OpenMP em Direção à Aproximação: Loop Perforation e Multithreading

João B. Oliveira[1], **Rogério A. Gonçalves**[1], **João Fabrício Filho**[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87.301-899 – Campo Mourão – PR – Brazil

joliveira.2022@alunos.utfpr.edu.br,{rogerioag,joaof}@utfpr.edu.br

Abstract. *This study investigates the integration of loop perforation into OpenMP to enhance the performance of applications on systems with limited computational resources. Four perforation techniques –* init*,* fini*,* large*, and* small *– were implemented, with experimental results demonstrating significant performance gains across various test configurations. The results indicate performance improvements up to* 43.33% *with the proposed techniques.*

Resumo. *Este estudo investiga a integração do* loop perforation *ao* `OpenMP` *para melhorar o desempenho de aplicações em sistemas com recursos computacionais limitados. Foram implementadas quatro técnicas de* perforation *–* init*,* fini*,* large *e* small *– com resultados experimentais que demonstram ganhos significativos de desempenho em várias configurações de teste. Os resultados indicam ganhos de até* $43,33\%$ *com as técnicas propostas.*

1. Introdução

A *Computação Aproximada* (CA) é um paradigma que explora a tolerância à imprecisão para obter ganhos de desempenho e eficiência energética em diversas aplicações, especialmente em áreas como Aprendizado de Máquina, Processamento de Sinais e Busca na Web. [Que et al. 2023, Che et al. 2009] A utilização do paradigma de aproximação em compiladores e ferramentas de alto desempenho vem sendo explorada, incluindo técnicas como o *loop perforation*. [Parasyris et al. 2021]

Loop perforation é uma técnica de otimização, que consiste em saltar algumas iterações do *loop* para reduzir o tempo de execução do programa em troca da perda de precisão. A quantidade de iterações a serem puladas e a lógica para determinar quais iterações são puladas variam de acordo com a implementação específica da técnica. O desafio é alcançar um equilíbrio entre o ganho de desempenho e a precisão do resultado final.

Este trabalho explora *loop perforation* em um ambiente de execução paralelo a fim de obter desempenho. Para tanto, expandimos as funcionalidades do `OpenMP` para o `LLVM/Clang` e introduzimos um novo construtor `approx` e estendendo o *runtime* do `OpenMP` para comportá-lo.

Nossos resultados mostram um ganho de até 43,33% na quantidade de ciclos da aplicação ao usar a técnica de *perforation*, com uma perda de precisão de 33,94%. Acre-

Agradecimentos à Fundação Araucária pelo auxílio financeiro e concessão de bolsa - Projeto 01947 - SISPEQ/UTFPR.

ditamos que esse trabalho contribua para o aprimoramento do desempenho em aplicações de dados aproximados.

Além disso, destacamos que nossa pesquisa vai na direção de extensões mais amplas do OpenMP, com a incorporação de outras técnicas de aproximação, como memoização e descarte de tarefas, que têm potencial de proporcionar melhorias ainda mais significativas no desempenho de aplicações paralelas.

2. Implementação

A integração do *loop perforation* no OpenMP foi realizada por meio da criação do construtor *approx*. Com esse construtor, o programador informa ao compilador quais regiões que devem ser aproximadas. O Código 1 apresenta o uso em conjunto com o construtor *for*, a cláusula *perfo* especifica o tipo de algoritmo e a taxa de *perforation*. Essa abordagem facilita a integração do paralelismo do OpenMP e também fornece uma interface comum para a implementação de novas técnicas de aproximação dentro do *framework*, utilizando extensões.

Código 1. Sintax da cláusula perfo

```
1 #pragma omp approx for [perfo]...
```

O processamento desta diretiva ocorre em duas etapas. Na primeira etapa, o código é analisado e transformado na fase de compilação. Por ser uma anotação no código, o compilador consegue identificar a região que deve ser aproximada, enquanto a cláusula *perfo* especifica a técnica e a taxa que será aplicada para a aproximação daquela região. Na segunda etapa, durante a execução do programa, ocorrem chamadas do *runtime*, responsáveis por distribuir as iterações do laço em execução entre as *threads* criadas na região paralela, conforme é comum em aplicações OpenMP. Assim há controle de quais iterações serão saltadas ou desprezadas.

Foram escolhidas quatro técnicas de *perforation* para implementação. A seleção baseou-se em algoritmos que dependem apenas do estado inicial do *loop*, demandando menos do *runtime* do que técnicas dinâmicas. A primeira técnica, denominada *init*, salta as primeiras interações especificadas pelo usuário. A segunda, chamada *fini*, realiza o salto das últimas iterações especificadas pelo usuário. A terceira, *large*, aumenta o valor da variável de indução conforme a taxa especificada pelo usuário. Por fim, a técnica *small* realiza saltos com base na Equação 1, no qual i é a variável de indução do *loop*, n é o número especificado pelo usuário que define a periodicidade da iteração e k é uma constante com valores entre $0 \leq k < n$.

$$i = i + 1 + (i\%n == k?1 : 0) \tag{1}$$

3. Configurações do Experimento

O Sistema Operacional utilizado realizar os testes foi GNU/Linux, especificamente a distribuição Ubuntu 22.04.4 LTS. O sistema executa em um processador Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz com 4 núcleos físicos e suporta 8 *threads*, 32GB de RAM e um SSD de 1TB.

A aplicação selecionada foi a *particle filter*, componente do *Rodinia Benchmark* [Che et al. 2009]. A aplicação estima a localização de um objeto, mesmo com medições ruidosas e com entendimento limitado do seu movimento.

O experimento consiste na execução da aplicação com sua configuração padrão com quatro variações dos algoritmos de *perforation*, cada uma utilizando 1, 2, 4 ou 8 *threads*, com entradas variando de *tiny* (pequeno), *train* (médio) e *test* (grande), e sendo repetidas dez vezes em cada configuração. A taxa de *perforation* foi limitada em 25% do tamanho do *loop* em todos os testes.

4. Resultados

A Figura 1 mostra a redução nos ciclos de `CPU` com o uso de algoritmos de aproximação em comparação com a aplicação padrão: *fini* obteve ganho de $76,47\%$, *init* de $43,33\%$, e *large* de $53,25\%$, enquanto *small* apresentou uma perda de desempenho de cerca de $351,65\%$. Isso se deve à necessidade de verificações repetidas durante a execução, exigindo intervenções frequentes do *runtime* para permitir a perfuração do código. No entanto, os benefícios de desempenho em termos de ciclos mostraram-se inversamente proporcionais à precisão das saídas, revelando uma baixa acurácia na identificação dos pontos, especialmente quando nas entradas menores. Esse cenário resulta em uma degradação na qualidade da saída devido à excessiva perfuração do código.

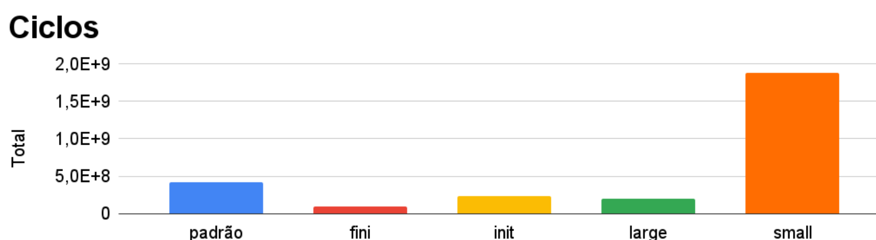

Figura 1. Média de ciclos de CPU por execução da aplicação com técnicas de *loop perforation*.

A Figura 2 apresenta o desempenho das execuções de cada programa com uma entrada diferente, destacando o erro médio das saídas calculado pelo MAPE (*Mean Absolute Percentage Error*). Os valores de x são representados em azul, os valores de y em vermelho e, por fim, a distância é mostrada em verde. De modo geral, tanto os valores de x quanto os de y desviaram consideravelmente dos resultados esperados, com exceção do algoritmo *small*, que permaneceu mais próximo dos resultados originais em todas as execuções. Além disso, observa-se que o erro na distância foi menos significativo em comparação com os erros nos pontos individuais.

À medida que o tamanho dos mapas utilizados como entrada na aplicação aumentava, os erros também se ampliavam, resultando em distorções significativas em relação aos valores originais. Conforme ilustrado na Figura 2, os algoritmos *fini*, *init* e *large* exibiram discrepâncias consideráveis em suas saídas quando comparados com o *small*. Importante ressaltar que, de maneira geral, os algoritmos apresentaram desempenho superior à medida que o tamanho do mapa de entrada aumentava, seguindo um padrão semelhante ao observado em outras aplicações. Tanto o algoritmo *init* quanto o *fini* demonstraram margens de erro similares, refletindo a natureza de suas implementações. Por

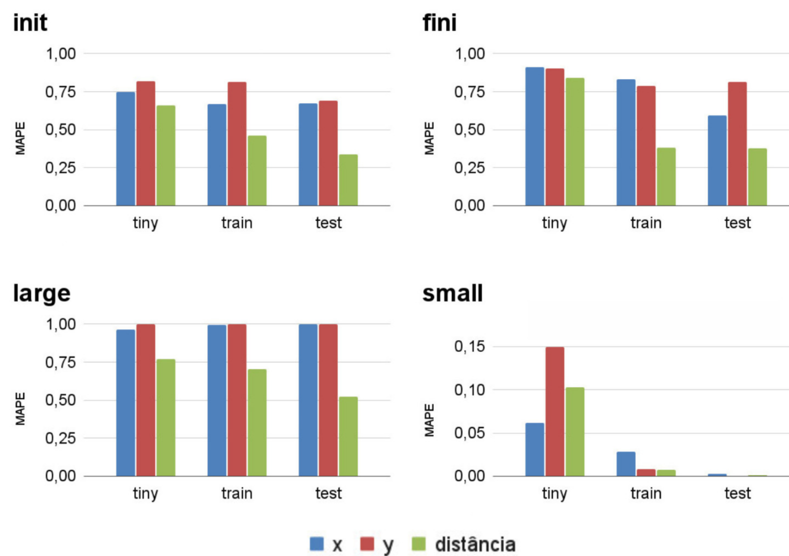

Figura 2. Erro percentual absoluto médio da saída para cada tamanho de entrada.

outro lado, o algoritmo *large* registrou os piores resultados, com saídas apresentando erros médios mais elevados em comparação com as demais aplicações. Em contrapartida, o *small* alcançou taxas de erro menores, chegando até mesmo a apresentar erros próximos a zero nas saídas de teste.

5. Considerações Finais

Este estudo apresentou a viabilidade e o potencial do *loop perforation* integrado ao `OpenMP` para melhorar o desempenho de aplicações em sistemas com recursos computacionais limitados. Os resultados dos experimentos evidenciam ganhos de desempenho em várias configurações de teste. No entanto, reconhecemos as limitações do *loop perforation* em diferentes tipos de aplicações, e sua eficácia em cenários específicos. Para futuras pesquisas, recomendamos explorar a implementação de técnicas adicionais para melhorar o desempenho em diferentes contextos de aplicação, bem como investigar estratégias para mitigar as limitações identificadas neste estudo.

Referências

[Che et al. 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54.

[Parasyris et al. 2021] Parasyris, K., Georgakoudis, G., Menon, H., Diffenderfer, J., Laguna, I., Osei-Kuffuor, D., and Schordan, M. (2021). HPAC: evaluating approximate computing techniques on HPC OpenMP applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, St. Louis Missouri. ACM.

[Que et al. 2023] Que, H.-H., Jin, Y., Wang, T., Liu, M.-K., Yang, X.-H., and Qiao, F. (2023). A Survey of Approximate Computing: From Arithmetic Units Design to High-Level Applications. *Journal of Computer Science and Technology*, 38(2):251–272.