

# Performance em NAS Parallel Benchmarks (NPB)

## - TPC1 -

CARLOS SÁ - A59905  
carlos.sa01@gmail.com

March 7, 2016

### Abstract

Este relatório é o resultado de um estudo sobre análise de desempenho utilizando aplicações do pacote NAS Parallel Benchmarks (NPB) no contexto da unidade curricular de Engenharia de Sistemas de Computação. Este estudo conta com a análise de três kernels diferentes cada um em três paradigmas de computação diferentes: Paradigma de memória partilhada (OpenMP), paradigma de memória distribuída (MPI) e a sua correspondente versão sequencial. Por forma a ser possível fazer uma análise comparativa dos resultados, os testes serão feitos no cluster SeARCH, em vários nodos, com diferentes compiladores flags, diferentes arquitecturas, e com dados de diferentes dimensões.

Ao longo de todo o trabalho, será explorado o comportamento dos programas e retirar algumas conclusões sobre o seu desempenho com recurso a várias medições de tempos de execução, tempos de ócio e bloqueio. Serão utilizados alguns comandos de UNIX por forma a compreender como é que alguns testes afetam o estado do sistema. Quer ao nível da percentagem de utilização do CPU, utilização de memória e de disco. A parte correspondente a essa análise estatística dos resultados será feita com recurso a gráficos criados apartir dos ficheiros com os resultados obtidos.

### 1. INTRODUÇÃO

O NAS Parallel Benchmarks corresponde a um conjunto de programas que têm como objectivo avaliar a performance computacional de supercomputadores. Neste estudo, o supercomputador utilizado como objecto de estudo será o cluster SeARCH do Departamento de Informática da Universidade do Minho. Este pequeno conjunto de programas possui um total de 5 kernels e 3 pseudo aplicações. Estes correspondem a benchmarks de avaliação de performance, sendo todos eles baseados em CFD (*Computational Fluid Dynamics*).

Os **kernels** [6] disponíveis neste conjunto são:

- **IS** - Código baseado na ordenação (sorting) de inteiros;
- **EP** - Algoritmo embaralhosamente paralelo, e baseado em números pseudo aleatórios Gaussianos, tipicamente utilizado em aplicações Monte Carlo. Praticamente não possui comunicação;
- **CG** - Método do gradiente conjugado, comumente utilizado em problemas que envolvam multiplicação matrix esparsa por vetor.
- **MG** - Algoritmo sobre malhas tridimensionais baseado na equação de Poisson. Uso intensivo de memória;
- **FT** - Algoritmo que realiza a multiplicação de matriz e transformada de Fourier 1D, 2D e 3D. Comunicação "todos-para-todos".

As **pseudo aplicações (ou simulações)** disponíveis neste [6] conjunto são:

- **BT** - Solver de equações de blocos tri-diagonais de tamanho 5x5;

- **SP** - Solver de sistemas de equações penta-diagonais. Uma solução SP requer cerca de 400 iterações.
- **LU** - Algoritmo de resolução de um sistema triangular inferior e superior de blocos  $5 \times 5$ . Requer cerca de 250 iterações;

Este conjunto de programas estão escritos na linguagem C e Fortran existindo as versões sequenciais, OpenMP® e MPI de cada um deles.

Para os diferentes benchmarks existem datasets de diferentes tamanhos identificados de acordo com as letras **S**, **W**, **A**, **B**, **C**, **D**, e **E** [2] sendo que:

- **S** - Tamanho mais pequeno. Apenas para pequenos testes;
- **W** - Tamanho workstation, em relação a **S** igualmente pequeno;
- **A, B e C** - Tamanho *standard*. Cada dataset é 4x maior que o anterior;
- **D, E, e F** - Dataset de maior tamanho, tipicamente utilizados para testes maiores. Cada dataset é cerca de 16x maior que o dataset anterior.

A ideia principal deste trabalho foi propôr aos alunos que escolhessem alguns dos testes e/ou simulações e avaliar a sua performance em nós de diferentes características de hardware do SeARCH Cluster. Assim, a ideia deste trabalho possui uma forte componente experimental, e avaliação crítica de resultados. Pretende-se que os alunos explorem os diferentes resultados obtidos com:

- Diferentes versões dos códigos (versão sequencial, paralela com OpenMP e paralela com MPI);
- Vários compiladores (Intel e GNU);

- Compiladores de diferentes versões;
- Diferentes opções de compilação;
- Microarquitecturas distintas;
- Classes (datasets) de vários tamanhos de dados;
- Diferentes tecnologias de comunicação (*Gigabit Ethernet* e *Myrinet* - especialmente importante para estudo das versões MPI dos testes);

Dado o mix de combinações possíveis, é necessário optimizar o processo de execução e tratamento estatístico dos resultados. Para tal, foi recomendado que os alunos fizessem processamento em lotes obrigando a desenvolver e aprofundar conhecimentos em scripting do sistema PBS. Numa fase posterior do trabalho pretende-se também que os alunos estudem o impacto resultante da execução dos kernels em termos de CPU, memória, disco entre outros. Para tal os alunos devem recorrer a comandos de UNIX tais como iostat, mpstat, top, e dstat.

## 2. KERNELS E DATASETS ESCOLHIDOS

Do conjunto dos 5 kernels disponíveis irei utilizar três para desenvolver este estudo: (IS, EP, CG). Cada um dos kernels foi explorado a nível de desempenho para 3 datasets de tamanhos distintos: A, B e C. Não escolhi S e W porque após a realização de alguns testes, concluí que estes tamanhos são de facto demasiado pequenos para o âmbito de análise de desempenho em cluster, e os tempos de execução obtidos eram demasiado baixos. Como pretendia explorar o desempenho dos kernels para as diferentes classes, diferentes compiladores, diferentes níveis de optimização, etc, optei por não escolher classes demasiado grandes. A versão do package do NPB utilizada neste estudo é a versão: **NPB 3.3.1**.

## 3. CARACTERIZAÇÃO DO SEARCH CLUSTER

Como iremos ver, uma boa análise de desempenho deve ser feita em diferentes máquinas por forma a obter resultados comparativos que possam ser discutidos e de onde se possam tirar conclusões sobre o comportamento dos programas. Com este objectivo em mente, irei utilizar alguns nós computacionais do cluster SeARCH de diferentes microarquitecturas. Este *cluster* é um recurso computacional localizado no Departamento de Informática da Universidade do Minho. Possui uma boa riqueza de nós computacionais heterogéneos em relação ao número e variedade de microarquitecturas. Os nós computacionais (compute nodes) são identificados através de um *rack* de 3 algarismos[3] que permite saber :

- A microarquitectura do nó (algarismo das centenas).
- A metade do número total de cores físicos do processador do nó (algarismo das dezenas);
- Suporte de acelerador: 1 - *Não* ou 2 - *Sim* (algarismo das unidades);

Para identificar a microarquitectura do nó é utilizada a seguinte numeração:

- 6 - Ivy Bridge;
- 5 - Sandy Bridge;
- 4 - Nehalem;
- 3 - Perny;
- 2 - AMD Magny-Cours;

Para identificar o #cores fisicos do nó, temos que multiplicar o valor das dezenas por 2. Por exemplo, um valor de 8 no algarismo das dezenas, significa que o nó possui um total de 12 cores físicos.

De modo, um nó **compute-662** representa um nó computacional com microarquitectura Ivy Bridge, com 12 cores físicos.

### 3.1. Nós computacionais utilizados

Dada a heterogeneidade de nodos existentes no *cluster SeARCH*, (como constatado na secção anterior), decidi utilizar 2 nós de microarquitecturas distintas:

- Intel® Nehalem (**compute-431**).
- Intel® Ivy Bridge (**compute-641**);

Então para o presente estudo será utilizado um nó do segmento 431 (Nehalem) e 641 (Ivy Bridge) - este último por possuir suporte de utilização da rede *Myrinet* de (10Gbps) - especialmente importante para fazer comparação de custos de comunicação em execução de programas MPI. A caracterização técnica de hardware dos nós é descrita nas seguintes tabelas:

<b>Manufacturer</b>	Intel® Corporation
<b>Processor</b>	2x Xeon X5650
<b>Microarchitecture</b>	Nehalem
<b>Processor's Frequency</b>	2.66 GHz
<b>#Cores</b>	6
<b>#Threads</b>	12
<b>Cache L1</b>	32KB
<b>Cache L2</b>	256KB
<b>Cache L3</b>	12288KB
<b>Associativity L1</b>	8-way
<b>Associativity L2</b>	8-way
<b>Associativity L3</b>	16-way
<b>Line Size</b>	64 Bytes
<b>Memory access bandwidth</b>	17,7 GB/s
<b>RAM Memory</b>	12GB
<b>Memory Channels</b>	3

**Table 1:** Hardware dos nós do segmento 431

<b>Manufacturer</b>	Intel® Corporation
<b>Processor</b>	E5-2650 v2
<b>Microarchitecture</b>	Ivy Bridge
<b>Processor's Frequency</b>	2.60 GHz
<b>#Cores</b>	8
<b>#Threads</b>	16
<b>Cache L1</b>	32 KB (I) + 32 KB (D)
<b>Cache L2</b>	256KB (I) + (D)
<b>Cache L3</b>	20MB
<b>Associativity L1</b>	8-way
<b>Associativity L2</b>	8-way
<b>Associativity L3</b>	20-way
<b>Line Size</b>	64 Bytes
<b>RAM Memory</b>	64GB

**Table 2:** Hardware dos nós do segmento 641

A escolha destes nós está relacionada fundamentalmente com as suas diferenças ao nível dos processadores, microarquitetura e nível de disponibilidade dos mesmos.

Por serem nós com microarquiteturas de gerações diferentes, achei que isso poderá ajudar a obter resultados bem distintos. Além disso, por questões de disponibilidade e de acordo com [3], podemos verificar que os nós com arquitetura *Ivy Bridge* e *Nehalem* são os nós de microarquiteturas diferentes que existem em maior número no cluster. A caracterização de hardware, para o processador e para a memória foram obtidos com recurso aos comandos da bash:

```

1 $ more /proc/cpuinfo > cpuinfo431.txt
2 $ more /proc/meminfo > meminfo431.txt

```

Quando ligado a um nó do segmento 431 e:

```

1 $ more /proc/cpuinfo > cpuinfo641.txt
2 $ more /proc/meminfo > meminfo641.txt

```

Quando ligado a um nó do segmento 641.

Contudo, existem certas informações como níveis de associatividade das caches e os tamanhos das caches de nível 1 e 2 que não estão disponíveis através dos comandos listados. Para essa informação complementar, recorri ao *spec.org*[5] e *CPU-Benchmark*[1].

#### 4. METODOLOGIA DE MEDIÇÃO

Todas as medições feitas neste estudo foram feitas atendendo a um conjunto de precauções, algumas nem sempre possíveis de contornar. Atendendo às tabelas 1 e 2 podemos saber o #cores de cada processador dos dois nós. Sendo que num nó 431 existem então dois sockets de 6 cores físicos e no 641 8 cores, sabemos que as duas máquinas destes segmentos possuem respectivamente 12 e 16 cores. Contabilizando o Hyper-Threading, sabemos que cada máquina suporta por hardware 24 e 32 threads.

Conhecendo estes valores, sabemos que uma boa prática de análise de performance deve ser feita executando e medindo os valores utilizando a totalidade dos cores da máquina. Neste estudo, todos os valores são obtidos nestas condições em que toda a máquina é reservada para executar os programas e obter os valores das métricas. Claro que existem outros processos que por omissão estão a correr nas máquinas e diminuir o máximo de intrusão possível nas medições, nem sempre é possível. Como o cluster SeARCH utiliza o sistema **PBS** a script utilizada para execução dos jobs contém estes pedidos para as execuções nas duas máquinas:

```

1 #PBS -l nodes=1:r431:ppn=24

```

e:

```

1 #PBS -l nodes=1:r641:ppn=32

```

Para a execução dos *kernels* em memória distribuída utilizando MPI, são utilizados mais nós. Assim tal como para as execuções sequenciais e paralelas com OpenMP, também as execuções MPI reservam a totalidade das máquinas que executam os kernels por forma a obter execução exclusiva dos programas e medição dos tempos. As medições de todos os gráficos utilizam as mesmas unidades (por exemplo, todos os tempos, estão em segundos). Como iremos ver mais à frente, o comando **dstat** foi utilizado para obter estatísticas de utilização do cpu, memória e disco. Um aspecto que foi tido em conta quando se realizou a execução deste comando para receber os resultados foi fazer *kill* para que o comando não ficasse em execução no nó de computação onde o benchmark é feito após os resultados serem guardados em CSV:

```

1 ( .... )
2 max_thread=32
3 sample_size=5
4 for file in *.x
5 do
6   for (( num_threads=1, num_threads <= $max_thread; num_threads+=2 ))
7   do
8     echo "Running ( Using $num_threads threads )"
9     export OMP_NUM_THREADS=$num_threads
10    for (( sample=1; sample <= $sample_size; ++sample ))
11    do
12      cd ..../Results/intel_2013_1_117/
13      $node_info
14      /home/a59905/dstat --cdm --output $file.csv >> /dev/null &
15      cd ../../bin
16      ./$file >> ..../Results/intel_2013_1_117/$node_info/"$file.txt"
17      kill $!
18      sleep 2
done

```

```

19      done
20 done
21 ( .... )

```

Também foi utilizado o comando *sleep* de 2 segundos entre execuções por forma a garantir tempos com alguma estabilidade.

## 5. COMPILADORES E FLAGS UTILIZADAS

Como iremos ver mais à frente, todo o estudo realizado conta com a exploração de resultados em compiladores distintos: **icc** da Intel (**intel 2013.1.117**) e **gcc** da GNU sendo que para este último foi avaliada a performance dos benchmarks em duas versões distintas: 4.9.0 e 4.9.3. Foram utilizados um total de **3 compiladores**. Em relação a **flags**, dado que este estudo pretende ser um estudo sobre avaliação de desempenho utilizei três optimizações automáticas de código ordenadas por nível de optimização: **-O0**, **-O2** e **-O3**. Para correr os kernels MPI, foi também utilizada a biblioteca **OpenMPI 1.8.4**.

## 6. MÉTRICAS ANALISADAS

Como o presente pacote do NAS já tem como objectivo ser utilizado para avaliação de performance, basta analisar os outputs gerados pela execução de cada um dos benchmarks para escolher métricas de análise possíveis. Vejamos o seguinte excerto do output gerado pela execução de um benchmark:

```

(...) 
EP Benchmark Results:
(...)

EP Benchmark Completed.
Class          =           A
Size           = 536870912
Iterations     =           0
Time in seconds =       2.16
Total threads  =           4
Avail threads  =           4
Mop/s total   =      248.43
Mop/s/thread   =      62.11
Operation type = Random numbers generated
Verification    =      SUCCESSFUL
Version        =      3.3.1
Compile date   = 26 Feb 2016
(...)

```

Podemos verificar que existe uma certa riqueza de métricas que podem ser utilizadas: Tempos de execução, número de threads, Mop/s, Mop/s por thread etc. Assim, escolhi como métricas de análise, o tempo de execução ( $T_{exec}$ ), o total de **MOP/s - Milhões de operações por segundo** para os diferentes kernels e para as diferentes classes. Para o caso concreto da experimentação realizada para os kernels MPI foi também analisado o número de MOPs por processo - *Milhões de operações por segundo*.

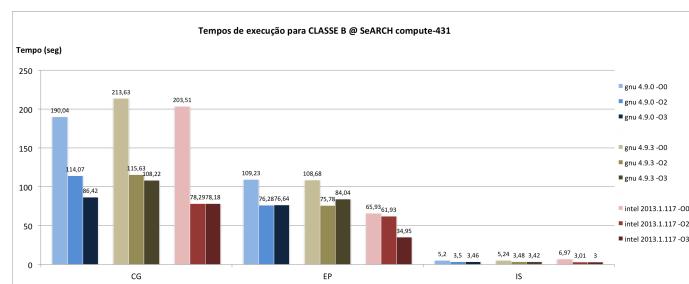
## 7. ANÁLISE SEQUENCIAL (NPB3.3-SER)

A primeira análise realizada foi aos kernels sequenciais. Nesta foram analisados os tempos de execução e Mop/s, para os diferentes kernels, diferentes classes, em nós de microarquiteturas distintas para os 3 compiladores e ainda três níveis de optimização: **-O0**, **-O2** e **-O3**.

### 7.1. Tempos de execução ( $T_{exec}$ ) - Sequencial

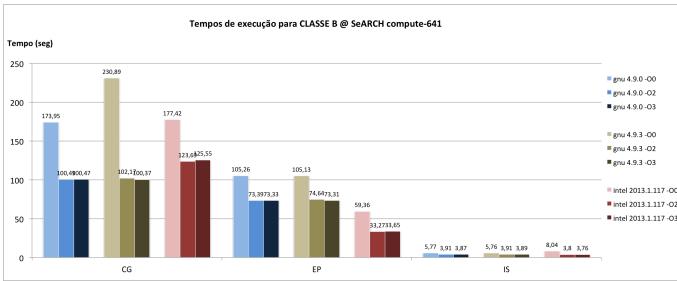
Sendo A a classe de menor tamanho, percebemos que o seu tempo de execução é o menor de todas as classes a rondar os 27 segundos para o kernel EP pelo apenas analisaremos as classes B e C. O gráfico da classe A, pode contudo, ser consultada em anexo na tabela: 35.

Em relação à classe B, podemos verificar uma maior performance geral no nó 641, para todos os kernels, em todas as flags em cada um dos compiladores. Globalmente para cada uma das classes, podemos verificar um comportamento semelhante nos diferentes compiladores: á medida que aumentamos o nível de optimização, (de O0 para O3 - da tonalidade mais clara, para a mais escura da cõr de cada um dos compiladores) os tempos de execução diminuem. A diferença é maior quando optimizamos de um nível O0 para O2 do que quando optimizamos de O2 para O3. Os compiladores Intel revelam-se mais eficientes para níveis de optimização superiores.



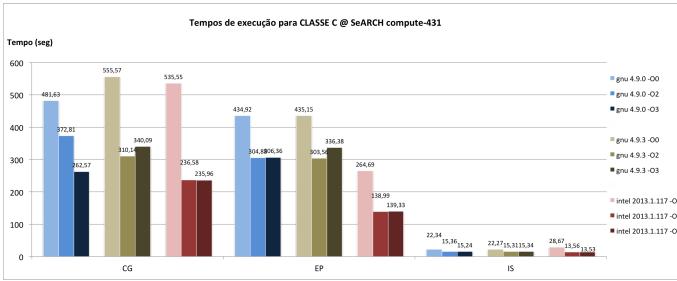
**Figure 1:** Tempos sequenciais para diferentes compiladores com optimizações **-O0**, **-O2**, **-O3** - nó 431

Note-se que a generalidade dos tempos para o kernel IS foram mais pequenos no nó 431. Uma microarquitetura de uma geração anterior, revela ser mais eficiente em algoritmos sequenciais de sorting de inteiros. Tal pode acontecer uma vez que, em sequencial o nó de microarquitetura Nehalem possui uma frequência de relógio superior.



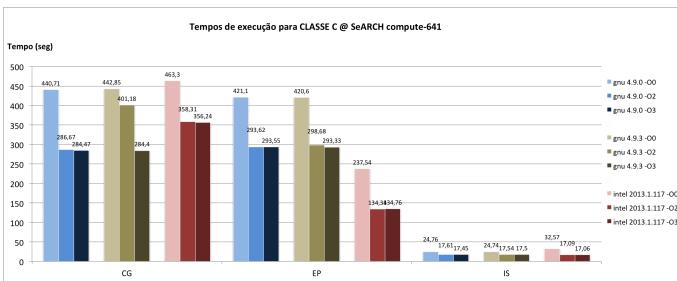
**Figure 2:** Tempos sequenciais para diferentes compiladores com optimizações -O0, -O2, -O3 - nó 641

O aumento da classe não provoca grandes alterações na eleição do melhor compilador para níveis de optimização superiores que continua a ser o icc da Intel. Contudo, no kernel CG os compiladores da GNU se revelaram mais eficientes no nível -O3 para classes menores neste tipo de algoritmos. O aumento da classe favorece o compilador da intel ao nível dos tempos de execução.



**Figure 3:** Tempos sequenciais para diferentes compiladores com optimizações -O0, -O2, -O3 - nó 431

Mesmo com datasets de maior tamanho, o compilador da intel aparenta ser mais favorável na microarquitectura Nehalem (segmento 431) do que na micro-arquitectura Ivy-Bridge (segmento 641) para o kernel CG. Para os restantes kernels, o compilador da intel revela conseguir gerar código mais eficiente para os diferentes níveis de optimização.



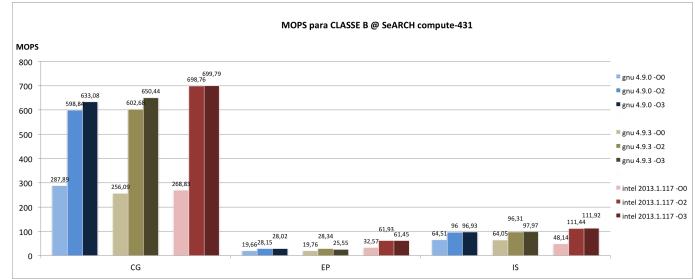
**Figure 4:** Tempos sequenciais para diferentes compiladores com optimizações -O0, -O2, -O3 - nó 641

## 7.2. Análise Milhões operações por segundo (Mop/s) para várias classes - 431 VS 641

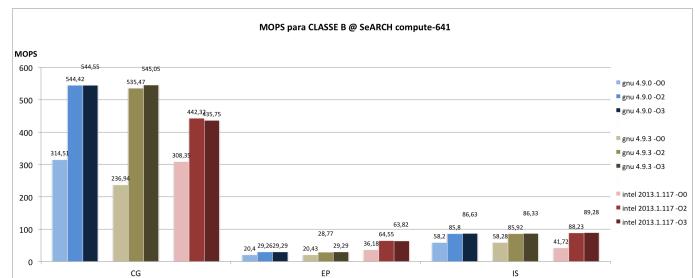
Depois de realizar a análise para o tempo de execução onde

o icc da intel se revelou mais vantajoso estudei também uma segunda métrica: Mop/s. Podemos perceber pelos gráficos seguintes que a eficiência dos compiladores não se reflete só ao nível do  $T_{exec}$  mas também ao nível do número de operações realizadas por unidade de tempo. Pela tabela da figura 5 (à semelhança do que acontece também no gráficos 6 numa microarquitetura mais recente, e ainda em classes de maior volume de dados - gráficos 7 e 8) quanto maior o nível de optimização, maior é o número de operações realizadas por unidade de tempo.

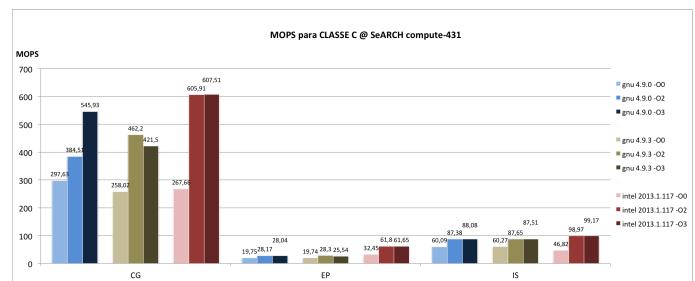
Uma conclusão óbvia a retirar, é que níveis de optimização maiores fazem com que o aproveitamento da taxa de operações realizadas por unidade de tempo sejam igualmente maiores, o que comparativamente aos resultados obtidos para  $T_{exec}$  faz com que estes sejam menores. Nos gráficos seguintes podem-se consultar os resultados obtidos para as classes B e C nas microarquiteturas Nehalem e Ivy-Bridge. Este estudo também foi realizado para a Classe A. O seu gráfico poderá ser encontrado em anexo na figura: 37.



**Figure 5:** Relação de Mop/s na Classe B no nó 431



**Figure 6:** Relação de Mop/s na Classe B no nó 641



**Figure 7:** Relação de Mop/s na Classe C no nó 431

Na microarquitectura mais antiga (Nehalem) comparativamente com a mais recente (Ivy-Bridge) os compiladores conseguem gerar código que realiza mais operações por segundo no kernel CG e IS, no entanto esta tendência já não se verifica no kernel EP em que os nós da micro-arquitectura Ivy-Bridge se revelam mais eficientes. O kernel EP executa um algoritmo embarcado paralelo. Por esta análise consegui concluir que algoritmos em que se consiga extraír um maior nível de paralelismo, tem um maior potencial para serem beneficiados a nível de desempenho em micro-arquitecturas mais recentes.

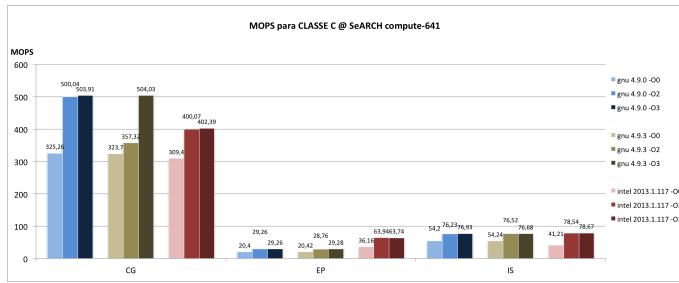


Figure 8: Relação de Mop/s na Classe C no nó 641

### 7.3. Performance compiladores: Classes A, B e C (431 VS 641)

Como foi possível perceber pela análise feita anteriormente, os tempos de execução são mais favoráveis quando utilizadas as optimizações -O3 para os três compiladores. Neste subcapítulo iremos fazer um confronto dos 3 compiladores nas duas micro-arquitecturas para as diferentes classes. Pela análise dos gráficos 9, 10 e 11 percebemos que os kernels têm um maior desempenho nos 3 compiladores na micro-arquitectura Ivy-Bridge para execuções dentro da mesma classe à excepção do kernel CG que apresenta tempos de execução mais pequenos na microarquitectura Nehalem para o GNU 4.9.0.

Como seria de esperar, em todos os compiladores, os tempos de execução são maiores para classes maiores uma vez que o volume de dados a processar são 4x superior das classes A para C.

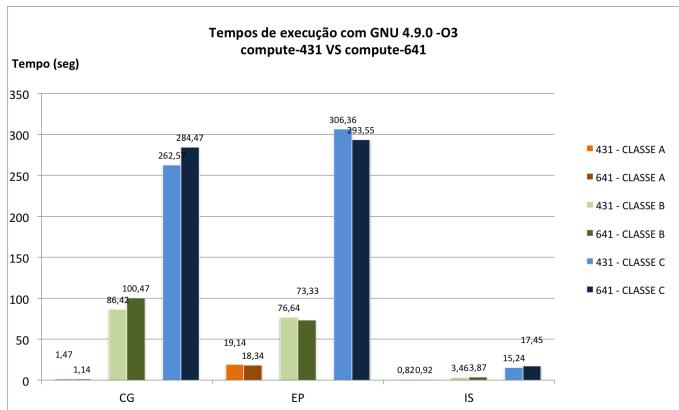


Figure 9: Análise performance GNU 4.9.0 nas microarquitecturas Nehalem e Ivy-Bridge

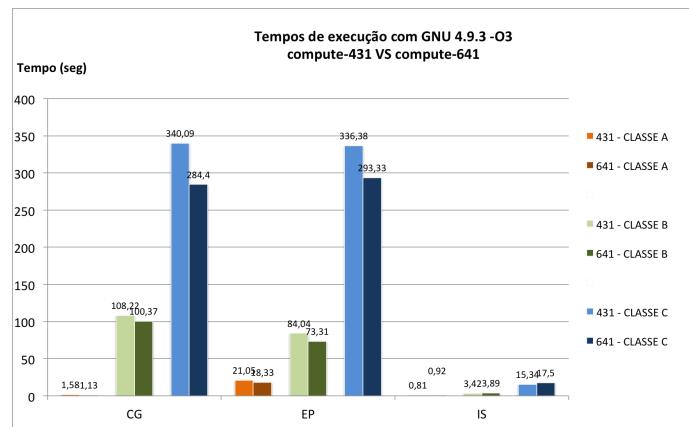


Figure 10: Análise performance GNU 4.9.3 nas microarquitecturas Nehalem e Ivy-Bridge

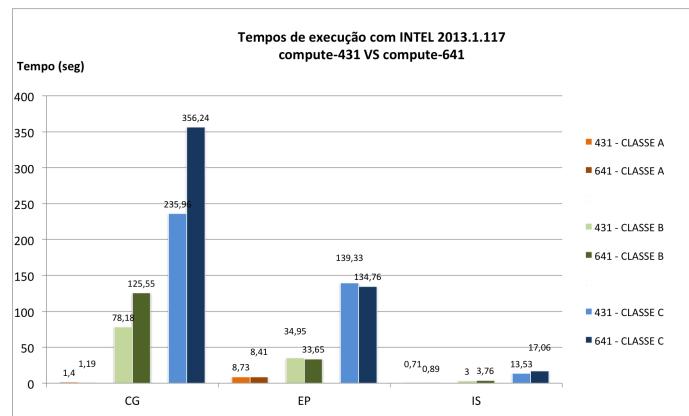


Figure 11: Análise performance Intel 2013.1.117 nas microarquitecturas Nehalem e Ivy-Bridge

## 8. ANÁLISE PARALELA - OPENMP (NPB3.3-OMP)

Concluída a análise dos kernels sequenciais, realizei igualmente o estudo para as versões paralelas com OpenMP. A análise dos kernels também foi feita com recurso a duas métricas diferentes: Tempo de execução e Mop/s. Todos os testes foram realizados utilizando a totalidade dos cores das máquinas em execução exclusiva dos kernels sem interferência de outros trabalhos em execução.

### 8.1. Tempos de execução ( $T_{exec}$ ) - 431 VS 641

Uma vez que necessitei de realizar múltiplas recolhas de tempos para execuções de cada kernel para um número variável de threads em paralelo, tive necessidade de incluir no meu script uma forma automática de fazer as diferentes execuções. Um excerto da forma como estes resultados foram retirados no script pode ser encontrado em 4.

Um dos objectivos de paralelizar os algoritmos é sem dúvida a obtenção de melhores tempos de execução. Como verifiquei uma tendência de melhores tempos de execução na versão sequencial utilizando a flag -O3, na análise da versão paralela

fixei esta flag e estudei os possíveis ganhos para os diferentes compiladores ao utilizar várias threads. Esta análise conta com um estudo feito também nas duas arquitecturas: Nehalem e Ivy-Bridge.

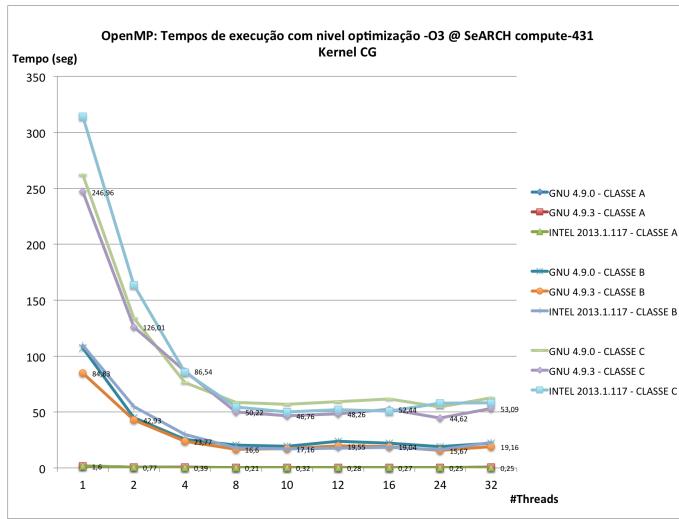


Figure 12: Tempos execução paralela com OpenMP - kernel CG para diferentes compiladores, e classes A, B e C em 431

Pela comparação dos resultados obtidos entre Nehalem e Ivy-Bridge pude verificar que a tendência ao aumentar o #threads é os tempos de execução serem menores na microarquitectura Ivy-Bridge.

Então nesta abordagem irei apenas averiguar o comportamento dos compiladores em função do número de threads para as diferentes classes apenas para a micro-arquitectura Ivy-Bridge.

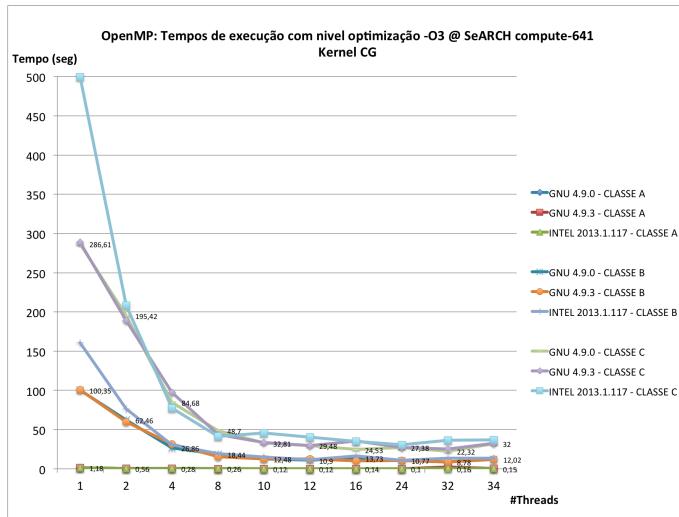


Figure 13: Tempos execução paralela com OpenMP - kernel CG para diferentes compiladores, e classes A, B e C em 641

Para o kernel CG os compiladores da GNU à semelhança do que aconteceu nos algoritmos sequenciais revelaram-se mais eficientes em termos de tempo de execução que o icc da intel,

para as diferentes classes de dados. Note-se que o estudo dos tempos foi feito até às 34 threads para o nó de arquitectura Ivy-Bridge. No entanto pela tabela 2 que possui a caracterização técnica do nó 641, percebemos que quanto ao número de cores físicos, o nó possui apenas 16 cores físicos (32 threads no total com *Hyper-Threading*).

A razão pela qual estou a fazer o estudo para as 34 threads, prende-se com o facto de apenas querer analisar e comprovar que acontece um facto esperado: o tempo de execução aumenta dado que se criam mais threads na execução do que o limite máximo de threads do processador. Percebemos então que o aumento do número de threads na execução de um programa paralelo deve ser feito de forma cuidada e atendendo à própria arquitectura e processador da máquina onde o programa corre.

Esta perda de performance é verificada embora não muito evidente neste estudo e curiosamente no kernel EP isto não aconteceu sendo que os melhores tempos de execução se obtém para 34 threads.

Ao contrário do que acontecia para o kernel CG onde os compiladores GNU se revelaram mais eficientes, para os restantes kernels (EP e IS) o compilador da intel demonstra tempos de execução mais curtos para todas as classes de dados.

Os gráficos dos tempos obtidos no nó 431 poderão ser consultados em anexo na figura 39.

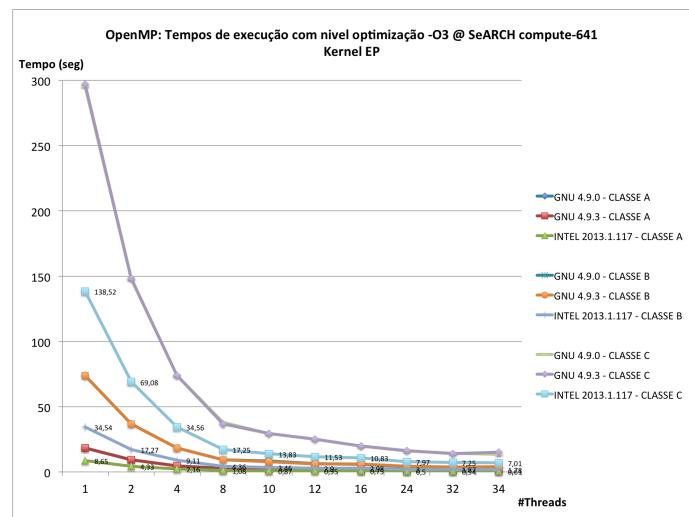


Figure 14: Tempos execução paralela com OpenMP - kernel EP para diferentes compiladores, e classes A, B e C em 641

Os gráficos dos tempos num nó 431 poderão ser igualmente consultados em anexo na figura 40

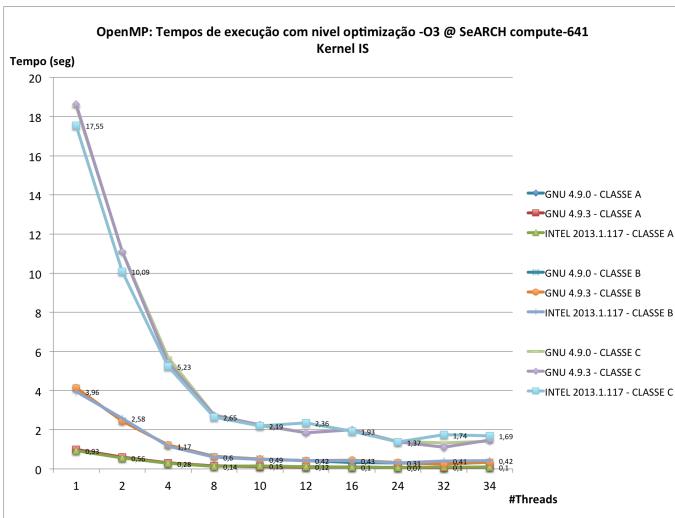


Figure 15: Tempos execução paralela com OpenMP - kernel CG para diferentes compiladores, e classes A, B e C em 641

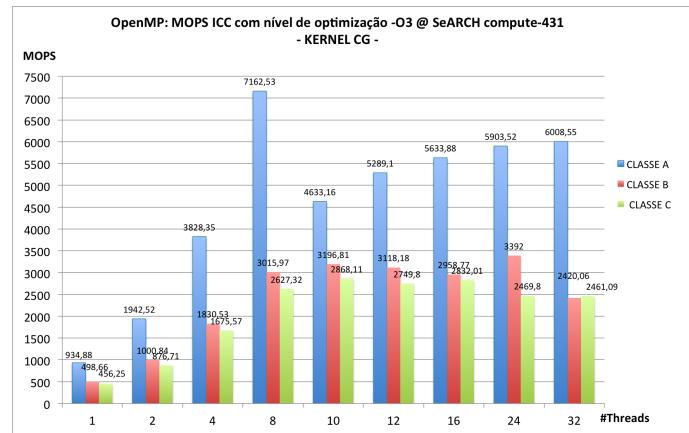


Figure 16: Mop/s para diferentes classes de dados - kernel CG em nó 431

## 8.2. Milhões operações por segundo (Mop/s) - 431 VS 641

Para avaliação desta métrica, utilizei o compilador da Intel visto ter revelado melhores resultados na execução paralela para a generalidade dos resultados obtidos. Fiz então um estudo para as diferentes classes para os diferentes kernels. Como podemos observar pelas figuras 16 e 17 os valores de milhões de operações por segundo são superiores na arquitectura *Ivy-Bridge*. Isto acontece para os vários kernels pelo que apenas os resultados para esta arquitectura serão explorados.

Os gráficos dos resultados para a microarquitectura 431 podem contudo, ser consultados em anexo na figura 41 para EP e na figura 42 para o kernel IS. O kernel CG regista o maior número de operações por segundo que os restantes kernels. É possível verificar que existe uma maior quantidade de operações por segundo a serem realizadas à medida que se aumenta o número de threads e à medida que se aumenta o volume de dados como seria de esperar. Este comportamento não se verifica para o kernel CG (figura 17) mas está bem evidente no kernel EP (figura 41) que possui um algoritmo capaz de tirar um maior partido do paralelismo e mais facilmente escalável. Um comportamento de que não estava à espera, foi verificado no kernel EP e CG.

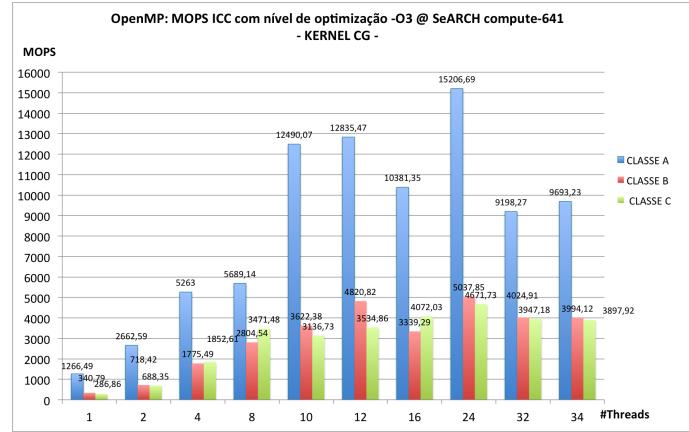


Figure 17: Mop/s para diferentes classes de dados - kernel CG em nó 641

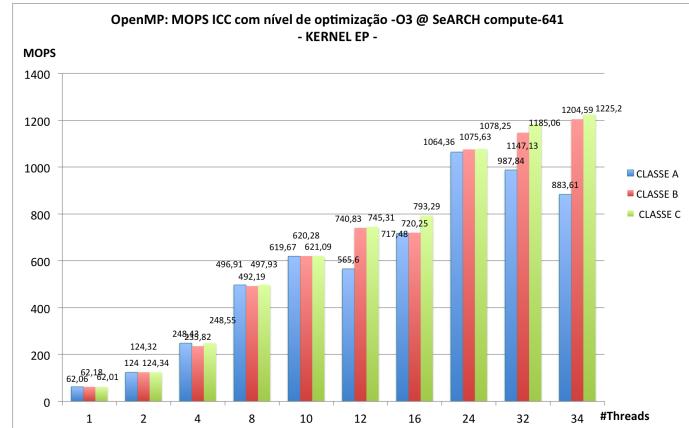


Figure 18: Mop/s para diferentes classes de dados - kernel EP em nó 641

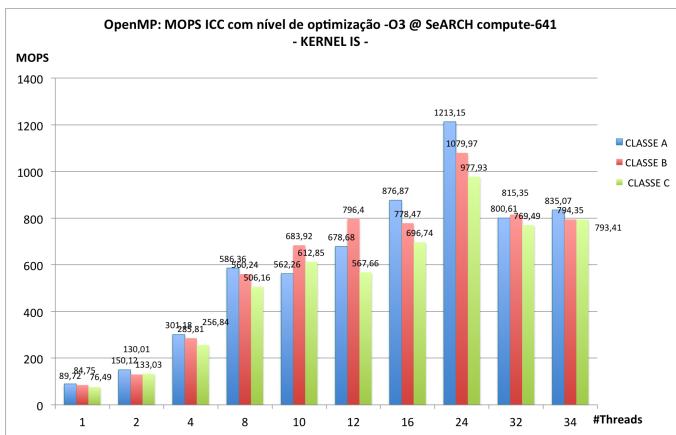


Figure 19: Mops/s para diferentes classes de dados - kernel IS em nó 641

### 8.3. Comparação Tempos de Execução OpenMP: 431 VS 641

Em jeito de avaliação final do comportamento do dos 3 kernels nas duas micro-arquitecturas na versão paralela com OpenMP construí um gráfico que me ajuda-se a perceber melhor o comportamento dos 3 kernels à medida que estes eram executados para um número crescente de threads. Assim o gráfico da figura 20 mostra esses tempos de execução. Todos os kernels foram compilados com **icc da versão 2013.1.117** e para a classe de dados **C**. Destes podemos perceber que todos os kernels são parallelizáveis. A microarquitectura **Nehalem** consegue tempos de execução mais baixo para os kernels CG e IS em execução sequencial (1 thread). Em execução sequencial é natural que o nó de microarquitectura Nehalem consiga melhores tempos de execução dado que o processador trabalha a uma frequência de clock superior. Contudo, esse efeito é diluído quando aumentamos progressivamente o número de threads, revelando mais uma vez que a microarquitectura Ivy-Bridge é mais eficiente a explorar paralelismo.

Cada vez mais os fabricantes como a Intel perceberam que não faz mais sentido persistir em arquitecturas sequenciais e frequências de relógio elevadas mas que o futuro passa por investir e explorar microarquitecturas que tirem maior partido do paralelismo pois é onde se consegue aplicações com tempos de execução mais curtos.

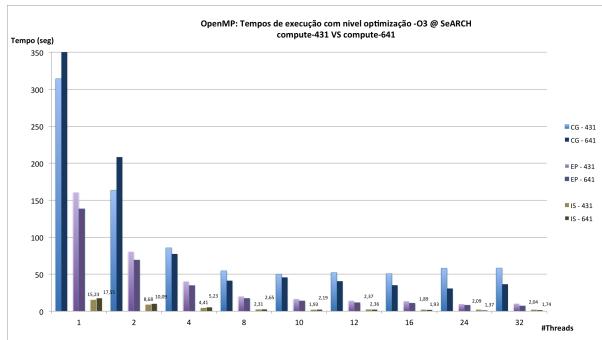


Figure 20: Tempos de execução para nível optimização -O3 compilado em intel.2013.1.117 e Classe C

### 8.4. Comparação Mop/s OpenMP: 431 VS 641

Uma segunda evidência de que os algoritmos dos kernels utilizados são escaláveis está ao nível da sua taxa de operações realizadas por unidade de tempo. Também pelo gráfico 21 podemos verificar que à medida que aumentamos o número de threads OpenMP, o total de operações realizadas por segundo tendencialmente aumenta. Mais operações realizadas por unidade de tempo representam, neste estudo, verificam-se para tempos de execução mais curtos, de grosso modo.

Também aqui verificamos que embora o Nehalem consiga bater o Ivy-Bridge em execução sequencial (1x thread), a arquitetura Ivy-Bridge mostra ter um maior potencial para explorar paralelismo à medida que aumentamos o número de threads. Isto é especialmente verificado para o kernel CG que como vimos anteriormente, é o kernel mais demorado.

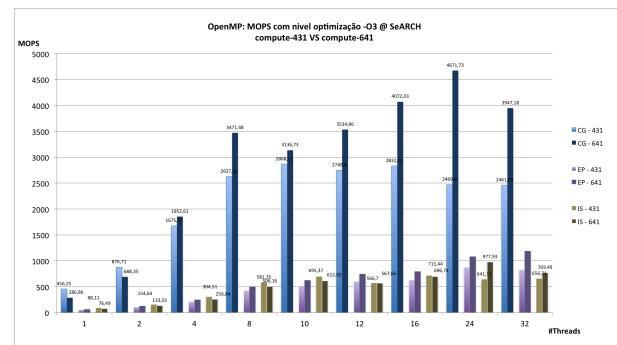


Figure 21: Mop/s para nível optimização -O3 compilado em intel.2013.1.117 e Classe C

## 9. ANÁLISE DISTRIBUÍDA: MPI - OPENMP (NPB3.3-MPI)

Concluída a análise dos kernels no paradigma de memória partilhada, avancei para a análise do desempenho dos kernels em memória distribuída. Nesta análise foram analisadas 3 métricas distintas: Tempo de execução, Mop/s e Mops por processo. Para tal, a análise foi feita utilizando 2 nós 641 do cluster SeARCH. Uma análise importante de fazer seria avaliar o desempenho dos algoritmos em memória distribuída utilizando rede **Gigabit ethernet** e **Myrinet** para avaliar diferenças de performance.

Contudo, por dificuldades de configuração do pedido PBS e indisponibilidade de nós livres com suporte de rede myrinet no SeARCH, não foi possível fazer esse ensaio e produzir resultados para fazer essa comparação.

### 9.1. Análise de tempos de execução: 8, 16 e 32 processos MPI em 2 nós 641

Pela observação dos resultados do gráfico podemos verificar que o número de processos que favorece os tempos de execução são 16 processos para a maioria dos kernels. Pelo relatório do MPI gerado (report bindings) pude verificar que estavam efectivamente a ser utilizados os dois processadores dos dois nós

pedidos. Contudo, verifica-se que para o kernel EP que utiliza um algoritmo embarrasosamente paralelo o número de processos MPI mais favorável são é de 32 processos. Sendo que cada nó 641 possui dois sockets de 8 cores físicos, temos um total de 16 cores físicos. Então, 32 processos mapeados por core num algoritmo embarrasosamente paralelo como o kernel EP revela ser, fazem um melhor aproveitamento dos cores disponíveis.

O tempo que demora a executar cada kernel para cada classe aumenta como seria de esperar, à medida que aumentamos o tamanho da classe. Isto deve-se ao facto do tamanho de cada classe ser de aproximadamente  $4x$  superior de uma classe para a seguinte.

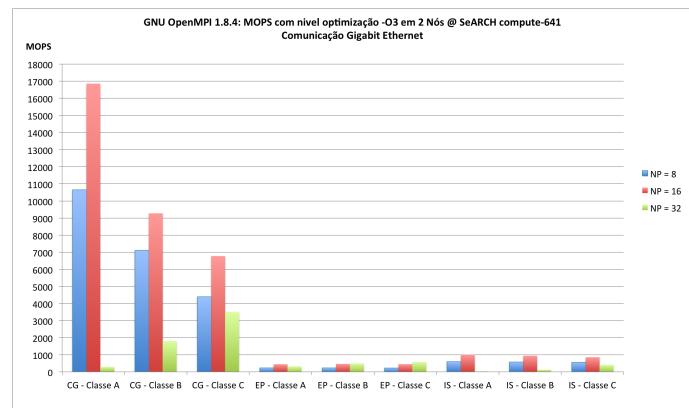


Figure 23: Mops para número variável de processos MPI em 2 nodos 641

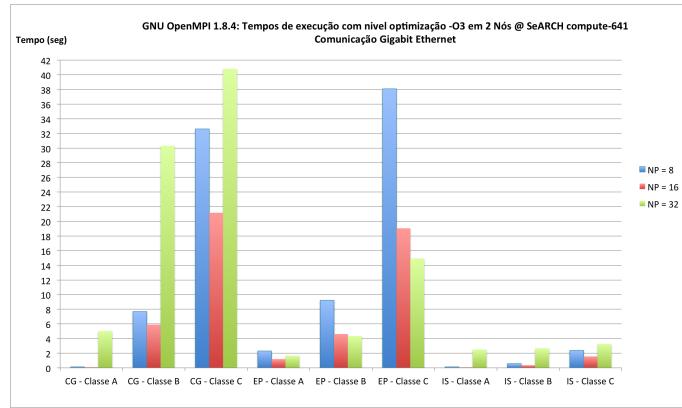


Figure 22: Tempos de execução para número variável de processos MPI em 2 nodos 641

## 9.2. Mop/s: 8, 16 e 32 processos MPI em 2 nós 641

Na análise do número de operações realizadas por unidade de tempo podemos verificar que também para 16 processos são realizadas um maior número de operações por segundo para a maioria dos kernels. À medida que aumentamos o tamanho dos dados, o número de Mop/s tende a diminuir. Podemos também verificar que comparativamente com os tempos de execução estudados anteriormente existe uma espécie de relação inversa entre os tempos de execução e o número de Mop/s (para este estudo). Quando temos tempos de execução mais curtos, o seu correspondente número de Mop/s é maior, e o contrário também se verifica.

## 9.3. Análise de Mops/processo para 8, 16 e 32 processos MPI em 2 nós 641

Através da análise dos Mops/processo podemos verificar qual a "eficiência" das operações executadas para o diferente número de processos MPI. A utilização de 16 processos, como podemos observar por 24 maximiza esta eficiência, para os diferentes kernels e para as diferentes classes.

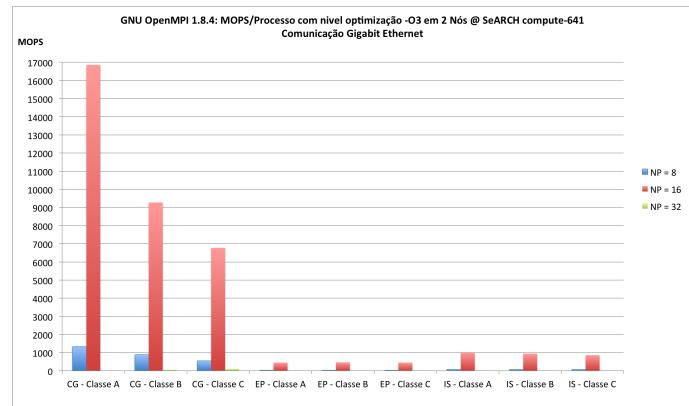


Figure 24: Mops/processo para número variável de processos MPI em 2 nodos 641

## 10. AVALIAÇÃO FINAL: SER VS OPENMP VS MPI

Depois de estudados os diferentes kernels nos diferentes paradigmas, criei um gráfico para tentar perceber qual o paradigma de computação que oferece um maior desempenho em termos de tempo de execução. Para tal, foram recolhidos os melhores tempos de cada kernel obtidos com compilador GNU 4.9.0 em cada um dos paradigmas e para MPI a biblioteca OpenMPI 1.8.4 para a maior classe de dados (classe C). Analisando o gráfico 25:

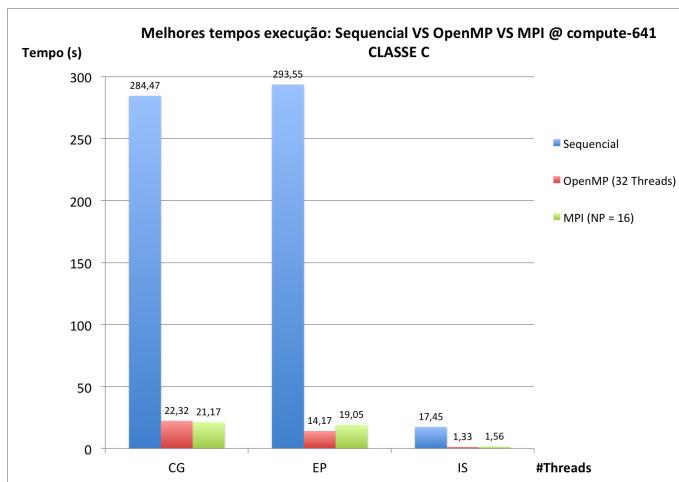


Figure 25: Comparação tempos de execução nos 3 paradigmas

Verificamos então que no kernel CG a versão MPI demonstrou ser aquela que tem maior desempenho. Já nos restantes kernels (EP e IS) as versões do kernel que revelaram ter um maior desempenho foram as versões OpenMP. Para este estudo, não podemos por isso eleger um paradigma favorito.

## 11. AFECTAÇÃO DO SISTEMA: MONITORIZAÇÃO COM DSTAT

Numa segunda fase do trabalho, realizei alguns testes de monitorização por forma a avaliar como é que a classe que coloca maior carga influencia o desempenho da máquina a nível de recursos de hardware. Para tal, utilizei o comando *dstat* de *UNIX* que me permitirá estudar a forma como o sistema é afetado em termos de utilização de CPU, memória, e disco nos nós do SeARCH.

Este comando permitirá gerar de forma automática, estatísticas sobre a utilização dos recursos em formato CSV. Com esse output construi gráficos que me permitissem perceber qual a afectação dos recursos de hardware. Entre as diferentes estatísticas que o comando *dstat* oferece, escolhi 3 que considerei as principais: Utilização de CPU, Memória e Disco.

Também pelo excerto da script que utilizei (e referenciada em 4) podemos verificar a execução do comando acompanhado das seguintes flags para gerar as estatísticas pretendidas:

```
1 /home/a59905/dstat -cdm --output $file.csv <-
    >> /dev/null &
```

Sendo que:

- c : Gera estatísticas para CPU nomeadamente a percentagem de CPU utilizada pelo sistema, tempo de utilizador, tempo de idle, tempos de espera por I/O, e interrupções de hardware e software;
- d : Gera estatísticas sobre a utilização dos discos no que toca a operações de leitura e escrita;

- m : Gera estatísticas sobre a utilização da memória: espaço utilizado, espaço livre, cache e buffers.

A nível de metodologia, optei por fazer este estudo para o Kernel EP, utilizando os melhores tempos de execução obtidos em cada um dos paradigmas e estudar o comportamento do sistema para esses resultados. A classe de dados utilizada foi a classe C. Por ser a classe de maior tamanho, os resultados obtidos a nível de tempo de execução deverão refletir resultados mais representativos deste estudo. Os resultados MPI utilizados são os resultados já explorados anteriormente para um número de processos igual a 32 que foi o número de processos para os quais os tempos de execução foram menores neste kernel.

Todas as estatísticas de CPU, Memória e Disco foram obtidas em máquinas 641.

### 11.1. Utilização de CPU: SER VS OpenMP VS MPI

Nos diferentes gráficos a que cheguei depois de estudar o comando *dstat*, pude verificar um conjunto de indicadores tais como:

- **usr** : Percentagem de tempo de CPU a executar código "user-mode" (fora do kernel). O tempo de gasto por outros processos, e o tempo que o processo fica bloqueado não é contabilizado. No fundo contabiliza a percentagem de tempo de CPU gasto a executar o processo.
- **sys** : Percentagem de tempo de CPU gasto a executar código contabilizando também o tempo gasto em system calls e chamadas ao kernel da máquina.
- **idl** : Percentagem de tempo que o processo permanece inativo sem realizar computação.
- **wait** : Percentagem de tempo que o programa fica a esperar por I/O (da memória, disco, ou rede) sem ainda ter recebido a informação para poder prosseguir[4];
- **hiq** : Relativo a interrupções causadas por hardware;
- **siq** : Relativo a interrupções provocadas por software;

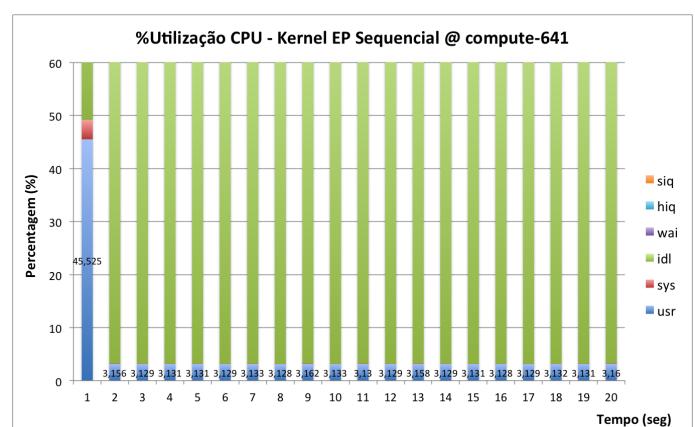


Figure 26: Utilização de CPU - Melhor execução Sequencial

Através da análise gráfica dos gráficos 26 e 27 podemos ver uma diferença bastante grande quanto à percentagem de utilização de CPU da versão sequencial, para a versão paralela em OpenMP do kernel EP. Uma conclusão imediata que podemos retirar destes dois gráficos logo à partida é que o paralelismo deste algoritmo introduz na computação um maior aproveitamento e utilização do CPU ao longo do tempo. O que seria de esperar uma vez que o próprio algoritmo por si só também escala razoavelmente bem. Em ambos verificamos que o primeiro segundo regista uma pequena percentagem de utilização do CPU a executar código do processo e de possíveis chamadas a sistema (sys) e os segundos seguintes são gastos praticamente a realizar computação do algoritmo.

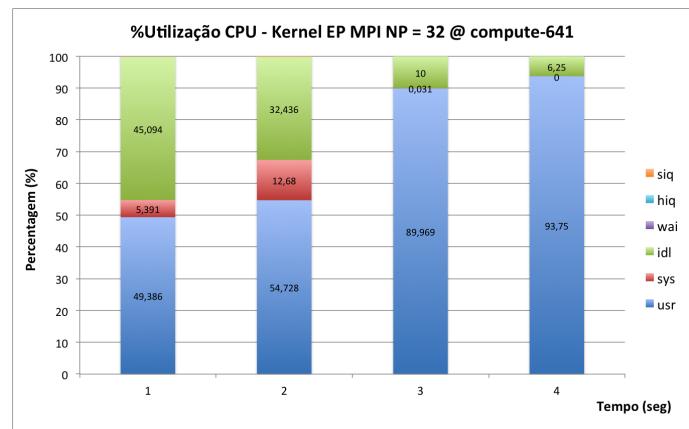


Figure 28: Utilização de CPU - Melhor execução MPI

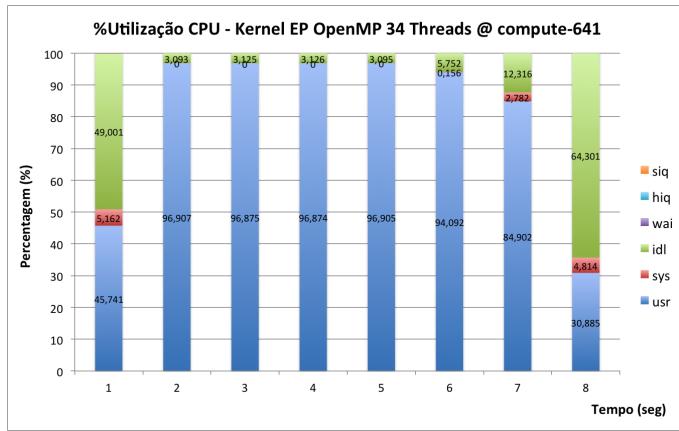


Figure 27: Utilização de CPU - Melhor execução paralela com OpenMP

Se atendermos à informação do gráfico que contabiliza a percentagem de utilização do CPU em MPI podemos perceber que continua a verificar-se a tendência anterior de um melhor aproveitamento do CPU em comparação com a versão sequencial do código. No entanto, enquanto que a percentagem de utilização do CPU na versão OpenMP (memória partilhada) é maior (na ordem dos 90%), em MPI (memória distribuída) essa percentagem de utilização é mais pequena nos instantes iniciais, mas aumenta progressivamente ao longo do tempo quando inicia a computação relacionada com o algoritmo.

Creio que esta tendência de, nos primeiros segundos de existir uma percentagem de tempo superior em MPI gasto em sys comparativamente com o que acontece em OpenMP se deva ao facto de haver um maior overhead a fazer o mapeamento dos processos pelos diversos cores em memória distribuída, do que em memória partilhada.

## 11.2. Utilização de Memória: SER VS OpenMP VS MPI

A análise da memória limita-se a ser uma análise em percentagem da relação entre o espaço livre e espaço ocupado durante a execução do benchmark. Como podemos verificar nos três gráficos 29, 30 e 31 no que toca à percentagem de memória utilizada, esta não varia muito de paradigma para paradigma estando sempre abaixo dos 2%.

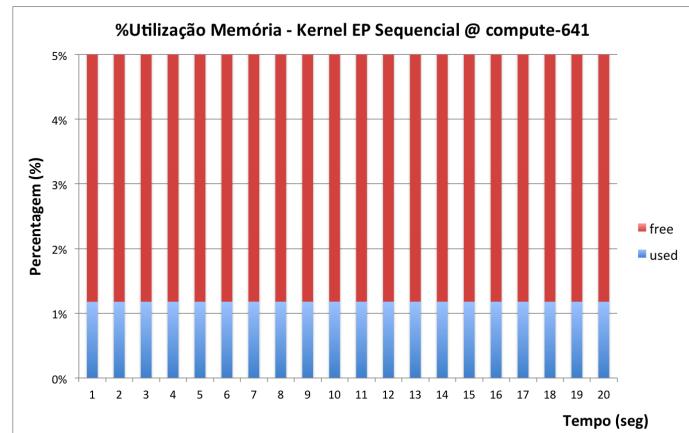


Figure 29: Leituras e escritas na Memória- Melhor execução Sequencial

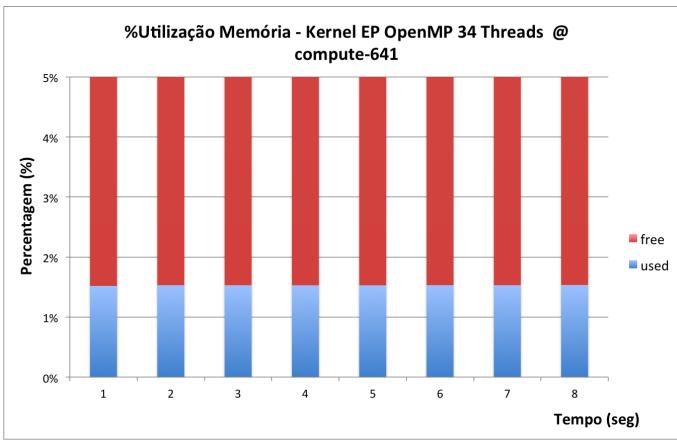


Figure 30: Leituras e escritas na Memória - Melhor execução paralela com OpenMP

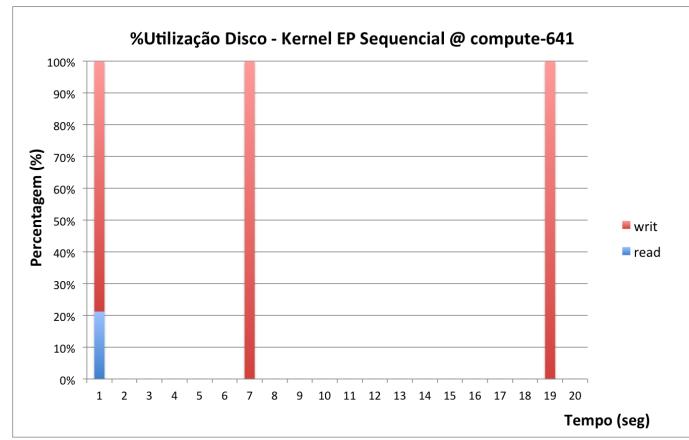


Figure 32: Utilização de disco: Kernel EP - Melhor execução Sequencial

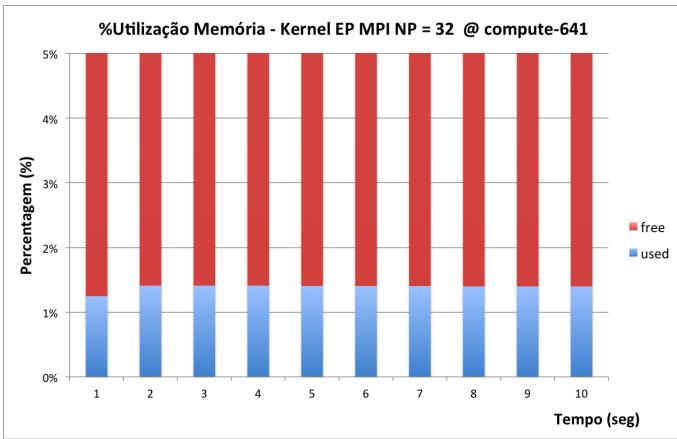


Figure 31: Leituras e escritas na Memória - Melhor execução MPI

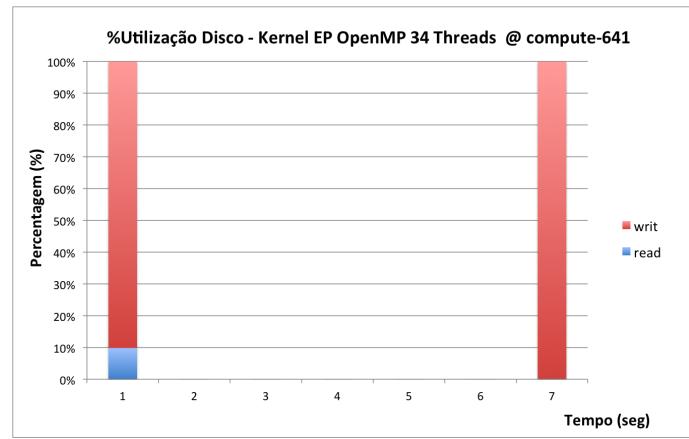


Figure 33: Utilização de disco: Kernel EP - Melhor execução paralela com OpenMP

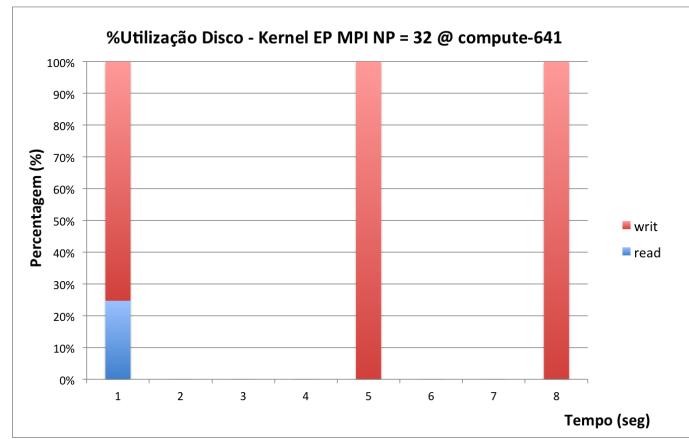


Figure 34: Utilização de disco: Kernel EP - Melhor execução MPI

### 11.3. Utilização de Disco: SER VS OpenMP VS MPI

Nos gráficos 32, 33 e 34 está representada a informação sobre a utilização dos discos nos diferentes paradigmas. A análise desta utilização é feita com base mais uma vez nas melhores execuções, *i.e* naquelas em que os tempos de execução são menores em cada um dos paradigmas, para a maior classe de dados (classe C) no kernel EP.

Comparativamente, a maior percentagem de leituras é feita em MPI rondando os 20% de leituras, um comportamento semelhante ao que acontece na versão sequencial do kernel. Globalmente, poucas são as operações de acesso a disco nos três paradigmas, sendo que que sempre que este é acedido geralmente é para fazer escritas e as leituras apenas são realizadas nos instantes iniciais.

## 12. TRABALHO FUTURO

Apesar de ter feito um estudo com objectivo de cobrir a maioria dos testes possíveis, existe um conjunto de testes alter-

nativos que poderiam futuramente ser explorados por forma a dar continuidade a este trabalho.

Um deles seria a comparação dos custos de comunicação para os kernels MPI ao executar os benchmarks utilizando *Giga-bit Ethernet* e *Myrinet*.

A análise dos resultados do comando dstat foram explorados neste estudo apenas para o Kernel EP. Um trabalho futuro interessante seria explorar o comando dstat (e outros comandos como iostat ou vmstat) para os restantes kernels para ter perfis de utilização de outros algoritmos que não escalem da mesma forma.

Apesar de terem sido explorados os diferentes resultados para diferentes micro-arquitecturas o SeARCH possui uma boa riqueza a nível de heterogeneidade de micro-arquitecturas. Por essa razão, refazer os mesmos testes para uma micro-arquitectura AMD e comparar os resultados entre Intel e AMD seria um confronto de resultados interessante.

### 13. CONCLUSÃO

Após a realização dos diversos testes, pude comprovar o impacto que os diferentes níveis de optimização têm na performance para os diferentes códigos, em diferentes tipos de algoritmos, com diferentes classes de dados, antes sequer de pensarmos em paralelismo.

A utilização de um paradigma de memória partilhada (OpenMP), para os kernels estudados revelou ter um impacto muito positivo nos tempos de execução e no número de operações realizadas por unidade de tempo à medida que aumentamos o número de threads utilizadas tirando um maior partido dos cores disponíveis. Em alguns testes a micro-arquitectura Nehalem revelou tempos de execução inferiores aos tempos registados numa micro-arquitectura mais recente (Ivy-Bridge) em single-thread por ter uma frequência de relógio superior. Quando o multi-threading é explorado, numa máquina com uma arquitectura mais recente que ainda por cima possui mais cores físicos, a micro-arquitectura nehalem acaba por ser ultrapassada. Das duas microarquitecturas estudadas, e pelo que investiguei do cluster SeARCH pude perceber que a intel tem desenhado micro-arquitecturas a pensar cada vez mais em execução de código paralelo, maximizando tempos de execução com processadores com mais cores, em detrimento de micro-arquitecturas single-thread para funcionar com processadores de frequências mais elevadas.

Depois dos diferentes testes nos diferentes paradigmas concluo também que globalmente o compilador icc da intel demonstra ser o compilador que gera código mais eficiente para as micro-arquitecturas estudadas.

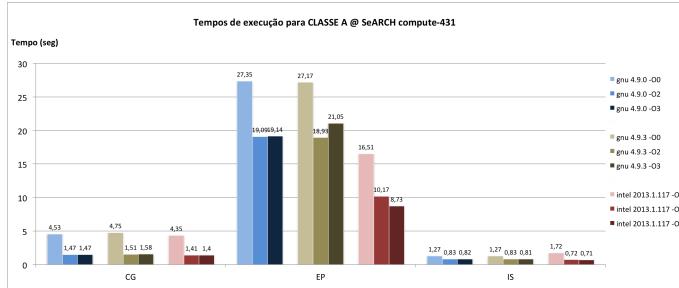
Dada a quantidade e variedade de testes que tive que realizar, tive a necessidade de criar uma script que realizasse os testes no cluster SeARCH, e me permitisse tratá-los de forma automática. Além do conjunto de competências inerentes à disciplina, este trabalho permitiu-me agilizar conhecimentos sobre scripting, profiling, sobre o sistema PBS e tratamento de informação com recurso a comandos unix tais como grep e awk.

### REFERENCES

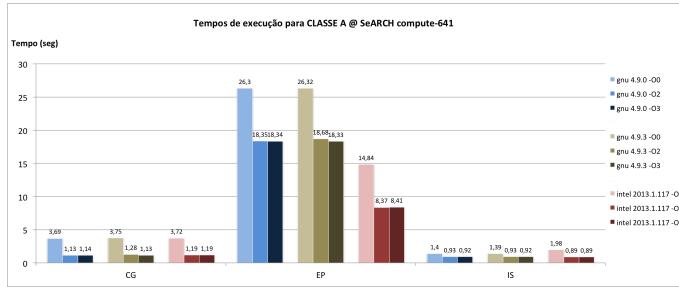
- [1] CPU-Benchmark. <http://puttin.xsrv.jp/?cat=2>.
- [2] NASA Advanced supercomputing division. <http://www.nas.nasa.gov/publications/npb.html>.
- [3] SeARCH Cluster Nodes. [http://search6.di.uminho.pt/wordpress/?page\\_id=55](http://search6.di.uminho.pt/wordpress/?page_id=55).
- [4] SliceHost - Using dstat to check I/O and swap. <http://articles.slicehost.com/2010/11/12/using-dstat-to-check-i-o-and-swap>.
- [5] Spec.org Web Site. <https://www.spec.org/cpu2006/results/res2014q3/cpu2006-20140908-31221.pdf>.
- [6] Subhash Saini and David H. Bailey. *NAS Parallel Benchmark (Version 1.0) Results 11-96*. NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, 1996.

## A. APPENDIX

### A.1. $T_{exec}$ Sequencial: Classe A

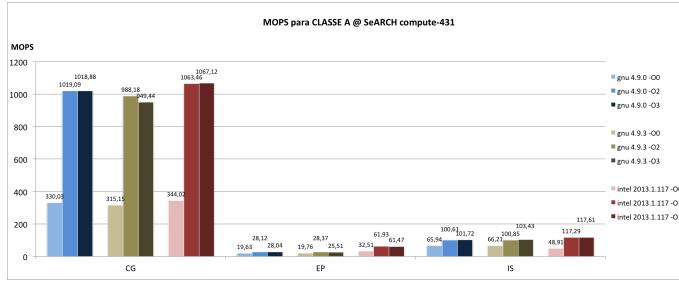


**Figure 35:** Comparação de compiladores para  $-O0$ ,  $-O2$ ,  $-O3$  num nó 431

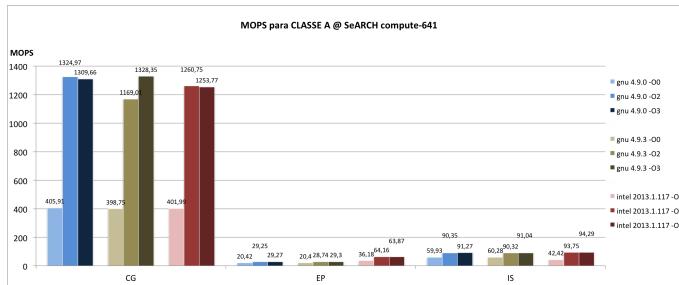


**Figure 36:** Comparação de compiladores para  $-O0$ ,  $-O2$ ,  $-O3$  num nó 641

### A.2. Mop/s Sequencial: Classe A

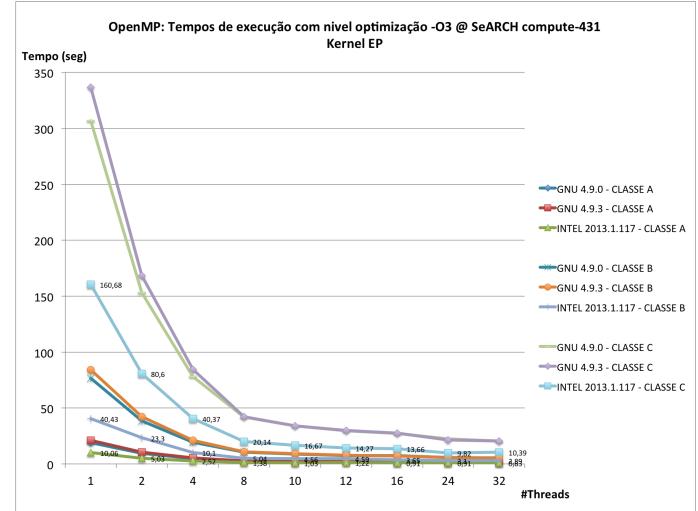


**Figure 37:** Comparação de Mop/s. Diferentes compiladores com  $-O0$ ,  $-O2$  e  $-O3$  em nó 431

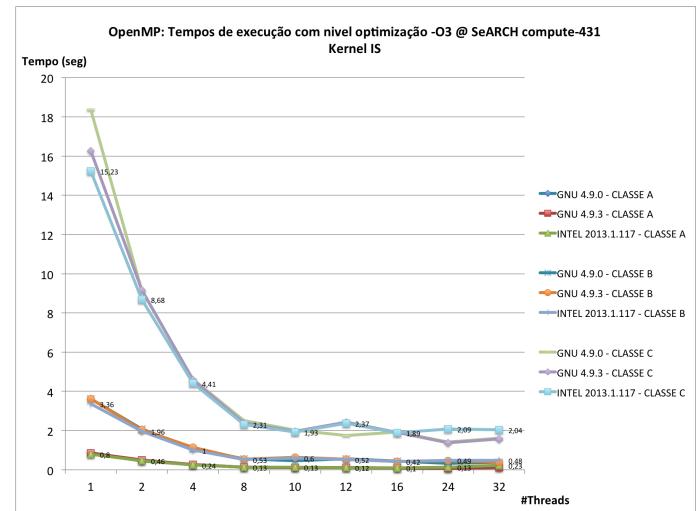


**Figure 38:** Comparação de Mop/s. Diferentes compiladores com  $-O0$ ,  $-O2$  e  $-O3$  em nó 641

### A.3. $T_{exec}$ para kernels paralelos com OpenMP



**Figure 39:** Performance de compiladores para diferentes classes - kernel EP



**Figure 40:** Performance de compiladores para diferentes classes - kernel IS

#### A.4. Mop/s para kernels paralelos com OpenMP

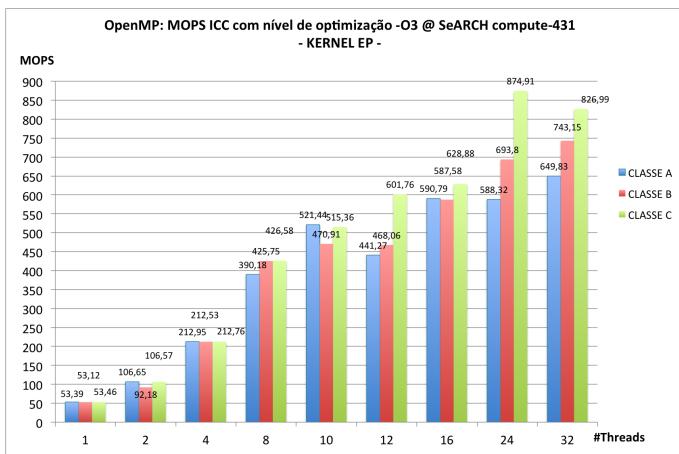


Figure 41: Mops obtidos com ICC para diferentes classes em nó 431 - kernel EP

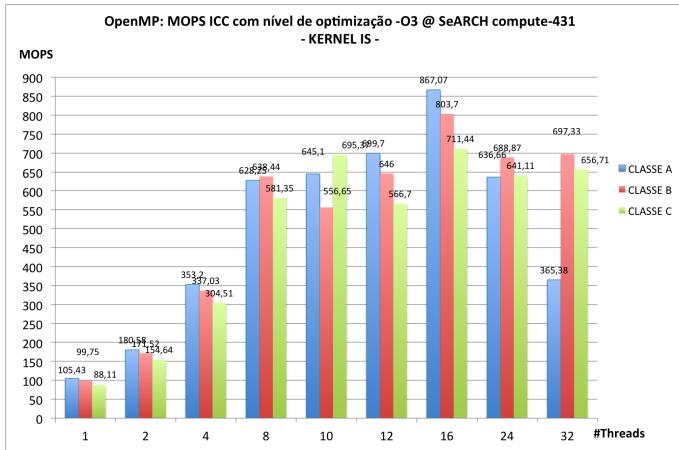


Figure 42: Mops obtidos com ICC para diferentes classes em nó 431 - kernel IS