Reprodução de Experimentos do NAS Parallel Benchmarks com OpenMP C++ e Fortran

[1]Eduardo M. Martins [1]Eduardo F. Eichner [1]Bernardo P. Fiorini [1]Dalvan J. Griebler

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

e.martins01@edu.pucrs.br, Eduardo.Eichner@edu.pucrs.br,

bernardo.fiorini@edu.pucrs.br, dalvan.griebler@pucrs.br

Abstract. *This article provides a reproduction of experiments that compare the efficiency of the original NAS Parallel Benchmarks, with a C++ version, both using parallel processing with OpenMP, and a comparison between two different classes of work. From that, the behavior of the executions in each situation was carefully evaluated.*

Resumo. *Este artigo proporciona uma reprodução de experimentos que comparam a eficiência do NAS Parallel Benchmarks original, com uma versão em C++, ambas utilizando processamento paralelo com o OpenMP, e ainda uma comparação entre duas diferentes classes de trabalho. A partir disso, foi avaliado cuidadosamente o comportamento das execuções em cada situação.*

1. Introdução

A partir da utilização da programação paralela, no contexto da computação, um processador consegue utilizar sua capacidade máxima, usufruindo de cada um de seus núcleos para realizar suas tarefas. Embora normalmente o processador não faça isso naturalmente, os programadores e desenvolvedores podem utilizar alguns métodos e ferramentas para tornar isso possível. Dessa forma, por mais que exista uma maior complexidade envolvida, é viável transformar uma aplicação sequencial (que utiliza somente um núcleo do processador por vez) em uma aplicação paralela (que divide o processo entre todos os núcleos disponíveis), tornando-a mais eficiente, com uma alta performance e aproveitando todo *hardware* disponível.

Esses métodos e ferramentas que tornam o processamento paralelo possível são formados principalmente por *frameworks*, que podem ser utilizados durante o desenvolvimento das aplicações. Dentre algumas soluções mais populares para a criação de aplicações paralelas, está o OpenMP [Chapman et al. 2007], muito utilizado em arquiteturas de memória compartilhada e está disponível para as linguagens de programação C, C++ e Fortran. Além dele, existe o Intel TBB [Voss et al. 2019], também muito popular e utilizado nas arquiteturas com memória compartilhada, disponível para a linguagem C++. Por fim um outro *framework* que será trabalhado é o FastFlow [Aldinucci et al. 2017], também para C++.

Nota-se, no entanto, que para uma aplicação ser paralelizada, depende do próprio desenvolvedor escolher qual a melhor metodologia a ser empregada, pois não é todo o algoritmo que está envolvido na paralelização, e sim alguns trechos específicos, principalmente onde estruturas e laços de repetição estão presentes. Além disso,

outros fatores que implicam na qualidade da programação paralela é a escolha do *framework*, bem como a escolha do compilador a ser utilizado e as *flags* indicadas no momento da compilação. Pode-se dizer então, que para cada tipo de problema, cada uma das combinações possíveis possui um comportamento diferente dos demais.

Para avaliar as características das diferentes formas de paralelização de códigos e facilitar a visualização de qual delas é mais eficiente em cada caso, são utilizadas aplicações pesadas que forçam diferentes partes do *hardware*. Assim, um dos conjuntos de aplicações mais conhecidos para essas comparações é o NPB (*NAS Parallel Benchmarks*), desenvolvido pela NASA, que é composto por cinco *Kernels* e três pseudo-aplicações desenvolvidas principalmente em Fortran e algumas em C.

Contudo, a partir do que foi descrito surge a motivação para a realização de uma reprodução de um artigo científico bem conceituado nesta área, denominado *"The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures"* [Löff et al. 2021], explorado na segunda seção deste artigo. Com isso, o objetivo principal deste projeto é replicar e comparar os resultados obtidos em um dos experimentos realizados pelos autores do [Löff et al. 2021], onde é comparado o tempo de execução do NPB, com o NPB-CPP[1] (uma versão do NPB traduzido para C++, feita pelos autores) usando diferentes números de *threads* e que posteriormente possibilitou a avaliação dos *frameworks* para programação paralela em C++.

Finalmente, este artigo está dividido em cinco seções, sendo elas Introdução, Fundamentação teórica, Metodologia de Reprodução, Resultados e análise de dados e Conclusões.

2. Fundamentação teórica

Nesta seção será apresentado a fundamentação teórica que possibilitou o desenvolvimento deste artigo, bem como uma breve descrição do funcionamento das ferramentas utilizadas no processo.

A principal referência para o desenvolvimento deste projeto é o artigo *"The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures"* [Löff et al. 2021], que contribuiu com a comunidade cientifica a partir do desenvolvimento de uma tradução do NPB original [Bailey et al. 1995], que foi desenvolvido pela Divisão de Supercomputação Avançada da NASA, para avaliar a performance de máquinas paralelas (desenvolvido majoritariamente em Fortran), para uma versão C++, respeitando a estrutura original dos *Kernels* e pseudo-aplicações. Além disso, os pesquisadores implementaram no NPB-CPP uma paralelização utilizando outros dois *frameworks* (Intel TBB e FastFlow), além do original com o OpenMP. Mostraram também, como o conceito da programação paralela pode ser eficientemente aplicado no NPB-CPP, utilizando principalmente a paralelização de estruturas como *Map* e *Map Reduce*. Por fim, os autores promovem também, uma discussão da performance obtida com o NPB-CPP utilizando os diferentes *frameworks* e arquiteturas.

2.1. NAS Parallel Benchmarks

Conforme já descrito, cada uma das aplicações do NPB original força alguma característica diferente do *hardware*, dessa forma é possível verificar o comportamento e velocidade das execuções em diferentes situações. Nesta seção serão apresentados os cinco *Kernels* e as três pseudo-aplicações de forma breve, visto que cada um possui suas especificações e peculiaridades.

[1]Código fonte do NPB-CPP: https://github.com/GMAP/NPB-CPP.

Embarrassingly Parallel(EP): Este *Kernel* é útil para medir a capacidade das operações de ponto flutuantes da arquitetura. São gerados desvios aleatórios gaussianos de acordo com uma fórmula específica e é tabulado um número de pares em anéis sucessivos.

Multi Grid(MG): *Kernel* utilizado para estressar a comunicação de dados a curta e longa distância. Faz a computação 3D da equação de Poisson, a partir de um método *V-cycle MultiGrid*.

Conjugate Gradient(CG): Um *Kernel* que estressa a comunicação de dados, mas também a memória. Calcula através de um método específico, uma aproximação do menor valor de uma matriz gigante e não estruturada.

Discrete 3D Fast Fourier Transform(FT): O foco desse *Kernel* é trabalhar a comunicação a longa distância. Computa uma equação diferencial parcial 3D utilizando FFT's.

Integer Sort(IS): Esse *Kernel* mede a capacidade de computação de inteiros e força a comunicação de dados. É baseado em um algoritmo de *Bucket-Sorting*, para a ordenação de números.

Block Tri-diagonal solver(BT): Esta é uma pseudo-aplicação que soluciona numericamente uma equação 3D de Navier-Stokes.

Scalar Penta-diagonal solver (SP): Uma pseudo-aplicação, muito parecida com a BT, que usa uma aproximação de *Beam-Warming* para decompor uma matriz 3D.

Lower-Upper Gauss–Seidel solver(LU): Resolve um sistema linear de equações, através do método *Symmetric Successive Over-Relaxation* (SSOR), é uma variação do de Gauss-Seidel.

É necessário também, considerar que existem diferentes níveis de complexidade que podem ser configurados em cada uma das aplicações. Esse grau de complexidade deve ser definido no momento da compilação das aplicações e é dada pela escolha de uma "classe", que variam conforme a descrição a seguir:

Classe S: Essa é a menor classe, existe para realização de pequenos testes.

Classes A, B e C: Essas são as classes padrão para os testes, o nível de complexidade entre elas aumenta em 4x ao subir uma classe.

Classes D, E e F: Essas são classes para testes muito pesados, o nível de complexidade entre elas aumenta em 16x ao subir uma classe, dessa forma a classe F é a mais pesada.

No escopo deste trabalho, as classes que foram utilizadas para os testes foram a B, para a realização de algumas comparações e testes descritos nas próximas seções, e a classe C, pois foi a utilizada nos testes feitos pelos pesquisadores desenvolvedores do NPB-CPP.

2.2. Frameworks

Nesta seção serão introduzidos mais detalhadamente os três *frameworks* estudados para o desenvolvimento do projeto, para uma melhor contextualização.

O OpenMP [Chapman et al. 2007] é uma API e conjunto de diretivas de compilação que possibilitam a criação de aplicações com processamento paralelo, possui suporte para Fortran, C e C++, e é o *framework* originalmente implementado nas paralelizações do NPB. Essa ferramenta é baseada em anotações do tipo *pragma*, que devem ser feitas diretamente no código antes das regiões que devem ser paralelizadas, dessa forma a partir disso as iterações dos *loops* são divididas entre as *threads* disponíveis no *hardware*. Em particular esse *framework* é o principal utilizado neste artigo,

pois por mais que os outros dois (Intel TBB e FastFlow) também foram trabalhados no [Löff et al. 2021], os experimentos selecionados para serem reproduzidos utilizam somente esse método de paralelização.

O Intel TBB [Voss et al. 2019] é uma biblioteca criada pela Intel para possibilitar a programação paralela em C++. O próprio *framework* é o responsável pela criação das *threads* e pela atribuição de tarefas, dessa forma, o programador fica responsável por criar e submeter as tarefas para que o *framework* execute quando for oportuno. São utilizadas técnicas para reutilizar e ou balancear as *threads*.

O FastFlow [Aldinucci et al. 2017], por fim, é um outro *framework open source* também desenvolvido para o moderno C++, suporta tanto a paralelização de dados como de *streams*, desenvolvido principalmente arquiteturas de memória compartilhada como *clusters*, com foco na performance e na programabilidade.

2.3. Map e Map Reduce

Nesta seção será discutido brevemente sobre as estruturas de *Map* e *Map Reduce* que precisam ser entendidas, bem como será demonstrado um exemplo da paralelização dessas estruturas com cada um dos *frameworks* trabalhados, pois estes são o foco da programação paralela.

A estrutura *Map* consiste em uma forma de *loop* que itera uma determinada quantidade de vezes, e cada uma das iterações são independentes umas das outras, ou seja, nenhuma variável é reaproveitada ou dividida entre os indexes durante as repetições. Para representar a paralelização do *Map*, são disponibilizados nos três diferentes *frameworks* uma estrutura denominada "*parallel for*", para referenciar a região a ser paralelizada. Entretanto, para cada um dos *frameworks* a notação é diferente, conforme mostrado na Figura 1.

```
//OpenMP parallel for
#pragma omp parallel for
for (i=0; i<n; i++){
  //Código...
}
}
```

```
//FastFlow parallel for
ff::ParallelFor pf
pf.parallel_for(0, n, 1, chunk, [&](int i){
  //Código...
}, workers);
}
```

```
//TBB parallel for
tbb::parallel_for(tbb::blocked_range<size_t>
(0, n, chunk),[&](const tbb::blocked_range<size_t>& r){
  for(i=r.begin(); i<r.end(); i++){
    //Código...
  }
}
}
```

Figura 1. Map paralelo.

Seguindo com o *Map Reduce*, este é uma estrutura de *loop*, onde as informações geradas durante ou após a execução de uma das iterações, influencia no resultado das outras repetições. O exemplo prático e comum de um caso envolvendo o *Map Reduce*, é um algoritmo para somar os valores de um vetor, dessa forma o resultado é incrementado em cada repetição. Assim para evitar perda de informações é necessário utilizar uma função de redução para juntar os dados parciais em um único dado ao final do *loop*. Para o desenvolvimento do *Map Reduce* nos *frameworks* trabalhados deve-se então fazer o uso de um "*parallel reduce*", conforme a Figura 2.

```
//OpenMP parallel reduce
#pragma omp parallel for reduction (+:x)
for (i=0; i<n; i++){
  //Código...
}
}
```

```
//FastFlow parallel reduce
ff::ParallelForReduce<double> pr;
pr.parallel_reduce(x, 0, n, chunk, [&A](int i){
  //Código...
},[&A](int i, double& x){x+=A[i];}, workers);
}
```

```
//TBB parallel reduce
tbb::parallel_for(tbb::blocked_range<size_t>
(0, n, chunk),[&](const tbb::blocked_range<size_t>& r){
  for(i=r.begin(); i<r.end(); i++){
    //Código...
  }
}
}
```

Figura 2. Map Reduce paralelo.

3. Metodologia e reprodução

Nesta seção será demonstrado quais experimentos que foram replicados do [Löff et al. 2021], de forma detalhada. Também será descrita qual foi a metodologia criada para recriar os experimentos em questão.

3.1. Detalhes da metodologia original

Na execução dos testes de performance realizadas no artigo original, que compara o NPB com o NPB-CPP, os pesquisadores obtiveram os resultados em uma plataforma denominada "Xeon". Como principais características de *hardware*, possui 64GB de memória RAM, dois processadores Intel Xeon E5-2695 com 12 *cores* cada, totalizando 24 *cores* e consequentemente disponibilizando um máximo de 48 *threads*. Esses processadores possuem um *clock* baseado na frequência de 2.4GHz, além disso, em relação à memória *cache*, as CPUs, têm um L1 privado de 32KB, um L2 privado de 256KB e um L3 compartilhado com 30MB de espaço. Em adição, é válido mencionar que o compilador utilizado nos testes foi o *GNU Compiler* (gcc). Por fim, a plataforma descrita se encontra na Universidade de Pisa, na Itália.

Com esse *hardware* definido, os pesquisadores do [Löff et al. 2021] rodaram cada um dos oito *Benchmarks* na classe C cinco vezes para cada número de *threads* disponível (de 1 até 48 *threads*), tanto para a versão original em Fortran quanto para a versão em C++, portanto o *frameworks* utilizado para a realização desse teste foi o OpenMP. Ao contrário do código fonte do NPB-CPP, os *Scripts* para executar e coletar os dados para gerar os gráficos do experimento não foram referenciados. Seguindo a metodologia descrita, os autores do projeto coletaram a quantidade de *threads* da vez, seu respectivo tempo de execução, o desvio padrão baseando-se no conjunto de repetições da quantidade de *threads* em questão, e ao plotar os gráficos encontraram os resultados que podem ser visualizados no artigo [Löff et al. 2021], na página 10, Figura 6.

3.2. Detalhes da metodologia adotada

Para conseguir reproduzir o teste descrito, foi necessário a definição de uma metodologia específica. Primeiramente, definiu-se alguns objetivos principais, sendo eles definir qual plataforma que irá rodar os testes finais. Criar um *Shell Script* para rodar os *Benchmarks* de forma automática já coletando os dados para plotar os gráficos no Gnuplot, seguindo a metodologia do teste original. Testar o *Shell Script* em uma plataforma alternativa antes de rodar na plataforma final. Criar os gráficos no Gnuplot a partir dos dados coletados, para ajustar a plotagem. Por último, rodar na plataforma definida, plotar os dados definitivos e discutir e comparar os resultados. Dessa forma o *WorkFlow* de execução de tarefas pode ser conferido no fluxograma da Figura 3.

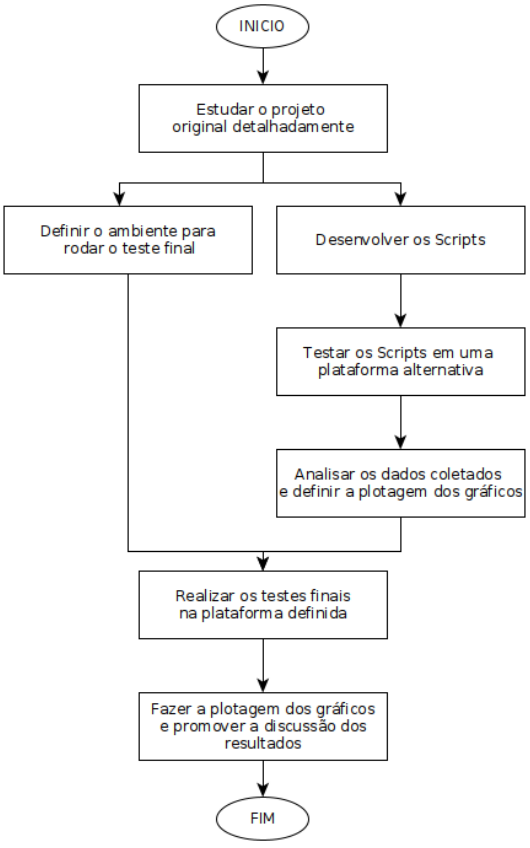


Figura 3. Fluxograma de tarefas.

A fim de tornar o teste final o mais próximo possível das condições de execução do teste original, bem como com o objetivo de obter resultados de uma máquina de maior performance, foi definido para ser a plataforma de execução final do projeto, um servidor de posse do Grupo de Modelagem e Aplicações Paralelas (GMAP) da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) chamado "Ostrich". Essa plataforma conta com uma memória RAM de 16GB, um processador AMD Ryzen 5 5600X de 6 *cores*, totalizando 12 *threads* que opera numa frequência de *clock* de 3.7GHz. Ainda possui um *cache* L1 de 192KB, um L2 de 3MB e um L3 de 32MB de espaço. Contudo, comparando as plataformas Xeon e Ostrich, percebe-se uma vasta diferença de *hardware*, entretanto, dado as limitações da pesquisa ainda sim considera-se a plataforma recém descrita uma ótima opção para a confecção dos testes finais. Para facilitar a visualização de todas as informações que foram descritas, um comparativo dos *hardwares* está presente na Tabela 1.

Tabela 1. Xeon x Ostrich

Item	Xeon	Ostrich
Modelo CPU	Xeon E5-2695	AMD Ryzen 5
Memória RAM	64GB	16GB
Quantidade de CPUs	2	1
Quantidade de *threads* total	48	12
Frequência de *clock*	2.4GHz	3.7GHz
Espaço livre no *cache* L1	32KB	192KB
Espaço livre no *cache* L2	256KB	3MB
Espaço livre no *cache* L3	30MB	32MB

Na sequência, criou-se o *Shell Script* para executar os *Benchmarks*[1]. Para rodar as aplicações, no próprio *Script* foi definido uma metodologia específica, onde o algoritmo começa executando da maior quantidade de *threads* disponível até a menor (de forma com que as execuções mais rápidas aconteçam primeiro. Assim é possível verificar os dados e procurar por erros de forma mais prática e eficiente, pois os *logs* são gerados em tempo real e são acessíveis antes do *Script* ser finalizado). Ainda, cada aplicação roda um total de dez vezes para cada número de *threads*, com isso é possível obter uma média de tempo bastante confiável e também calcular o desvio padrão para cada uma das médias encontradas. Em adição à essa etapa, foi garantido também que nenhum *Benchmark* do mesmo tipo fosse executado mais de uma vez seguida, para evitar qualquer tipo de interferência relacionada com dados pré-carregados na memória *cache* que pudessem ser reaproveitados de uma execução para a outra.

Com o *Script* em mãos, optou-se por testá-lo em uma máquina qualquer para verificar sua funcionalidade e também, gerar *logs* reais, que pudessem ser utilizados para dar início a criação dos gráficos através do Gnuplot. Dessa forma, foi executado o conjunto dos oito *Benchmarks* classe S, tanto na versão em Fortran, quanto na versão C++, em uma máquina Dell com 16GB de memória RAM e processador i7 com 4 núcleos contabilizando 8 *threads*, porém sem nenhum fim comparativo, somente para testes e ajustes.

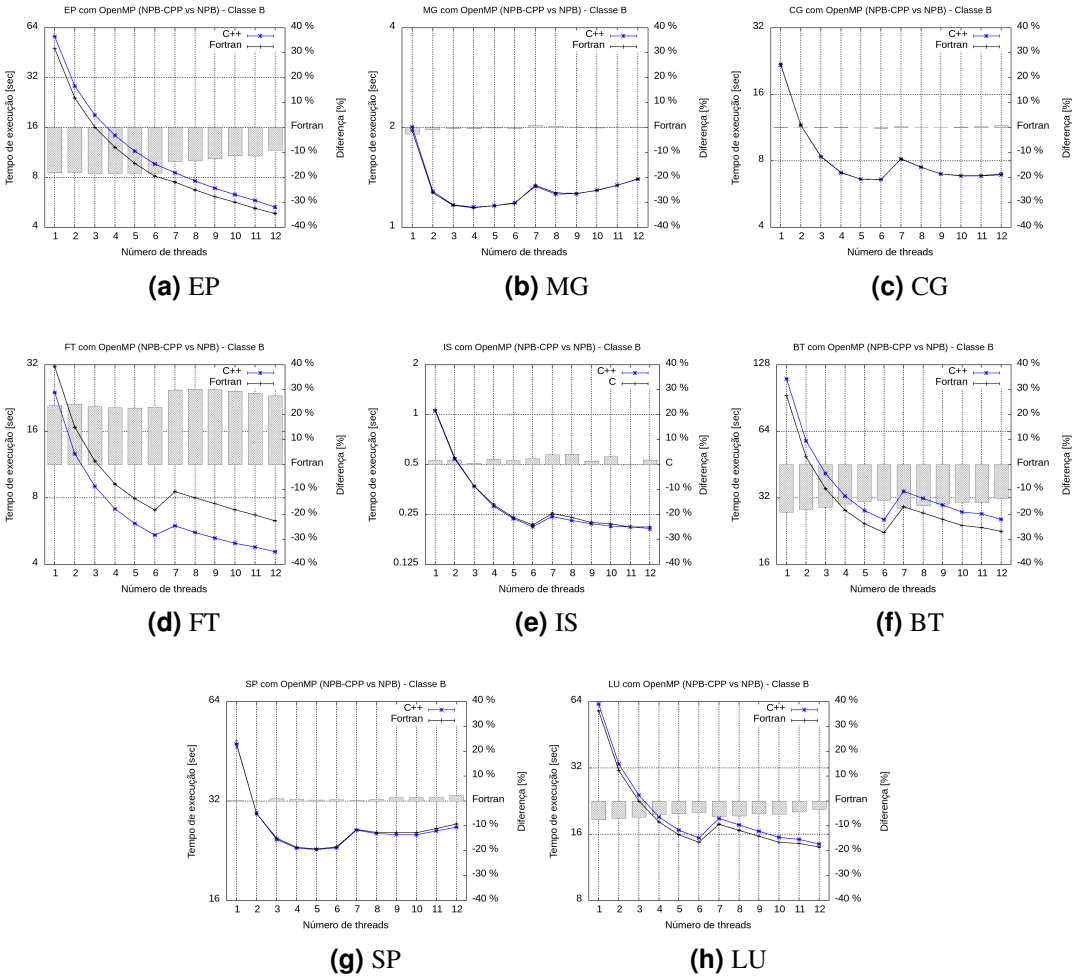
Na posse dos primeiros resultados, realizaram-se um série de ajustes para que existissem apenas dois *logs* saídas para cada *Benchmark*, uma para o NPB e outra para o NPB-CPP, assim configurou-se esses *logs* em um formato específico com três colunas,

[1]Repositório do *Script* criado: https://github.com/Eduardo-Machado-Martins/NPB-IOT-Scripts-1.0.git.

sendo a primeira o número de *threads*, a segunda o tempo médio, e a terceira o desvio padrão. Esse padrão foi adotado justamente para facilitar o *plot* dos gráficos no Gnuplot, que a partir do que foi descrito, define somente um gráfico para cada uma das aplicações, com as linhas do NPB e NPB-CPP sobrepostas em função dos eixos "Número de *threads*"x "Tempo de execução em segundos".

Com tudo pronto, foi executado primeiramente a classe B na plataforma Ostrich, não só para fins de teste, mas também para realizar alguma possível comparação entre os comportamentos da classe B contra a classe C.

Finalmente, após a realização de todas as etapas anteriores à essa, realizou-se o teste final, com a execução do NPB e NPB-CPP, ambos com a classe C na plataforma Ostrich, seguindo fielmente a metodologia descrita até aqui. A finalização da execução deu-se em aproximadamente 60 horas após o inicio, onde a plataforma foi utilizada exclusivamente para o teste. Outrossim, é válido mencionar que todos os *logs* de saída brutos, de todas as execuções foram mantidos em arquivos separados, filtrados por número de *threads* e por *Benchmarks*, para que se necessário, os dados possam ser resgatados e conferidos em algum momento.



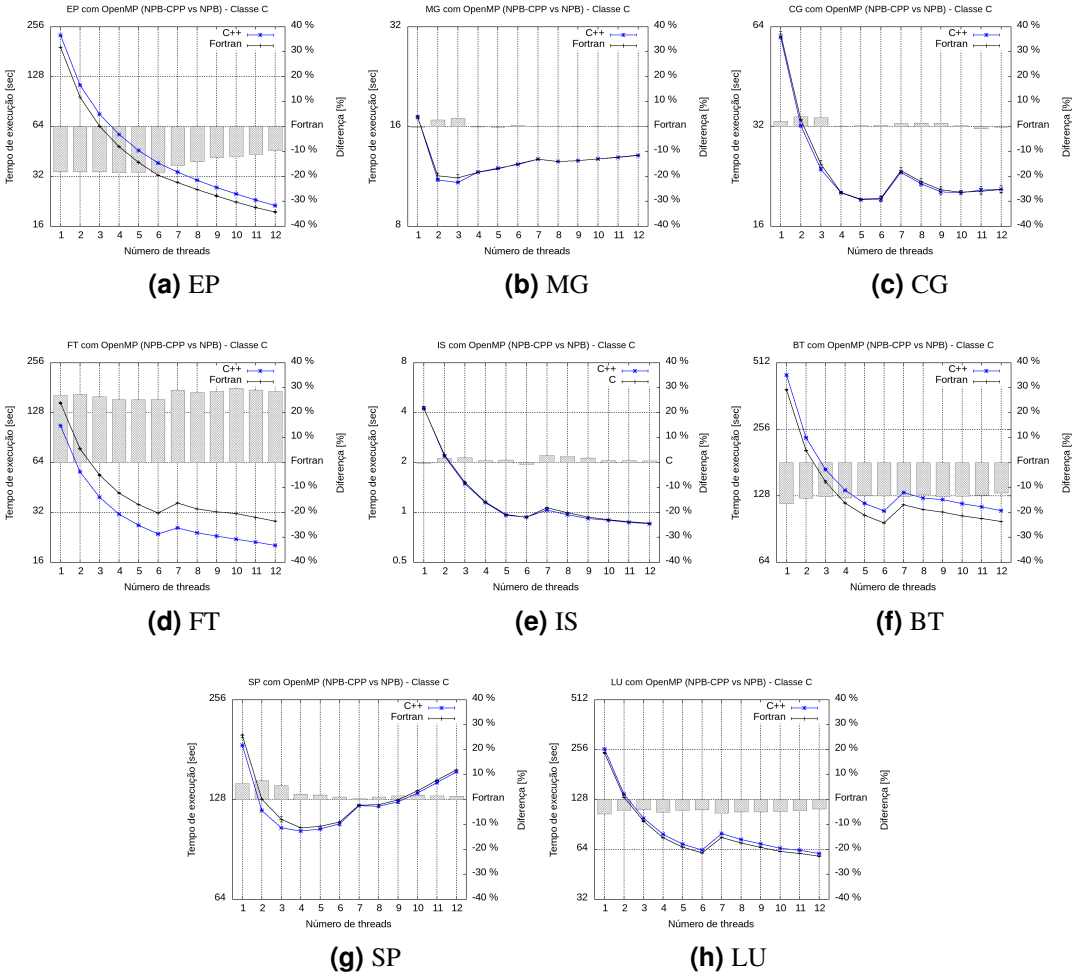
(a) EP **(b)** MG **(c)** CG

(d) FT **(e)** IS **(f)** BT

(g) SP **(h)** LU

Figura 4. Gráficos de comparação do NPB com o NPB-CPP na classe B.

.



(a) EP **(b)** MG **(c)** CG

(d) FT **(e)** IS **(f)** BT

(g) SP **(h)** LU

Figura 5. Gráficos de comparação do NPB com o NPB-CPP na classe C.

.

4. Resultados e análise de dados

Nesta seção, os resultados dos experimentos serão apresentados, juntamente com uma análise metódica das informações que foram obtidas através deles, e também comparações com os resultados obtidos no [Löff et al. 2021].

A partir dos experimentos realizados executando o NPB e NPB-CPP com a classe de trabalho B, obteve-se as informações necessárias para a criação dos gráficos presentes na Figura 4. Do mesmo modo, ao executar com a classe C, foi criado os gráficos da Figura 5.

Primeiramente, para visualizar as informações das Figuras 4 e 5, é necessário interpretá-las da seguinte forma, cada gráfico individual representa uma aplicação e em todos eles a configuração é a mesma. Conforme já mencionado, as linhas representam a velocidade de execução do NPB e NPB-CPP em função do número de *threads*. Em cada ponto definido nas linhas, está presente o desvio padrão das execuções. Ainda sim, atrás das linhas, as barras demonstram a porcentagem de variação entre as linhas tendo como referência a linha do NPB, dessa forma, se a barra indica um valor negativo significa que a versão original é mais rápida, e caso contrário, indica que a versão em C++ é mais eficiente. Por fim, é necessário ressaltar que o eixo do tempo esta plotado em escala logarítmica.

4.1. Classe B contra classe C

A primeira análise feita, é em relação aos testes do NPB-CPP em classes diferentes (B e C). No geral, por se tratar das mesmas aplicações e mesmo *hardware*, sendo apenas a carga de trabalho a diferença, os gráficos praticamente mantiveram o mesmo padrão, ou um muito similar.

Uma observação que deve ser feita é que para a pseudo-aplicação SP, houve uma diferença mais aparente no padrão gráfico. Neste caso, para a classe C o tempo de execução aumentou mais drasticamente após a etapa de *Hyper-Threading* (para este *hardware*, começa a partir da utilização de mais de 6 *threads*), onde os núcleos começam a trabalhar com mais de uma *thread* por vez. Essa mudança provavelmente está relacionada com a alguma característica do próprio *Benchmark* no momento de distribuir as tarefas, pois o aumento ocorreu tanto para versão em Fortran quanto para a versão em C++, e o desvio padrão em todas as execuções se manteve muito baixo.

4.2. Original contra reprodução

Ao realizar a comparação entre os resultados obtidos com a classe C na plataforma Xeon no [Löff et al. 2021], com os resultados obtidos na plataforma Ostrich, presente na Figura 5, pode-se observar alguns detalhes importantes.

Nota-se, para a plataforma Ostrich, que em todas as execuções, o desvio padrão ficou muito baixo, a ponto de mal ser percebido nos gráficos, enquanto para a Xeon algumas pequenas alterações podem ser observadas nas execuções do MG e CG, porém nada muito significativo.

Percebe-se, no momento em que o *Hyper-Threading* é ativado (quando um dos núcleos começa a usar mais de uma *thread*), ocorre um ligeiro aumento no tempo, e conforme os outros núcleos também aderem a esse processo, o tempo volta a descer. Esse padrão se manteve em todos os gráficos e acontece por conta de um mal balanceamento na divisão de tarefas entre as novas *threads* geradas.

Em relação a comparação das velocidades em si do [Löff et al. 2021] com a reprodução feita, verifica-se que para os *Benchmarks* BT, SP e LU os resultados foram muito parecidos, mantiveram as mesmas informações, tanto na verificação da versão mais rápida quanto na porcentagem da diferença de entre as velocidades observadas nas barras.

Quando observamos os resultados do IS, MG e CG, no artigo referenciado as execuções mostram resultados um pouco mais variados e irregulares, com porcentagens mais significativas e maiores diferenças entre as versões NPB e NPB-CPP. Enquanto para a plataforma Ostrich, essa diferença não ficou tão evidente, o tempo de execução ficou parecido e assim consequentemente gerando uma variação muito baixa na indicação da diferença em porcentagem entre as linhas.

Para o FT, em ambas as plataformas a versão C++ foi consideravelmente melhor, entretanto, percebe-se uma variação muito grande no percentual de diferença, quando observado o teste na Ostrich. Essa variação bate mais de 30% em alguns pontos, enquanto nos testes com a Xeon chegou no máximo um pouco mais de 10%.

Por último, em relação ao EP, houve uma notória inversão em relação a qual versão do *Benchmark* foi mais rápido. Enquanto para a plataforma Xeon a versão em C++ foi melhor para praticamente todos os níveis de paralelismo e com uma baixa porcentagem de diferença, quando observados os testes na Ostrich observa-se outra coisa. Nota-se que a versão em Fortran foi ligeiramente mais rápida, batendo um percentual de quase 20% em alguns pontos.

5. Conclusões

Esse artigo proporciona uma reprodução dos experimentos realizados pelo [Löff et al. 2021], para a verificação da eficiência do *NAS Parallel Benchmarks*, traduzido em C++ com a uso do *framework* OpenMP. Fora isso, ainda apresenta uma breve comparação entre os padrões de comportamento encontrados quando a carga de trabalho das execuções é alternada da classe B para a classe C.

A partir disso, com os testes realizados, concluiu-se que ao trocar a carga de trabalho envolvida nos testes, com a plataforma Ostrich, o comportamento gráfico tende a se manter o mesmo, aumentando o tempo de maneira proporcional. Ainda assim, algumas pequenas diferenças são causadas por características particulares dos próprios *Benchmarks*.

Em relação à comparação entre as execuções do [Löff et al. 2021] na plataforma Xeon, e a replicação na Ostrich, pode-se dizer que o NPB e o NPB-CPP possuem tempos de execução muito próximos em todos os testes, e que as diferenças percebidas devem-se às características individuais de cada plataforma onde foram realizados os experimentos.

Um adendo que pode ser feito, é em relação à estabilidade das máquinas, que apresentaram um desvio padrão muito baixo nos testes, se mostrando muito confiáveis e estáveis.

Finalmente, como proposta para pesquisas futuras existe a ideia de aplicar esses mesmos testes envolvendo o NPB e NPB-CPP, em ambientes embarcados. Mais especificamente, comparar as execuções em uma placa Raspery-PI e em uma NvidiaJdson, pois ainda não existe muitos estudos ou pesquisas sobre o processamento paralelo nesse tipo de *hardware*.

Agradecimentos

O desenvolvimento deste artigo científico contou com a ajuda de diversas pessoas, dentre as quais agradecemos:

Ao professor orientador Dalvan J. Griebler que nos acompanhou semanalmente, dando o auxílio necessário para a elaboração do projeto, juntamente com os mestrandos Júnior Löff e Renato Hoffmann. Agradecemos também a Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), que nos disponibilizou toda a infraestrutura e o ambiente de estudos para o projeto, bem como o Grupo de Modelagem de Aplicações Paralelas (GMAP), que nos proporcionou os computadores e servidores para os testes presentes no artigo.

Referências

Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2017). Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing*.

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M. (1995). The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center.

Chapman, B., Jost, G., and Van Der Pas, R. (2007). *Using OpenMP: portable shared memory parallel programming*. MIT press.

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.

Voss, M., Asenjo, R., and Reinders, J. (2019). *Pro TBB: C++ parallel programming with threading building blocks*. Springer.