Universidade do Minho

Engenharia dos Sistemas de Computação

1º Trabalho

Grupo 8

Bruno Ferreira PG41065
Vasco Figueiredo PG41102

13 de abril de 2020

Índice

1. Resumo

Neste relatório, temos como principal objetivo estudar os impactos de aplicações num sistema de computação. O relatório tem duas tarefas principais. Como primeira tarefa, vamos utilizar o pacote NAS Parallel Benchmarks (NPB) para avaliar a performance dos vários programas no sistema, verificando o impacto das várias versões dos programas, e analisando também as diferentes opções de compilação, as diferentes tecnologias de comunicação e diferentes dimensões de dados. Como segunda tarefa, iremos avaliar a performance do sistema durante a execução dos programas falados na parte anterior. Com a ajuda de comandos, iremos analisar o estado do sistema e como este é afetado pelos vários programas.

2. Introdução

Nesta primeira parte do relatório, temos como objetivo analisar a performance dos vários programas do pacote NAS Parallel Benchmarks (NPB).

NAS Parallel Benchmarks é um pequeno conjunto de programas desenvolvidos para ajuda a avaliar a performance de supercomputadores criado pela NASA. As "benchmarks" são derivadas de "computational fluid dynamics" (fluidodinâmica computacional), "meshes" adaptáveis não estruturadas, I/O paralelos, aplicações "multi-zone" e grelhas computacionais. O pacote consiste em 5 kernels (IS, EP, CG, MG, FT) e 3 pseudo-aplicações (BT, SP, LU). O programa inclui também versões "multi-zone" que são programas projetados para avaliar paralelismo em múltiplos níveis ou implementações híbridas de paralelismo. Cada programa do pacote possui várias classes de benchmark, que vão modificar o tamanho (ou peso) computacional do teste. Estas classes são: S, que possui um tamanho pequena, ótima para pequenos testes; W, que possui o tamanho de uma "workstation" de anos 90; A, B e C, que tem um tamanho padrão, aumentando 4x o tamanho entre cada uma; e finalmente, D, E e F, que tem um tamanho grande, 16 vezes maior do que cada uma das classes anteriores.

Como só era necessário escolher alguns destes programas, decidimos utilizar para testes, os programas EP (um programa embaraçosamente paralelo), MG (uma "multi-grid" numa sequência de "meshes", com comunicações de pequena e longa distância, sendo estes testes "memory intensive") e BT (um "Block Tridiagonal solver"), o que nos pareceu na altura escolhas distintas entre si que iam dar resultados distintos, com as classes W, A e B.

Para efetuar as várias medições e gráficos, os vários programas foram testados no cluster SeARCH, nas máquinas 641. As máquinas possuem um dual cpu Intel Xeon E5-2650 v2 com 16 cores e 64 GiB de memória.

Para obter os valores, foram efetuados 10 testes e depois foi calculada a média dos 5 melhores resultados. Apesar do número de testes ser relativamente pequeno, valores obtidos com um conjunto maior de testes apresentam os resultados extremamente parecidos (com variações mínimas).

2.1. Compilação

O primeiro fator que vamos avaliar é o impacto da flag -O na otimização de código sequencial. Estas otimizações são efetuadas pelo compilador e para ativar, é necessário indicar a flag O[n] pretendida no código de compilação (n representa o nível de otimização). Nesta secção iremos avaliar os primeiros 4 níveis (0, 1, 2 e 3). Para mais à frente podermos entender os resultados, iremos falar brevemente sobre cada nível.

- Com o nível 0, apenas disponível em sistemas Linux, o compilador desativa todas as otimizações possíveis. Isto reduz o tempo de compilação apesar de gerar o código não otimizado.
- Com o nível 1, o compilador otimizada o tamanho do código, melhora a "code locality" e melhora o tempo de execução. Além disso desativa o "loop unrolling". Este nível otimiza moderadamente, sem reduzir o tempo de compilação drasticamente. Mais adequado para aplicações com tamanhos

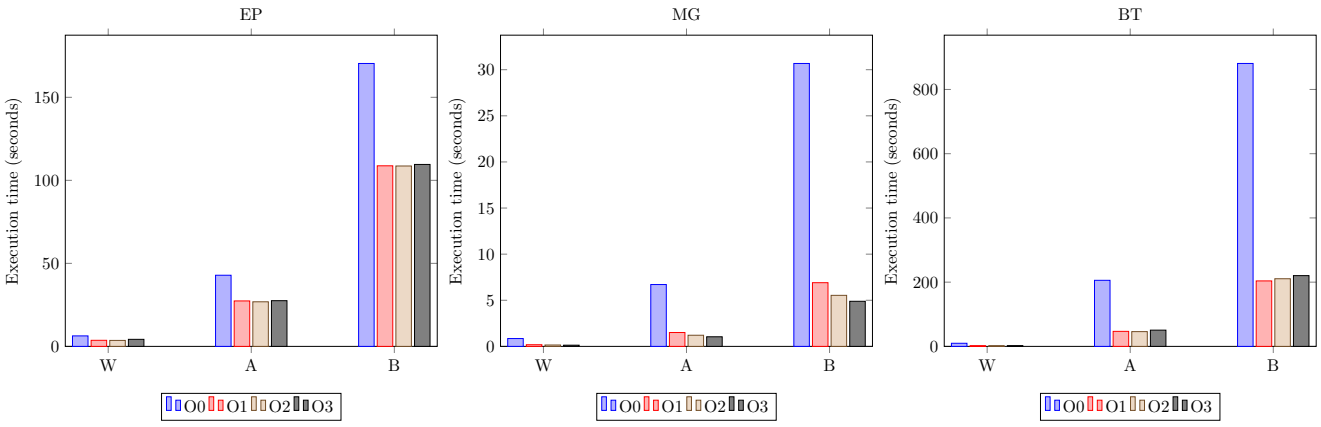
grande de código, várias branches e tempos de execução pouco influenciados por código dentro de loops.

- Com o nível 2, o compilador otimiza a velocidade do código. Esta é a otimização padrão. Este nível permite várias otimizações, tais como: "dead-code elimination", "loop unrolling", "variable renaming", "exception handling optimizations", "peephole optmizations" entre várias outras. Em sistemas baseados em Itanium (microprocessador da Intel), esta opção permite otimizações à velocidade, "software pipelining", especulação e "predication". Em geral, faz uma otimização completa, mas tem o pior tempo de compilação.

- Com o nível 3, o compilador utiliza todas as otimizações do nível 2, mas permite o uso de otimizações mais agressivas, tais como "prefetching", transformações de acessos à memoria e loops, "scalar replacement" e tenta vetorizar loops. Este nível pode não melhorar a performance do programa, a não ser que este possua loops ou acessos à memoria. Pode apresentar resultados piores do que apenas usar o nível O2. Esta opção é melhor usada em aplicações com loops que usem bastante cálculos de virgula flutuante e que processem grandes data sets.

2.2. Resultados

Correndo os testes com as definições que foram ditas anteriormente, obtemos os seguintes resultados para cada tipo de teste e classe de tamanho em função dos tempos de execução:



Como podemos observar, qualquer otimização é melhor que nenhuma sendo o impacto bastante grande independentemente do tipo de teste e a classe de tamanho. Mas como podemos analisar nos gráficos, não há uma otimização que seja melhor que as outras em todos os casos, sendo que os resultados mudam de aplicação para aplicação.

No caso do EP, nenhum nível teve um desempenho significativamente melhor do que as outras, tendo o nível O2 obtido resultados ligeiramente melhores. Já nas outras 2 aplicações nota-se uma diferença maior.

No caso da aplicação MG, o nível O3 obteve resultados bastante melhores que os outros dois níveis, em todas as classes. Isto pode dever-se ao código do MG possuir vários loops ou beneficiar bastante de vetorização, visto que o nível O3 beneficia bastante códigos que dependam de loops ou que possam ser vetorizados.

Já na aplicação BT, os resultados foram ao contrário: o nível O2 obteve resultados melhores do que os outros dois níveis nas duas primeiras classes (mesmo que a diferença não tenha sido tão significativa para o O1), mas perde para o O1 na classe B. A perda de performance na classe B nos níveis superiores de otimização, pode ser uma consequência das estratégias mais agressivas que estes níveis possuem. Ao

usar estratégias de compilação mais agressivas, o compilador pode estar a prejudicar mais o tempo do que a ajuda-lo, visto que o nível O1, que possui uma otimização mais moderada, apresenta tempos melhores.

Estes gráficos mostram que não existe um nível de otimização superior, sendo que o tipo de código usado pode beneficiar mais de um nível do que de outros. No entanto, provam que usar uma otimização é definitivamente melhor, visto que os resultados são bastante mais lentos sem qualquer tipo de otimização.

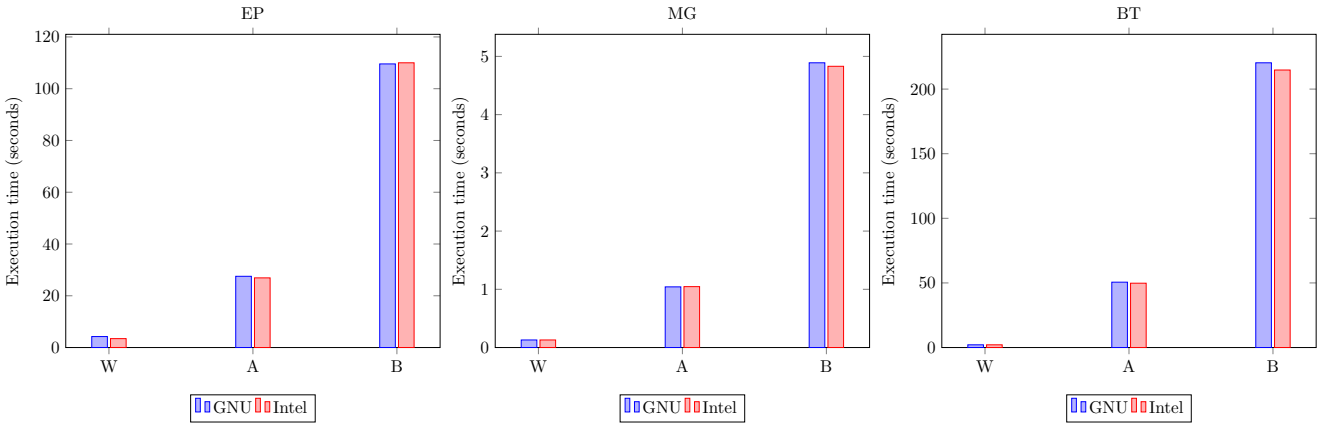
3. Compilador

O segundo fator que vamos avaliar é o uso de diferentes compiladores, mais especificamente, avaliar o uso do compilador da GNU e do compilador compilador da Intel.

O GNU Compiler Collection (GCC) é um compilador produzido pela GNU Project. Originalmente desenvolvido para o sistema operativo GNU, este compilador suporta várias línguas de programação e várias sistemas operativos. Foi desenvolvido para ser um software 100% grátis mas que respeita a liberdade do utilizador. O Intel C++ Compiler foi criado pela Intel Corporation, podendo ser usado para compilar aplicações escritas em C e C++. Tal como o GCC, está disponivel para Linux, Windows e Mac Os X.

Para efetuar estes testes, executamos mais uma vez as três aplicações (com o nível de otimização O3), mas utilizando os dois compiladores.

3.1. Resultados



Como podemos observar nos gráficos, em quase todos os casos, o compilador da Intel provou ter melhores resultados, apesar da diferença ter sido pequena. Os únicos casos aonde o compilador da GNU foi superior ao compilador da Intel foi na aplicação EP, classe B e na aplicação MG classe A. Só com estes dados, é difícil explicar ao certo porque é que o compilador da Intel apresentou melhores resultados. Uma explicação possível é o facto do processador usado para os testes ser da Intel. Como o compilador é da mesma empresa que o processador, este pode usar melhor os recursos fornecidos pelo processador e muito provavelmente, estará mais otimizado para executar nestes processadores do que um compilador de outra empresa.

No pacote NPB, existem pelo menos 4 implementações diferentes para cada aplicação: uma sequencial, uma utilizando OpenMP, outra usando OpenMPI e uma usando uma implementação híbrida entre OpenMPI e OpenMP. Até agora, temos avaliado a versão sequencial das aplicações, para poder avaliar o impacto das diferentes opções de otimização ou de diferentes compiladores. Nas próximas secções,

iremos avaliar as implementações que beneficiam de *multithreading* e *multiprocessing* baseando-nos nas otimizações de compilação O3.

4. Multithreading

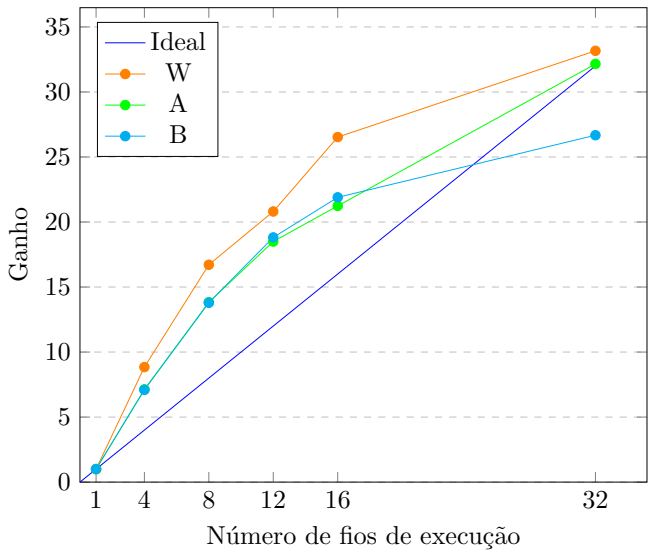
Multithreading é a habilidade de um CPU de criar vários "threads" (fios) de execução, capazes de executar tarefas em paralelo. Ao usar multithreading, podemos melhorar o tempo de execução de um programa, ao maximizar o uso de um core através de formas de paralelismo como "thread-level parallelism" e "instruction level parallelism".

Para utilizar multithreading, o pacote NPB usa a API OpenMP, que permite utilizar multithreading em programas escritos em C, C++ e Fortran.

Com o uso de multithreading, podemos distribuir o código de uma aplicação por um número de threads. Dependendo do grau de paralelização que a aplicação pode ter, com o aumento do número de threads, o código vai ter um ganho de performance em relação ao código sequencial. Se a aplicação o permitir, os ganhos de performance poderão ter os valores espectáveis, que numa aplicação que pudesse ser perfeitamente paralelizada, com 2 threads, seria 2, ou seja, o código ficaria 2 vezes mais rápido. Nos próximos gráficos, iremos comparar o ganho que a aplicação teve, em relação ao valor obtido sua versão sequencial, com o aumento do número de threads.

4.1. EP

A aplicação EP que, como representa o nome de *embaraçosamente paralelo*, é esperado escalar bastante bem com o aumento do número de fios de execução. Isto também se pode verificar como podemos ver nos seguintes gráficos dos nossos testes:



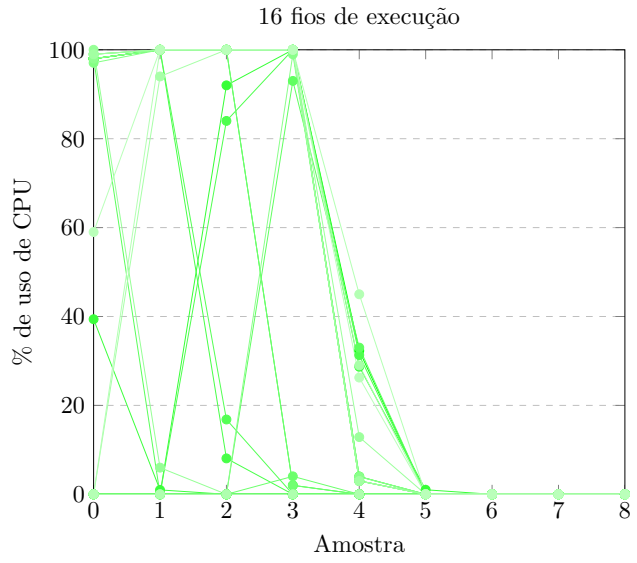
A maioria dos pontos no gráfico passam acima do que era esperado para um ganho ideal. Isto pode-se explicar com os nossos testes sequenciais terem sido possivelmente prejudicados por fatores externos que não estão no nosso controlo, como a temperatura dos processadores, o que levaria a uma imposição de um limite da velocidade de relógio. No entanto, decidimos não refazer estes testes não só por uma questão de trabalho e tempo mas também porque seria difícil realmente ter uma situação perfeita e queremos abstrair-nos dos resultados o máximo possível e não escolher à mão certos resultados.

Podemos ver que de facto este programa escala bastante bem, podendo-se observar talvez duas quebras pequenas no declive das curvas do ganho. Uma no 8 aonde passa-se da primeira socket da máquina para

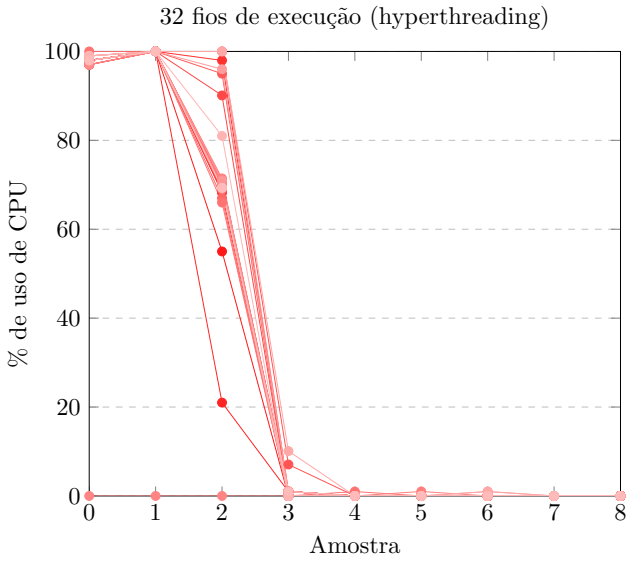
a segunda e no 16 aonde se passa de 16 núcleos físicos para 32 núcleos lógicos.

Mesmo tendo sido prometido como um programa altamente escalável, achámos interessante que mesmo com hyperthreading o ganho tenha sido substancial, embora não linear. Para observar melhor este facto, decidimos medir com o comando *mpstat -P ON 1* o uso dos núcleos lógicos dos CPUs. O argumento *-P ON* possibilita a visualização de todos os núcleos lógicos enquanto que o argumento final *1* define em segundo a taxa de amostragem, sendo que o comando irá repetir-se de 1 em 1 segundo.

Também só fizemos estas medições para 16 e 32 fios de execução pois o comportamento com menos seria expectável e daí desinteressante.



Média: 7.1647%



Média: 9.1335%

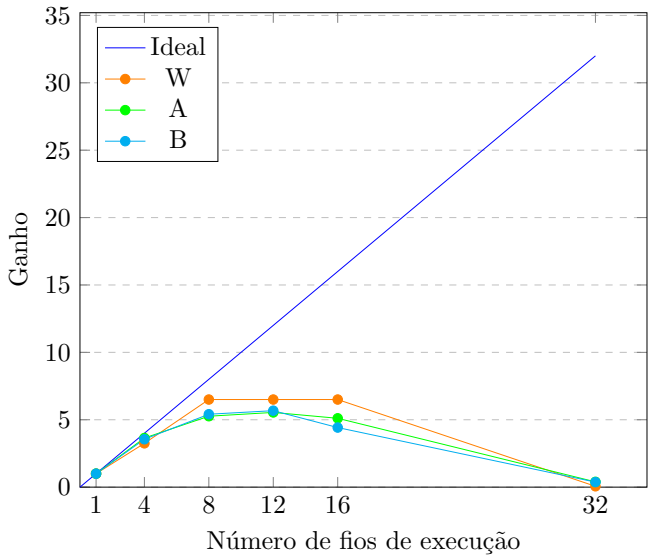
Cada linha no gráfico representa então o uso em percentagem de um núcleo lógico ao longo da execução do programa. As medições foram feitas de uma forma estática com uma duração de 30 segundos e, por isso, a média representa todas as amostragens e não só durante a execução do programa.

Pode-se observar no gráfico que o hyperthreading utiliza muito mais os núcleos do processador, sendo que na segunda amostragem temos quase todos os núcleos a 100% de utilização, havendo um que está perto dos 0% que não sabemos exatamente porquê.

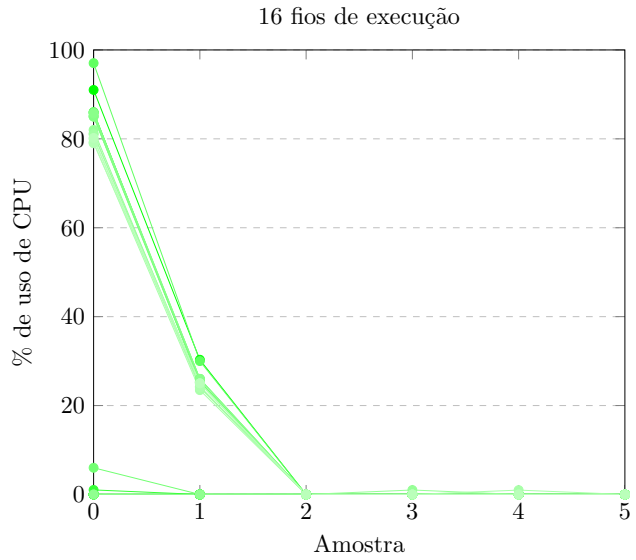
Algo que se pode também reparar no gráfico da esquerda, é que este tem muitas mudanças no processamento de núcleo para núcleo, sendo que um vai de 100% para 0% e outros fazem o contrário já a meio da execução do programa.

4.2. MG

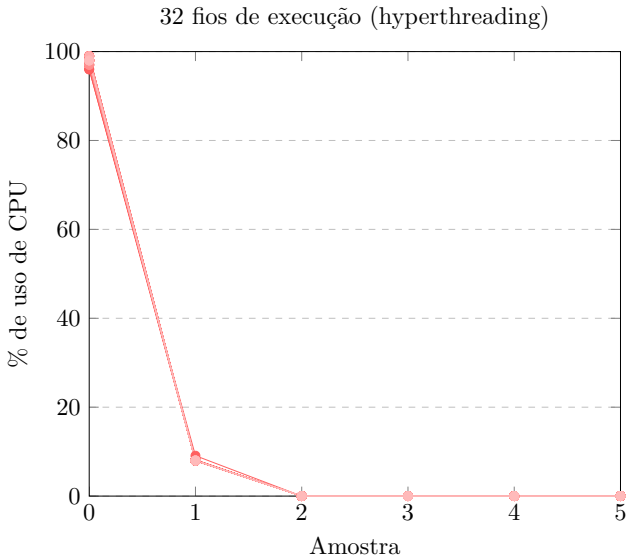
A aplicação MG de *multi-grid* é descrita como intensa em termos de memória e com comunicação o que significa algum certo nível de dependência de dados. Logo, é de esperar que esta aplicação não escale bem como a anterior.



Como se pode verificar no gráfico, esta aplicação tem ganhos bons até 8 fios de execução, mas a partir daí os benefícios de esperados de paralelismo já não aparecem, pois a curva do ganho estabiliza e desce com o uso de hyperthreading.



Média: 1.9147%

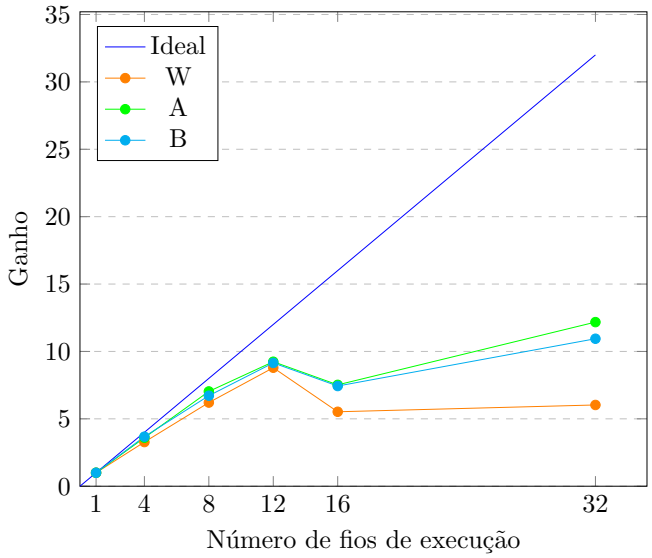


Média: 3.6612%

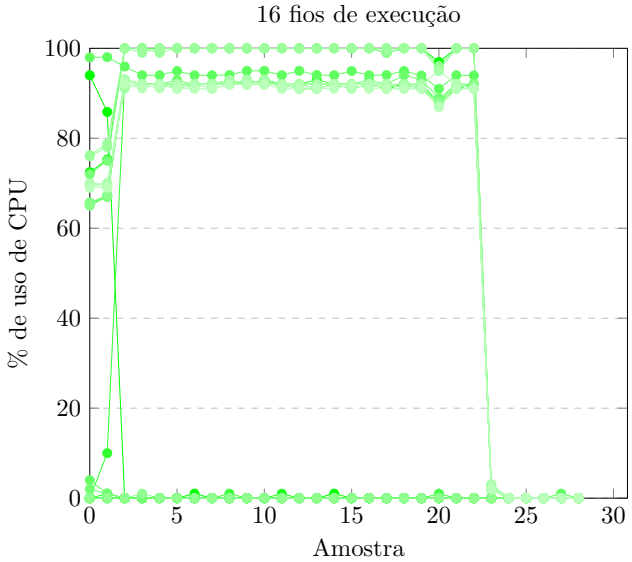
Apesar de termos medições dos usos percentuais dos núcleos da máquina, não os consideramos válidos pois temos demasiado pouca informação para tirar boas conclusões. Isto é devido à taxa de amostragem ser tão grande em relação ao tempo total de execução do programa, em que não podia ser menor pois 1 segundo era o mínimo e o programa em si corria em poucos segundos. O ideal seria ter corrido para um tamanho de classe maior exclusivamente para este teste, principalmente visto que este foi o único caso em que o hyperthreading não ajudou, no entanto, não conseguimos saber se é por causa de grão demasiado fino ou um problema do hyperthreading em si.

4.3. BT

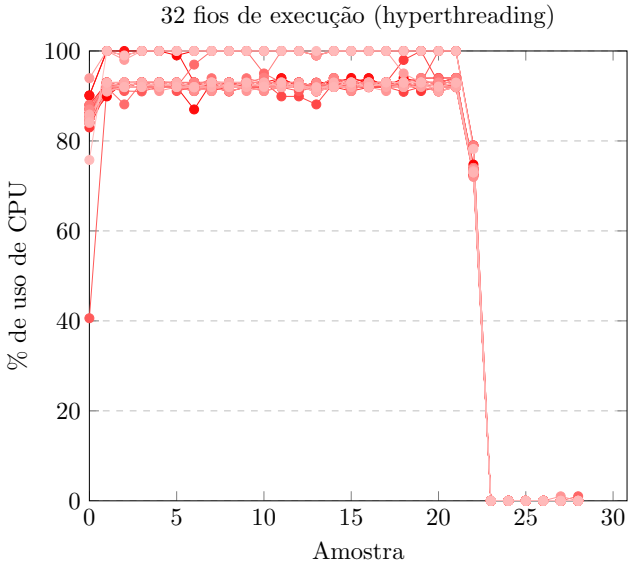
A aplicação BT de *block Tri-diagonal solver*, usada para resolver certos sistemas de equações, contém também dependência de dados limitando assim a sua escalabilidade. No entanto, esta aplicação ao ser compilada, reparámos que esta exige que a estrutura de dados bidimensional seja repartida igualmente por processos de uma forma quadrada.



Devido à forma como a estrutura de dados é repartida (em sequência de quadrados como 1, 4, 9, 16, etc...), seria de esperar que com os 16 fios de execução houvesse um ganho decente, no entanto há uma queda grande de 12 para 16. O programa escolhe como repartir a estrutura baseando-se no número de núcleos disponíveis mas talvez haja uma necessidade de algum núcleo livre e assim ter escolhido uma repartição em 9 em vez de 16, o que explicaria os seguintes resultados.



Média: 36.5703%



Média: 73.0824%

No gráfico da esquerda podemos então confirmar que está longe de usar os 16 núcleos que lhe foram

atribuídos, pois, feito uma contagem à mão dos resultados, 16 dos 32 núcleos lógicos estiveram basicamente em espera durante a execução do programa.

No gráfico da direita já podemos ver uma melhor utilização dos núcleos totais pois a partição do problema deverá ter sido em 36 blocos quadrados, dando trabalho a todos os núcleos. Apenas podemos estimar pois o relatório da execução não nos diz esta informação.

5. Multiprocessos

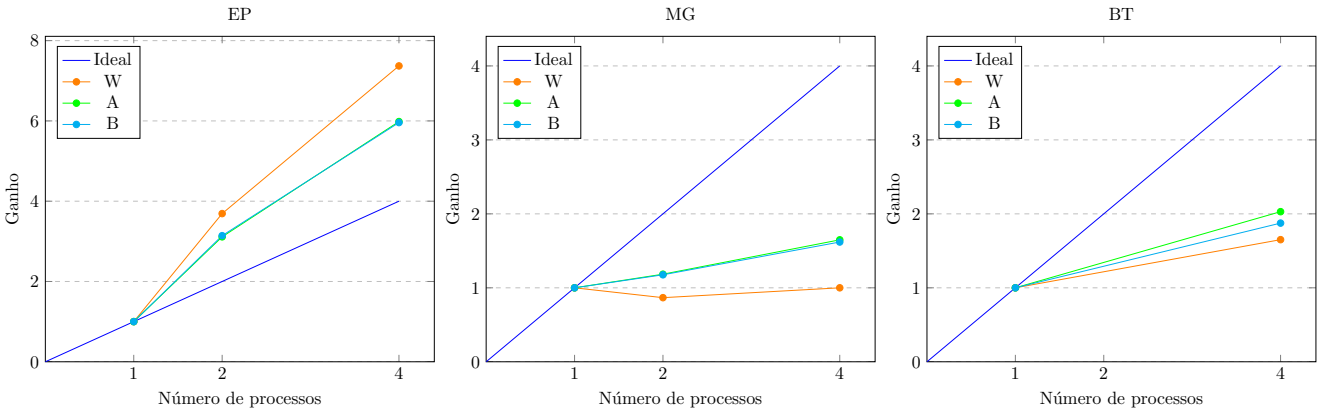
Multiprocessing, ou multiprocessos, é o uso de um ou mais cores dentro de um único computador. Um sistema multiprocessado é capaz de distribuir os processos pelos vários CPUs. Para beneficiar-se desta capacidade, o pacote NPB usa a biblioteca OpenMPI.

O OpenMPI é uma biblioteca de passagem de mensagens muito utilizada para sistemas de memória distribuída, o que nos permite deixar de estar limitados ao número de núcleos dos processadores numa máquina, podendo passar a utilizar várias máquinas em conjunto como um cluster.

O OpenMPI permite ao utilizador escolher a forma como os processos são geridos pelas máquinas. Pode ser por nó, por socket ou por núcleo. Neste relatório iremos avaliar a performance das aplicações quando distribuímos por núcleo e quando distribuímos por nó.

5.1. Mapeamento de um processo por nó

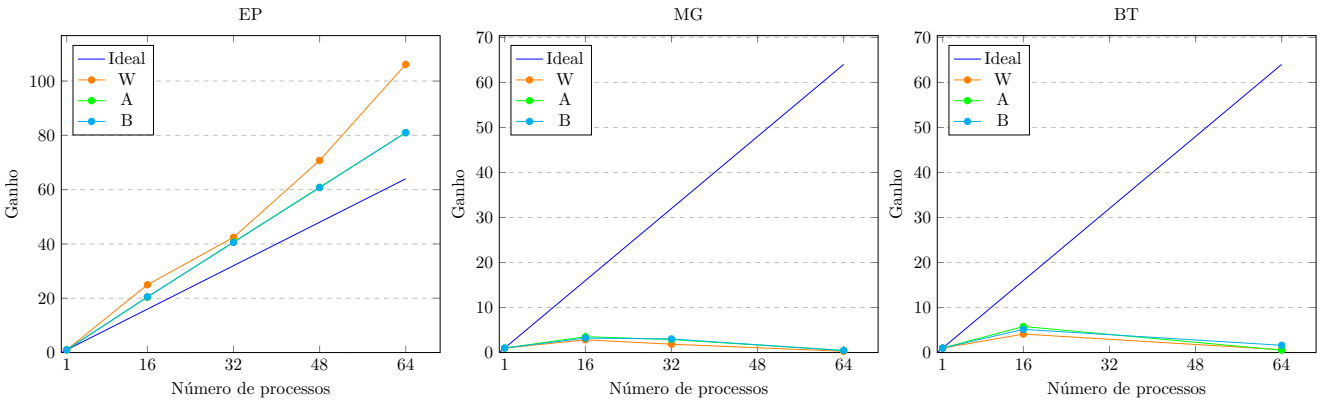
Os testes que primeiro decidimos correr foi mapear apenas um processo por máquina. Isto dá uma melhor ideia do impacto da comunicação por rede entre as várias máquinas antes de passarmos para os seguintes testes.



Como se pode observar, temos um comportamento parecido com os testes do OpenMP em que o EP escala bem, mas no entanto, os outros dois não. O BT apenas tem pontos em 1 e 4 processos porque não é impossível dividir um quadrado em dois quadrados, como já foi referido as limitações desta aplicação anteriormente.

5.2. Mapeamento por núcleo

De seguida, corremos o mesmo tipo de teste mas agora existem processos iguais ao número total de cores físicos das máquinas. O comportamento é semelhante ao OpenMP, mas estamos a lidar com processos que têm um peso maior e não com fios de execução que são mais leves e simples. No entanto, uma vantagem aqui é que não estamos limitados ao número de núcleos de uma máquina.



Novamente podemos ver o mesmo comportamento que anteriormente, portanto não há muito mais a adicionar.

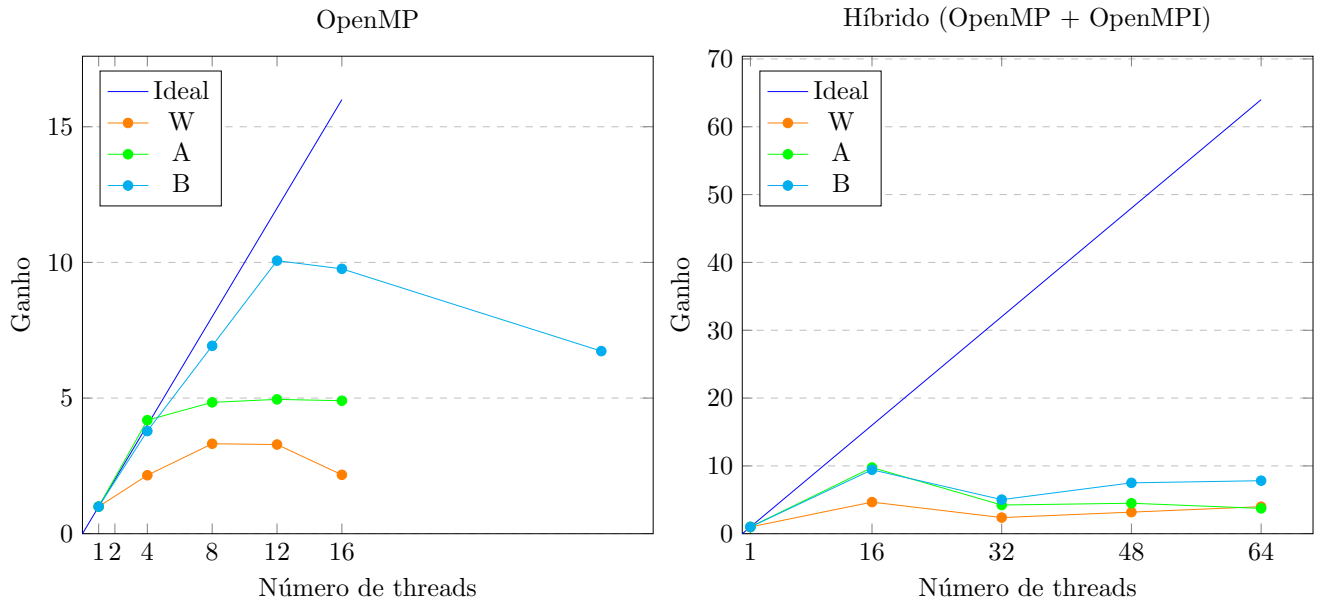
6. Híbrido

A ultima implementação que vamos analisar é a implementação híbrida. A implementação híbrida tira partido das vantagens de multithreading e de multiprocessing, para assim tentar obter uma performance ainda maior ao buscar o multithreading mais leve e simples e a capacidade de escalar com várias máquinas. Para isto, é usado um implementação híbrida entre OpenMP e OpenMPI.

No pacote NPB, para a implementação híbrida, não estão disponíveis as mesmas aplicações que para as outras implementações. Nos nossos testes, decidimos usar a aplicação BT-MZ, que é uma versão do BT mas que usa várias meshes, ou zonas (MZ significa Multi-Zone) de tamanhos irregulares.

6.0.1. BT-MZ

Para conseguir efetuar os testes necessários à versão híbrida e descobrir qual é o ganho, testamos a versão sequencial e multithreading da aplicação.



No gráfico da esquerda temos então os resultados do OpenMP. Tal como anteriormente, testamos com

2, 4, 8, 12 , 16 e 32 fios de execução (sendo 32 hyperthreading). Infelizmente, apenas a classe B conseguiu executar hyperthreading tendo as restantes dado um erro de número máximo de fios de execução excedido, o que não conseguimos explicar.

Podemos ver que todas as classes apresentam bons ganhos até às 12 fios de execução. Na classe W, já notamos que não houve ganhos dos 8 fios de execução para as 12. Tanto a classe W como a classe B perdem ganhos quando passam dos 12 fios de execução, enquanto que a classe A, não apresenta qualquer tipo de ganhos (ou perda de desempenho) a partir dos 12 fios de execução. Das 3 classes, a classe B é a que apresenta um maior ganho, sendo que quase consegues acompanhar os ganhos ideias até os 12 fios de execução. Em geral, este problema apresenta uma escalabilidade decente, mas podemos ver que os ganhos são prejudicados com o aumento excessivo do número de fios de execução. Isto quer dizer que a partir de um certo ponto, diminuir ainda mais o grão de trabalho não compensa o custo de estar a criar mais fios de execução, ou de ter estes a processar tão pouco trabalho. Seria então expectável que com tamanhos de dados maiores a aplicação continuasse a escalar com o mesmo aspeto de curva.

No gráfico da direita, estão os resultados da versão híbrida. Vemos que as classes têm ganhos (apesar de não ideais) nos 16 fios de execução, mas quando passam para os 32, todas eles tem ganhos inferiores. Apesar disso elas conseguem ter alguns ganhos com 48 fios de execução e, no caso da classe W e B, os ganhos aumentam com 64 fios de execução. No entanto, pode-se concluir que este tipo de aplicação não compensa ir para uma versão híbrida nesta situação, visto que, com 16 fios de execução que correspondem ao limite de núcleos físicos de uma máquina, atingimos o máximo de ganho com o paralelismo.

7. Futuro trabalho

Embora estejamos satisfeitos com o nosso trabalho, há bastantes pontos em que gostaríamos de ter feito melhor, mas devido a questões de tempo e trabalho, optámos por o não fazer.

Estas resumem-se a:

- repetir alguns testes ou possivelmente todos num ambiente mais controlado para evitar situações como o sequencial base ter sido prejudicado e acabarmos com curvas de ganhos superiores à de ganho ideal;
- apesar de termos medido a utilização do disco e o espaço da memória virtual com os comandos *iostat* e *vmstat*, não conseguimos concluir nada de relevante pois todas as situações eram semelhantes com um comportamento normal. Isto fez-nos pensar que deveríamos ter corrido com tamanhos de dados bastante superiores para atingir outros limites no hardware para além da capacidade de processamento aritmético;
- não conseguimos encontrar nenhum comando já instalado no cluster que nos permitisse medir o tráfego de rede de uma forma simples, o que seria útil para as medições em OpenMPI;
- fazer uma melhor comparação com o compilador de Intel testando várias opções e também diferente hardware, possivelmente hardware que não seja fabricado pela Intel como processadores da AMD e afins.

8. Conclusão

Este relatório foi escrito com dois objetivos em mente: analisar a performance de várias aplicações do pacote NAS Parallel Benchmarks analisando os vários impactos de otimizações e compiladores diferentes, avaliar como diferentes implementações melhoram (ou pioram) os resultados obtidos e como diferentes dimensões de dados afetam estas variáveis. O segundo objetivo era avaliar o impacto destas aplicações na performance do sistema e como estas utilizavam os vários recursos disponibilizados pelo sistema.

Através da criação de scripts criados para avaliar estes dados e com o uso do cluster disponibilizado

pela universidade, obtivemos um conjunto de resultados, dos quais conseguimos obter várias conclusões. Demonstramos que as otimizações efetuadas pelo compilador são bastante boas a diminuir o tempo de execução dos programas mas dependem bastante do tipo de código com que estamos a trabalhar pois não há um vencedor claro.

Analisamos o efeito de implementações diferentes de paralelização (multithreading e multiprocessing) nas diferentes aplicações e como as diferentes "arquiteturas" das aplicações influenciam os ganhos obtidos com uso de diferentes implementações e, consequentemente, como é estas vão utilizar os recursos do computador. No final analisamos a versão híbrida de multithreading e multiprocessing e comparamos com a implementação multithreading, concluindo que a versão híbrida não apresenta resultados satisfatórios para apoiar o uso desta versão.

Como indicamos na secção anterior, não analisamos todas as componentes que queríamos, nem as analisamos com o nível de detalhe que desejávamos, o que impediu-nos de aprofundar mais em certos tópicos. Face a este problema, estamos motivados para em próximos trabalhos analisar mais "in-depth" os tópicos em questão, para obtermos uma análise mais detalhada e profunda, da qual fiquemos ainda mais orgulhosos.