Parallel Patterns with C and OpenMP (1.0)

Francisco Cardoso 42997, Nelson Santos 48288

Abstract—O presente projeto teve como finalidade a transformação de oito padrões sequenciais em versões paralelas. A implementação foi alcançada com a utilização da framework OpenMP, que através de diretivas auxiliou a paralelização dos padrões sequenciais. Para validação da implementação, foram realizados diversos testes com o objetivo de analisar conceitos como speedup, eficiência e o custo de todos os padrões. Nem todos os padrões obtiveram ganhos significativos com a paralelização, e isto deve-se a vários fatores: natureza e aptidões de alguns padrões para operações mais de índole sequenciais; ora por outros fatores relacionados com a framework, nomeadamente *overheads* criados pelo carregamento da library, o instanciamento de threads, ou gestão do locks.

Index Terms—Concorrência, Paralelismo, Padrões, OpenMP, Speed up, Eficiência, Custo.

1 Introdução

A optimização de problemas é um desafio transversal em várias áreas da nossa sociedade. Um Engenheiro Informático, diariamente depreende-se com a necessidade de desenvolver conteúdos que consigam alcançar velocidades de computação mais elevadas.
Assim, o presente projeto, desenvolvido no âmbito da disciplina de Concorrência e Paralelismo, visa desenvolver versões paralelas face a implementações sequenciais de alguns padrões disponibilizados, levando a otimizações. Para o desenvolvimento das versões paralelas, será utilizada a *framework* OpenMP, que permite a utilização de diretivas/anotações no código desenvolvido, que possibilitando alcançar a paralelização mais rapidamente e eficientemente. Além da implementação, serão demonstrados os resultados dos testes realizados e onde as seguintes métricas serão analisadas: *speedup*, eficiência e custo.
Todos os gráficos no presente relatório são referentes ao tamanho de *array* de cem milhões. Decidiu-se analisar esta dimensão porque em todos os padrões é o onde se consegue "visualizar" melhor as diferenças entre os respetivos *workers*. Para a mitigação de erros marginais nas medições do tempo de execução, cada teste foi repetido trinta vezes. A implementação foi testada em duas máquinas distintas: node 12 do cluster do DI; e num computador com processador MD Ryzen 7 3700X, Octa-Core, 3.6GHz. Decidiu-se optar por analisar os resultados obtidos pela segunda máquina, pois por ser mais recente, e naturalmente com maiores capacidades de computação, obteve-se maiores ganhos de tempo de execução em todos os padrões.

2 Análise dos Padrões

2.1 MAP

Implementação: analisando o padrão, constatou-se da ausência de qualquer dependência entre as operações de leitura/escrita dos *arrays* de *input*/*output*, aplicando a *elemental function*. Assim, a paralelização foi alcançada com a aplicação da diretiva do *OpenMP #pragma omp parallel for num_threads(x)*.
Análise de resultados: observando o gráfico, verifica-se que o existem ganhos significativos com a versão paralela do padrão Map. O tempo de execução mais baixo é com 16 threads (0.1127 s), no entanto com o tempo de execução com 8 *threads* é muito semelhante (0.1144 s). Confirma-se ainda o aumento e degradação do tempo de execução com 32 e 64 *threads* respetivamente.

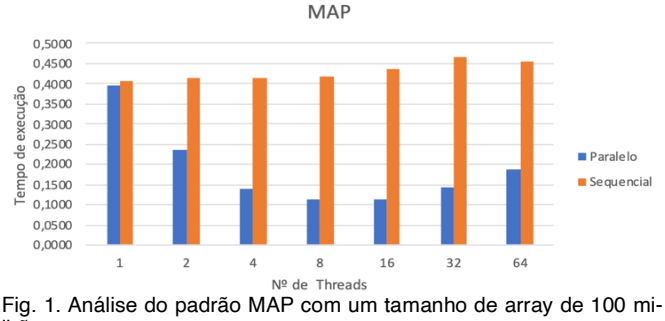

Fig. 1. Análise do padrão MAP com um tamanho de array de 100 milhões.

2.2 REDUCE

Implementação: para a paralelização deste padrão poderíamos optar por duas hipóteses: *Vectorization* ou *Tiling*, conforme o capítulo 5 de [1]. Começou-se por utilizar a *Vectorization,* no entanto a implementação não apresentava resultados determinísticos. Assim, decidiu-se optar pela vertente de Tiling, onde se obtiveram os resultados esperados: ganhos de *speedup* e determinismo.

Assim, a nossa implementação divide o *array* de *input* pelas *threads* existentes (constante definida no início do ficheiro patterns.c), e através de um *array* auxiliar, com a ajudar da diretiva *#pragma omp parallel for*, cada *thread* paralelamente faz a combinação dos pares da sua quota parte do array. Como as leituras e escritas são realizadas pelas *threads* em cada uma das suas zonas no *array* auxiliar, leva a que o resultado final seja determinístico. No final é realizado um ciclo *for* que coleta os dados parciais resultantes da aplicação da função combinatória e escreve estes no *array* de *output*, *dest*.
Análise de resultados: observando o gráfico, verifica-se que os ganhos são logo visíveis a partir da utilização de 2 *threads*, obtendo o melhor resultado com a utilização de 16 *threads*. Tal como no padrão MAP, a utilização de 32 e 64

threads não apresentam ganhos em tempo de execução.

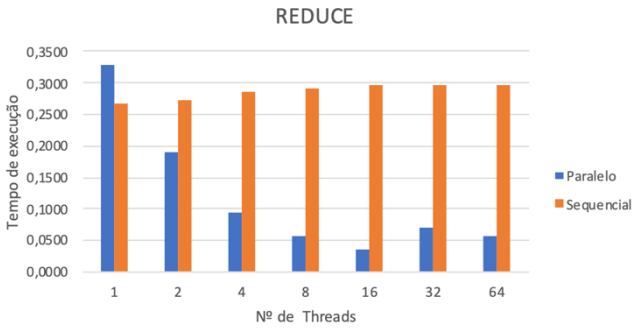

Fig. 2. Análise do padrão REDUCE com um tamanho de array de 100 milhões.

2.3 SCAN

Implementação: seguindo a sequência descrita em [3], o padrão Scan foi implementado em 2 fases: a primeira fase sendo "*Up Pass*" que consiste na criação de uma árvore binária com a soma do valor sum dos nós filhos para cada nó e respetiva alocação de memória; e a segunda fase, o "*Down Pass*", que consiste na computação dos valores *fromLeft* para cada nó e a computação final do *fromleft* com o *sum* do nó raiz. Para a implementação deste padrão usámos uma estrutura de dados do tipo "*node*", que contem o valor *sum*, *fromleft*, index e dois apontadores: um para cada nó filho, *left* e *right*.

Análise de resultados: observando o gráfico, verifica-se que existem perdas significativas de eficiência comparando esta versão paralela do algoritmo com a versão sequencial. Comparando os melhores valores obtidos do scan em paralelo, com 8 *threads* em que o tempo foi de 17.173 s, com o tempo da versão sequencial de 1,379 s notamos que a versão paralela demora quase 12.5 vezes mais do que a sequencial. Com a nossa implementação não existem vantagens na paralelização, no entanto, acreditamos que com computações mais complexas, tenhamos maiores ganhos de tempo de execução neste padrão.

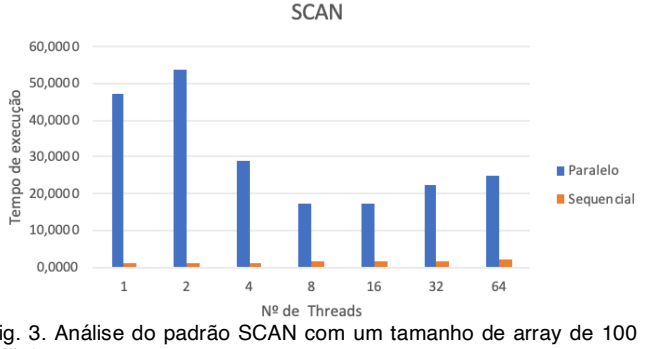

Fig. 3. Análise do padrão SCAN com um tamanho de array de 100 milhões.

2.4 PACK

Implementação: Seguindo a sequência descrita em [3], o padrão Pack foi implementado em dois passos: no primeiro passo foi utilizado o padrão Scan para criar um vetor *bitsum*, este vetor servirá para saber a ordem pela qual os elementos não descartados no padrão vão ser escritos no *array* de *output*; no segundo passo paralelizamos a escrita dos valores de *input* não descartados, no *array* de *output* usando a ordem estabelecida no *array bitsum*. O descarte de elementos é feito através de um filtro de inteiros com valores 0 e 1, em que as posições em que o filtro tenha o valor 1 são escritas e as posições com o valor 0 são descartadas. Como neste passo cada valor de input será escrito uma única vez e para cada escrita é atribuída uma posição diferente no vetor, não há dependências entre escritas, sendo simples a paralelização deste padrão através da diretiva do *OpenMP #pragma omp parallel for num_threads(x).*

Análise de resultados: o Pack é outro dos padrões em que com a nossa implementação não é benefica a sua paralelização. Este padrão apresenta em média, aproximadamente o dobro do tempo a executar comparando com a sua versão sequencial. Estes resultados podem ser explicados pela utilização do Scan sequencial para criar o *array bitsum* necessário para ordenação dos *inputs*. Optámos por usar o Scan *sequencial* pois a nossa implementação do Scan em paralelo apresentava resultados piores.

Com acesso a uma versão paralela do padrão Scan que apresente um melhor desempenho, acreditamos que a paralelização do Pack se torne viável.

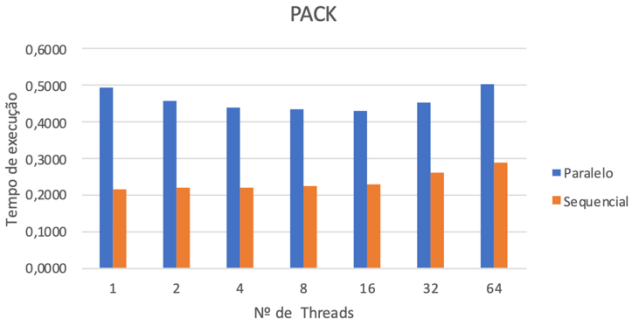

Fig. 4. Análise do padrão PACK com um tamanho de array de 100 milhões.

2.5 GATHER

Implementação: Neste padrão só são executadas operações de leitura, e como as operações de leitura não apresentam dependências entre si, constatou-se que para a paralelização deste padrão bastava a aplicação da diretiva do OpenMP *#pragma omp parallel for num_threads(x).*

Análise de resultados:

Analisando o gráfico em baixo, verifica-se que existem ganhos significativos com a versão paralela do padrão Gather. O tempo de execução mais baixo é com 16 threads (0.0333 s). Como o computador onde executámos os testes tem 8 cores verifica-se que aumentando o número de threads a partir das 16, o tempo de execução começa a aumentar pois as threads adicionais são fictícias e aumenta

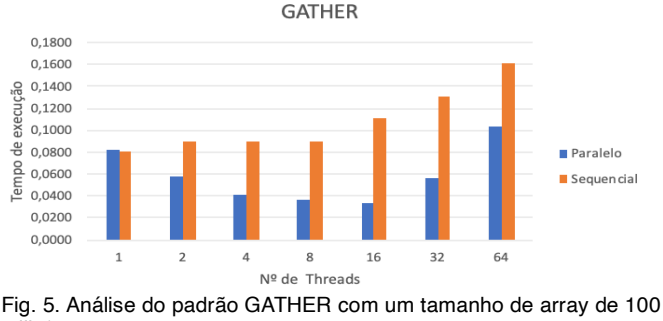

Fig. 5. Análise do padrão GATHER com um tamanho de array de 100 milhões.

significativamente o custo de gestão das threads.

2.6 SCATTER

Implementação: Na nossa implementação, como entendemos que o determinismo seria fundamental e importante, em detrimento do speedup, foi utilizado o *Prioraty Scatter*. Nesta versão, em caso de ocorrer uma colisão, ou seja, duas escritas na mesma posição de uma coleção, prevalece a escrita mais que ocorrer mais tarde temporalmente. O local identificado onde aplicar a paralelização do algoritmo sequencial, seria no ciclo for, que iterava todas as posições do *array* de *input*. Assim, para resolver as colisões foi utilizado um *array* auxiliar (*auxFilter*), onde com recurso à função da biblioteca stdlib.h da linguagem C, 'calloc', cada posição do *array* auxiliar foi instanciada com valor 0. No ciclo for foi utilizada a diretiva do OpenMP *#pragma omp parallel for num_threads(x)*, que desta forma divide o tamanho do array de input pelas *threads* disponíveis. Em cada escrita verifica-se se o valor do índice atual (i) é superior ao valor da posição que se irá escrever, no array auxiliar. Caso seja superior, significa que já ocorreu uma escrita anterior à presente, pelo que esta tem prioridade e é realizado um *overwrite* no *array* de *output*.

Análise de resultados: obeservando o gráfico, verifica-se que com 1 e 2 *threads* o *speedup* da versão paralela é superior. No entanto a partir das 4 *threads* os ganhos vão aumentando, ainda que ligeiros, atingindo com 32 threads o melhor valor de speedup. Os ganhos genéricos da versão paralelizada é pouco, comparado com a implementação da versão atómica, que também foi testada. No entanto, como o determinismo prevaleceu, decidiu-se manter a versão *Prioraty Scatter*.

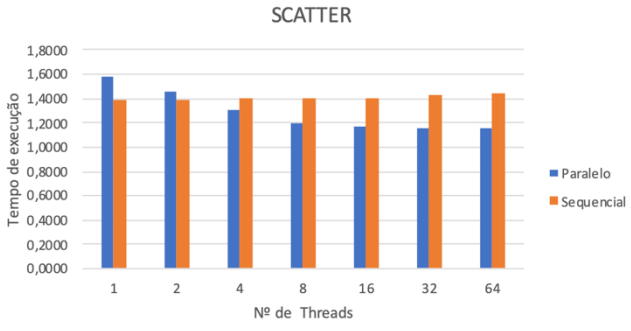

Fig. 6. Análise do padrão SCATTER com um tamanho de array de 100 milhões.

2.7 PIPELINE

Implementação: a única forma que se conseguiu paralelizar o Pipeline foi utilizando os conceitos referidos em [2] onde essencialmente se removem as dependências com recurso a diagonais, usando como referência o pseudo-codigo referido na pág. 39 de [2].

Temos assim dois ciclo *for*, onde o exterior itera o total de diagonais, e o ciclo *for* interior itera as etapas a realizar. A paralelização foi atingida com a utilização da diretiva *#pragma omp parallel for oredered* no ciclo interior. O ciclo exterior, como refere [2] não é possível paralelizar. Para que as tarefas fossem executadas ordenadamente, além da flag *ordered* na diretiva do *parallel for*, é necessário adicionar a diretiva *#pragma omp ordered* junto à tarefa a realizar.

Análise de resultados: Comparado com os outros padrões, a nossa versão do Pipeline não apresenta ganhos, porque como já foi referido anteriormente, e como está em [1], o Pipeline é maioritariamente utilizado em tarefas de ordem sequencial, sendo mais útil ser utilizado com outros padrões. Preferimos manter esta versão porque entendemos que seria importante implementar um Pipeline "puro" com pior desempenho, em detrimento de uma implementação com características de Map, mas com tempos de execução melhor, como realizado por alguns grupos.

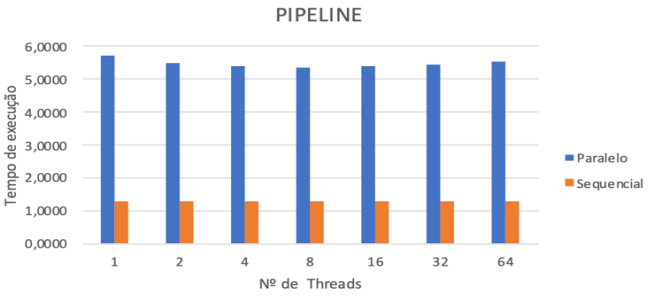

Fig. 7. Análise do padrão PIPELINE com um tamanho de array de 100 milhões.

2.8 FARM

Implementação: O padrão Farm, consiste na distribuição de tarefas de um *thread master* a vários workers disponíveis. Para atingir este objetivo a nossa implementação consiste numa primeira fase, em calcular o número de tarefas a atribuir a cada worker, para que a atribuição de tarefas seja feita de uma forma equilibrada, e numa segunda fase, a atribuição de uma tarefa pelo *thread* master a cada *worker*. Esta criação e atribuição de tarefas é conseguida através da diretiva *#pragma omp task*.

Estas tarefas consistem na aplicação de uma função a um elemento de uma coleção, sem qualquer dependência sobre outros elementos e sempre escrita numa posição diferente do array de *output*, sendo simples paralelizar com a diretiva *#pragma omp parallel*.

Análise de resultados: observando o gráfico, verifica-se que os ganhos são logo visíveis a partir da utilização de 2 *threads*, obtendo o melhor resultado com a utilização de 16 *threads*. Tal como no padrão MAP e Reduce, a utilização de 32 e 64 *threads* não apresentam ganhos de tempo de execução.

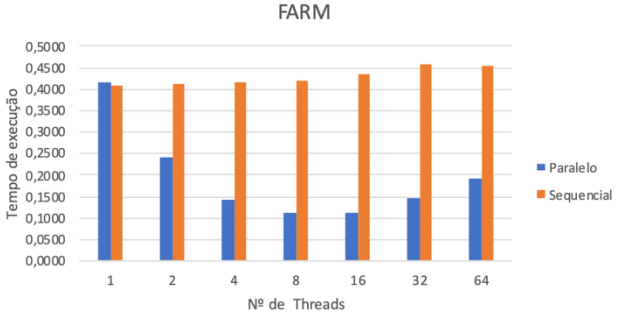

Fig. 8. Análise do padrão FARM com um tamanho de array de 100 milhões.

3 ANÁLISE DE RESULTADOS

3.1 Speedup

A avaliação do *speedup* visa comparar e calcular o rácio entre o tempo despendido na computação da mesma operação, entre um *worker* ou 'n' *workers*. Assim o *speedup* é calculado da seguinte forma: $speedup = Sp = T_1 / T_p$, segundo o cap. 2 de [1]. Sendo T_1 a latência despendida por um *worker* e T_p a latência despendida por 'p' *workers*.

Na análise dos resultados, verifica-se que o padrão que apresenta melhores resultados de speedup é o Reduce, seguido do Map e Farm. O padrão que apresenta pior resultado de speedup é o Scan, porque a simplicidade das operações não justifica o investimento realizado na criação das *threads*. Verifica-se ainda perda de speedup a partir de 16 threads de todos os padrões devido às características naturais da máquina onde foram realizados os testes (8 cores/16 threads), aliadas às operações aritméticas relativamente simples, não compensando o investimento na criação de 32/64 threads.

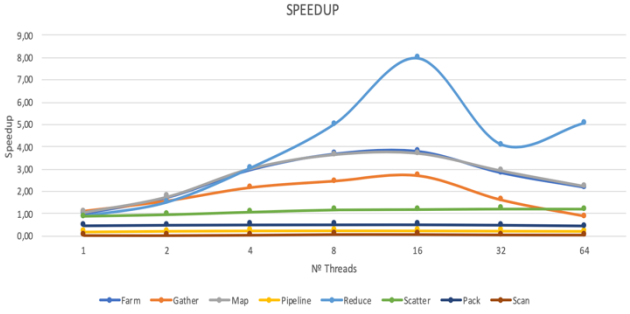

Fig. 9. Análise da métrica speedup com um tamanho de array de 100 milhões, para todos os padrões.

3.2 Eficiência:

A eficiência apresenta o retorno do investimento de hardware, neste caso o nº de workers, segundo o cap. 2 de [1]. Assim a eficiência é calculada da seguinte forma: $eficiência = S_p / P$. Sendo S_p o speedup e P o total de 'p' *workers*.

A eficiência decresce de uma forma geral em todos os padrões, mas este decréscimo é mais notório no padrão Reduce quando passa utilizar 32 threads em vez de 16, levando a uma quebra de eficiência de aproximadamente 40%. Outro padrão que sofre uma quebra mais acentuada na eficiência é o Map quando deixa de utilizar 8 threads em vez de 4.

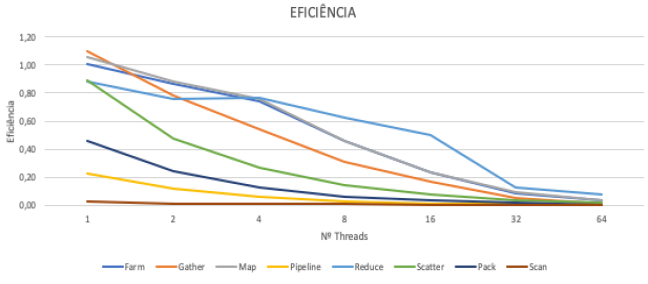

Fig. 10. Análise da métrica eficiência com um tamanho de array de 100 milhões, para todos os padrões.

3.2 Custo:

O custo visa determinar o valor despendido para que se possam ter ganhos em *speedup*, segundo o cap. 2 de [1]. A fórmula do custo é a seguinte: $custo = p \; x \; T_p$, sendo p o total de *workers* e T_p a latência despendida pelos *workers*. Os padrões que apresentam maior custo, para que se possam ter ganhos em *speedup* é o Scan e Pipeline.

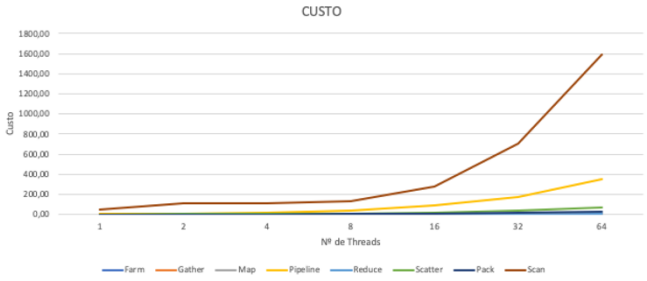

Fig. 11. Análise da métrica custo com um tamanho de array de 100 milhões, para todos os padrões.

4 CONCLUSÃO

Entendemos que o objetivo do projeto foi alcançado. Embora com expectativas diferentes das iniciais, nomeadamente, pensámos inicialmente que todos os padrões na sua versão paralela, iriam apresentar grandes ganhos em termos de *speedup* em relação à implementação sequencial, no entanto, como demonstrado anteriormente, nem todos os padrões obtiveram ganhos de speedup quando transformados numa versão paralela. Esta situação ocorre devido a vários fatores: natureza e aptidões de alguns padrões para operações mais de índole sequenciais; ora por outros fatores relacionados com a framework, nomeadamente *overheads* criados pelo carregamento das *libary* que a framework utiliza, o instanciamento de threads, ou gestão do *locks*. [4]

Mas esta situação levou-nos a refletir que a paralelização é um desafio constante e que realmente existem situações que os algoritmos sequenciais têm melhor desempenho que os algoritmos paralelos, cabendo ao utilizador a escolha da melhor versão face ao que se pretende implementar.

A paralelização é um desafio que estará presente na vida de qualquer programador, pois a otimização de problemas é transversal em várias áreas, pelo que a utilização da *framework* OpenMP permitiu-nos obter conhecimento sobre a tipologia deste tipo de *framework*, que desconhecíamos até este momento.

Entendemos também que este foi um dos projetos mais completos que realizamos durante o curso, pois permitiu-nos colocar em prática o *workflow* do *git*, programação em C, utilização do cluster do DI, utilização de ferramentas como o *PERF* e *Valgrind*, teste e análise de resultados, portanto, muito ambíguo e completo.

Achamos igualmente que o projeto esteve muito bem estruturado, timings acertados, com regras claras e justas, contribuindo para uma motivação constante em realizar um bom trabalho.

5 AGRADECIMENTOS

Gostaríamos de agradecer ao grupo 1 (Jorge Pereira e Joana Parreira) pela disponibilidade demonstrada relativamente à resolução de problemas relacionados com o cluster do DI; e ainda ao grupo 6 (Pedro Pais e Miguel Figueira) pelas ideias partilhadas ao longo do projeto, nomeadamente na implementação do Pipeline.

5 DIVISÃO DO TRABALHO

Ambos os alunos realizaram quatros padrões respetivamente, e nomeadamente:

- Francisco Cardoso: Scan, Pack, Gather e Farm;
- Nelson Santos: Map, Reduce, Pipeline e Scatter;

Em virtude da maior disponibilidade do colega Nelson Santos, entendemos que se deva beneficiar e premiar a maior dedicação, nomeadamente na elaboração do relatório e realização de testes no cluster do DI, pelo que entendemos que seja justa a seguinte proporção:

- Francisco Cardoso: 40%.
- Nelson Santos: 60%.

REFERÊNCIAS

[1] M. McColl, A. D. Robison, J. Reinders, *Structured Parallel Programming*. MK, 2012.

[2] Y. Solihin, *Fundamentals of Parallel Computer Architecture*. Solihin Books, 2009.

[3] J. Lourenço, "Parallel Algorithms," *Concurrency and Parallelism*, 2020.

[4] Intel, IBM, "Performance Obstacles for Threading: How do they affect OpenMP code?" *MUD History*, https://software.intel.com/content/www/us/en/develop/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code.html. 2009.