Analisando o NAS Parallel Benchmarks em C++ e Fortran com OpenMP em Sistemas Embarcados

[1]Eduardo M. Martins [1]Eduardo F. Eichner [1]Bernardo P. Fiorini [1]Dalvan 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 study dedicated to analyzing the behavior of efficiency tests with NAS Parallel Benchmarks in C++ and Fortran, using OpenMP on an embedded system called Raspberry Pi 4, with an ARM processor. Based on this, the results obtained were analyzed based on previous studies carried out on multi-color platforms.*

Resumo. *Esse artigo proporciona um estudo dedicado a analisar o comportamento de testes de eficiência com o NAS Parallel Benchmarks em C++ e Fortran, utilizando o OpenMP em um sistema embarcado denominado Raspberry Pi 4, com um processador ARM. A partir disso foi analisado os resultados obtidos com base em estudos anteriores feitos em plataformas multicores.*

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 outros exemplos conhecidos e que servem de referência, como o Intel TBB [Voss et al. 2019], e o FastFlow [Aldinucci et al. 2017], ambos 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 compila-

dor 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.

Em adição, tendo em vista a acessibilidade e popularidade dos sistemas embarcados, que são basicamente compostos pela combinação do *hardware* e *software* designados para uma função específica, no geral microcontroladores como as placas Raspberry Pi, NividiaJetson e até mesmo o próprio Arduino, por exemplo. Nessa pesquisa, explorou-se o cruzamento dos temas, visto que é uma área pouco trabalhado dentro do contexto da computação e programação paralela.

Dessa forma, a partir do que foi descrito surge a motivação para o estudo com o objetivo de analisar a eficiência de um *hardware* embarcado baseado em um processador ARM, através de testes para obter o desempenho do mesmo, a partir do uso dos *benchmarks* mencionados anteriormente e também de comparações com estudos prévios realizados no passado. Portanto, a partir do que foi descrito, definiu-se como objeto base para o estudo, uma placa Raspberry PI 4 Model B.

Finalmente, este artigo está dividido em cinco seções, sendo elas Introdução, Fundamentação teórica, Metodologia, 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++[1], respeitando a estrutura original dos *kernels* e pseudo-aplicações. Além disso, os pesquisadores 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.

Além disso, no que se refere aos sistemas embarcados, no [Papakyriakou et al. 2018], verificou-se em testes com o Raspberry Pi 2, com um processador ARM, ele apresentou resultados promissores quando testado com o *High-Performance Linpack Benchmark* (HPL) [Dongarra et al. 2003], usando processamento paralelo, firmando a escolha do objeto de estudo.

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

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.

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 S, para a realização de alguns testes rápidos, descritos nas próximas seções, e as classes B e C para os testes finais.

2.2. OpenMP

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*.

2.3. Map e Map Reduce

Nesta seção será discutido brevemente sobre as estruturas de *Map* e *Map Reduce*, bem como será demonstrado um exemplo da paralelização utilizando o OpenMP.

A estrutura *Map* consiste em uma forma de *loop* que itera uma determinada quantidade de vezes, e cada uma das iterações é independente uma das outras, ou seja, nenhuma variável é reaproveitada ou dividida entre os indexes durante as repetições. Para representar a paralelização do *Map*, É disponibilizado no *framework* uma estrutura denominada "*parallel for*", para referenciar a região a ser paralelizada. A notação *pragma* pode ser observada, conforme mostra a Figura 1.

```
//OpenMP parallel for
#pragma omp parallel for
for (i=0; i<n; 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* no *framework* trabalhado, 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...
}
}
```

Figura 2. Map Reduce paralelo.

2.4. Raspberry Pi

Dentro do universo dos sistemas embarcados, o Raspberry Pi é uma solução muito popular e acessível, sendo um microcomputar de código aberto com processador ARM. O uso desta ferramenta é voltado para diversas funções específicas, conforme as necessidades do usuário.

Referente ao processamento paralelo, ele já foi alvo de alguns estudos, onde foi trabalhado o conceito de *cluster's* utilizando o Raspberry, como no [Papakyriakou et al. 2018], onde foi criado um *cluster* com doze Raspberry Pi's e realizado testes com o *High-Performance Linpack Benchmark* (HPL) [Dongarra et al. 2003]. Já no [Doucet and Zhang 2017], foram utilizados cinco deste dispositivo, para verificar o *speed up* em um teste específico que calula o valor de Pi.

Além disso, em outros estudos, como no [Matthews et al. 2018], pode-se observar discussões sobre como o Raspberry pode ser utilizado para introduzir o contexto de programação paralela para estudantes.

Entretanto, ainda existe o interesse em verificar o comportamento da execução do *NAS Parallel Benchmark* em um Raspberry Pi 4, com o conceito de programação paralela do OpenMP para C++ e Fortran. Esses testes, tendo como base o que foi desenvolvido anteriormente em experimentos como os realizados pelo [Löff et al. 2021], e alguns estudos prévios[2].

[2]Resultado do estudo prévio: https://github.com/Eduardo-Machado-Martins/NPB-IOT-Scripts-1.0.git.

3. Metodologia

Nesta seção será demonstrado os experimentos realizados de forma detalhada, bem como será descrita qual foi a metodologia empregada para obter os resultados.

3.1. Etapas

Para realização dos testes, foi necessário a criação de uma metodologia específica. Ao iniciar a pesquisa, definiu-se alguns objetivos principais, como definir qual plataforma embarcada 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. Testar o *Shell Script* com a Classe de trabalho S, antes de realizar os testes finais. Criar os gráficos no Gnuplot a partir dos dados coletados, para ajustar a plotagem. Por último, rodar na plataforma definida as classes de trabalho B e C, plotar os dados definitivos e discutir e comparar os resultados. Dessa forma o *WorkFlow* de tarefas pode ser conferido no fluxograma da Figura 3.

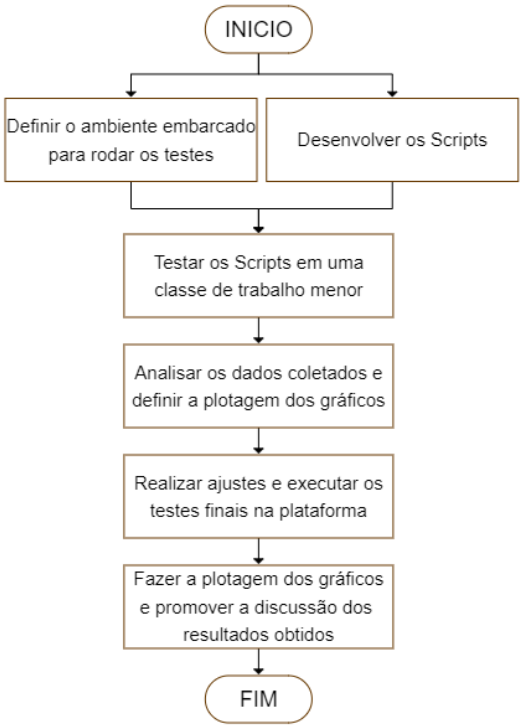


Figura 3. Fluxograma de tarefas.

3.2. Ambiente de testes

Conforme já mencionado anteriormente, o ambiente embarcado definido para rodar os testes foi uma placa Raspberry Pi 4 Model B disponibilizada pelo Grupo de Modelagem e Aplicações Paralelas (GMAP) da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS).

Para trabalhar com esse dispositivo, foi necessário instalar um sistema operacional, dessa forma, por ser um ambiente para realização de testes, optou-se por instalar no microSD do Raspberry um Ubuntu Server versão 20.04 para 64bit.

Em relação ao *hardware* disponível na placa, é importante saber que ela possui uma arquitetura ARM, que contém um *socket* com um processador Cortex-A72 de 4 núcleos sem a tecnologia de *Hyper-Threading*. Além disso, possui uma velocidade de *clock* interno máximo de 1,8GHz e uma memória RAM de 8GB.

3.3. Método de execução

Na sequência, criou-se o *Shell Script* para executar os *Benchmarks*[3]. 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

[3]Repositório do *Script* criado: https://github.com/Eduardo-Machado-Martins/NPB-IOT-Scripts-2.0.git.

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 classe de trabalho menor, 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++, no próprio Raspberry Pi 4, 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, 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".

Finalmente, após a realização de todas as etapas anteriores à essa, foi executado primeiramente a classe C na plataforma embarcada. A excussão dessa etapa levou cerca de oito dias para ser finalizada. Após esse período, foi executado a classe de trabalho B, que levou cerca de quatro dias para ficar pronta. É importante ressaltar que durante esse período a plataforma foi utilizada exclusivamente para os 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.

3.4. Base de testes em estudos anteriores

Antes de analisar os resultados, nesta seção será brevemente descrito a metodologia implementada pelo [Löff et al. 2021], que foi a base para a realização deste projeto, e que servirá de parâmetro para possíveis comparações com plataformas multi-cores.

Na execução dos testes de performance realizadas no artigo mencionado, que compara o NPB com o NPB-CPP, os pesquisadores obtiveram os resultados em uma plataforma com as seguintes características. 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* (possui *Hyper-Threading*). 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.

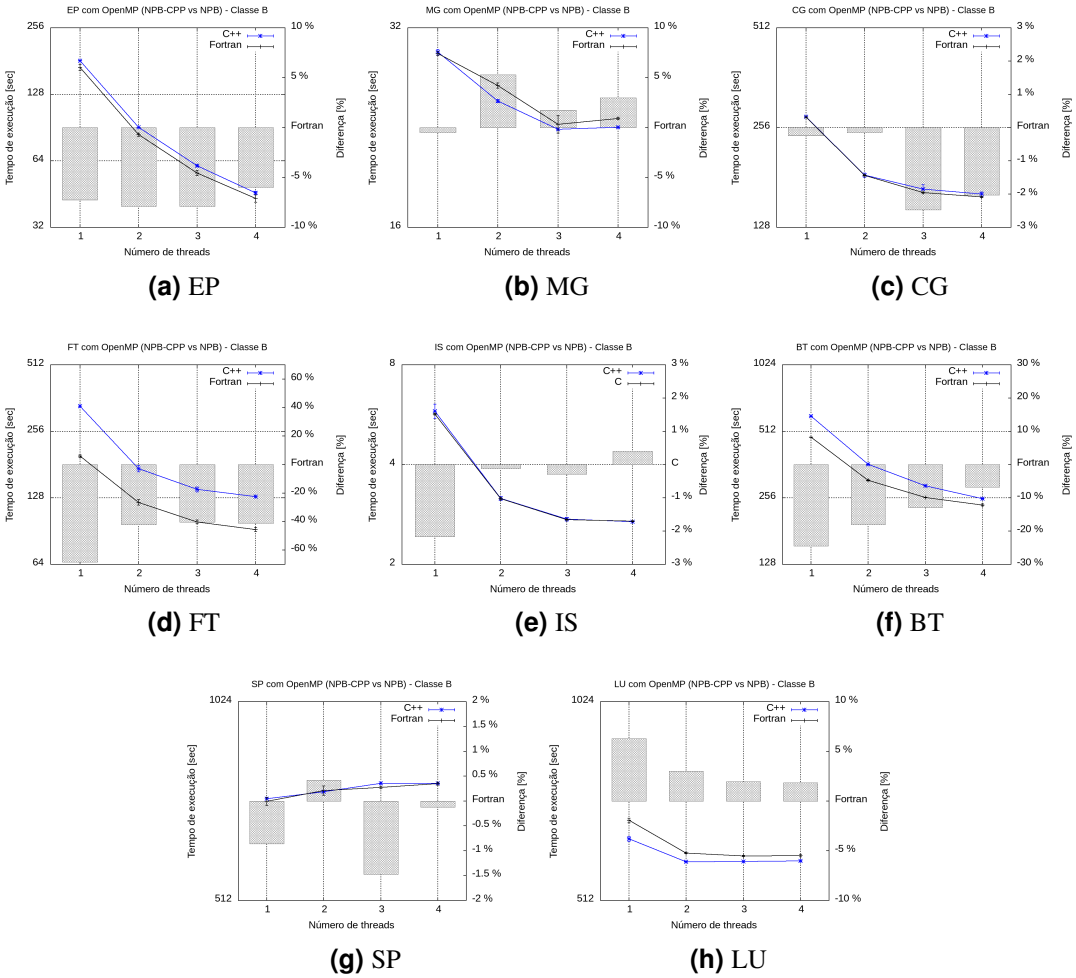
Com esse *hardware* definido, os pesquisadores do [Löff et al. 2021] rodaram cada um dos oito *Benchmarks* na classe C cinco vezes para cada número de *threads* disponível (de 1 até 48 *threads*), tanto para a versão original em Fortran quanto para a versão em C++, portanto o *frameworks* utilizado para 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.

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 algumas breves 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.



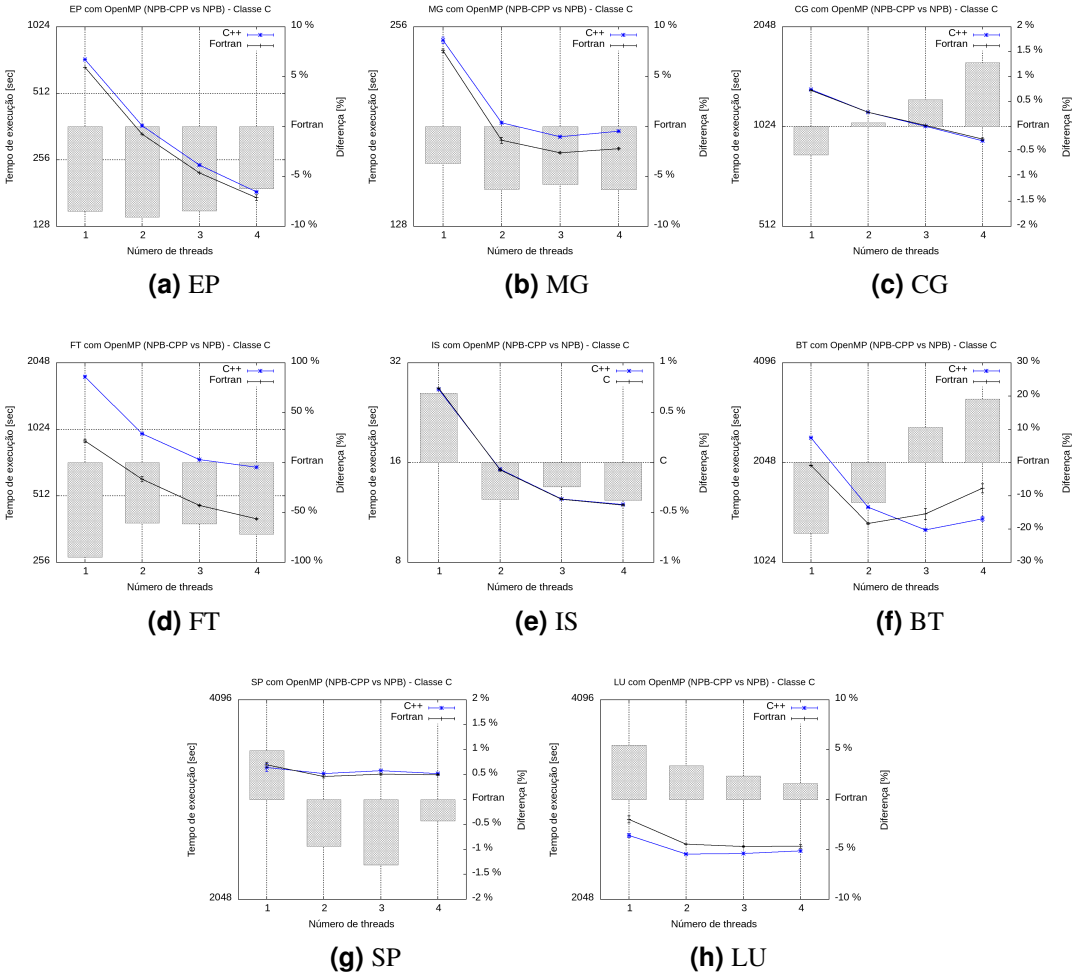
(a) EP **(b)** MG **(c)** CG

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

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

Figura 4. Comparação do NPB com o NPB-CPP na classe B com Raspberry Pi 4.

.



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

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

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

Figura 5. Comparação do NPB com o NPB-CPP na classe C com Raspberry Pi 4.

.

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 mo geral ficaram com padrões similares, com algumas exceções. Entretanto, obviamente como a carga de trabalho da classe C é cerca de quatro vezes maior do que a classe B, os tempos de execução foram maiores para a classe C.

Para as aplicações EP, MG, CG, IS, SP e LU, os padrões se mantiveram muito próximos, todos com valores muito parecidos até mesmo no percentual de diferença.

Em relação ao *kernel* FT, observa-se um comportamento similar em ambos gráficos, onde a versão do NPB em Fortran foi ligeiramente mais veloz do que a versão traduzida para C++. Entretanto, ao mudar da classe B para classe C, essa diferença aumentou drasticamente, batendo mais de 20% em alguns pontos.

Para a pseudo-aplicação BT, houve uma mudança não só no percentual de diferença, como também no padrão gráfico. Quando executado na classe B, a versão em Fortran foi mais eficiente para os quatro níveis de paralelização. Na classe C, a partir da terceira *thread* a versão C++ se saiu melhor, invertendo o percentual de diferenças.

Após algumas análises, concluiu-se que essas variações notadas entre as classes de trabalho estão relacionadas 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. Raspberry Pi contra multi-core

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

Mesmo sendo plataformas completamente diferentes, pode-se observar alguns características interessantes ao comparar as aplicações em uma plataforma multi-core. Primeiramente, nota-se que na Xeon é possível identificar o padrão gráfico gerado pela etapa de *Hyper-Threading*, após um núcleo começar a utilizar mais de uma *thread*, fato que não é percebido no Raspberry, pois ele não possui essa tecnologia.

Além disso, os tempos de execução percebidos foram bastante diferentes, pois embora seja a mesma classe de trabalho sendo aplicada, as plataformas possuem características completamente diferentes. Dessa forma, comforme o esperado, por ser uma máquina com um *hardware* bastante superior ao Raspberry Pi, a plataforma Xeon atingiu melhores tempos de execução.

Por fim, em relação aos padrões gráficos, por mais que seja difícil comparar, visto a diferença no número de *threads* observada entre as máquinas, percebe-se alguns fatores. Nas duas plataformas, no geral as execuções do NPB e NPB-CPP foram muito próximas, e com desvio padrão baixo, demonstrando a confiabilidade das máquinas.

4.3. Análise geral

De maneira geral, observando e analisando as Figuras 5 e 6, nota-se um desvio padrão consideravelmente baixo, a ponto de quase não ser diferenciado dos pontos nos gráficos.

Além disso, na execução da pseudo-aplicação SP, percebe-se um padrão gráfico muito diferente do que foi encontrado em todos outros testes, pois apresentou um tempo de execução muito parecido para cada nível de paralelismo. Com exceção do que foi mencionado, e da variação do percentual de diferença entre as execuções do NPB e NPB-CPP verificado nos gráficos referentes ao *kernel* FT, os resultados foram considerados satisfatórios.

5. Conclusões

Esse artigo proporciona um estudo do comportamento das execuções dos *Benchmarks* da NAS, baseado nos experimentos realizados pelo [Löff et al. 2021]. Com isso, verificou-se a eficiência do *NAS Parallel Benchmarks*, traduzido em C++ com a uso do *framework* OpenMP, em uma placa Raspberry Pi 4 Model B. Assim, a partir disso realizou-se uma análise do processamento paralelo em um sistema embarcado. Essas análises foram feitas em cima de comparações entre as execuções de duas classes de trabalho diferentes (B e C), bem como com a comparação dos resultados obtidos em uma plataforma multi-core realizados pelo [Löff et al. 2021].

A partir disso, com base nos resultados obtidos, concluiu-se que ao trocar a carga de trabalho envolvida nos testes, com o sistema embarcado Raspberry Pi, o comportamento gráfico tende a se manter o mesmo, aumentando o tempo de maneira proporcional. Ainda assim, algumas pequenas diferenças e avarias 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 no Raspberry Pi, pode-se dizer que o NPB e o NPB-CPP possuem tempos de execução muito próximos em todos os testes, e que as grandes 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 testes envolvendo o NPB e NPB-CPP, e fazer comparações de *hardware* com outras placas, mais especificamente, utilizando o Raspberry Pi e uma NvidiaJdson, pois ainda não existe muitos estudos ou pesquisas sobre o processamento paralelo nesse tipo de *hardware*. Além disso, é necessário aprofundar a pesquisa pra entender quias são as características dos *Benchmarks* que causas as alterações nos padrões gráficos, bem como entender por que algumas paralelizações não foram tão eficientes.

Agradecimentos

A criação deste artigo contou com a ajuda de diversas pessoas, as quais agradecemos:

Ao professor orientador Dalvan Griebler que nos acompanhou semanalmente, dando o auxílio necessário, juntamente com os mestrandos Júnior Löff e Renato Hoffmann. Agradecemos também a Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), que nos disponibilizou todo o ambiente de estudos, bem como o Grupo de Modelagem de Aplicações Paralelas (GMAP), que nos proporcionou os equipamentos para os testes.

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.

Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820.

Doucet, K. and Zhang, J. (2017). Learning cluster computing by creating a raspberry pi cluster. In *Proceedings of the SouthEast Conference*, pages 191–194.

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.

Matthews, S. J., Adams, J. C., Brown, R. A., and Shoop, E. (2018). Portable parallel computing with the raspberry pi. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 92–97.

Papakyriakou, D., Kottou, D., and Kostouros, I. (2018). Benchmarking raspberry pi 2 beowulf cluster. *International Journal of Computer Applications*, 179(32):21–27.

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