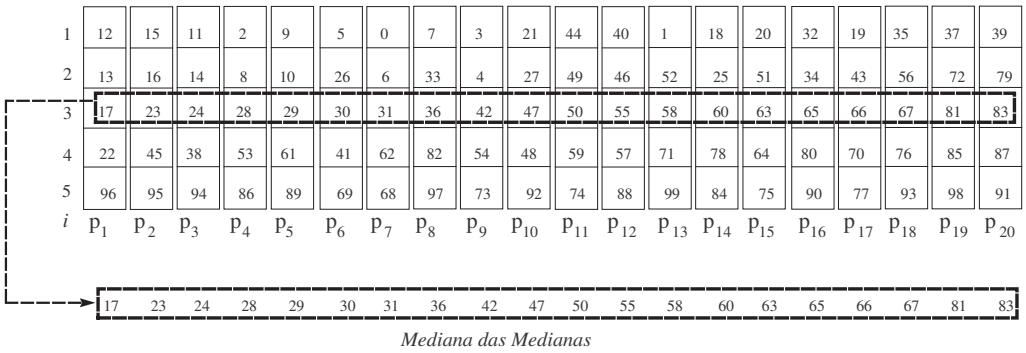


Figura 6.26: Cálculo de medianas locais.

No próximo passo (linha 3), cada processador transfere m_i e n_i para o processador 1, como é ilustrado pela Figura 6.27.

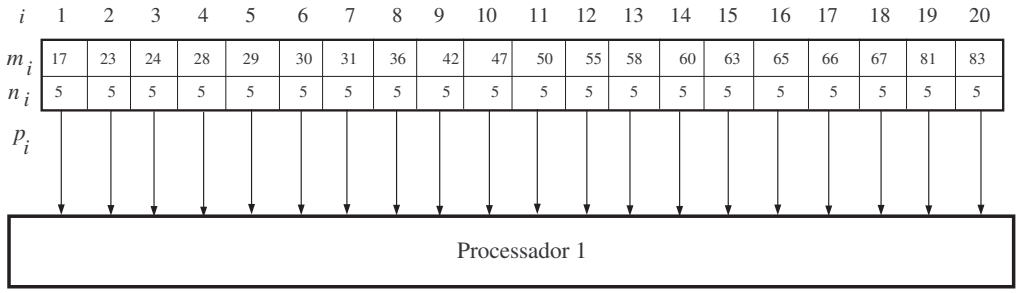


Figura 6.27: Envio de medianas ponderadas e número de elementos locais.

Em seguida, o processador 1 calcula a mediana ponderada M (linha 4), como é ilustrado pela Figura 6.28.

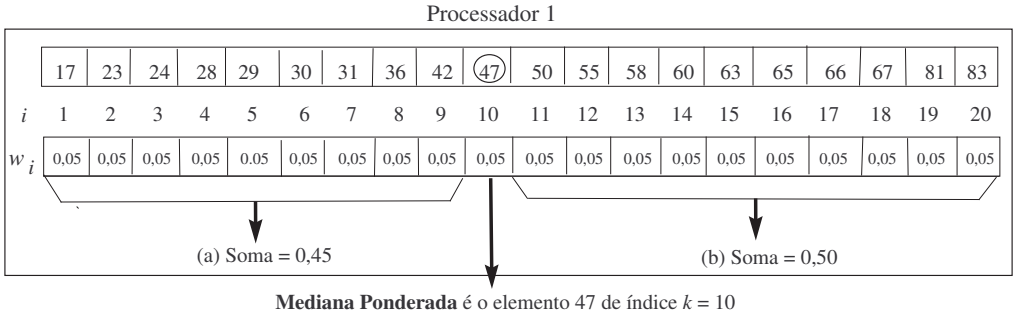


Figura 6.28: Cálculo da mediana ponderada.

No próximo passo, o processador 1 irradia a mediana ponderada M para os demais processadores (linha 5), como é ilustrado pela Figura 6.29

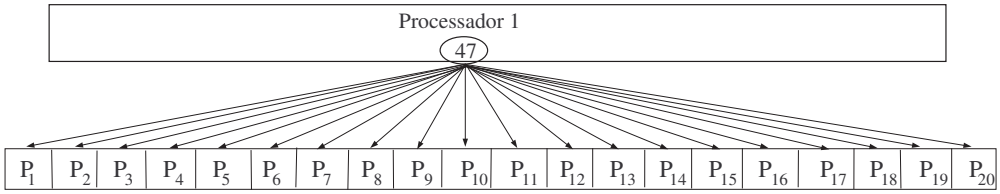


Figura 6.29: Irradiação da mediana ponderada.

Em seguida, cada processador i calcula os numeros de elementos locais menores l_i, iguais e_i e maiores g_i que a mediana ponderada M (linha 6), conforme ilustrado pela Figura 6.30.

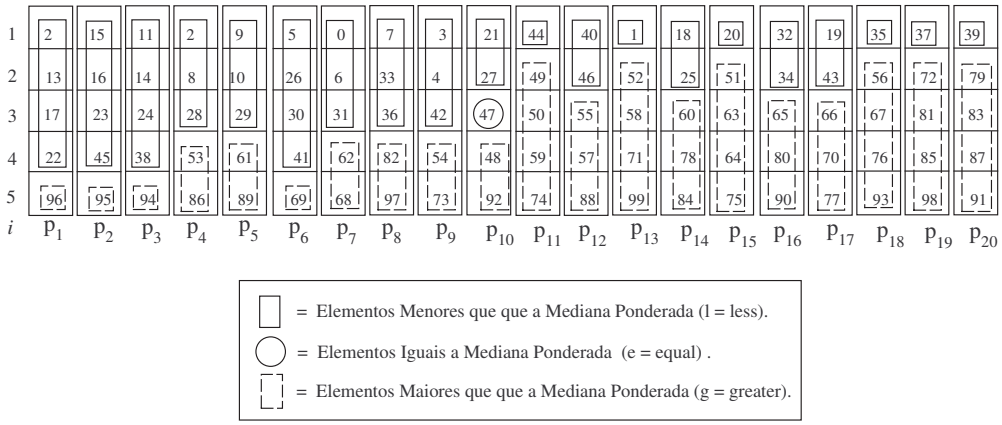


Figura 6.30: Cálculos locais de elementos menores, iguais e maiores que a mediana ponderada.

No próximo passo, cada processador i transfere l_i, e_i e g_i para o processador 1 (linha 7), como é ilustrado pela Figura 6.31.

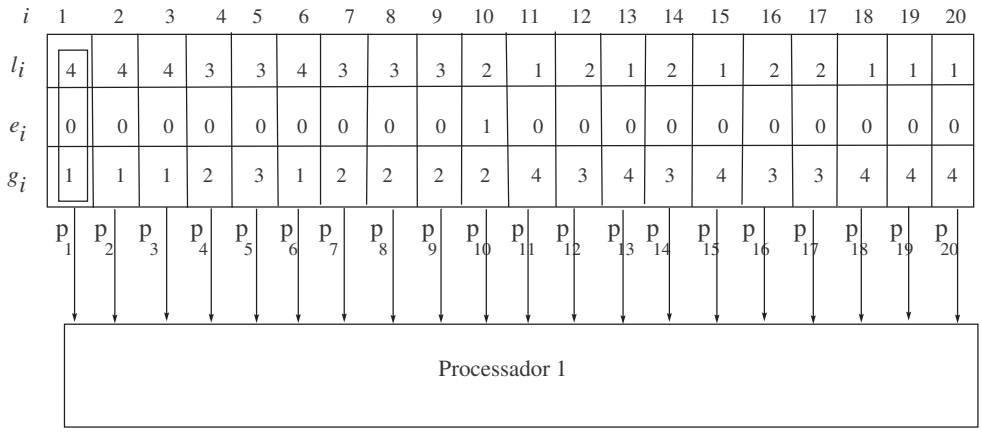


Figura 6.31: Transferência de elementos para o processador 1.

Em seguida, o processador 1 calcula os elementos totais menores (L), iguais (E) e maiores (G) que a mediana ponderada M (linha 8), como é ilustrado pela Figura 6.32.

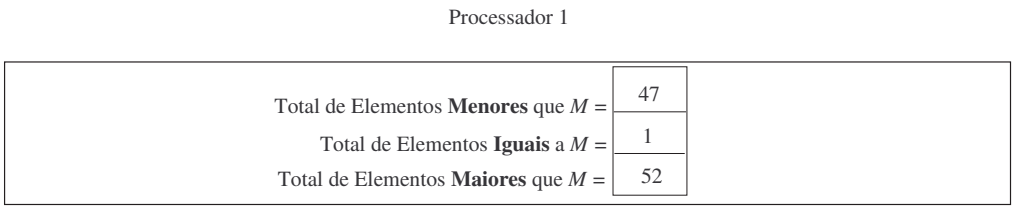


Figura 6.32: Cálculos totais.

No próximo passo, o processador 1 irradia L, E e G para os demais processadores (linha 9), como é ilustrado pela Figura 6.33.

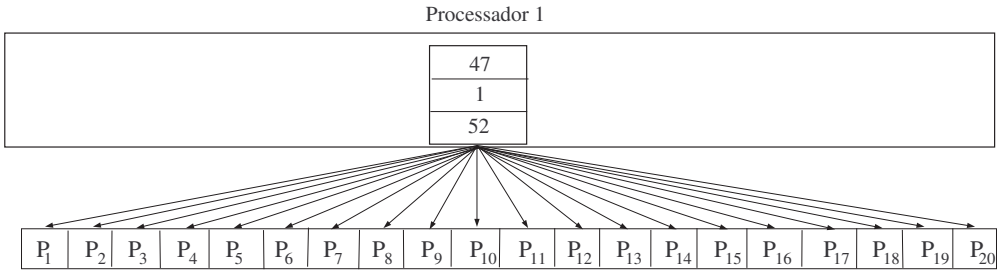


Figura 6.33: Irradiação de L, E e G .

Em seguida, cada processador executa o passo das linhas de 11 a 13, como é ilustrado pela Figura 6.34.

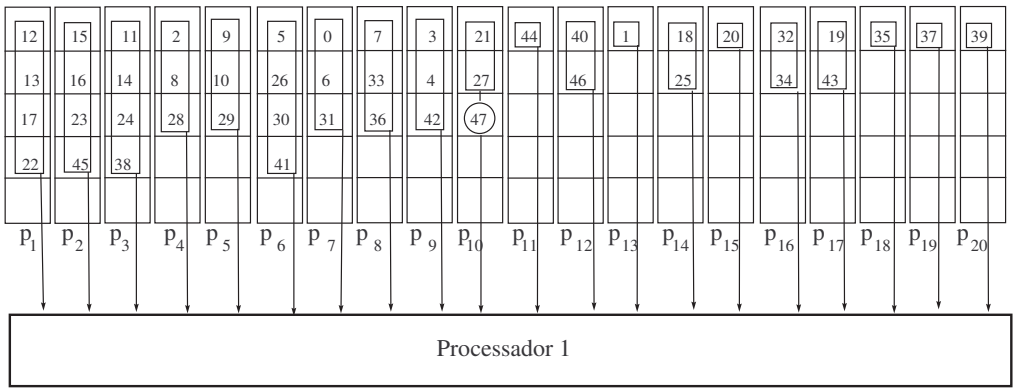


Figura 6.34: Aplicando algoritmo de partições localmente.

Nos últimos passos todos os n elementos restantes são transferidos para o processador 1, que termina sequencialmente a resolução do problema, como é ilustrado na Figura 6.35

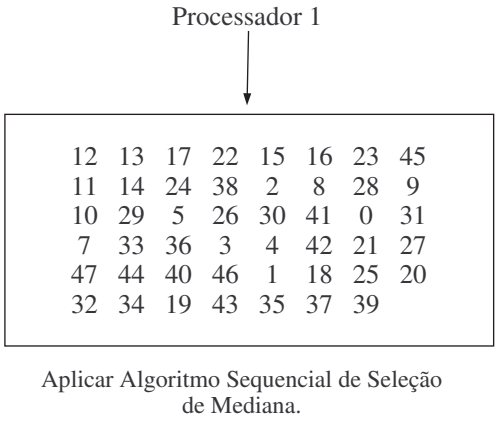


Figura 6.35: Solução local no processador 1.

6.5 A Implementação *Multi/Manycore* do Algoritmo BSP/CGM de Saukas e Song

Em termos de implementação para o problema da k-seleção, voltamos nossa atenção para o algoritmo CGM de Saukas e Song (Algoritmo 23). Nossa escolha considerou a possível eficiência e o ineditismo de implementação paralela deste algoritmo em ambiente de memória compartilhada e com muitos núcleos, particularmente em uma GPU utilizando CUDA. Como vimos, o Algoritmo 23 possui muitos passos paralelos que são sequencialmente executados.

O nosso trabalho de mapeamento e implementação para este algoritmo consumiu um considerável tempo, principalmente na análise e na escolha de soluções apropriadas para a implementação em CUDA de cada passo. Na entrada do algoritmo, consideramos um vetor A de inteiros de tamanho n armazenado em memória global. Além disto, p *threads* são instanciadas, cada uma com um subvetor A_i de tamanho n/p. Antes de qualquer execução as seguintes variáveis devem ser consideradas:

- O tamanho da entrada, ou seja, o número de elementos;
- A granulosidade do problema;
- A quantidade de *threads*, que representam nossas unidades de processamento;
- O número de Blocos de *threads*;
- O tamanho de cada bloco.

O tamanho de entrada, a granulosidade e o tamanho do bloco devem ser definidos antes da execução do algoritmo. As demais variáveis possuem as seguintes fórmulas de relação:

- quantidade de *threads* $= \frac{Tamanho\_da\_entrada}{Granulosidade}$;
- número de blocos $= \frac{Quantidades\_de\_threads}{Tamanho\_de\_cada\_bloco}$.

Uma vez definidos os valores de cada variável, os seguintes passos são executados:

1. Cada *thread* i determina sequencialmente a sua mediana m_i e a escreve na memória global;

2. Através de uma ordenação chave-valor de medianas e pesos ($\frac{qtd\_local\_de\_elementos}{qtd\_local}$), seguida de uma soma de prefixos, o *id* da mediana ponderada m das m_i medianas recebidas é localizado e armazenado na memória global;

3. Cada *thread* i determina a quantidade de elementos em A_i menores que m, a quantidade de elementos em A_i iguais a m e a quantidade de elementos em A_i maiores que m. Essas informações são armazenadas na memória global;

4. Através de uma redução calcula-se o valor de L, E e G, correspondentes às somas dos elementos menores, iguais e maiores que a mediana ponderada, respectivamente;

5. Se $L < k \leq L + E$, então a resposta é encontrada. Caso contrário:

   (a) se $k \leq L$, em paralelo os elementos maiores ou iguais a m são eliminados de A_i;

   (b) se $k > L + E$, em paralelo os elementos menores ou iguais a m são eliminados de A_i.

A nossa primeira implementação em GPU do Algoritmo 23 possui uma série de *kernels* CUDA de execução. De modo a facilitar a compreensão, optamos por relacionar com cada passo do algoritmo os respectivos *kernels*, que devem ser executados. Os relacionamentos e as tarefas executadas por cada *kernel* são ilustradas pela Tabela 6.3. Os *kernels* que utilizam algoritmos da biblioteca CUDPP são iniciados por **cudpp**.

Passos do Algoritmo 23	***Kernels* de execução (Versão 1)**
Cálculo de medianas locais m_i	1) **selectMedianGPU**: Retorna a mediana local utilizando o Algoritmo sequencial **Find.**
Cálculo da mediana ponderada M	1) **cudppRadixSort**: Ordena o vetor de medianas. 2) **cudppScan**: Realiza uma soma de prefixos baseada nos pesos de cada mediana. 3) **FindMedianOfMedians** = Localiza a mediana ponderada via operações atômicas, que evitam condições de corrida.
Cálculo de elementos locais menores, iguais e maiores que M	1) **FindNumberOfElements**: algoritmo sequencial executado em cada partição para contagem de elementos menores, maiores e iguais a M.
Cálculo global de elementos menores, iguais e maiores que M	1) **FindLEG**: Localiza os três valores globais de elementos menores, iguais e maiores que M. Utiliza operações atômicas para evitar condições de corrida.
Compactação de partições locais (Redução de elementos).	1) **CompactaMenores**: algoritmo sequencial executado em cada partição, que desconsidera os elementos maiores que M. 2) **CompactaMaiores**: algoritmo sequencial executado em cada partição, que descondidera os elementos menore que M.

Tabela 6.3: Relação entre os passos do Algoritmo 23 e os respectivos *kernels* CUDA da implementação em GPU.

6.6 Implementações e Resultados

Nesta seção apresentamos os resultados obtidos por duas versões de implementação do Algoritmo CGM de Saukas e Song (Algoritmo 23).

1. A primeira versão foi implementada em MPI seguindo os passos descritos do Algoritmo 23. A codificação desta versão está presente no anexo do trabalho de Saukas e Song [41];

2. A segunda versão é a nossa primeira implementação em CUDA, que segue os mesmos passos, mas obedecendo as particularidades descritas na Seção 6.5.

6.6.1 Resultados – Versão MPI

Para a versão MPI realizamos testes com 32, 64, 128 e 256 processadores. A Tabela 6.4 e a Figura 6.36 ilustram os resultados obtidos. Nota-se que o teste com 64 processadores obteve os melhores tempos de execução. Comparando com a versão sequencial (algoritmo mediana das medianas), o maior valor de *speedup* foi até 19 vezes melhor.

n (milhões)	Tempo Seq.	32 proc.	**64 proc.**	128 proc.	256 proc.	*Speedup* (Seq./64 proc)
1,048.576	102.7200	297.6533	28.7024	356.7599	371.392	3.5788
2,097.152	204.25581	275.921	33.6710	295.7938	557.313	6.0662
4,194.304	408.78598	365.6411	41.9090	219.3500	397.007	9.7541
8,388.608	806.08101	385.516	59.1890	234.6198	843.475	13.6189
16,777.216	1.599.39032	454.5355	95.1613	363.5950	971.121	16.8072
33,554.432	3.166.95723	497.9685	172.0890	343.3083	956.127	18.4030
67,108.864	6.261.39448	1.015.868	322.1739	319.1359	1766.897	19.4348
134,217.728	12.597.26318	1.688.6234	640.9353	522.0278	1323.723	19.6545

Tabela 6.4: Tempos de execução (**Milisegundos**): Sequencial (Mediana das Medianas) e Versões MPI (32, 64, 128 e 256 processadores).

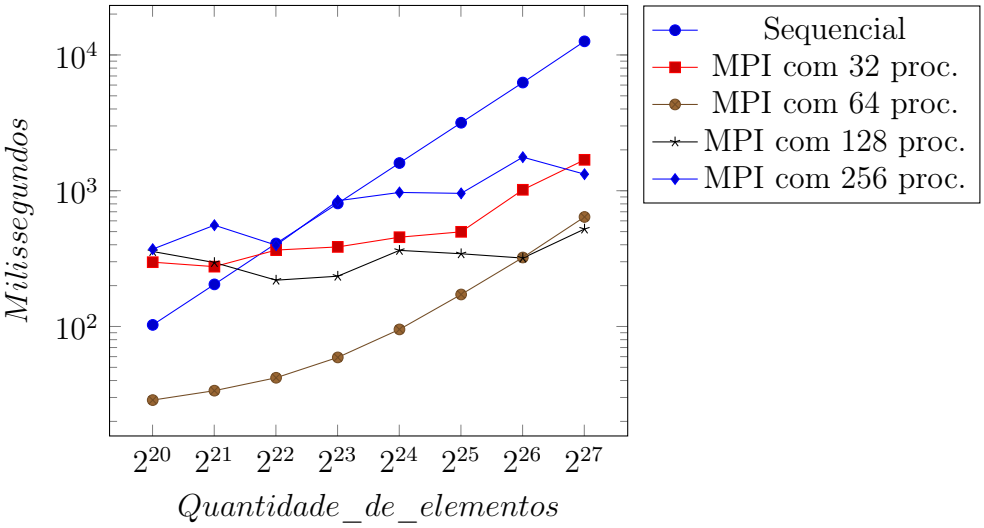


Figura 6.36: Sequencial × Versões MPI.

6.6.2 Resultados em CUDA

Nosso teste para versão CUDA considerou diferentes valores para o tamanho da entrada e para o tamanho da granulosidade do problema. Nosso objetivo era avaliar o comportamento do algoritmo em diferentes cenários de entrada e de granulosidade. O tamanho de entrada variou de 2^{20} até 2^{27} elementos. Já a granulosidade variou do tamanho 32 até 262144 (2^5 até 2^{18}). Mantivemos o tamanho do bloco fixo em 256. Realizamos buscas pelos diferentes índices k de elementos: 32, 64, 128, 256, 512, 1024, 4096, 8192, 16384, 32768 e 131072. Os valores buscados modificaram em muito pouco o tempo de execução final. Os resultados obtidos considerando a variação da granulosidade, o tamanho de entrada e os tempos finais são ilustrados pela Figura 6.37 e pela Tabela 6.5. Estes resultados expressam a busca por $k = 32$.

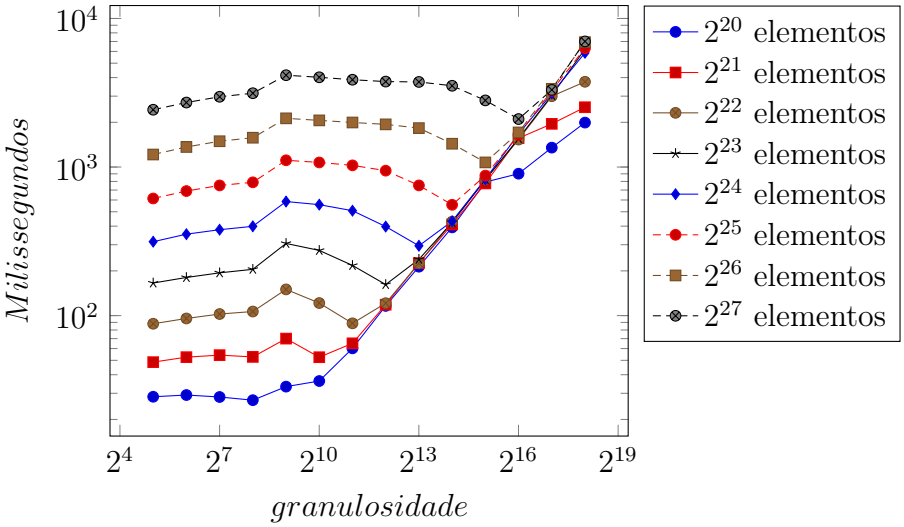


Figura 6.37: Granulosidade × Tempo de execução em GPU.

Gran.	2^{20} elem.	2^{21} elem.	2^{22} elem.	2^{23} elem.	2^{24} elem.	2^{25} elem.	2^{26} elem.	2^{27} elem.
32	**28.4292**	**48.6315**	**88.1270**	**165.6752**	**314.3291**	**614.2681**	**1214.1082**	**2429.8110**
64	29.2020	52.5208	95.6606	180.5849	353.3047	689.1633	1364.1342	2717.0103
128	28.3264	54.2045	102.4754	194.0165	378.6311	752.1265	1491.7426	2971.6794
256	26.9596	52.6972	106.5208	204.8876	399.1209	789.1514	1575.1687	3139.6343
512	33.2624	70.0142	150.1040	306.0136	585.8326	1110.3882	2127.1980	4149.6968
1024	36.2817	52.4821	121.5011	274.7000	558.0660	1073.7626	2060.6533	4018.5388
2048	60.2002	65.0943	88.7058	217.6210	508.2274	1026.4238	1994.7703	3869.9915
4096	116.2277	118.4272	121.1914	161.8813	397.4619	945.3781	1937.8453	3760.5083
8192	213.4423	225.7672	224.2203	239.2212	295.1200	752.3214	1828.7002	3739.3708
16384	392.4767	410.9009	428.9262	423.5645	431.7398	556.8693	1435.0470	3523.3835
32768	792.4399	775.7010	826.6927	821.8873	828.9627	875.9778	1074.2168	2808.6895
65536	901.9362	1564.0802	1530.9674	1555.3778	1667.5636	1645.3586	1709.5624	2103.5315
131072	1350.8899	1953.0784	2996.3987	3037.1279	3126.4478	3265.0776	3360.6968	3304.5332
262144	1991.3154	2527.3818	3747.8652	6217.8584	5871.3428	6313.7725	6929.6724	7007.1094

Tabela 6.5: Tempos em GPU: Granulosidade × Número de elementos.

De acordo com a Figura 6.37 e a Tabela 6.5 podemos concluir que o valor de granulosidade fixa em 32 retorna os melhores resultados. É possível observar que, para cada entrada, em algum momento o valor de tempo de execução cai, para em seguida se elevar abruptamente. A elevação de tempo é esperada, visto que em nosso problema, quanto maior a granulosidade, menor é o número de *threads* ativas trabalhando concorrentemente.

Tratando-se de desempenho, comparamos os tempos obtidos pelo algoritmo em GPU (com granulosidade fixa em 32) e os tempos da versão sequencial mediana das medianas. Os resultados são ilustrados pela Figura 6.38 e pela Tabela 6.6. Como pode ser observado, o maior valor de *speedup* foi de até 5 vezes melhor do que a versão sequencial.

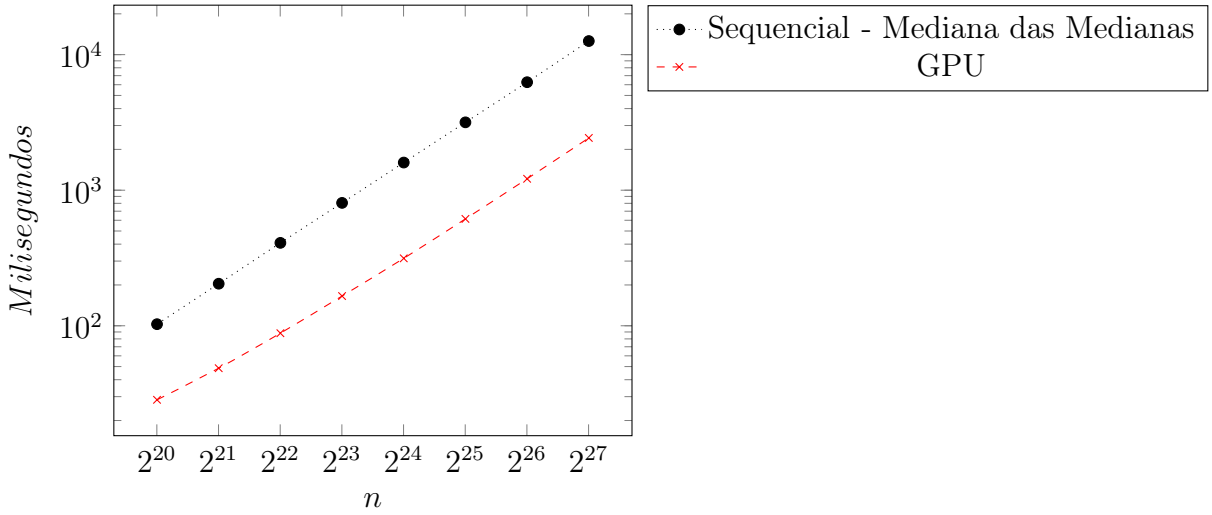


Figura 6.38: Sequencial × GPU.

n milhões	Gran.	N. *threads*	Tamanho Bloco	Tempo GPU	Tempo Seq.	*Speedup*
1,048.576	32	1024	256	28.4292	102.7200	3.6139
2,097.152	32	2048	256	48.6315	204.25581	4.2001
4,194.304	32	4096	256	88.1270	408.78598	4.6386
8,388.608	32	8192	256	165.6752	806.08101	4.8654
16,777.216	32	16384	256	314.3291	1.599.39032	5.0883
33,554.432	32	32768	256	614.2681	3.166.95723	5.1557
67,108.864	32	65536	256	1.214.1082	6.261.39448	5.1572
134,217.728	32	131072	256	2.429.8110	12.597.26318	5.1844

Tabela 6.6: Tempos de execução (**Milisegundos**) com granulosidade fixa em 32: GPU × Sequencial (Mediana das Medianas).

Alternativas de otimização – Versão 2

Buscando melhorar o tempo de nossa primeira implementação em GPU do Algoritmo 23, adotamos novas estratégias que pudessem diminuir o tempo local dos passos mais custosos. Além disto, também realizamos algumas alterações do ambiente de execução. Particularmente, o passo que consome maior tempo é a seleção local de medianas. Em média, acerca de 40 % do tempo total é dedicado a este passo. Neste passo, em nossa versão anterior, havíamos utilizado o algoritmo sequencial **Find**, que era executado concorrentemente. Reconhecendo que algoritmos de ordenação também podem retornar o valor de medianas, realizamos testes com três deles. A estratégia de maior êxito está sublinhada (vide Tabela 6.7) . O segundo passo para o qual conseguimos alguma melhoria significativa, foi o cálculo global de elementos menores, iguais e maiores que a mediana ponderada. Neste passo trocamos as antigas operações atômicas por três reduções executadas em paralelo. A Tabela 6.7 ilustra as alterações realizadas nesta segunda versão em GPU do Algoritmo 23.

Passos do Algoritmo 23	***Kernels* de execução (Versão 2)**
Cálculo de medianas locais m_i	1) **selectMedianGPU** : utilizando o algoritmo de ordenação *SelectionSort* até metade do vetor. A operação foi efetuada em memória compartilhada. Resultado foi satisfatório comparando-se com as demais tentativas. 2) **selectMedianGPU** : utilizando o algoritmo de ordenação *InsertionSort*. Apesar de ser o melhor dos algoritmos mais simples para ordernação, mostrou-se custoso. Operação efetuada em memória compartilhada. Resultado insatisfatório. 3) **selectMedianGPU** : utilizando o algoritmo de ordenação *QuickSort*. Parece que a recursividade acarretou um custo extra em nosso teste. Operação efetuada em memória compartilhada. Resultado insatisfatório.
Cálculo global de elementos menores, iguais e maiores que M	1) **cudppReduce**: operações de redução são executadas para localizar os três valores.

Tabela 6.7: Relação entre os passos do Algoritmo 23 e os respectivos *Kernels* CUDA da implementação em GPU.

Uma estratégia complementar foi a utilização do ambiente de memória compartilhada para a execução do passo de seleção local de medianas. O objetivo era acelerar o término do passo. Agora, as *threads* trazem de maneira coalescente os elementos da memória global para memória compartilhada. O acesso coalescente é obtido quando cada *thread* acessa a posição de memória com o número de seu *thread id*. Neste caso, a *thread* 1 acessa a primeira posição de memória, a *thread* 2 acessa a segunda posição e assim por diante. Um exemplo deste processo é ilustrado pela Figura 6.39.

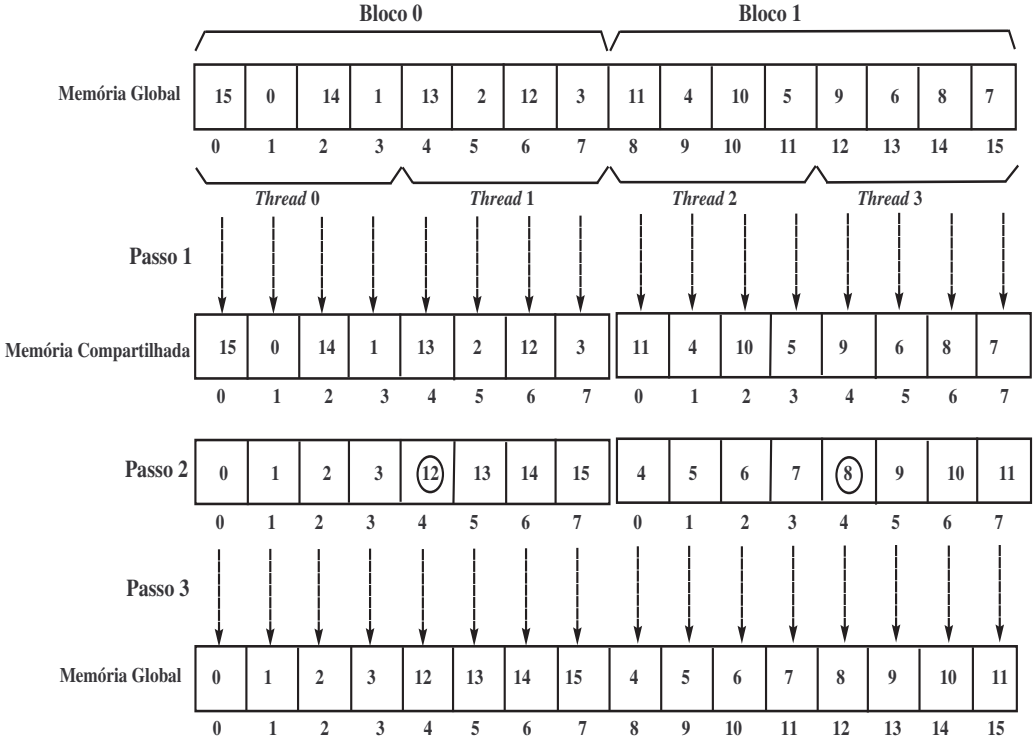


Figura 6.39: Trazendo elementos de maneira coalescente para memória compartilhada.

No exemplo (Figura 6.39), 16 elementos que formam dois blocos de com 8 elementos são carregados para a memória compartilhada. Uma vez terminado o passo, os blocos são novamente carregados para memória global. Apesar de veloz, um dos obstáculos para utilização da memória compartilhada é o seu tamanho, visto que restringe o tamanho de blocos que podem ser carregados. Utilizando elementos do tipo *int*, o número de posições de memória que podem ser carregadas para memória compartilhada é dado por: tamanho do bloco $\times$

Granulosidade. Considerando 256 como tamanho de bloco e 32 como granulosidade (exemplo típico de execução), temos que o número de posições de memória será 8192. Este número ainda deve ser multiplicado por 4 (tamanho *int*). O resultado alcança $32Kbytes$, que pode ser o limite da memória compartilhada.

Os resultados obtidos por esta nossa segunda implementação CUDA são apresentados pela Tabela 6.8. Agora, o maior valor de *speedup* é até nove vezes melhor do que a versão sequencial. Esta é uma melhoria de 40 % em relação a versão 1. Entretanto, este resultado ainda é inferior ao alcançado pela versão MPI.

n milhões	Gran.	N. *threads*	Tamanho Bloco	Tempo GPU	Tempo Seq.	*Speedup*
1,048.576	32	32768	256	19.3496	102.7200	5.3086
2,097.152	32	65536	256	32.5688	204.25581	6.2715
4,194.304	32	131072	256	55.0208	408.78598	7.4297
8,388.608	32	262144	256	99.6801	806.08101	8.0867
16,777.216	32	524288	256	180.2541	1.599.39032	8.8729
33,554.432	32	1048576	256	350.2770	3.166.95723	9.0413
67,108.864	32	2097152	256	658.2865	6.261.39448	9.5117
134,217.728	32	4194304	256	1.317.1656	12.597.26318	9.5639

Tabela 6.8: Tempos de execução (**Milissegundos**) com granulosidade fixa em 32: GPU (Versão 2) × Sequencial (Mediana das Medianas).

6.6.3 A comparação com a ordenação CUDPP

A operação de ordenação de um vetor n por si já retorna na posição correta, cada um dos k-ésimos menores elementos. Verificamos que a ordenação oferecida pela biblioteca CUDPP de CUDA é bastante eficiente. Neste contexto, realizamos comparações de nossa melhor solução para o problema da seleção (algoritmo MPI (64 proc.)), com um algoritmo de ordenação oferecido pela biblioteca CUDPP, no caso *cudppRadixSort*. Os resultados comprovam a eficiência desta ordenação sobre a seleção, conforme é ilustrado pela Tabela 6.9 e pela Figura 6.40. Entretanto, nosso algoritmo de seleção admite entradas continuamente crescentes. Este é um dos gargalos da biblioteca CUDPP, que apesar de potencializar os melhores recursos da GPU, possui restrição de tamanho de entrada.

No teste comparativo representado pela Tabela 6.9, o *speedup* entre a versão MPI de seleção e a ordenação CUDPP diminui progressivamente. Além disto, a ordenação CUDPP não conseguiu tratar entradas a partir do tamanho 2^{28}. Já a versão MPI admitiu entradas cada vez maiores. No último teste as entradas tinham tamanho 2^{30}.

n milhões	MPI (64 proc)	Ordenação CUDPP (cudppRadixSort)	*Speedup* (MPI / CUDPP)
(2^{20}) - 1,048.576	28.7024	1.9473	14.7396
(2^{21}) - 2,097.152	33.6710	3.0699	10.9681
(2^{22}) - 4,194.304	41.9090	5.4294	7.7189
(2^{23}) - 8,388.608	59.1890	9.3832	6.3079
(2^{24}) - 16,777.216	95.1613	22.2204	4.2826
(2^{25}) - 33,554.432	172.0890	43.6195	3.9452
(2^{26}) - 67,108.864	322.1739	86.8942	3.7076
(2^{27}) - 134,217.728	640.9353	175.6858	3.6481
(2^{28}) - 268,435.456	1.526.172	*(out of memory)	-
(2^{29}) - 536,870.912	2.953.0633	*(out of memory)	-
(2^{30}) - 1,073.741.824	5.515.2456	*(out of memory)	-

Tabela 6.9: Comparação de Tempos (**Milisegundos**) de execução Seleção MPI× Ordenação CUDA.

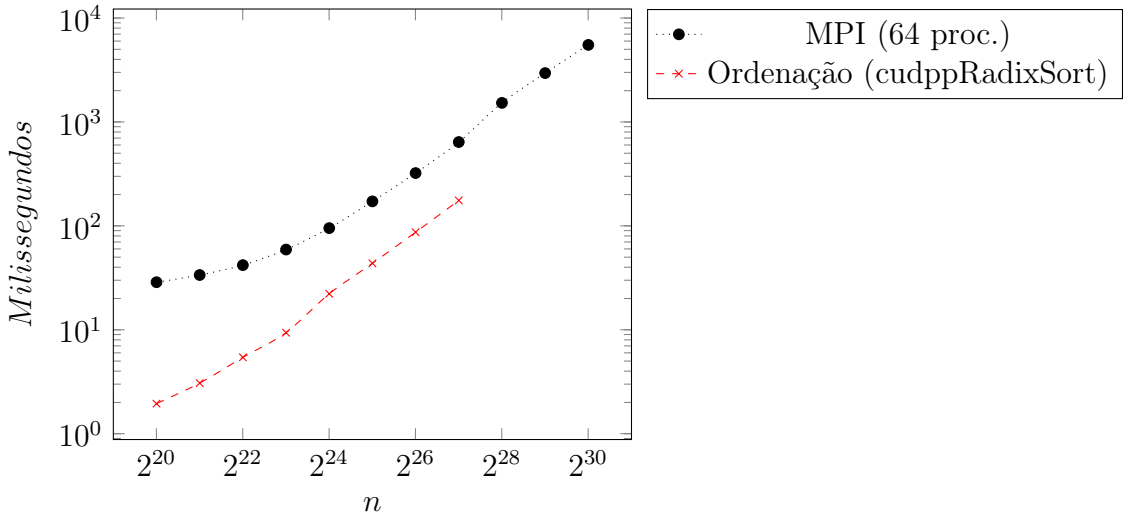


Figura 6.40: MPI (64 proc.) × Ordenação *cudppRadixSort*.

6.7 Sugestões de Problemas Futuros em Seleção

Destacamos duas sugestões de trabalhos futuros a respeito do problema da seleção:

1. Um problema relacionado com a seleção de um único elemento k consiste no problema de determinar q **diferentes** k-ésimos elementos $k_1, k_n, \ldots, k_q$ de um mesmo conjunto $A = a_1, a_2, \ldots, a_n$. Neste caso, ao invés de aplicar consecutivamente o algoritmo básico de seleção q vezes, pode-se adaptar este algoritmo para determinar estes elementos simultaneamente. É possível uma adaptação no Algoritmo 15, que garante uma grande economia nas fases de comunicação, uma vez que a determinação simultânea de q elementos pode ser realizada empregando-se o mesmo número de rodadas de comunicação, no pior caso, de uma única execução do algoritmo básico (para q menor ou igual a p), embora a quantidade total de informação transmitida seja até q vezes maior. A economia é obtida nas fases de processamento local, pois a determinação destes q elementos pode ser feita assintoticamente com o mesmo tempo de processamento local nenessário para a determinação de um único k-ésimo elemento (para $q = \mathrm{O}(p)$). Um algoritmo para este problema, acompanhado de demonstrações e teoremas, constam em um trabalho de Saukas e Song [41];

2. Uma outra sugestão é a implementação em GPU para o problema de seleção múltipla utilizando agora sequências bitônicas. Para tal, caso a sequência original não seja bitônica,

devemos primeiramente utilizar o o método de transformação em sequência bitônica. Em seguida, considerando q inteiros $k_1, k_2, \ldots, k_q$, onde $k_1 < k_2 < \cdots < k_q$, o problema da seleção múltipla deve obter q números que são os k_i-ésimos menores números ($1 \leq i \leq q$) da sequência dada. Vejamos possíveis etapas para este algoritmo paralelo:

- Com um *split* bitônico, a sequência de n números é particionada em 2 partições de tamanho $n/2$ cada, onde todos os números da primeira partição são menores que os números da segunda partição;
- Aplicando-se um novo *split* bitônico a cada uma das partições geradas, a sequência de n números fica particionada em 4 partições de tamanho $n/4$ cada, em que os números de cada partição são sempre menores que os da partição seguinte;
- Assim, depois de m *splits* bitônicos, a sequência de n números ficará particionada em 2^m partições de tamanho $n/2^m$, em que os números de cada partição são sempre menores que os da partição seguinte. O número m é escolhido com o seguinte critério: o tamanho de cada partição (ou seja $n/2^m$) deve ser tal, que os números são facilmente ordenados em um SM (*Streaming Multiprocessor*) da GPU. Esse valor depende das características da GPU e será considerado conhecido. É fácil identificar, através de um cálculo simples, a qual das 2^m partições cada um dos k_i menores elementos deve pertencer;
- Os números de cada uma dessas partições são transmitidos à memória compartilhada de um SM e ordenados, obtendo-se as respostas desejadas.

6.8 Considerações Finais do Capítulo

Neste capítulo discutimos sobre os problemas da seleção do k-ésimo menor elemento e da seleção de medianas. Dentre as soluções paralelas apresentadas para o primeiro problema, optamos pelo estudo e posterior implementação em GPU do algoritmo de Saukas e Song [41]. Alcançamos bons resultados de *speedup*, mas ainda inferiores aos resultados da versão MPI. Entretanto, a amplitude de testes, que ainda podem ser realizados em cada passo é grande. Acreditamos, que o algoritmo pode ter uma melhoria significativa de desempenho e alcançar ou superar a versão MPI, desde que o passo final queé executado apenas sequencialmente seja implementado. Particularmente, para a solução do problema específico de seleção de medianas, trabalhamos com duas abordagens: seleção bitônica e seleção por árvores diametralmente opostas. As duas apresentaram desempenho superior à versão sequencial da k-seleção, porém a versão por árvores apresentou uma curva de desempenho mais interessante. Neste caso, mesmo possuindo tempo inicial maior, conforme o aumento do tamanho das entradas, o tempo final do algoritmo cresce, porém não se eleva abruptamente como os demais, chegando ao ponto de se tornar melhor.

CAPÍTULO 7

CONCLUSÕES

No presente trabalho apresentamos uma extensão do modelo de computação paralela BSP/CGM para projetar algoritmos paralelos para arquiteturas *multi/many core*. Considerando que essa arquitetura é a adotada pelos principais fabricantes de *hardware*, e que a maioria dos algoritmos sequenciais não obtém ganho de desempenho quando implementados diretamente nessa arquitetura, é fundamental termos um modelo teórico para o projeto de algoritmos paralelos em arquiteturas *multi/many core*. Utilizando esse modelo, abordamos problemas que possuem uma grande dependência entre suas subtarefas. Para esses problemas, uma simples distribuição das subtarefas entre os processadores não leva a uma diminuição do tempo de execução, quando comparado a uma execução em um único processador.

Um dos problemas que abordamos é o problema da subsequência de soma máxima. A solução sequencial desse problema utiliza programação dinâmica e como a complexidade do melhor algoritmo sequencial é $O(n)$, a obtenção de um algoritmo paralelo para esse problema, cujo tempo de execução seja inferior que ao tempo de execução do algoritmo sequencial é um grande desafio. Propusemos também soluções paralelas usando a extensão do modelo BSP/CGM para o problema da submatriz de soma máxima e do hiper-retângulo de soma máxima. A solução desses problemas é uma generalização do caso unidimensional (subsequência). Em várias situações, a simples obtenção da soma máxima não é suficiente para responder perguntas mais específicas sobre regiões que estamos procurando numa sequência ou num hiper-retângulo (dimensão ≥ 2). Em função disso, propusemos soluções paralelas para a obtenção da maior e da menor subsequência de soma máxima, número de subsequências disjuntas de soma máxima, maior e menor submatrizes de soma máxima, número de submatrizes de soma máxima, k-ésima submatriz de soma máxima e a submatriz de densidade relativa máxima. Não é de nosso conhecimento a existência de algoritmos paralelos para o modelo BSP/CGM para esses problemas relacionados. Apresentamos também soluções para o problema da seleção do k-ésimo elemento de uma sequência.

Como vimos, os problemas abordados possuem uma diversidade de aplicações em diversas áreas da ciência, com destaque em biologia computacional, visão computacional, análise de volumes rochosos e de ordem. Os resultados que obtivemos com as implementações desses algoritmos indicam que a extensão do modelo BSP/CGM é adequada para o projeto e análise de algoritmos em arquiteturas *multi/many core*. Para confirmar a viabilidade do modelo, implementamos todos os algoritmos utilizando uma GPGPU. Além disso, como tínhamos a hipótese de que o desempenho dos algoritmos na GPGPU poderia ser melhor que numa

arquitetura com memória compartilhada, realizamos também implementações em MPI e OpenMP. A seguir descrevemos com mais detalhes cada um dos resultados obtidos:

- Para o problema básico da subsequência de soma máxima formulamos um novo algoritmo usando a extensão do modelo BSP/CGM. Para projetar esse algoritmo, nos baseamos no algoritmo proposto por Perumalla e Deo [35] no modelo PRAM. Uma particularidade deste novo algoritmo é o formato de sua saída. A partir dela obtivemos soluções para três problemas relacionados: a maior subsequência de soma máxima, a menor subsequência de soma máxima e o número de subsequências disjuntas de soma máxima. Não é de nosso conhecimento a existência de algoritmos BSP/CGM para estes três problemas relacionados. Nossos novos algoritmos obtiveram bom desempenho, confirmados por resultados experimentais. Em termos de complexidade, os algoritmos utilizam p processadores e consomem tempo paralelo de $O(n/p)$, com um número constante de rodadas de comunicação para o problema básico da subsequência de soma máxima e $O(\log p)$ rodadas de comunicação, com $O(n/p)$ computação local por rodada, para os algoritmos dos três problemas relacionados;

- Para o problema da submatriz de soma máxima também formulamos um novo algoritmo BSP/CGM. O algoritmo também utiliza as ideias propostas por Perumalla e Deo [35] no modelo PRAM, entretanto, como estamos trabalhando com um modelo realístico de computação paralela, tivemos que modificá-lo para equalizar a carga de trabalho e para manter a integralidade de dados nos processadores, especificamente das submatrizes tratadas. Procuramos assim, diminuir o número de etapas de comunicação. Além disto, nosso novo algoritmo executa a computação de um arranjo de saída extra, que armazena o número de elementos de cada submatriz de soma máxima. A partir destes arranjos, conseguimos obter soluções para uma série de problemas relacionados. São eles: (a) a maior submatriz de soma máxima; (b) a menor submatriz de soma máxima; (c) o número de submatrizes de soma máxima; (d) a seleção da k-submatriz de soma máxima e (e) a submatriz de soma máxima com maior densidade relativa. Não é de nosso conhecimento a existência de algoritmos BSP/CGM para estes problemas. As implementações desses algoritmos paralelos numa GPGPU obtiveram desempenho competitivo quando comparadas com as implementações sequenciais ou com resultados paralelos já conhecidos. Em termos de complexidade, nossos algoritmos utilizazam p processadores em tempo paralelo $O(n^3/p)$ com um número constante de rodadas de computação. Para o problema bidimensional apresentamos ainda uma aplicação do algoritmo. No caso ele foi utilizado para a detecção de áreas mais brilhantes em diagnóstico por imagens;

- No caso do hiper-retângulo de soma máxima, propomos uma solução para o problema de soma máxima cuja entrada é cúbica (tridimensional). Não é de nosso conhecimento a existência de algoritmos BSP/CGM para este problema. Basicamente, partimos das mesmas ideias utilizadas na versão bidimensional do problema. Assim como na versão bidimensional, procuramos manter um correto balanceamento de carga e a integralidade de dados nos processadores. Considerando que trata-se de uma algoritmo cuja solução sequencial é n^3, a versão em GPU do algoritmo apresentou resultados interessantes. Os melhores *speed-ups* atingiram a marca de centena de vezes com relação ao algoritmo sequencial. Em termos de complexidade, o algoritmo utilizou p processadores em tempo paralelo $O(\frac{n^5}{p})$, com $O(1)$ rodadas de comunicação. É importante destacar que a solução para o caso tridimensional pode ser generalizada para $d > 3$;

- Para o problema da seleção do k-ésimo menor elemento apresentamos versões BSP/CGM para três algoritmos paralelos. Um dos desafios no projeto de algoritmos paralelos para o problema da mediana é o fato de que a implementação do algoritmo de ordenação na biblioteca CUDA cudpp é muito rápido. Como salientamos, a solução sequencial $O(n)$ é um resultado muito teórico e as constantes são muito grandes. Por outro lado, a melhor implementação usa a mesma ideia de pivôs do *Quicksort* e os resultados obtidos com a paralelização do *Quicksort* não obtêm bons *speed-ups*. Em função disso, exploramos outras ideias. A mais promissora foi o algoritmo para o cálculo de medianas ponderadas no modelo BSP/CGM proposto por Saukas e Song [41, 42]. O algoritmo foi implementado usando memória distribuída em MPI. Neste trabalho desenvolvemos uma versão do algoritmo usando memória compartilhada em CUDA. Os resultados obtidos são promissores, mas os tempos obtidos com MPI foram melhores. O balanceamento de carga no MPI foi mais simples que na GPGPU. No caso da comparação com a biblioteca de ordenação CUDPP, a nossa implementação CUDA obteve um melhor desempenho quando o tamanho da entrada era superior ao limite de entrada do CUDPP.

7.1 Resultados Obtidos

Neste nosso trabalho apresentamos resultados significativos para problemas e aplicações utilizando a arquitetura *multi/many core*. Os problemas tratados possuem aplicabilidade em diversas áreas da computação. Descrevemos os resultados de uma aplicação na área médica. Usando ideias semelhantes é possível analisar voxels [1] com a soma máxima em hiper-retângulos de dimensão 3. Não é do nosso conhecimento algoritmos paralelos BSP/CGM para vários dos problemas abordados. Os problemas que foram tratados não possuem uma paralelização imediata e acreditamos que a extensão do modelo BSP/CGM para projeto de algoritmos paralelos para a arquitetura *multi/manycore* seja um passo inicial importante no estabelecimento de um modelo robusto para projeto de algoritmos paralelos nessa arquitetura. Até o presente momento, são estes os resultados advindos deste trabalho:

1. Publicação em conferência de um artigo internacional intitulado: *Efficient BSP/CGM algorithms for the maximum subsequence sum and related problems* [32]. Este artigo é uma compilação dos algoritmos e resultados obtidos de nossa pesquisa sobre o problema da subsequência de soma máxima e problemas relacionados. O trabalho consta dos anais da *International Conference On Computational Science* de 2015 (ICCS-2015). Posteriormente, o artigo foi disponibilizado também pelo periódico *Procedia Computer Science*, no Volume 51, ano 2015, nas páginas 2754–2758;

2. Publicação em conferência de um artigo internacional intitulado: *Efficient BSP/CGM Algorithms for the Maximum Subarray Sum and Related Problems* [31]. Este artigo é uma compilação dos algoritmos e resultados obtidos, de nossa pesquisa sobre o problema da submatriz de soma máxima e problemas relacionados. O trabalho consta dos anais da *International Conference on Computational Science and Its Applications* de 2015 (ICCSA-2015).

3. Submissão de um artigo para uma revista. Atualmente este artigo está em fase de revisão pelos editores da revista. O trabalho é intitulado: *Solving the maximum subsequence sum and related problems using BSP/CGM model and CUDA*. Trata-se de uma versão completa do primeiro artigo publicado. Neste artigo acrescentamos a descrição formal

dos algoritmos, novos experimentos e detalhes aF do problema da subsequência de soma máxima e dos três problemas relacionados;

4. Além dos dois artigos internacionais já publicados e do artigo de revista submetido, estamos trabalhando em mais dois outros artigos, um para uma revista e outro para um congresso internacional. Eles correspondem aos estudos e experimentos de nossos trabalhos sobre os problemas do hiper-retângulo de soma máxima e da seleção do k-ésimo menor elemento, respectivamente.

Para fins de consulta, os códigos fontes deste trabalho estão disponibilizados e organizados no seguinte repositório: https://github.com/AndersonLima37/CodigosTese.

7.2 Trabalhos Futuros

No sentido de continuação da pesquisa, elencamos a seguir alguns pontos a serem investigados em trabalhos futuros:

- Ampliar o arcabouço teórico da extensão do modelo BSP/CGM para obtermos um modelo teórico robusto para projetar algoritmos paralelos que envolvam tanta a memória compartilhada quanto a distribuída;
- Melhorar o balanceamento de carga dos SM's no problema da seleção do k-ésimo elemento de uma sequência;
- Projetar um algoritmo paralelo para arquiteturas *multi*/*manycore* para o problema de computar todas as somas maximais de uma sequência;
- Implementar os algoritmos num ambiente multiGPU's.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Voxel. https://en.wikipedia.org/wiki/Voxel. Accessed: 2015-10-11.

[2] Carlos E. R. Alves, Edson N. Cáceres, and Siang W. Song. BSP/CGM algorithms for maximum subsequence and maximum subarray. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 139–146. Springer Berlin Heidelberg, 2004.

[3] Sung E. Bae. *Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem*. PhD thesis, University of Canterbury, Christchurch, New Zealand, 2007.

[4] Jon Bentley. Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873, September 1984.

[5] Jon Bentley. *Programming Pearls*. Addison-Wesley, Pearson-Education, 1986.

[6] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[7] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4), August 1973.

[8] Shiva Chaudhuri, Torben Hagerup, and Rajeev Raman. Approximate and exact deterministic parallel selection. In *Mathematical Foundations of Computer Science 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 352–361. 1993.

[9] Richard John Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26(6):295–299, 1988.

[10] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Ed. Morgan Kaufmann, 2 edition, 2012.

[11] NVIDIA Corporation. CUDPP: Data parallel primitives library, 2014. [online; accessed 06-June-2014], http://cudpp.github.io/.

[12] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *A. Parallel Computer Architecture: A Hardware/Software Aprroach*. Ed. Morgan Kaufmann Publishers, San Francisco, California, USA, 1999.

[13] Edson N. Cáceres, Henrique Mongelli, and Siang W. Song. Algoritmos paralelos usando cgm/pvm/mpi: Uma introdução. In *XXI Congresso da Sociedade Brasileira de Computação*, pages 219–278, 2001.

[14] F. Dehne, A. Ferreira, E. N. Cáceres, S. W. Song, and A. Roncato. A. efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Algorithmica*, 33(2):183–200, 2002.

[15] Frank Dehne, Andreas Fabri, and Andrew Rau-chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6:298–307, 1994.

[16] Rob Farber. *CUDA Application Design and Development*. Ed. Morgan Kaufmann, 2011.

[17] Cleber S. Ferreira, Raphael Y. Camargo, and Siang W. Song. A parallel maximum subarray algorithm on GPUs. In *IEEE International Symposium on Computer Architecture and High Performance Computing Workshops*, pages 12–17, October 2014.

[18] Robert W. Floyd and Ronald L. Rivest. Algorithm 489: The algorithm select for finding the ith smallest of n elements. *Commun. ACM*, 18(3), March 1975.

[19] National Center for Biotechnology Information. NCBI: Genomes-bacterias, 2014. [online; accessed 12-Oct-2014], ftp://ftp.ncbi.nlm.nih.gov/genomes/Bacteria.

[20] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] Matthieu Garrigues and Antoine Manzanera. Exact and approximate median splitting on distributed memory machines. *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 529–534, 2012.

[22] Silvia M. Götz. *Communication-Efficient Parallel Algoritms for Minimum Spanning Tree Computation*. PhD thesis, University of Paderborn, May 1998.

[23] Alexandros V. Gerbessiotis and Constantinos J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 223–232, New York, NY, USA, 1996. ACM.

[24] Luciano Gonda and Henrique Mongelli. Uma implementação de algoritmos BSP/CGM de ordenação. In *VI Workshop em Sistemas Computacionais de Alto Desempenho*, pages 89–96, 2005.

[25] J. S. Hillmore. Certification of algorithms 63, 64, 65: Partition, quicksort, find. *Commun. ACM*, 5(8):439–, August 1962.

[26] National Cancer Institute. Cancer imaging archive, 2015. [online; accessed 11-Jan-2014], https://public.cancerimagingarchive.net.

[27] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[28] David B. Kirk and Wen-Mei W. Hwu. *Programando para Processadores Paralelos: Uma Abordagem Prática à Programação em GPU*. Ed. Elsevier, Rio de Janeiro, Brazil, 2011.

[29] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Ed. Morgan Kaufmann Publishers, 2 edition, 2012.

[30] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2 edition, 1973.

[31] Anderson C. Lima, Rodrigo G. Branco, and Edson N. Cáceres. Efficient BSP/CGM algorithms for the maximum subarray sum and related problems. *Lecture Notes in Computer Science*, pages 392–407, 2015. Computational Science and Its Applications.

[32] Anderson C. Lima, Rodrigo G. Branco, Edson N. Cáceres, Roussian R.A. Gaioso, Samuel Ferraz, Siang W. Song, and Wellington S. Martins. Efficient BSP/CGM algorithms for the maximum subsequence sum and related problems. *Procedia Computer Science*, 51:2754–2758, 2015. International Conference On Computational Science - Computational Science at the Gates of Nature.

[33] Sandra L. Marcus, John H. Brumell, Cheryl G. Pfeifer, and B.Brett Finlay. Salmonella pathogenicity islands: big virulence in small packages. *Microbes and Infection*, 2(2):145–156, 2000.

[34] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6:40–53, 2008.

[35] Kalyan Perumalla and Narsingh Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5:367–363, 1995.

[36] Ke Qiu and Selim G. Akl. Parallel maximum sum algorithms on interconnection networks. Technical report, Queens University Dept. of Com., Ontario, Canada, 1999.

[37] M. Suhail Rehman, Kishore Kothapalli, and P. J. Narayanan. Fast and scalable list ranking on the GPU. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 235–243, New York, NY, USA, 2009. ACM.

[38] Rüdiger Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396–409, 1985.

[39] Salah Saleh, Marwan Abdellah, Ahmed A. A. Raouf, and Yasser M. Kadah. High performance cuda-based implementation for the 2d version of the maximum subarray problem (MSP). In *Cairo International Biomedical Engineering Conference (CIBEC)*, 2012.

[40] Nicola Santoro, Michael Scheutzow, and Jeffrey B. Sidney. On the expected complexity of distributed selection. *Journal of Parallel and Distributed Computing*, 5(2):194–203, 1988.

[41] Einar L. G. Saukas and Siang W. Song. Algoritmos de seleção para máquinas paralelas com memória distribuída, 1997. Master's thesis.

[42] Einar L. G. Saukas and Siang W. Song. Efficient selection algorithms on distributed memory computers. *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–26, 1998.

[43] Maria Cecília R. Sena and Joseaderson Augusto C. Costa. Programa campus ambassador HPC - laboratório de computação científica e visualização. In *Tutorial OpenMP C/C++*. LCCV-UFAL, Macéio, Alagoas, Brasil, 2008.

[44] Douglas R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Sci. Comput. Program.*, 8(3):213–229, June 1987.

[45] ITU International Telecommunication Union. Recommendation itu-r bt. 470-6,7. conventional analog television systems, 1998. [online; accessed 03-Jan-2015], http://www.itu.int/rec/R-REC-BT.470-6-199811-S/en.

[46] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[47] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, January 2011.

[48] Zhaofang Wen. Fast parallel algorithms for the maximum sum problem. *Parallel Computing*, 21(3):461–466, 1995.