Conversão do NAS Parallel Benchmarks para C++ Standard

Arthur S. Bianchessi, Leonardo Mallmann, Renato B. Hoffmann, Dalvan Griebler

[1] Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

(arthur.bianchessi, leonardo.mallmann, renato.hoffmann)@edu.pucrs.com,
dalvan.griebler@pucrs.br

Resumo. *A linguagem C++ recebeu novas abstrações de paralelismo com a definição das políticas de execução dos algoritmos da biblioteca padrão. Entretanto, a adequabilidade e o desempenho dessa alternativa ainda necessita ser estudado em comparação com outras alternativas bem estabelecidas. Portanto, o objetivo deste trabalho foi explorar a vasta gama de opções de recursos da biblioteca padrão C++ para avaliar a aplicabilidade e desempenho a partir de cinco kernels do NPB. Através dos experimentos em um ambiente multithreaded, foi constatado que a incorporação de estruturas de dados da biblioteca padrão, assim como a abstração para acesso multidimensional criada, não apresentam impacto notável no tempo de execução. Já os algoritmos com políticas de execução paralela demonstraram uma perda de desempenho estatisticamente significativa.*

1. Introdução

O padrão da linguagem C++ desenvolveu um ecossistema abrangente de paralelismo, incluindo políticas de execução paralela para algoritmos nativos da biblioteca [Josuttis 2012]. Disponíveis em sua maioria a partir do C++ 2017, os algoritmos permitem que o usuário se utilize de um paradigma funcional para definir funções recorrentes como transformações, iterações, reduções, ordenamentos, buscas, entre outros. Esses algoritmos permitem definir uma política de execução que pode ser sequencial, vetorial, paralela e paralela vetorial. A partir disso, a biblioteca padrão do C++ é capaz de abstrair o código paralelo de baixo nível gerado com Intel oneTBB (Threading Building Blocks). Entretanto, como ainda é uma solução relativamente recente, ainda são necessários estudos para comprovar sua eficácia.

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]. Os códigos que simulam o comportamento de modelos de fluidodinâmica computacional surgiram em meados da década de 90 e foram desenvolvidos, e mantidos até hoje, em Fortran.

Apesar do NPB já existir há mais de duas décadas, ele ainda é popular no segmento de HPC para testes de hardware e também de software. Considerando que muitas novas APIs e bibliotecas para paralelismo como SkePU, Nvidia CUDA, HPX e Thrust são feitas em C++ com o uso das abstrações mais recentes da biblioteca. Portanto, surge a necessidade de obter uma versão do benchmark que faz a utilização de tais estruturas e funções nativas presente a partir da versão C++ 17. Devido a sua maturidade e grande volume de trabalhos científicos envolvidos, o NPB pode servir como uma base estabelecida de comparação para novas técnicas e algoritmos de paralelismo.

Versões paralelas do NPB estão disponíveis em OpenMP para arquitetura de memória compartilhada e MPI (Message Passing Interface) para arquiteturas distribuídas. Desde então, o NPB tem sido utilizado em diferentes domínios de pesquisa, principalmente para avaliar estratégias, algoritmos e ferramentas de paralelismo. Apesar do ganho de popularidade do C++ em virtude da maior completude e abrangência da sua biblioteca padrão, nenhuma conversão completa do NPB foi feita para o C++ com a incorporação das novas abstrações.

Dito isso, o principal objetivo deste trabalho foi obter uma nova versão do NPB com o intuito de avaliar as estratégias do C++ moderno em relação a 2 critérios: (1) adequabilidade para cálculos matemáticos presentes no NPB e (2) desempenho. Para tal, inicialmente foi realizada uma conversão da versão sequencial utilizando abstrações do padrão da linguagem como `complex` e as representações de matrizes multidimensionais utilizando `vector`. A partir dessa versão, foram adicionados algoritmos de execução padrão do C++ em combinação com políticas de execução paralelas. Sendo assim, as contribuições científicas foram listadas abaixo:

1. Análise das funcionalidades e desempenhos das estruturas da biblioteca padrão do C++ nos kernels NPB.
2. Análise do desempenho do paralelismo dos *algorithms* com políticas de execução paralela nos kernels do NPB.
3. Comparação dos novos códigos portados com a versão C já existente.

Dados os objetivos apresentados, o restante do artigo está organizado como segue. A seção 2 pretende descrever a literatura acerca do NPB e suas transcrições prévias e também sobre o uso da biblioteca STL na contexto de HPC. Na sequência, a Seção 3 descreve o processo de portar o código para estruturas modernas de dados, as dificuldades encontradas e a motivação de certas escolhas, como o porquê de criar versões lineares e matriciais, bem como a explicação de como as novas estruturas funcionam. Na seção de 5, será apresentado um comparativo dos resultados obtidos em cada *kernel* em relação a uma versão prévia dos benchmarks na linguagem C++. Ao final, a seção 6 traz um breve resumo do que foi feito no trabalho, as ferramentas exploradas dentro da biblioteca padrão e um resumo dos resultados obtidos.

2. Trabalhos Relacionados

No artigo [Griebler et al. 2018], os autores realizaram a transcrição dos 5 *kernels* da linguagem Fortran para C++. O objetivo era paralelizar o NPB com interfaces de programação paralela para avaliação de desempenho. Contudo, a versão disponibilizada não utiliza funcionalidades da biblioteca padrão do C++. Posteriormente, foi introduzida uma proposta alternativa com versões mais recentes das aplicações do NPB chamada de NPB-CPP [Löff et al. 2021]. Nessa foram introduzidas otimizações para aumentar a portabilidade através de estruturas lineares de dados e alocação dinâmica de memória. Este trabalho parte da base de código em C++ para incorporar as estruturas de dados e algoritmos paralelos presentes na biblioteca padrão do C++ (*C++ Standard Library*) visando medir o impacto que tais abstrações causam no tempo de execução de cada kernel.

Outro trabalho [Di Domenico et al. 2022] também partiu dos códigos do NPB C++ [Löff et al. 2021]. Contudo, ele focou na tradução do código sequencial e paralelo

para a linguagem de programação Python. Além disso, os autores conduziram experimentos para analisar o desempenho e esforço de programação. Apesar de explorar outra linguagem, foram utilizadas bibliotecas da linguagem Python para analisar como o uso do ecossistema do Python impacta no desempenho em comparação com Fortran e C.

O artigo [Seo et al. 2011] focou na análise de desempenho das aplicações do NPB paralelizadas com o uso da ferramenta OpenCL para GPU. Para isso, os autores partiram da versão Fortran paralelizada com OpenMP e realizaram a conversão para a linguagem C também utilizando o OpenMP. O objetivo dos autores era obter uma versão do NPB para testar estratégias de programação em OpenCL e avaliar a sua eficiência. Desta forma, o trabalho não apresentou detalhes da conversão e nem todas as versões executaram cargas de trabalho maiores. Recentemente, [Do et al. 2019] estendeu o trabalho anterior apresentando otimizações nos códigos paralelos em OpenCL e CUDA. O artigo não apresenta detalhes sobre as estratégias de conversão do código sequencial já que foca apenas nas estratégias paralelas. Diferentemente, este trabalho foca na estratégia e desempenho das estruturas e algoritmos padrão da linguagem C++.

No artigo [Mietzsch and Fuerlinger 2019], os autores comparam a expressão do paralelismo da biblioteca padrão do C++ com a biblioteca TBB. Para analisar a eficiência dos algoritmos na expressão do paralelismo, os autores comparam o tempo de execução de uma implementação própria dos *kernels* do NPB aplicando os `algorithms` e as versões em OpenMP e TBB desenvolvidas no artigo [Löff et al. 2021]. A análise que é proposta neste trabalho, contudo, se estende no processo de incorporação das estruturas da biblioteca padrão na base de código em C++, ressaltando possíveis *overheads* de tempo de execução e memória cache durante o processo. Além disso, conduziu-se uma análise sobre o desempenho das estruturas de alocação contígua ao comparar os tipos de acesso aos dados: memória coalescida ou multidimensional através do aninhamento de *containers*.

No artigo [Lin et al. 2022], os autores avaliam o desempenho dos algoritmos diante de três pseudo-aplicações, sobre os quais exploram diferentes alternativas para percorrer dados contíguos com os recursos da biblioteca STL (*biblioteca de templates padrão*) do C++17. Em sua análise de resultados, os autores comparam a eficiência da expressão de paralelismo em CPUs entre as ferramentas: *OpenMP*, *Kokkos*, *SYCL* e C++ STL versão 17. Outra contribuição da foi a utilização do compilador `nvc++` para gerar códigos paralelos para GPUs a partir da base de código em C++17 puro e, a partir disso, conduziram uma comparação com o paralelismo expresso através da ferramenta *CUDA*. Em nosso trabalho, foi avaliado o uso do C++ STL sobre o NPB além de conduzir uma análise acerca das estruturas que necessitam ser incorporadas antes do uso dos `std::algorithms` espaços contíguos de dados na memória.

3. Conversão do NPB para o C++ Padrão

Antes de utilizar os algoritmos com políticas de execução paralela do C++, a primeira etapa, descrita nesta Seção, envolve a preparação da base de código em relação, principalmente, a adesão de estruturas de dados contíguas classificadas na biblioteca padrão como *containers*. Contudo, antes de pensar na incorporação de tais estruturas, observou-se cada código procurando substituir funções e estruturas antigas em pontos não críticos do código como operações de input e output *stream*, leitura de arquivos e

outras abstrações básicas não presentes como *strings*. Na sequência, passou-se a observar a manipulação dos ponteiros (*raw pointers*), para identificar se estes apontavam para uma porção contígua de memória ou para uma variáveis apenas. O uso do ponteiro torna-se um grande alvo dentro da transcrição visto que estruturas modernas lidam com o problema de gerenciamento de alocação de memória automaticamente. Uma das práticas adotadas foi a utilização dos ponteiros inteligentes (p.e., `unique_ptr` ou `shared_ptr`) C++ para gerenciamento da memória. Outra substituição realizada foi de ponteiros por objetos de classes padrão como *string* e *outputstream*.

Todos *kernels*, com exceção do *EP*, lidam com acessos multidimensionais na memória. Para lidar com isso, foi necessário declarar globalmente um ponteiro e, dentro das sub-rotinas, realizar uma conversão na forma de acesso ao espaço de memória, tornando-o bidimensional ou tridimensional, como demonstrado no Código 1.

```
1      static dcomplex (*u0)=(dcomplex*)malloc(sizeof(dcomplex)*(
       NTOTAL));
2      //...
3      static void compute_indexmap(void* pointer_twiddle,
4                          int d1,int d2, int d3){
5   double (*twiddle)[NY][NX] = (double(*)[NY][NX])pointer_twiddle;
6           //...
7      }
8   };
```

Código 1. Manipulação do tipo de acesso à memória pelo ponteiro.

Depois das etapas anteriores, ainda havia estruturas condicionais que alternavam a alocação entre dinâmica e estática. Portanto, 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. Entretanto, surgiu um problema nos casos em que a memória referenciada pelo vetor era parcialmente manipulada em funções auxiliares. Como a estrutura *vector* não tinha nenhum *slice* adequado, optou-se por enviar o ponteiro intrínseco da classe *vector* que pode ser acessado através do método `data()`.

Quanto ao acesso multidimensional a um espaço contíguo, pesquisou-se na biblioteca padrão da linguagem por uma estrutura nativa para representar tal abstração. No entanto, não se obteve resultados, encontrando apenas uma referência para uma futura implementação no C++23 da estrutura `std::mdspan`. Logo, 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.

```
1  template <typename T>
2  class Matrix2D {
3      public:
4          int rows, columns;
5          std::vector<std::vector<T>> matrix;
6          std::vector<T>& operator[] (int index) {
7              return matrix[index];
8  };};
```

Código 2. Visualização da sobrecarga do operador *[]* na classe Matrix2D.

A sobrecarga do operador [] sobre a classe *Matrix2D* (Código 2) permite que seja acessada, através da lógica de índices sobre ponteiros, a estrutura intrínseca da classe que armazena os dados, neste caso a variável `std::vector<std::vector<T>> matrix`. Utilizando dessa abstração foi possível descartar a utilização de ponteiros e substituir pela declaração da estrutura de acesso multidimensional correspondente ao tipo de acesso da variável.

Para varrer *Matrix2D*, dentro dos *std::algorithms* de forma *data-oriented*, no entanto, é necessário o suporte ao uso de *iterators* para a classe. Para tanto, foi criada uma estrutura que entrega o suporte necessário, a qual será discutida com mais profundidade na próxima seção.

Ainda sobre a conversão em alguns pontos peculiares de cada kernel, destaca-se a utilização da estrutura de abstração de números complexos no kernel FT, `std::complex`. Na base de código em C, a abstração para o tipo de dado foi construída manualmente pelos autores, dando suporte para as operações de adição, subtração, divisão e multiplicação. Ao substituir a classe previamente criada pela disponibilizada pela biblioteca STD, no entanto, notou-se, mesmo durante os testes preliminares, uma grande deficiência de desempenho na função de multiplicação entre dois números complexos, disparada através da sobrecarga do operador "*". Sendo assim, a classe standard C++ para números complexos desempenhou substancialmente inferior à implementação original.

Outro kernel de destaque, mas ainda sobre as mudanças em detrimento dos ponteiros, foi o MG, o qual possui diversas funções comunicando-se ativamente entre si e, principalmente, acessos irregulares na memória principal. Neste kernel observou-se que muitas das vezes espaços contíguos não transitavam entre as funções em sua forma integral, mas sim porções específicas do mesmo. Para isso, passava-se como argumento para uma função um endereço para um ponto intermediário do espaço contíguo, o qual era recebido como um ponteiro.

Dentro deste formato do código, portanto, o problema encontrado foi lidar com `std::vector`, pois não haveria forma de enviar como parâmetro uma fatia específica do *container* e receber no outro lado da função a mesma estrutura sem evitar uma cópia e, portanto, abandonando a referencial original. No entanto, explorando a biblioteca STD, encontrou-se a estrutura `std::span` a qual, por si só, não mantém uma referência própria para o endereço de memória, não necessitando alocar ou desalocar memória na construção do objeto. De qualquer forma, ela providencia uma forma de iterar sobre os valores de um *container*, como `std::array` e `std::vector`, sem realizar uma cópia dos valores. Dessa forma, foi possível dar entrada para a função uma fatia específica do vetor na forma `{myVec[x], myVec[y]}` e receber na forma de um `std::span`.

Ainda sobre o *MG*, para adaptar-se ao problema de envio de parcela específica da memória na versão Matrix, teve de ser adaptado um construtor que recebe um ponteiro para o início da faixa de memória e, a partir deste, constrói uma Matrix2D partindo do ponto de início de memória especificado.

4. Algoritmos STL com Políticas de Execução Paralela

O conjunto de algoritmos da biblioteca STL (biblioteca de *templates* padrão) provem uma série de abstrações com diferentes estratégias para operar sobre um conjunto

de elementos. Tais abstrações, em sua maioria, comportam-se como funções de alta ordem, recebendo uma operação (*callback*) definida pelos usuários na forma de funções *lambda*, no caso do C++ moderno. Este vasto conjunto envolve algoritmos para buscas, reduções, ordenamentos e manipulações. Um exemplo é o `std::for_each` ou `std::for_each_n` para percorrer o *container* através de seu índice, como em uma iteração comum.

Alguns algoritmos STL C++ possuem políticas de execução que possibilitam o paralelismo com dependência na biblioteca TBB [Voss et al. 2019]. Dentre os algoritmos para computação orientada a dados, os mais utilizados neste trabalho foram o `std::for_each` e `std::transform_reduce`.

O algoritmo *transform_reduce* é dividido em duas fases: transformação e redução. O usuário define a operação de transformação, que retorna um elemento depois da transformação realizada sobre o elemento da sequência, e também a redução, que unifica o resultado dos elementos que já foram operados pela etapa de transformação. O Código 3 demonstra o código escrito na forma tradicional utilizando *for* para iterar sobre os elementos no formato de índice, enquanto que o Código 4 apresenta a transcrição para o algoritmo de *transform_reduce*.

```
1 for(int j = 0; j < lastcol-firstcol+1; j++) {
2     d   = x[j] - r[j];
3     sum += d*d;
4 }
```

Código 3. Rotina de execução em *loop* presente no *kernel* CG.

```
1  sum = std::transform_reduce(
2   x.cbegin(),
3   x.cbegin()+(lastcol - firstcol + 1),
4   r.cbegin(),
5   0.0,
6   std::plus<double>(),
7   [&d] (const double x_el, const double r_el) {
8    d = x_el - r_el;
9    return pow(d, 2);
10  }
11  );
```

Código 4. Exemplo da aplicação do algoritmo *transform_reduce*.

No Código 4 é possível identificar as duas etapas realizadas dentro do escopo do algoritmo. Entre as linhas 8-10 é realizada a operação de transformação, a qual, neste caso, realiza uma subtração entre os elementos dos vetores x e r seguida de uma potência sobre o resultado desta subtração. Por fim, na linha 6 é realizada a redução do valores da operação anterior através da função `std::plus<T>` que realiza uma soma entre dois elementos.

Outros algoritmos frequentemente utilizados, porém com menor complexidade de incorporação, são `std::fill` e `std::copy`. O algoritmo `std::fill` recebe como argumentos iteradores de início e parada e o valor estático para preencher a fatia especificada de posições do *container*. Já `std::copy` realiza a iteração simultânea de dois vetores a partir de seus iteradores individuais, permitindo com que os valores de uma

vetor sejam copiados para outro. Para garantir eficiência no processo de cópia dos valores, este algoritmo evita realizar uma sequência de múltiplas cópias, realizando uma cópia em volume (p. ex., *bulk copy*) através de funções como `std::memmove`.

4.1. Políticas de execução paralelas

A implementação do paralelismo de NPB-ALG foi desenvolvida com base nas implementações de NPB-CPP. Durante o processo de adaptação, uma das principais divergências foi a substituição de regiões críticas por operações de redução utilizando o algoritmo `transform_reduce`. Adicionalmente, no contexto da aplicação, houve a necessidade de lidar com os kernels `EP` e `MG`, os quais retornam múltiplas variáveis. Para abordar essa situação, optou-se por criar uma estrutura denominada "`returnable`", que possibilitou o retorno e a redução eficiente das variáveis provenientes desses kernels. Essas modificações se revelaram fundamentais para garantir a escalabilidade e o desempenho adequado do paralelismo em NPB-ALG. Além disso, devido a certos algoritmos, como `std::inner_product`, não possuírem integração com *execution policies*, esses foram adaptados para algoritmos que aceitam execução paralela, como apresentado no código 5. Essa integração com técnicas já estabelecidas em NPB-CPP demonstrou a viabilidade e a eficiência do paralelismo em NPB-ALG, corroborando o sucesso da abordagem adotada neste trabalho de pesquisa para otimização do desempenho da aplicação.

```
1 result = std::inner_product(v.begin(), v.end(), v.begin(), 0.0);
2
3 result = std::transform_reduce(policy, v.begin(), v.end(),
4                                 v.begin(), 0.0,
5                                 std::plus<double>(),
6                                 std::multiplies<double>());
```

Código 5. Mudança de std::algortihm para aplicação de execution policy

Visto a relação na expressão de paralelismo entre os algoritmos paralelos da biblioteca STD e a biblioteca TBB, utilizou-se como base os códigos já paralelizados com o uso da ferramenta citada. Para expressar o paralelismo nativo da biblioteca STD, é necessário o uso da *feature* `std::execution`. Este recurso dá acesso a quatro modos de operação disponíveis para os algoritmos, sendo que dois destes são paralelos: `std::execution:par` e `std::execution::par_unseq`. O segundo, em especial, possui um nível de otimização a mais dentro da execução paralela, permitindo a execução de múltiplas funções em uma só *thread*, caso a vetorização esteja habilitada.

4.2. Aplicação dos Algoritmos nas classes Matrix

Para que os algoritmos da biblioteca possam iterar sobre uma estrutura de dados, é necessário que haja uma estrutura que possua métodos que permitam a navegação entre os elementos. A estrutura de dados utilizada nos códigos durante a conversão, o *container* `std::vector`, fornece acesso ao construtor do seu Iterador através dos métodos `begin()` e `end()`. Dentro da estrutura de um Iterador como o que é usado pelo *container* citado, são necessários alguns métodos de sobrecarga de operador fundamentais para se movimentar entre os elementos adjacentes.

Para então possibilitar que nossa estrutura de abstração de matrizes pudesse ser iterada por meio dos algoritmos, portanto, procurou-se compreender os métodos disponibilizados em estruturas de iteração em sequências contíguas como no `std::vector` e,

assim, personalizar para nossa estrutura. Os primeiros métodos desenvolvidos na classe *Matrix2DIterator* foram as sobrecargas dos operadores ++, para permitir o avanço do Iterador, e ==, para identificar o encontro dos Iteradores de início e fim. Para implementar tais operadores em nossa classe, entretanto, não foi possível apenas replicar a forma como foi feito no Iterador do `std::vector`, pois este, a cada operação de incremento (++) sobre a classe, apenas repassava o incremento para o ponteiro herdado do *container*, o qual segue um endereçamento linear até o fim da sequência. Portanto, para realizar o incremento em nosso iterador, era necessário, além do ponteiro, saber informação da linha e coluna atual e os limites em cada dimensão. Dessa forma, era possível saber quando estava se iterando pelo mesmo vetor ou não, identificando uma quebra na contiguidade dos endereços naquele espaço, necessitando resgatar novamente a referência a partir do endereço original do objeto. O Códido 6 contém um trecho do código da classe Iterador apresentando a sobrecarga do operador ++.

```
1
2 Matrix2DIterator(Matrix& matrix, size_t Max_row=0, size_t Max_col
    =0, size_t row = 0, size_t col = 0)
3         : currPtr(nullptr), row_(row), col_(col), row_size(
    Max_row), col_size(Max_col) {
4             matrixPtr = &matrix;
5             currPtr = &(matrix[row_][col_]);
6 }
7
8 Matrix2DIterator& operator++() {
9         col_++;
10        currPtr++;
11        if (col_ >= col_size and row_ < row_size-1) {
12            col_ = 0;
13            row_++;
14            currPtr = &((*matrixPtr)[row_][col_]);
15        }
16        return *this;
17 }
```

Código 6. Sobrecarga do operador ++ no Iterador da classe Matrix2D.

5. Resultados

Com o objetivo de analisar o impacto das estruturas fundamentais que sustentam os algoritmos paralelos no código, procedeu-se à realização de testes que viabilizam a comparação entre a versão anterior do NPB-CPP, desprovida das estruturas fornecidas pela biblioteca padrão, e a abordagem desenvolvida no âmbito deste estudo. Todas as avaliações experimentais foram conduzidas em um sistema equipado com dois processadores *Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz*, totalizando 24 núcleos e 48 *threads*, com 188 GB de memória RAM. A análise das execuções foi realizada com o auxílio da ferramenta `perf stat`, que possibilita a coleta de métricas de desempenho do sistema durante a execução dos testes.

Os resultados ora apresentados são derivados da média e desvio padrão de 10 iterações para cada núcleo de execução, empregando a classe de carga referente à classe B do NPB-CPP. Tanto a compilação dos núcleos com o compilador `GCC 12.2.1` quanto com o `Clang 17.0.0` foi realizada utilizando a flag de otimização `-O3`, a qual denota o mais alto nível de otimização disponível.

Tabela 1. Média e desvio padrão do tempo de execução sequencial dos kernels compilados com GCC e Clang.

		NPB-CPP	NPB-STD		NPB-ALG	
			Linear	Matriz	Linear	Matriz
EP	GCC	91.0 ± 0.0	92.3 ± 0.2	–	105.6 ± 0.2	–
	Clang	72.2 ± 0.3	74.0 ± 0.3	–	85.3 ± 0.2	–
FT	GCC	53.7 ± 0.4	43.6 ± 0.3	36.6 ± 0.1	82.3 ± 0.2	47.1 ± 0.7
	Clang	58.4 ± 0.3	65.6 ± 3.4	45.8 ± 0.2	84.8 ± 0.5	55.1 ± 0.8
CG	GCC	42.5 ± 0.1	42.8 ± 0.1	42.8 ± 0.1	46.4 ± 3.6	49.3 ± 5.5
	Clang	111.2 ± 0.3	43.1 ± 0.1	43.1 ± 0.1	46.6 ± 3.8	48.1 ± 3.2
MG	GCC	4.5 ± 0.0	5.2 ± 0.0	5.6 ± 0.0	7.8 ± 0.0	7.1 ± 0.0
	Clang	4.5 ± 0.0	4.9 ± 0.0	4.7 ± 0.0	8.1 ± 0.0	7.1 ± 0.0
IS	GCC	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.4 ± 0.0	2.4 ± 0.0
	Clang	2.1 ± 0.0	2.2 ± 0.0	2.1 ± 0.0	2.4 ± 0.0	2.4 ± 0.0

No contexto específico do uso do compilador Clang, optou-se por empregar a biblioteca padrão do GCC, uma vez que as implementações dos algoritmos padrão em `libc++` não se encontravam integralmente concretizadas, garantindo assim uma base sólida para as comparações. Em relação às versões paralelas, foram avaliadas sob o emprego do mais alto grau de paralelismo possível (48 *threads*), dado a inexistência de meios para controlar o nível de paralelismo por meio da biblioteca padrão.

5.1. Análise das Execuções Sequenciais

Na Tabela 1, foi possível perceber que todas versões que fizeram o uso da classe Matrix na versão NPB-STD, com exceção do *kernel* FT, obtiveram um desempenho similar às versões lineares, apesar da não contiguidade dos elementos em memória em virtude do uso da abstração com o aninhamento das estruturas `std::vector`.

Sobre o desempenho do *kernel* FT na versão Matrix, observou-se que as versões lineares obtiveram um desempenho consideravelmente inferior às outras versões. Análises das execuções revelaram que as versões lineares realizaram 3 vezes mais referências para a memória cache, isto é, tentativas de acesso aos dados. Isso se deve ao fato do compilador não ter realizado a vetorização automática de todos os laços de repetição. Isso foi confirmado através de uma análise do código *assembly*, que demonstrou que não haviam instruções vetoriais da família AVX, e que também haviam menos instruções vetoriais da família SSE.

Ainda no *kernel* FT, onde necessita-se de uma estrutura de dados de números complexos, foi possível observar uma grande deficiência no desempenho ao utilizar a classe *std::complex*. Especificamente na operação de multiplicação entre números complexos, realizada através da sobrecarga do operador "*", o tempo de execução observado para a versão com o uso do recurso da biblioteca STD foi o dobro comparado a versão onde utiliza-se de uma estrutura criada manualmente. Em virtude da grande lentidão observada, a versão com uso desta estrutura não foi continuada na implementação dos algoritmos.

Na Tabela 1, o tempo de execução do *kernel* MG quando compilado com `CLANG` foi o único com resultado a favor do acesso linear. Esse resultado está associado à uma

Tabela 2. Média e desvio padrão do tempo de execução paralelo dos kernels compilados com GCC e Clang.

		NPB-CPP	NPB-STD		NPB-ALG	
			Linear	Matriz	Linear	Matriz
EP	GCC	2.75 ± 0.11	2.57 ± 0.05	–	4.25 ± 0.20	–
	Clang	2.38 ± 0.10	2.23 ± 0.05	–	3.90 ± 0.09	–
FT	GCC	4.04 ± 0.43	5.15 ± 0.53	3.86 ± 0.06	5.54 ± 0.19	3.88 ± 0.05
	Clang	4.91 ± 0.35	6.08 ± 0.23	4.12 ± 0.05	5.36 ± 0.21	4.10 ± 0.05
CG	GCC	4.03 ± 0.14	4.33 ± 0.11	3.88 ± 0.18	7.68 ± 0.07	7.73 ± 0.05
	Clang	4.30 ± 0.04	4.59 ± 0.04	4.28 ± 0.05	7.68 ± 0.06	7.73 ± 0.07
MG	GCC	1.15 ± 0.08	1.23 ± 0.06	1.41 ± 0.04	1.86 ± 0.01	2.39 ± 0.10
	Clang	0.97 ± 0.06	1.36 ± 0.03	1.48 ± 0.09	1.87 ± 0.02	2.40 ± 0.09
IS	GCC	0.16 ± 0.03	0.25 ± 0.06	0.28 ± 0.09	0.33 ± 0.00	0.34 ± 0.01
	Clang	0.18 ± 0.04	0.26 ± 0.05	0.26 ± 0.06	0.33 ± 0.01	0.33 ± 0.01

adaptação feita no código para incluir um construtor da classe `Matrix2D` que recebia como parâmetro um ponteiro para um vetor linear.

Para fornecer uma visão geral da Tabela 1, tomando como base o tempo de execução NPB-CPP, observou-se que a versão NPB-STD obteve um decréscimo de 3% no tempo total com uso do acesso linear e de 8% no uso de estruturas para acesso multidimensional. Referente à diferença entre as versões NPB-STD e NPB-ALG, notou-se um aumento no tempo de execução, principalmente para os *kernels* FT e MG, com diferenças de 52% e 33%, respectivamente. Ao realizar análise dos contadores de hardware para o *kernel* MG, observou-se um total de instruções executadas 55% maior, 3% de diferença entre *cache-references*, e diferenças de 0.08% em branch-miss.

5.2. Análise das Execuções Paralelas

A Tabela 2 exibe os resultados provenientes das execuções em paralelo (com 48 threads) do benchmark `NPB-STD`. A tabela apresenta a média e o desvio padrão dos tempos de execução. Durante essas execuções, observou-se uma deterioração substancial no desempenho de `EP`, com seu tempo de execução excedendo em cerca de 1.7 vezes o da versão padrão. A análise da execução do binário revelou um aumento notável de 3.6 vezes na ocorrência de `cache misses` e um acréscimo de 34% no número de instruções. Também se constatou um desempenho inferior em `MG` e `IS`, com `MG` registrando um tempo médio 56% mais longo e `IS` 31% maior. Uma análise mais detalhada da execução identificou razões distintas para esse comportamento. Em `MG`, verificou-se um aumento de 213% no número de instruções em comparação com a versão `NPB-STD`, sendo as discrepâncias relacionadas ao acesso à memória e `cache misses` mantidas abaixo de 5%. No caso de `IS`, houve um aumento substancial de `cache misses`, com um incremento de 83% nesse aspecto, enquanto o número de instruções aumentou em 9%.

Os resultados do `NPB-ALG` revelaram uma tendência de desempenho inferior na grande maioria das versões analisadas, com essa discrepância alcançando significância estatística, conforme confirmado pelo teste `U de Mann-Whitney` ($p < 0.02$). No entanto, merece destaque o caso do kernel `FT` quando compilado com o compilador

`clang`, que apresentou o único ganho de desempenho em comparação com a versão `NPB-ALG`. Nesse cenário, a versão `Linear` demonstrou uma notável significância estatística ($p = 0.0002$), exibindo uma redução de 11.8% no tempo de execução em relação à sua contraparte na versão `NPB-STD`.

6. Conclusões

Neste trabalho, foi realizada uma análise acerca das estruturas de dados, abstrações, algoritmos e políticas de execução paralela do C++ moderno. Para isso, foram realizadas implementações dos 5 kernels do NPB benchmark utilizando diferentes estruturas C++ e algoritmos dos conjuntos `std::algorithm` e `std::numeric`, assim como a própria aplicação e avaliação dos mesmos em um cenário *multithreaded*. Em comparação com C, as abstrações de estruturas de dados C++ obtiveram uma perda de desempenho entre 3 e 8%, trazendo também desafios de programação para criar iteradores multidimensionais customizados. Em relação aos *algorithms*, o desempenho foi estatisticamente inferior conforme corroborado pelo teste `U de Mann-Whitney` ($p < 0.02$).

Como direção para investigações subsequentes, é proposto a análise da exequibilidade da utilização de algoritmos paralelos de forma concomitante ao compilador da empresa de unidades de processamento gráfico, NVIDIA, com o intuito de automatizar a geração de código destinado a tal dispositivo. Adicionalmente, uma comparação entre o paralelismo apresentado pelo *NPB-ALG* e as implementações baseadas em *Thread Building Blocks* da Intel poderia ser realizada, visando uma avaliação mais abrangente da aplicabilidade da Biblioteca Padrão. Além disso, é sugerido a avaliação da implementação da biblioteca padrão nas pseudo-aplicações presentes no conjunto de testes de referência *Nas Parallel Benchmark*, acrescentando maior profundidade à análise empreendida.

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.

Di Domenico, D., Cavalheiro, G. G. H., and Lima, J. V. F. (2022). Nas parallel benchmark kernels with python: A performance and programming effort analysis focusing on gpus. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 26–33.

Do, Y., Kim, H., Oh, P., Park, D., and Lee, J. (2019). SNU-NPB 2019: Parallelizing and Optimizing NPB in OpenCL and CUDA for Modern GPUs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 93–105.

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.

Josuttis, N. M. (2012). *The C++ standard library: a tutorial and reference*. Addison-Wesley.

Lin, W.-C., Deakin, T., and McIntosh-Smith, S. (2022). Evaluating iso c++ parallel algorithms on heterogeneous hpc systems. *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 36–47.

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.

Mietzsch, N. and Fuerlinger, K. (2019). Investigating performance and potential of the parallel stl using nas parallel benchmark kernels. *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 136–144.

Seo, S., Jo, G., and Lee, J. (2011). Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148.

Voss, M., Asenjo, R., Reinders, J., Voss, M., Asenjo, R., and Reinders, J. (2019). Tbb and the parallel algorithms of the c++ standard template library. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*, pages 109–136.