Impacto da biblioteca padrão do C++ nos Kernels do NAS Parallel Benchmarks

Leonardo Mallmann[1], **Arthur Bianchessi**[1], **Dalvan Griebler**[1]

[1] Escola Politécnica, Grupo de Modelagem de Aplicações Paralelas (GMAP), Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, Brasil

leonardo.mallmann@edu.pucrs.br, arthur.bianchessi@gmail.com, dalvan.griebler@pucrs.br

Resumo. *A programação paralela nativa na linguagem C++ ganhou força com os std algorithms e suas políticas de execução paralela. Para que seja possível a aplicação destes recursos, porém, é necessário a incorporação no código das estruturas de dados sobre as quais tais funções possam operar. Mesmo adicionando uma camada de abstração maior através de tais estruturas, observou-se um tempo de execução similar à versão em C.*

1. Introdução

O NAS Parallel Benchmarks (NPB) é um conjunto de aplicações desenvolvido e mantido pela divisão de supercomputação da NASA para avaliar o desempenho de arquiteturas paralelas [Bailey et al. 1994]. O NPB contêm cinco *kernels* e três pseudo aplicações que simulam o comportamento de modelos de fluidodinâmica computacional e relacionados com ciência aeroespacial. Além disso, o NPB conta com cargas de trabalho pré-definidas que simulam computações realistas a fim de testar diferentes características computacionais. Os códigos surgiram em meados da década de 90, quando a linguagem predominante para computação de alto desempenho era o Fortran 90. Desde então, o NPB tem sido utilizado em diferentes domínios de pesquisa, principalmente para avaliar estratégias, algoritmos e ferramentas. Dentro desta crescente utilização dos algoritmos NPB, destaca-se a aplicação das técnicas de paralelismo sobre os códigos através de diferentes frameworks.

Fortemente impulsionado pelo próprio movimento de modernização das linguagens e evolução das ferramentas dentro do ecossistema de HPC (High Performance Computing), o conjunto de códigos acabou sendo disponibilizado na linguagem C++ (Raw C) no artigo [Griebler et al. 2018], no qual os autores realizaram a transcrição dos 5 *kernels* da linguagem Fortran para C++, com o objetivo de paralelizar os códigos com interfaces de programação paralela escritas em C++. Ainda que transcrito para a nova linguagem, o código ainda está muito próximo de uma implementação escrita utilizando apenas C. Por exemplo, no código ainda são encontradas estruturas e rotinas do C como `malloc`, `printf`, `char*` e outros. Posteriormente, os autores introduziram uma alternativa com versões mais recentes das aplicações do NPB chamada de NPB-CPP [Löff et al. 2021], na qual os autores introduziram otimizações para aumentar a portabilidade através de estruturas lineares de dados e alocação dinâmica. Nosso trabalho parte desse estudo dos códigos na versão C++ e evolui a conversão ao focar em estruturas de dados e métodos presentes na biblioteca padrão da linguagem (*C++ Standard Library*). Focou-se em usufruir de todas as funcionalidades na biblioteca padrão do C++ para representar as estruturas e gerenciar os dados, entrada e saída de tela, arquivos e ponteiros inteligentes.

Portanto, a motivação de transcrever o conjunto para a linguagem C++ se baseia na curiosidade sobre o impacto na eficiência que uma linguagem com uma série de abstrações traz. Além disso, muitas das ferramentas de paralelismo da atualidade como SkePU, Nvidia CUDA, HPX, Thrust e a própria C++ STL Library são baseadas no C++, sendo assim, é do interesse dos desenvolvedores destes frameworks a disponibilização dos benchmarks para construir uma nova avaliação. Este artigo apresenta na Seção 2 como foi feita a conversão. Posteriormente, a Seção 3 descreve e discute os resultados alcançados. Por fim, as conclusões são descritas na Seção 4.

2. Conversão do NPB para o C++ Standard

A transcrição realizada neste trabalho situou-se na compreensão dos códigos do conjunto, conhecimento das funções e estruturas disponíveis na biblioteca padrão do C++ e como essas poderiam ser incorporadas dentro da lógica atual de cada código. A parte que requer mais tempo de análise dentro dos códigos é a manipulação de porções contíguas na memória, os *arrays*, e a forma como eram feitas suas referências em funções através do uso dos ponteiros. Inclusive, um grande empenho na transcrição foi direcionado a mitigar o uso deste operador, pois o mesmo se torna perigoso dentro do contexto de gerenciamento da memória e atualmente existem estruturas que controlam o ciclo de vida dos mesmos. Durante as etapas da conversão, observou-se o quanto a incorporação das novas estruturas e métodos presentes na biblioteca padrão impactam na eficiência, através da análise do tempo de execução. Para evidenciar o comportamento dos *kernels* junto de tais estruturas e métodos, será feita uma comparação entre o tempo de execução das versões C (Raw C) e C++ (STD) sendo compiladas nas ferramentas *CLANG* e *GCC*.

O processo de transcrição consistiu na análise prévia de cada *kernel* para identificar funções e estruturas que poderiam ser substituídas por equivalentes na biblioteca padrão do C++ como funções de leitura de arquivo, *outputstream* e estruturas primárias que ainda não existiam como a *string*. Após isto, procurava-se pelas funções de alocação contígua de memória e identificação do ciclo de vida dos ponteiros através das funções.

Após ter-se uma visão geral de todos os 5 *kernels* do conjunto, foi possível identificar que todos eles, com exceção do *EP*, lidam com acessos multidimensionais na memória. Para isso declara-se globalmente um ponteiro através da função *malloc* e, dentro das sub-rotinas, realizava-se uma conversão na forma de acesso ao espaço de memória alocado através de uma operação no ponteiro, tornando-o bidimensional ou tridimensional.

Visualizando o código como um todo, ainda haviam estruturas condicionais que mudavam o tipo de alocação entre dinâmica e estática, dependendo da escolha do usuário. Tendo consciência dessa opção, usou-se a estrutura *std::vector* para realizar a alocação contígua, pois com esta é possível pré definir o tamanho na compilação e/ou delegar a expansão de memória para a estrutura a medida que se aproxima da sua capacidade máxima.

Observando a biblioteca padrão da linguagem, notou-se que não havia uma estrutura nativa para representar *arrays* de acesso multidimensional. Para contornar o impasse, surgiu a ideia de desenvolver duas versões para cada *kernel*: uma versão com acesso linear à memória, com a garantia de acesso coalescido à memória, e outra versão onde o acesso é multidimensional, abstraído por uma classe que tem como base estruturas do

tipo *std::vector* aninhadas. No código abaixo é apresentada uma das três classes genéricas criadas:

```
template <typename T>
class Matrix2D {
    public:
        int rows, columns;

        std::vector<std::vector<T>> matrix;

        std::vector<T>& operator[] (int index) {
            return matrix[index];
        };
};
```

Código 1. Visualização da sobrecarga do operador *[]* na classe Matrix2D.

Esta classe genérica faz uso da sobrecarga do operador [] para que seja possível acessar a estrutura de vetores aninhados que estão declarados dentro do objeto. Para transitar os *containers* entre as sub-rotinas e permitir a modificação dentro destas, utilizou-se a passagem por referência, em que é passado para a função o objeto, o qual é recebido com o operador & indicando que não é para ser criada uma cópia e sim para manter a referência para o objeto original. Desta forma, foi possível contornar a maioria dos casos de utilização dos ponteiros e incorporar o *std::vector* para que possam ser utilizadas as funções de alta ordem sobre o mesmo.

3. Experimentos

Para observar o impacto da incorporação das estruturas no código foram realizados testes que permitem a comparação entre a versão antiga, sem o uso das estruturas da biblioteca *std*, e a versão implementada neste trabalho. Todos os experimentos foram executados em uma máquina com um processador do modelo Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz e memória RAM de 188Gb. Todos os resultados apresentados provém da média de 10 execuções para cada *kernel* e pseudo aplicação, com a carga referente à classe B. Todos *kernels* foram compilados, tanto no uso do GCC quanto do CLANG, com a flag de otimização -O3, a qual, em ambos compiladores, se refere ao maior nível de otimização.

		NPB-CPP		Linear		Matriz		Matrix c/ Std::Complex		Std::Complex	
	Compilador	Tempo	Desvio Padrão	Tempo	Desvio Padrão	Tempo	Desvio Padrão	Tempo	Desvio Padrão	Tempo	Desvio Padrão
EP	Clang	72.208	0.3487	73.981	0.3293						
	GCC	90.883	0.1668	92.662	0.1663						
FT	Clang	58.363	0.3169	65.588	3.4350	45.762	0.2368	51.794	0.1909	73.289	0.2341
	GCC	41.228	0.2470	43.59	0.3043	33.916	0.1247	45.029	0.2316	54.089	0.3139
CG	Clang	111.22	0.3246	43.118	0.0644	43.124	0.0570				
	GCC	51.023	3.0635	51.597	2.7990	49.706	4.5744				
MG	Clang	4.516	0.0052	4.944	0.0052	4.739	0.0099				
	GCC	4.603	0.0506	5.093	0.0327	5.293	0.0316				
IS	Clang	2.078	0.0042	2.152	0.0063	2.098	0.0103				
	GCC	2.109	0.0074	2.129	0.0120	2.12	0.0149				

Tabela 1. Tempo de execução dos 5 kernels e suas versões entre os dois compiladores: CLANG e GCC

Na tabela 1, onde é apresentado o tempo de execução dos cinco *kernels*, é possível perceber que todas versões que fizeram o uso da classe Matrix, com exceção do *kernel* FT, obtiveram um desempenho muito parecido em relação às versões lineares, cujo ponto

era de preocupação, pois suspeitou-se da contiguidade dos elementos na memória quando usou-se a abstração com o *arrays* aninhados. Sobre o *kernel* FT, observou-se no teste de cache, realizado com a ferramenta *valgrind*, que houve um melhor aproveitamento do primeiro e último nível de cache. Ainda neste código, onde haviam computações de números complexos, foi possível observar uma grande deficiência no módulo *std::complex*, mais especificamente na operação de multiplicação entre números complexos, realizada através da sobrecarga do operador "*".

Observando o tempo de execução entre os dois compiladores, notou-se uma maior diferença para esta métrica nos *kernels* FT e EP. No FT, foi observado um melhor desempenho do compilador GCC, acredita-se que pelo melhor aproveitamento da memória cache. Ao rodar os experimentos sobre a memória cache para este *kernel*, percebeu-se que o compilador CLANG gerou um código menos otimizado com quase 100.000.000.000 instruções a mais e também com mais do que o dobro de leituras ao primeiro nível de cache.

4. Conclusões

Neste trabalho foi explorado a aplicação das estruturas e funções presentes na biblioteca padrão do C++, visando mitigar nos códigos transcritos antigas práticas do C++ (Raw C) como o uso de ponteiros, principalmente, e incorporar os *std containers*, sobre os quais é possível aplicar os algoritmos com políticas de execuções paralelas do C++. Os resultados mostraram que, apesar da incorporação de estruturas que trazem uma maior camada de abstração, percebeu-se tempos de execução não muito destoantes da versão antiga. Para trabalhos futuros, será mapeada a implementação da nova estrutura *std::mdspan*, a qual, partindo de um *std container* unidimensional, cria uma abstração para que o acesso se torne multidimensional. Além disso, serão exploradas as políticas de execução paralelas que possam ser aplicadas aos algoritmos presentes na biblioteca *std*. Por fim, visando compreender todo o conjunto NPB, também espera-se aplicar estas estruturas e os estudos deste artigo nas pseudo-aplicações do conjunto.

Referências

Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. (1994). The NAS Parallel Benchmarks. Technical report, NASA Ames Research Center, Moffett Field, CA - USA.

Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018). Efficient nas benchmark kernels with c++ parallel programming. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 733–740.

Löff, J., Griebler, D., Mencagli, G., Araujo, G., Torquati, M., Danelutto, M., and Fernandes, L. G. (2021). The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757.