Utilização de técnicas de computação de alto desempenho para resolver problema de busca de caminhos: Uma comparação entre as técnicas

Eduardo A. de O. Borges[1], **Júlio C. M. Álvares**[1],
Marcus V. R. Campos[1], **Laerte M. Rodrigues**[1]

[1]Departamento de Engenharias e Computação – Instituto Federal de Minas Gerais (IFMG)
Caixa Postal 5 – 38.900-000 – Bambuí – Minas Gerais – Brasil

{edborges96,juliocmalvares07, marcusrodriguesc321}@gmail.com,

laerte.rodrigues@ifmg.edu.br

Abstract. *The work aims to propose a solution to the problem Finding Paths, which consists in finding the number of paths starting from the origin for a given set of coordinates in a three-dimensional Cartesian plane. The original problem only calculate how many paths are possible, for the job, besides calculating how many paths are possible, they will also be computed and shown to the user. Because it is characterized as an NP problem, several high performance computing techniques will be used to try to optimize the execution time of the algorithm, such as the use of threads, treating the problem as a problem of division and conquest.*

Resumo. *O trabalho tem como objetivo propor uma solução para o problema Finding Paths, que consiste em encontrar a quantidade de caminhos partindo da origem para um dado conjunto de coordenadas em um plano cartesiano de três dimensões. O problema original busca apenas calcular quantos caminhos são possíveis, para o trabalho, além de calcular quantos caminhos são possíveis, os mesmos também serão computados e mostrados ao usuário. Por se caracterizar como um problema NP, serão utilizadas diversas técnicas de computação de alto desempenho para tentar otimizar o tempo de execução do algoritmo, como a utilização de threads, tratando o problema como um problema de divisão e conquista.*

Introdução

A computação de alto desempenho é uma sub-área da ciência da computação que busca diminuir o tempo de relógio de algoritmos para a resolução de diversos problemas, na maioria das vezes, muito complexos [Coulouris et al. 2005].

Dentre as diversas técnicas da computação de alto desempenho, as mais comuns envolvem paralelização e distribuição da carga de trabalho do algoritmo, sendo que, cada uma das sub-partes será responsável por uma parte da execução do algoritmo. Ou seja, a carga de trabalho é sempre a mesma, apenas o tempo de relógio é otimizado.

Para definirmos programação sequencial, temos que pensar no paradigma imperativo como uma base, logo, a programação sequencial envolve uma consecutiva e ordenada execução de instruções, uma após a outra [Mivule 2011].

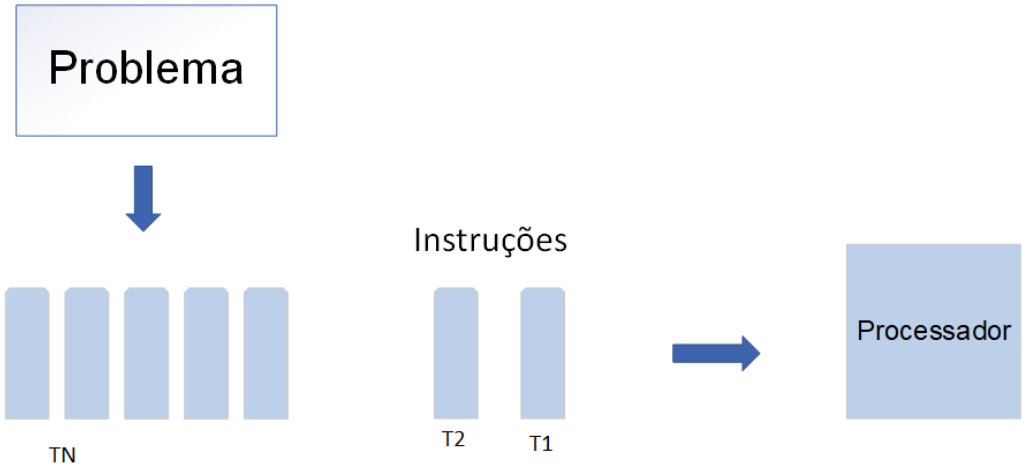

Figura 1. Diagrama para ilustração da programação sequencial.

Como pode-se observar na Figura 1, o problema é dado em uma série de instruções, nomeadas de *Tn*, onde são enviadas e executadas sequencialmente pelo processador.

Com a programação sequencial definida, temos que a programação paralela trata-se da execução de diversos programas sequenciais de forma simultânea, sobre sub-partes do mesmo problema. A computação paralela ajuda na realização de grandes cálculos dividindo a carga de trabalho em mais de um núcleo, ou processador, todos trabalhando no cálculo de forma simultânea. A maioria dos supercomputadores empregam princípios de computação paralela para operar [Techopedia 2018].

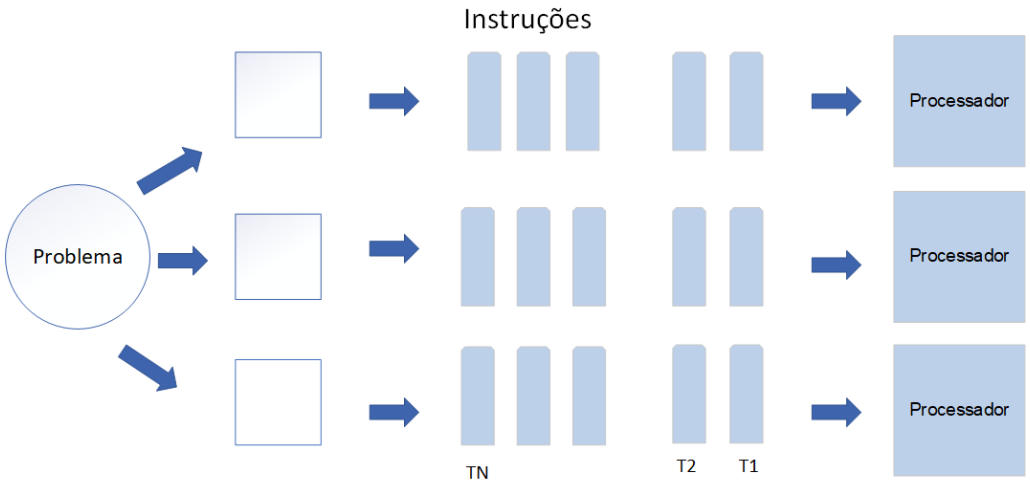

Figura 2. Diagrama para ilustração da programação paralela.

Na Figura 2 observa-se que o problema é quebrado em várias partes e distribuído entre vários diferentes processos para que os mesmos executem cada um de forma sequencial.

Uma técnica ainda mais potente é a programação paralela e distribuída. Um sis-

[1] As instruções na Figura 2 estão sendo executadas de forma sequencial por cada *thread*.

tema distribuído é aquele no qual os componentes de hardware e software, localizados em computadores interligados em rede, comunicam-se e coordenam suas ações apenas enviando mensagens entre si [Coulouris et al. 2005]. Então, em cada computador ligado na rede os processos funcionam de forma paralela.

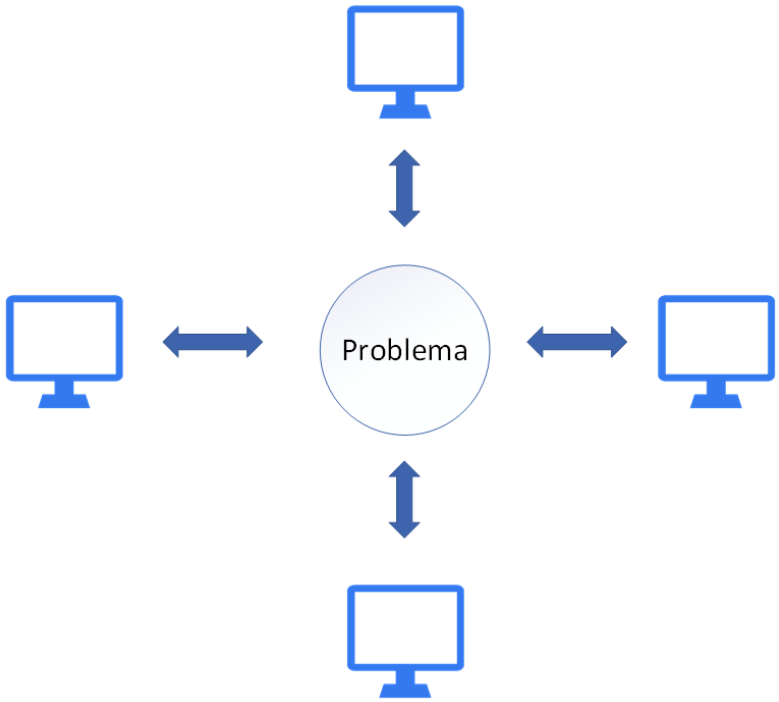

Figura 3. Diagrama para ilustração de sistemas distribuídos.

Na figura acima, o problema é sub-dividido entre vários computadores diferentes para que cada um possa resolver uma parte do problema.

Por fim, utilizamos a técnica de programação em *GPU*, utilizando o *cuda-toolkit* com suporte para placas gráficas *NVIDIA*. A *GPU* é uma poderosa unidade que conta com memória principal e unidade de controle própria. A programação em cuda tem como princípio o formato *host* e *device*, sendo eles so computador e a placa gráfica. Tal formato faz com que o *host* trate da lógica do programa e o *device* trate os cálculos do problema. [CUDA 2018]

Para testar o poder de cada uma das técnicas, um problema foi escolhido para ser trabalhado com todas e seus tempos de execução comparados.

Um problema NP *(nondeterministic polynomial)* encontra-se em uma classe que se caracteriza como não resolvíveis de forma determinística em tempo polinomial, ou seja, tais problemas não podem ser resolvidos a fim de dar a melhor resposta global em um tempo cabível, sendo que os mesmos podem chegar a milênios em tempo de processamento. [Cormen et al. 2009]

Tais problemas geram características como execuções *IO-Bound* ou *CPU-Bound* e, em raros casos, ambos.

Problemas *CPU-Bound* são aqueles que demandam muito tempo de execução em CPU, ou seja, têm muitos cálculos. De forma análoga, problemas *IO-Bound* são aqueles

que demandam de muitas interrupções de leitura ou escrita, fazendo com que seu tempo de CPU seja baixo. [Abraham and Baer 2000]

O problema em questão chama-se *Finding Paths*, retirado da plataforma *UVA Online Judge* e consiste em encontrar quantos caminhos são possíveis da origem até um conjunto de coordenadas *(n, m, k)* em um plano tri-dimensional sem repetir nenhum caminho e tendo um conjunto definido de passos possíveis. O conjunto de passos $\{(x + 1, y, z), (x, y + 1, z), (x, y, z + 1), (x + 1, y + 1, z), (x + 1, y, z + 1), (x, y + 1, z + 1), (x + 1, y + 1, z + 1)\}$ limita as possibilidades de caminhos. Um caminho só é válido se, e somente se passar por pelo menos um dos estados do sub-conjunto $\{(x+1, y+1, z+1),(x, y+1, z+1), (x+1, y, z+1), (x+1, y+1, z)\}$.

Apenas calcular a quantidade de caminhos possíveis não trata-se de um problema de extrema complexidade, sendo que, é possível modelar matematicamente o comportamento do crescimento da quantidade de caminhos, assim, o tempo de cálculo para a quantidade de caminhos torna-se linear. Como é desejado que o problema tenha extrema dificuldade, iremos calcular quantos caminhos podem ser gerados e, além disso, gerar todos os caminhos e mostrá-los para o usuário. Agora, o problema trata-se de um NP, pois gerar uma quantidade muito grande de caminhos não é trivial e demanda uma grande quantidade de armazenamento.

Para propor uma solução para o problema, foi feito um programa na linguagem de programação C++ e utilização do suporte da mesma para técnicas de computação de alto desempenho, como *threads*, utilizando a biblioteca *OpenMP*. Trata-se de uma biblioteca para programação *multi-threading* com memória compartilhada de utilização simples e objetiva [Chapman et al. 2008].

Para a técnica de sistemas distribuídos, utilizamos a biblioteca *OpenMPI* para dividir o processamento em várias máquinas. A mesma tem diversas diretivas que fazem o trabalho pesado para o programador e que deixam a sincronização entre os processos mais simples. Para [Graham et al. 2005], o *OpenMPI* é uma biblioteca para criação de aplicações *high performance* de multi processos.

Desenvolvimento

Para o desenvolvimento do trabalho, primeiramente foi feito um programa para calcular e mostrar a quantidade de caminhos de forma sequencial, utilizando listas encadeadas para armazenar as expansões dos caminhos em cada posição do plano cartesiano. Para agilização do processo, o código sequencial e paralelo é o mesmo, mudando apenas a quantidade de *threads* que serão executadas no mesmo.

Já executando o código com apenas uma *thread*, ou seja, de forma sequencial, notou-se que o uso de memória principal estava muito elevado, chegando à 6 GB para o conjunto (5, 5, 5) de coordenadas. Quantidade um tanto elevada, sendo que era esperado coordenadas maiores e menor uso de memória. Para contornar esse problema, utilizamos uma diretiva de sincronização da biblioteca *OpenMP* chamada *nowait*. Tal diretiva faz com que certas regiões sejam executadas de forma sincronizada entre as *threads* para que, em regiões futuras, o acesso a variáveis compartilhadas não tenham problemas como leitura fantasma [OpenMP 2018]. Assim descobrimos que nossas *threads*, em determinados momentos, estouravam a linha de proteção das condições de destino, inserindo

infinitamente novas posições na memória, fazendo com que a o uso da mesma cresça indefinidamente.

Para a execução do código paralelo e distribuído, o ideal seria a montagem de um *cluster*. Como não há recursos disponíveis para a montagem do mesmo, o código foi executado utilizando a diretiva de compilação do *OpenMPI* chamada *np x*, sendo "x"a quantidade de processos que serão criados dentro do sistema operacional. Assim, o código executa em diferentes processos mas sofre de problemas como escalonamento de processos do sistema operacional, podendo ocorrer vários fatores que comprometem o tempo de execução do mesmo.

Devido a tal problema, é esperado um *speed up* bem baixo ou não existente, sendo que o código está sendo executado em um ambiente completamente hostil e não preparado para o mesmo.

Para melhores comparações entre as técnicas, o mesmo algoritmo foi executado em dois computadores diferentes, com a mesma quantidade de memória RAM, divergindo apenas os processadores. Um dos computadores conta com um processador Intel i7-7500U e o outro conta com um processador AMD FX6300. Para execução dos algoritmos em CUDA, foi utilizado uma placa de vídeo GeForce 940MX de 4GB da NVidia.

O algoritmo em CUDA foi tratado da mesma forma que os demais, sendo que tal abstração do problema não é a ideal para trabalhar com essa técnica. Assim, foram encontrados problemas durante a execução dos algoritmos, impossibilitando que os dados dos mesmos sejam coletados.

A abstração da solução proposta deixa a execução do algoritmo tanto de forma *IO-Bound* como de forma *CPU-Bound*, assim, o algoritmo gera muitas interrupções no sistema operacional nas suas instruções.

Resultados

A quantidade de soluções pra o conjunto de coordenadas desejado está disposto na tabela a seguir.

Coordenadas	Quantidade de Soluções
(1, 1, 1)	7
(1, 2, 2)	71
(2, 2, 2)	319
(2, 3, 3)	3487
(3, 3, 3)	14401
(3, 4, 4)	165939
(4, 4, 4)	664471
(4, 4, 5)	2193751
(4, 5, 5)	7961951
(5, 5, 5)	31436497

Tabela 1. Quantidade de soluções para cada conjunto de coordenadas escolhido.

Os resultados das execuções dos algoritmos estão dispostos nas tabelas abaixo, em segundos.

Coordenadas	Sequencial	Paralelo	Paralelo e Distribuído
(1, 1, 1)	0.000157	0.026332	0.011615
(1, 2, 2)	0.004498	0.051649	0.021488
(2, 2, 2)	0.017263	0.047676	0.015723
(2, 3, 3)	0.156577	0.077664	0.103047
(3, 3, 3)	0.620496	0.333466	0.475226
(3, 4, 4)	7.200473	3.417465	5.046896
(4, 4, 4)	28.533579	22.385052	17.478684
(4, 4, 5)	93.742440	64.149772	63.736599
(4, 5, 5)	337.259533	202.153245	219.063759
(5, 5, 5)	1258.483783	1105.591386	2170.412446

Tabela 2. Distribuição dos tempos de execução (em segundos) de cada técnica no processador Intel.

Coordenadas	Sequencial	Paralelo	Paralelo e Distribuído
(1, 1, 1)	0.000724	0.002925	0.005511
(1, 2, 2)	0.003687	0.009612	0.008967
(2, 2, 2)	0.014090	0.027250	0.017188
(2, 3, 3)	0.131676	0.194929	0.126708
(3, 3, 3)	0.522095	0.756202	0.481820
(3, 4, 4)	6.119439	8.537793	5.521914
(4, 4, 4)	25.397143	34.130704	21.627552
(4, 4, 5)	78.296651	117.288454	72.922474
(4, 5, 5)	284.788489	417.413515	258.487852
(5, 5, 5)	1147.230041	1635.354696	1019.447437

Tabela 3. Distribuição dos tempos de execução (em segundos) de cada técnica no processador AMD.

Discussão dos Resultados

Como observamos na Tabela 1, o crescimento da quantidade de soluções é extremamente brusca para um conjunto pouco divergente de coordenadas. Tal característica nos dá base para confirmar a tipologia do problema como um NP, sendo que para pequenas mudanças na entrada, a saída diverge muito, além de que o tempo de execução dos mesmos também.

Abaixo serão dispostos gráficos para as comparações entre as técnicas.

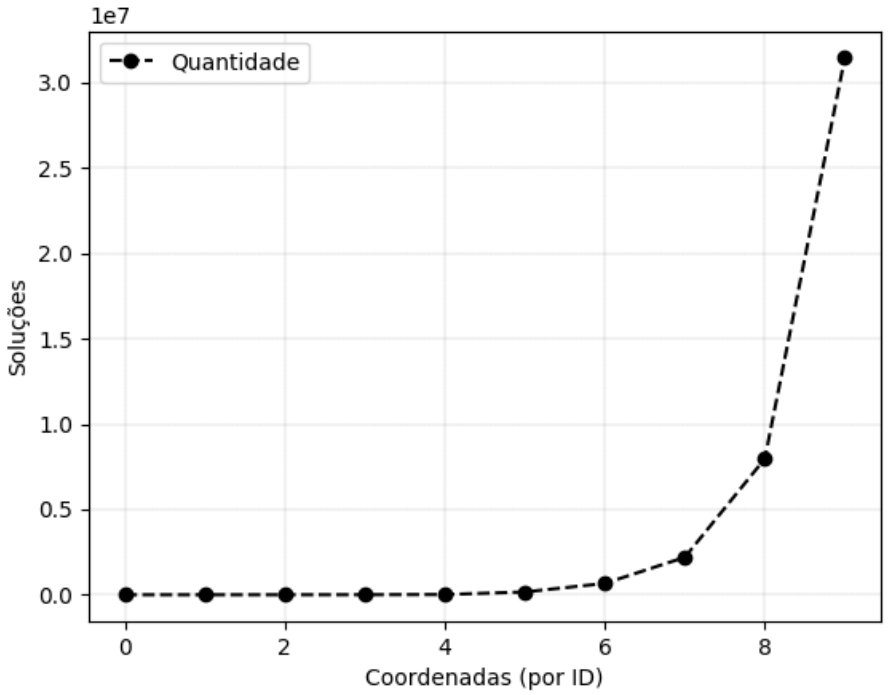

Figura 4. Curva de crescimento das soluções para o conjunto de coordenadas.

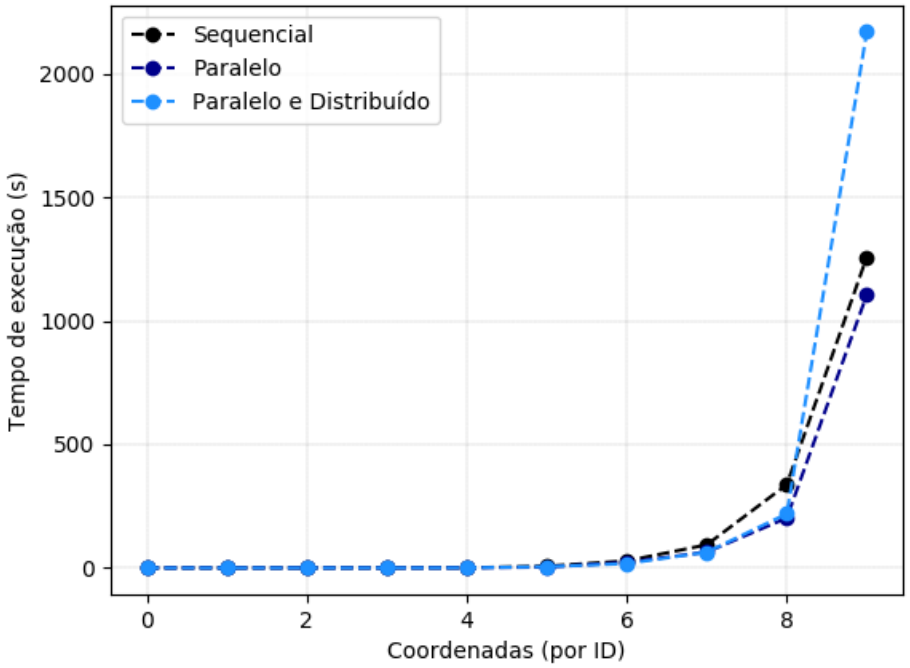

Figura 5. Curvas de crescimento dos tempos de execuções de cada técnica para o conjunto determinado de coordenadas para o processador Intel.

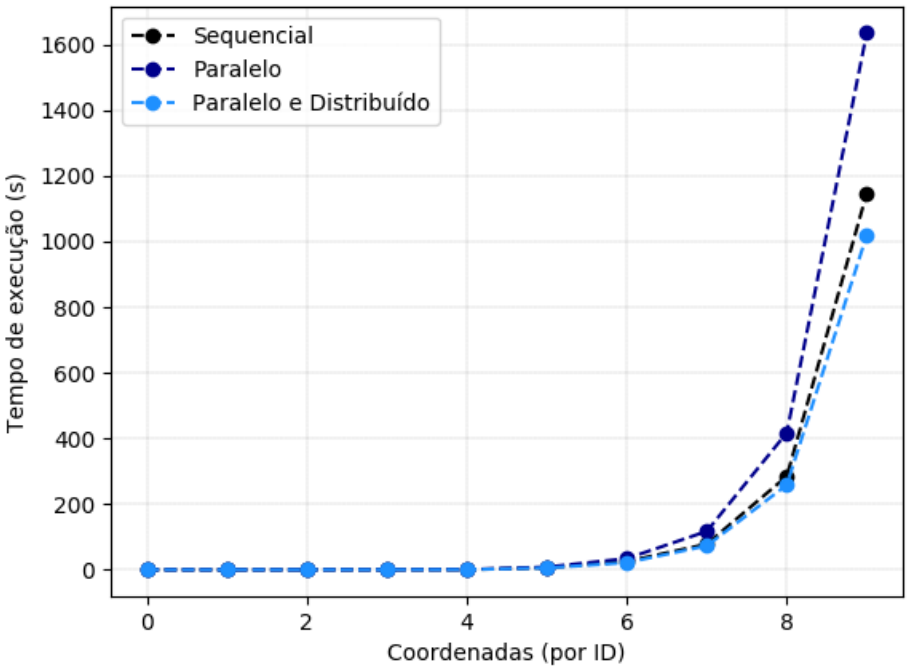

Figura 6. Curvas de crescimento dos tempos de execuções de cada técnica para o conjunto determinado de coordenadas para o processador AMD.

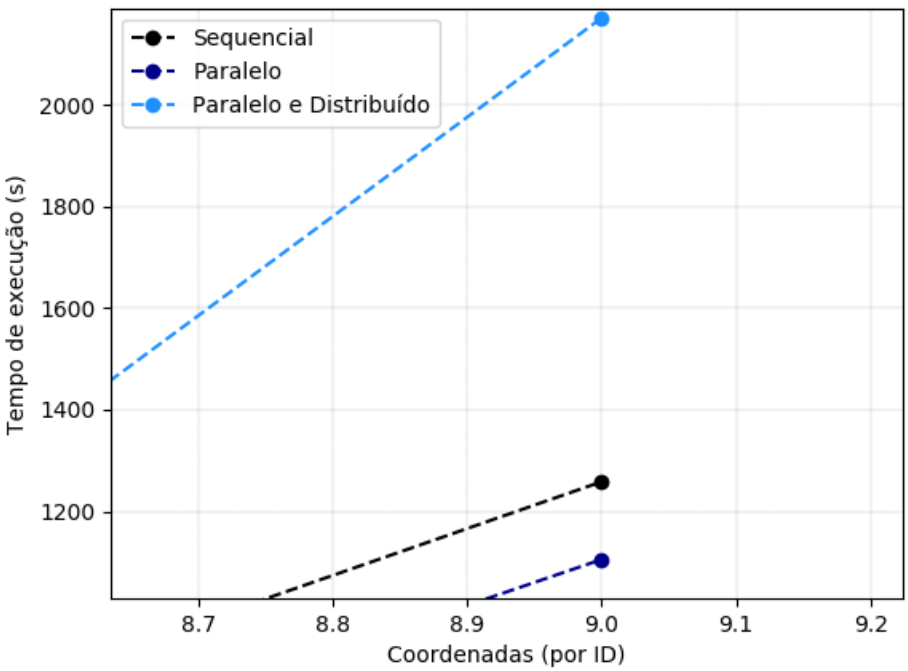

Figura 7. Pontos de tempo de execução para as coordenadas (5,5,5) para o processador Intel.

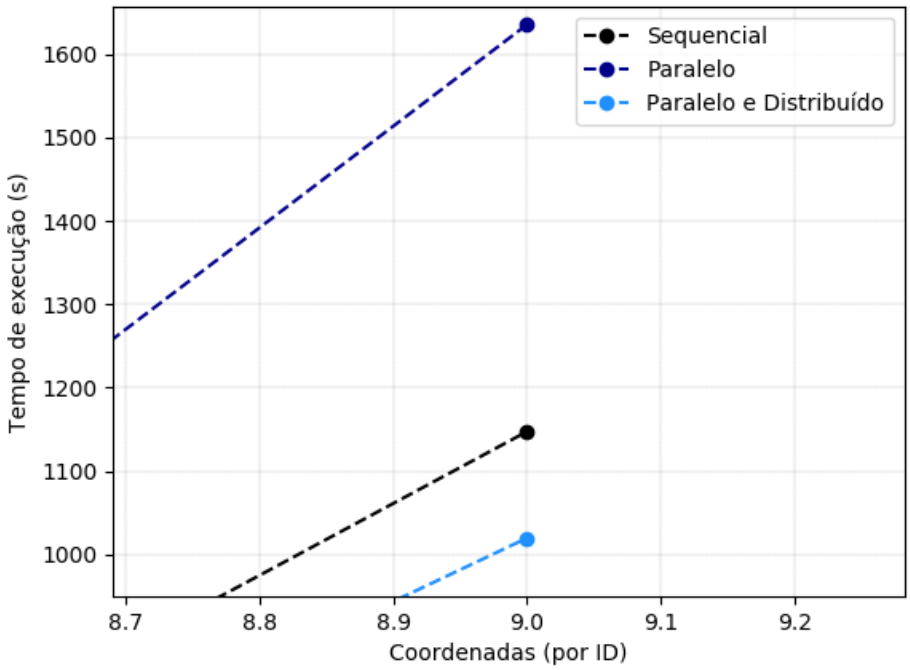

Figura 8. Pontos de tempo de execução para as coordenadas (5,5,5) para o processador AMD.

Observando a Figura 4 podemos notar que a curva de crescimento da quantidade de soluções se torna altamente acentuada a partir do conjunto de coordenadas (3,3,3), ou seja, a quantidade de soluções para o conjunto de coordenadas cresce extremamente para uma divergência baixa nas coordenadas.

Como mostra a Figura 2, a presença do *Hyper-Threading* nos processadores Intel deixa a execução dos algoritmos *multithreading* mais otimizados, porém, o escalonamento dos processos para as execuções paralelas e distribuídas não foram satisfatórias para o ponto (5, 5, 5), mas ainda sim trazendo *speedup* para a execução nos outros pontos.

Já observando a Figura 3, a falta de um sistema *Hyper-Threading* deixa a execução dos algoritmos paralelos e paralelos e distribuídos menos otimizadas, mas observando a execução sequencial, a mesma é altamente otimizada.

As Figuras 7 e 8 mostram uma perspectiva mais aproximada dos pontos (5, 5, 5), para melhor observação.

Em uma observação geral, concluímos que o processador AMD é mais otimizado em geral para as execuções de tarefas complexas para esse contexto.

Como os resultados da execução do CUDA não puderam ser coletados devido a diversos fatores, sendo eles problemas inerentes à natureza da solução proposta para o problema e que a utilização desenfreada da memória de vídeo acarreta no travamento do sistema operacional, fazendo com que o processo seja abortado, não foi possível mostrar resultados para que os mesmos sejam comparados com as outras técnicas.

Conclusão

Concluímos que os objetivos do trabalho foram parcialmente concluídos. Devido ao problema encontrado no CUDA, não foi possível fazer uma comparação completa entre as quatro técnicas.

Todavia, com os dados que obtivemos, é possível entender que as técnicas de computação de alto desempenho são realmente válidas e, na prática, demonstram resultados como os descritos na literatura.

Também podemos observar nos resultados que o sistema *Hyper-Threading* implementado nos processadores Intel faz com que os mesmos trabalhem de forma otimizada para processos *multithreading*, mas que a execução sequencial também não deixa a desejar.

De forma análoga, o processador AMD que não conta com o sistema *Hyper-Threading*, não teve execução altamente otimizada para o código paralelo e para o paralelo e distribuído, mas teve a execução sequencial mais otimizada em relação ao processador Intel, contando também com uma melhor execução como um todo.

Referências

Abraham, S. and Baer, G. P. (2000). *Operating systems concepts*. JOHN WILEY.

Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

Coulouris, G. F., Dollimore, J., and Kindberg, T. (2005). *Distributed systems: concepts and design*. pearson education.

CUDA (2018). Cuda zone, nvidia acelerated computing. `https://developer.nvidia.com/cuda-zone`. Accessed: 2018-07-01.

Graham, R. L., Woodall, T. S., and Squyres, J. M. (2005). Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer.

Mivule, K. (2011). Difference between sequential and parallel programming. `https://mivuletech.wordpress.com/2011/01/12/difference-between-sequential-and-parallel-programming/`. Accessed: 2018-07-01.

OpenMP (2018). The openmp api specification for parallel programming. `https://www.openmp.org/`. Accessed: 2018-07-01.

Techopedia (2018). Parallel computing. `https://www.techopedia.com/definition/8777/parallel-computing`. Accessed: 2018-07-01.