TPC2: Análise de Desempenho em Supercomputadores Utilizando NAS Parallel Benchmark (NPB)
- Engenharia dos sistemas de Computação -

autor: Daniel Malhadas

***Resumo*—O presente documento apresenta um estudo aprofundado sobre análise de desempenho em supercomputadores, em particular, alguns nodos específicos do Cluster SeARCH. Para esta análise de desempenho foi utilizado o pacote NAS Parallel Banchmark (NPB) de forma a corretamente poder chegar a conclusões relevantes relativas ao desempenho de cada nodo do cluster mencionado e poder, de forma realista, comparar também os nodos entre si.**

***Index Terms*—NAS Parallel Benchmark, Análise de Desempenho, Computação Paralela e Distribuída.**

I. Introdução

A. Contextualização e Motivação

Com este projeto pretende-se recorrer ao uso de NAS Parallel Benchmarks (NPB) de forma a analisar o desempenho de nodos do cluster SeARCH. Estas análises consistem em executar repetidamente os mesmos benchmarks fazendo pequenas variações (por exemplo a ferramente de compilação, a rede para comunicação entre processos, o tamanho dos dados, etc) em diferentes máquinas de forma a poder-se entender os pontos fortes e fracos de cada máquina e também poder comparar esses mesmos pontos entre diferentes máquinas podendo assim com relativa certeza poder determinar que máquinas seriam mais indicadas para determinados tipos específicos de problemas e porquê.
Tendo em conta que o número de possibilidades de testes possíveis de realizar é praticamente infinita e sendo todos esses testes relativamente relevantes houve a necessidade de limitar um pouco o domínio dos testes a realizar. Esse domínio será explicitado ao longo deste capítulo nos próximos subcapítulos. No entanto, mesmo limitando o domínio, temos ainda uma enorme quantidade de dados a organizar, assim sendo o uso de comandos auxiliares como iostat, mpstat, top e dstat e de ferramentas como GNU plot permitem-nos um tratamento mais rápido e mais eficiente dos dados obtidos com os benchmarks mencionados e tornam-se então uma necessidade para uma análise de desempenho aprofundada.

B. O que são as NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB) é um pacote com um pequeno conjunto de programas idealizados para avaliar o desempenho de supercomputadores paralelos. Estas benchmarks derivam de aplicações na àrea *Computational Fluid Dynamics (CFMD)* e consistem em 5 kernels e 3 pseudo-aplicações na especificação original do NPB 1. Os tamanhos dos problemas NPB são pré-definidos e organizados por diferentes classes consoante o tamanho.[1]

II. Considerações Iniciais

A. Caracterização das Máquinas Utilizadas

Com o intuito de realizar análises de desempenho em nós específicos do cluster SeARCH, teve-se o cuidado de observar calmamente a grande quantidade de nós disponíveis e assim escolher nós relevantes para serem analisados em conjunto. Isto porque certas combinações de nodos não seriam boas para este trabalho, pois não há grande interesse em escolher, por exemplo, nós com microarquiteturas semelhantes, já que estaremos a comparar análises de desempenho obtidas em duas máquinas que operam de forma semelhante sendo que a sua comparação não nos iria trazer nada de novo.
Houve ainda um cuidado especial de, embora se terem escolhido nodos com mircroarquiteturas distintas, de escolher nodos com capacidades semelhantes essencialmente a nível da frequência do processador e número de cores físicos. Isto porque assim a análise será mais "justa", visto que, por exemplo, o desempenho de uma aplicação com um número de threads que exceda bastante o número de cores físicos numa máquina será em princípio menor do que numa outra máquina onde não exceda o número de cores físicos, fazendo com que a comparação não seja "justa"já que a máquina com menos cores físicos pode realmente ser melhor que a outra para um menor número de threads, mas isto poderá estar demasiado ofuscado ao observar o panorama geral.
Pelas razões referidas optaram-se então pelas máquinas do cluster SeARCH caracterizadas nas tabelas seguintes:

Designação	Nodo 652
Fabricante	Intel
Modelo do CPU	Dual CPU E5-2670v2
Microarquitetura do CPU	Ivy Bridge
Frequência do Clock	2.50 GHz
#Cores	20, com 40 threads (2 por core)
Cache	L1d: 32kB, L1i: 32kB, L2: 256kB, L3: 25600kB
Largura de Banda de acesso à memória	59.7GB/s
Memória RAM	64GB
Rede para Comunicação	Gigabit Ethernet e Myrinet 10Gbps

Designação	Nodo 431
Fabricante	Intel
Modelo do CPU	Dual CPU X5650
Microarquitetura do CPU	Nehalem
Frequência do Clock	2.67 GHz
#Cores	12, com 24 threads (2 por core)
Cache	L1d: 32kB, L1i: 32kB, L2: 256kB, L3: 12288kB
Largura de Banda de acesso à memória	32GB/s
Memória RAM	48GB
Rede para Comunicação	Gigabit Ethernet e Myrinet 10Gbps

Esta informação foi obtida apartir de várias fontes distintas. Sendo elas: - o comando Unix **lscpu**; - o site **cpu-world.com**; - o site **ark.intel.com**; - o site **http://search6.di.uminho.pt/wordpress/?page_id=55**;

Note-se que, embora apenas se tenham escolhido dois nodos do cluster, estes dois nodos permitem ter em conta o panorama geral da maioria dos nodos presentes no cluster, já que, depois de consultar o site **http://search6.di.uminho.pt/wordpress/?page_id=55** se pode observar que a microarquitetura mais comum nos nodos é Ivy Bridge seguida da Nehalem, por isso os nodos escolhidos são representativos de grande parte do domínio de nodos presentes no cluster.
Teve-se também o cuidado que, os nodos utilizados, permitissem ambos o uso de Gigabit Ethernet e de Myrinet para assim poderem ser comparadas as duas máquinas quanto à comunicação entre processos com cada um desses tipos de rede.

B. Caracterização dos Kernels utilizados

As 8 benchmarks especificadas no NPB 1 consistem em 5 kernels e 3 pseudo-aplicações. Visto que o objetivo deste estudo é o de comparar o desempenho em arquiteturas distintas com diferentes paradigmas de memória, diferentes compiladores, diferentesformas de comunicação entre máquinas, etc, temos maior interesse nos kernels do que nas pseudo-aplicações. Por esse motivo apenas analisamos os kernels. Apresenta-se de seguida cada um dos 5 kernels específicados no NPB 1 junto com uma breve descrição de cada:[1]

IS - Integer Sort
Kernel de ordenação de interior inteiros usando o algoritmo bucket sort[2]. Como se trata de uma ordenação sabemos implicitamente que consiste em acessos aleatórios à memória sendo assim possível testar comunicação irregular ora via memória partilhada ora via memória distribuída. Permite ainda perceber o desempenho do tratamento de número inteiros.
EP - Embarassingly Parallel
Kernel embaraçosamente paralelo que gera variáveis Gaussianas aleatórias usando o *Marsaglia Polar method*[2]. esta geração de variáveis Gaussianas traduz-se em instruções muito paralelizáveis sendo então de esperar que estas escalem bastante com o número de threads. Podemos ainda perceber o desempenho de números de virgula flutuante já que as variáveis Gaussianas referidas são isso mesmo.
CG - Conjugate Gradient
Kernel que estima o menor autovalor de uma grande matriz simétrica positiva usando a iteração inversa com o método *Conjugate Gradient* como uma sub-rotina para resolver sistemas de equações lineares[2]. Aqui iremos ter acesso irregular à memória e comunicação também irregular oque poderá levar a uma má escalabilidade a nível de threads/processos e memória. É relevante ver como cada nodo se irá comportar para um kernel deste tipo.
MG - Multi-Grid on a sequence of meshes
Kernel que aproxima a solução de uma equação de Poisson discreta tridimensional usando o método*V-cycle multigrid*[2]. Baseia-se em comunicações de curto e longo alcance numa grelha sendo também bastante intensivo a nível de memória. Espera-se então que a comunicação intensiva necessária para este kernel seja um fator limitante para a escalabilidade a nível de processos e memória em diferentes nodos, podendo assim talvez o paradigma de memória partilhada ter vantagens sobre o paradigma de memória distribuída.
FT - discrete 3D fast Fourier Transform
Kernel que resolve uma equação diferencial tridimensional parcial usando a transformada rápida de Fourier[2]. O funcionamento deste kernel tem por base "comunicação de todos-para-todos"sendo então bom para estudar o desempenho quando o fator limitante é a comunicação. Embora haja outros kernels que também tenham a comunicação entre processos como um fator limitante, este é o único onde a comunicação é o único fator limitante. No entanto, como se sente que a comunicação já é suficientemente estudada nos restantes kernels, optou-se por deixar este de parte, focando-se mais o estudo nos 4 anteriores.

Na descrição de cada kernel indica-se as razões pelas quais é relevante estudá-lo. Dos 5 apresentados apenas se abdicou do estudo do FT, pelas razões indicadas na sua descrição.

C. Caracterização dos Conjuntos de Dados Utilizados

Como referido anteriormente, os conjuntos de dados utilizados para os NPB consistem em tamanhos pré-definidos que se dividem em várias classes identificadas por letras distintas. As classes referidas encontram-se caracterizadas para cada kernel na seguinte tabela:[3]

Benchmark	Parameter	Class S	Class W	Class A	Class B	Class C	Class D	Class E
CG	no. of rows	1400	7000	14000	75000	150000	1500000	9000000
	no. of nonzeros	7	8	11	13	15	21	26
	no. of iterations	15	15	15	75	75	100	100
	eigenvalue shift	10	12	20	60	110	500	1500
EP	no. of random-number pairs	2^{24}	2^{25}	2^{28}	2^{30}	2^{32}	2^{36}	2^{40}
FT	grid size	64 x 64 x 64	128 x 128 x 32	256 x 256 x 128	512 x 256 x 256	512 x 512 x 512	2048 x 1024 x 1024	4096 x 2048 x 2048
	no. of iterations	6	6	6	20	20	25	25
IS	no. of keys	2^{16}	2^{20}	2^{23}	2^{25}	2^{27}	2^{31}	
	key max. value	2^{11}	2^{16}	2^{19}	2^{21}	2^{23}	2^{27}	
MG	grid size	32 x 32 x 32	128 x 128 x 128	256 x 256 x 256	256 x 256 x 256	512 x 512 x 512	1024 x 1024 x 1024	2048 x 2048 x 2048
	no. of iterations	4	4	4	20	20	50	50

Considera-se que o espaço ocupado pelos conjuntos de dados em memória são o fator mais relevante e não os valores dos dados em si, por isso de forma a escolher tamanhos que sejam realmente relevantes teve-se em conta os seguintes critérios relacionados com o espaço em memória:

1 - Conjuntos de dados que caibam na Cache L1 das máquinas utilizadas

As máquinas usadas e descritas anteriormente têm L1d: 32kB e L1i: 32kB, apenas nos interessa dados, por isso temos um total de 32kB na Cache L1 para os nossos dados, logo este nível da cache poderá conter 32000B/4B=8000 valores inteiros ou de vírgula flutuante de 4B cada. Mas não terminamos por aqui, podemos ainda ter em atenção o tamanho de uma linha da cache. Neste caso cada linha terá 64B, por isso cabem 16 valores por linha, logo o número ideal valores no conjunto de dados deve ser um múltiplo de 16 de forma a maximizar o proveito tirado da localidade espacial. 8000/16 = 500, logo a dimensão ideal seria de 8000 valores de 4B ou 4000 valores de 8B (double). No entanto, note-se que não se encontrou uma classe cujos conjuntos de dados para os diferentes kernels caibam todos dentro da cache L1, não foi possível então escolher nenhum conjunto de dados com estas características. **2 - Conjuntos de dados que caibam na Cache L2 das máquinas utilizadas**

Calculando da mesma forma vemos que, para a cache L2, o número ideal de valores seria 64000 inteiros ou 32000 doubles. Na classe S vemos que há alguns conjuntos de dados que cabem nesta cache L2.

3 - Conjuntos de dados que caibam na cache L3 das máquinas utilizadas

Calculando da mesma forma, para o nodo com menor cache L3 (o nodo 431 com cache L3 de 12288kB), vemos que o número ideal de valores seria 3072000 inteiros ou 1536000 doubles. Na classe S, W, A, B e C vemos que há alguns conjuntos de dados que cabem nesta cache L3.

4 - Conjuntos de dados que não caibam em nenhum dos níveis de Cache anteriores e que forcem o CPU a carregá-lo da DRAM

Para este conjunto de dados precisamos somente de conjuntos que sejam significantemente maiores que a cache L3. A, B, C, D e E têm vários conjuntos de dados que verificam esta condição.

Concluímos agora que todas as classes têm conjuntos de dados que são de alguma forma relevantes, no entanto, utilizá-las todas não seria de todo relevante. Por exemplo, as classes S e W enquadram-se as duas num escopo com conjuntos de dados pequenos em comparação com as outras. São demasiado parecidas e como tal, o uso das duas ao mesmo tempo seria redundante. Depois de alguns testes para se determinar qual seria mais relevante a usar concluiu-se que ambas são demasiado pequenas em comparação com as restantes e, por essa razão, tornam-se obsoletas para os nodos utilizados que as correm demasiado rapido, não obtendo dados particularmente interessantes. O mesmo se conclui para a classe D e E, mas pela razão oposta. Ambas se enquandram num escopo demasiado grande em comparação com as restantes. São tão grandes até que, ao fim de alguns testes se preveu que não se obteriam bons resultados em tempo útil se estas fossem utilizadas. Por estes motivos optou-se por utilizar as classes **A, B e C**, sendo que estas classes respeitam, dos critérios mencionados a negrito anteriormente, o critério 2, 3 e 4. Ao "desistir"das outras classes não perdemos nada no panorama geral, visto que nenhuma classe respeitava o critério 1, ou seja, continuamos a ter conjuntos de dados relevantes para a 2, 3 e 4 apesar de apenas utilizarmos as classes A, B e C.

De forma a usar as classses selecionadas com os kernels selecionados observou-se a seguinte tabela com as distribuições dos NPB disponíveis[1]:

Version	Benchmarks Included	Problem Classes	Programming Models Used	Major Changes
NPB 3.3.1	IS, EP, CG, MG, FT, BT, BT-IO, SP, LU, UA, DC, DT	S,W,A,B,C,D,E	MPI, OpenMP, serial	added Class E
NPB 3.3.1-MZ	BT-MZ, SP-MZ, LU-MZ	S,W,A,B,C,D,E,F	MPI+OpenMP, OpenMP, serial	nested OpenMP version
GridNPB 3.1	ED, HC, VP, MB	S,W,A,B	Globus, Java, serial	added Globus version
NPB 3.0	IS, EP, CG, MG, FT, BT, SP, LU	S,W,A,B,C	OpenMP, HPF, Java	new programming paradigms
NPB 2.4.1	IS, EP, CG, MG, FT, BT, BT-IO, SP, LU	S,W,A,B,C,D	MPI	added BT-IO, Class D
NPB 2.3	IS, EP, CG, MG, FT, BT, SP, LU	S,W,A,B,C	MPI, serial	added CG, serial version

Observou-se que se teria de usar a distribuição **NPB3.3.1**, pois esta é a única que, para os kernels pretendidos e as classes pretendidas, permite avaliar tanto a versão sequencial, como uma versão em memória partilhada (OpenMP) e uma versão em memória distribuída (MPI).

D. Caraterização do Procedimento de Medição Utilizado

Entende-se que seja relevante realizar medidas de desempenho relativas à execução de 3 principais tipos de paradigmas: **Paradigma Sequencial**; **Paradigma Paralelo - Memória Partilhada**; **Paradigma Paralelo - Memória Distribuída**. Para cada um destes paradigmas iremos testar com as versões adequadas de cada kernel, sendo elas respetivamente, **as versões sequenciais**; **as versões com OpenMP**; **as versões com MPI**.

De forma a realizar bons testes que adjectivem as capacidades máximas das máquinas utilizadas entende-se que a máquina deve estar a ser usada na sua totalidade. Isto é, no mínimo, todos os cores físicos da máquina devem ser utilizados em todos os testes a realizar. O uso de um maior número de threads/processos do que cores físicos nos testes relacionados com os paradigmas paralelos é também bastante interessante

para observar de forma direta o impacto de HyperThreading na execução, nas medições dos kernels em paradigma paralelo isto é tido em conta e faz-se também um pequeno estudo relacionado com o assunto.
Embora se reserve toda a máquina em cada teste sabe-se também que haverá sempre algum "ruído"implícito, por exemplo, por causa de processos externos ao controlo do utilizador da máquina que correm por omissão no background, ou por mudanças súbitas da frequência de clock talvez por um aquecimento excessivo da máquina em questão. Por estas razões indicadas nem sempre é fácil garantir que os testes realizados utilizam a máquina a cem por cento, no entanto, por falta de uma maneira de o fazer iremos assumir que o pedido de um job com o comando qsub com argumentos **-lnodes=1:r652:ppn=40** ou **-lnodes=1:r431:ppn=24** serão suficientes para garantir a exclusividade de cada nodo.

De forma a preparar cada execução, olhando para as ferramentas disponíveis nos dois nodos a utilizar, decidiu-se fazer todos os testes compilando com recurso ao compilador gcc e, de seguida, novamente todos os testes com o compilador icc. Para o icc tencionava-se usar a única versão disponível nas duas máquinas icc 2013.1.117. No entanto ao tentar usar esta ferramente uma falha na licança impediu o seu uso em ambos os nodos, por essa razão optou-se antes pelo gcc versão 4.7.2 já que o gcc versão 4.7.0 (não presente nos nodos usados) é a versão diretamente compativel com o icc referido. Para a outra versão do gcc escolheu-se a mais recente presente nos dois nodos (4.9.3), já que assim temos comparação direta entre ferramentas mais distintas o que se torna mais relevante. Para a compilação de kernels com MPI utiliza-se a ferramenta mpicc, sendo a versão do MPI usada a 1.6.2. Quanto às flags de optimização, usa-se apenas O2 e O3 comparando-se de seguida as execuções com estas flags com a execução do código compilado sem nenhuma flag de optimização.

Olhamos agora para um exemplo de um excerto de um script utilizado para executar os kernels para os testes:

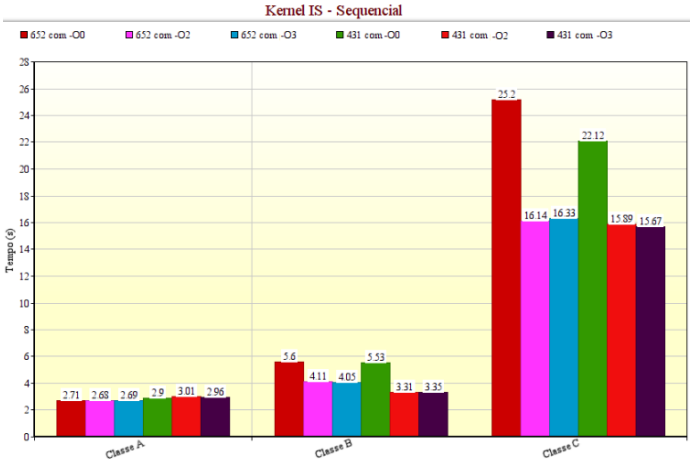
```
for ((threads = 1; threads <= max_threads;
    threads++))                          (3)
do
   for ((seq=1; seq <= \$sample_size; seq++))
   do
      for execc in *.x
      do
(1)      /home/a72293/resultados_gcc/dstat
         -tvfnml --output "\$execc"_"\$seq"_"
         \$node_info".csv >> /dev/null &
(2)      ./\$execc > DSTAT_"\$execc"_"\$seq"_"
         \$node_info".txt
         kill \$!
         sleep 4                         (4)
      done
   done
done
```

Note-se no ponto circundado 1 como a ferramenta dstat é utilizada com o propósito de reunir dados relativos a ocupação em memória, ocupação do CPU, a transferência de dados em rede, etc, causada pela execução de cada kernel, que ocorre no segundo ponto circundado. Caso estivessemos a querer testar o paradigma de memória partilhada com OpenMP, bastaria adicionar uma linha no ponto circundado 3 que atualizasse a variável de ambiente OMP_NUM_THREADS para o valor da variável threads. Note-se que, para os testes sequenciais, max_threads será igual a 1. Para o caso da situação de memória distribuída com MPI, podemos modificar a linha do ponto circundado 2, com o intuito de usar a ferramenta mpirun que permite executar programas compilados com mpicc e dar o valor da variável threads como argumento a mpirun.
Por último, repare-se no ponto circundado 4, onde se tem o cuidado de usar o comando kill para de imediato impedir a ferramenta dstat de continuar a executar sem que haja necessidade. Depois é usado o comando sleep para que assim haja um compasso de espera entre cada medição de forma a tentar manter alguma imparcialidade na mesma.
Cada teste é repetido 10 vezes fazendo-se a média dos 5 melhores. No caso dos 5 melhores terem variações demasiado grandes entre si então repete-se os testes de forma a que os 5 melhores referidos não tenham uma divergência entre si de mais do que 10%.

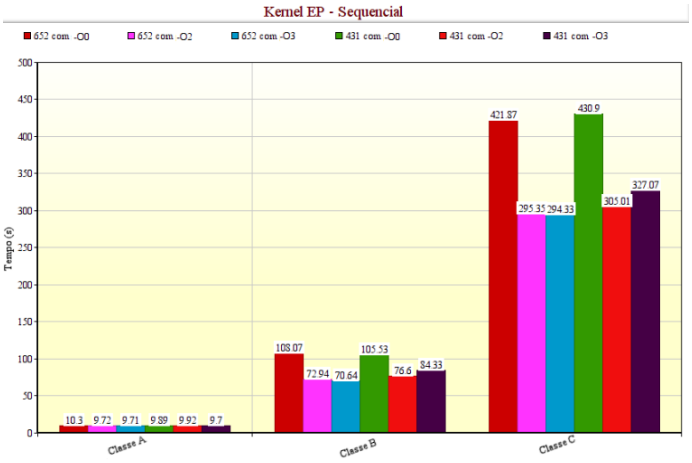
III. Medições de Desempenho - Paradigma Sequencial

Para cada kernel mediu-se o tempo em segundos que demorou em cada máquina com cada classe para cada compilador com diferentes flags de otimização. Apresentam-se agora gráficos demonstrativos dos resultados obtidos sendo cada gráfico relativo a um kernel distinto:

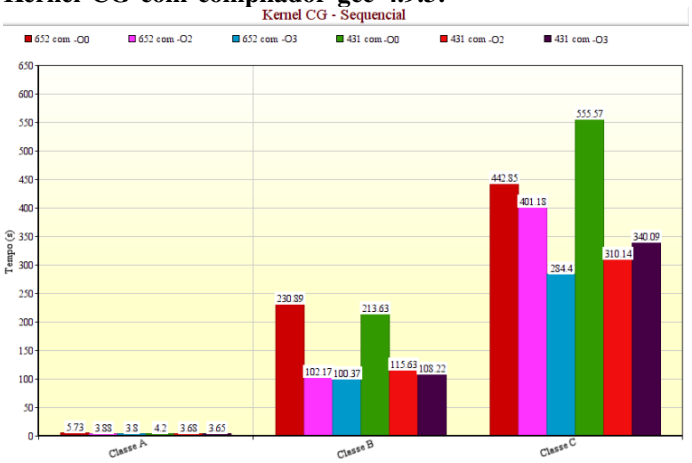
Kernel IS com compilador gcc 4.9.3:



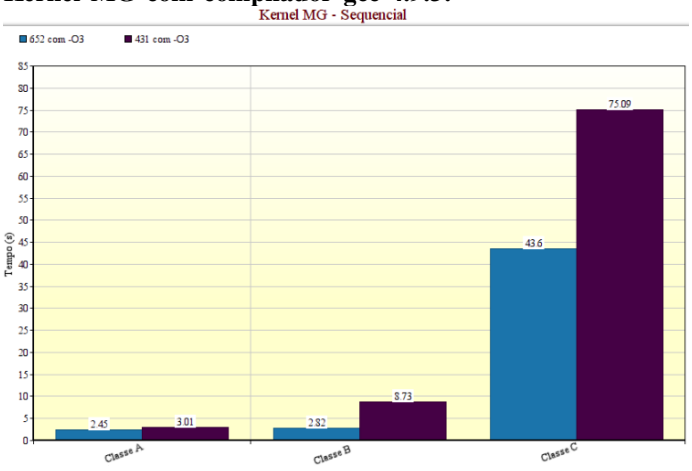
Kernel EP com compilador gcc 4.9.3:



Kernel CG com compilador gcc 4.9.3:



Kernel MG com compilador gcc 4.9.3:



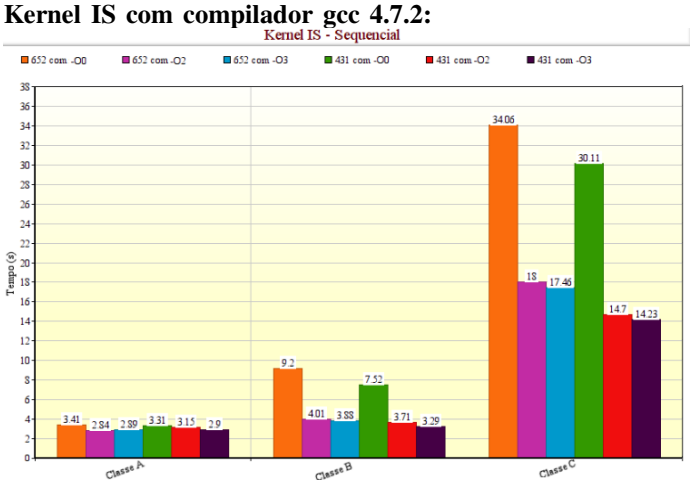
Olhando para os gráficos podemos observar fácilmente o impacto do uso de diferentes classes em cada kernel, em cada nodo e o impacto de variar as flags de optimização para o primeiro compilador.

Note-se que houve problemas com os resultados obtidos com o kernel MG já que estes foram valores sem sentido que decerto estavam errados por alguma situação exterior que não foi possível identificar (eram sempre obtidos valores perto do zero), por essa razão, não foi possível obter resultados sem flags de otimização nem com flag -O2. Os resultados obtidos com -O3 pensa-se serem confiáveis, no entanto, tendo em conta a situação e o desconhecimento da sua causa, não se pode ter cem por cento de certeza.

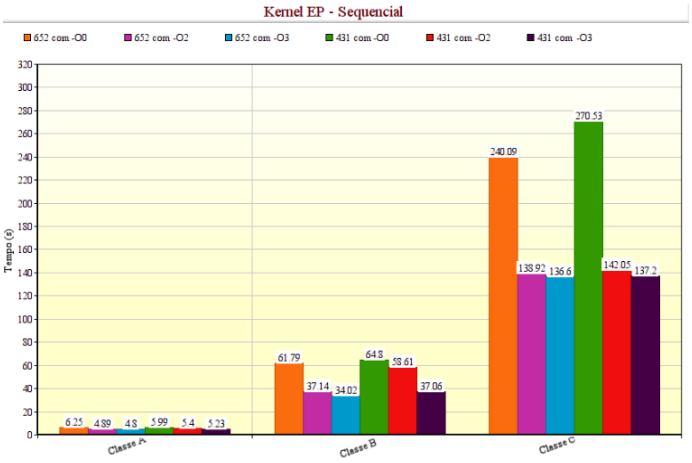
Olhando para os gráficos no geral vemos que com todos os kernels a classe A tem valores demasiado semelhantes nas diferentes situações o que leva a que não possamos concluir nada de especial olhando para os resultados da mesma. O pequeno tamanho da classe A em comparação com as restantes é a causa mais provável para este comportamento. Por esta razão, em testes posteriores não iremos avaliar a classe A.
Vemos também um aumento no tempo de execução à medida que se usam classes maiores, como seria expectável. No geral usar flag de otimização -O2 neste compilador é sempre melhor em termos de tempo de execução em comparação com o uso de nenhuma flag, no entanto com a flag -O3 os ganhos, quando existem, são mínimos. Uma possível razão para isto é a flag O2 fazer já a maioria das optimizações possíveis aos kernels utilizados, sendo que a flag O3 pouco mais pode fazer já que, dependências de dados, situações que posssam dar azo a pointer aliasing, etc podem estar a impedir uma maior performance com o uso da flag -O3.
Notamos ainda que, na maioria dos casos, o nodo 652 tem uma melhor performance tanto sem flags de otimização como com falgs de otimização. O kernel IS é talvez o que mais diverge desta afirmação o que pode indicar uma maior capacidade do nodo 431 para o tratamento de inteiros e/ou acessos aleatórios à memória (como é característico de algoritmos de ordenação).

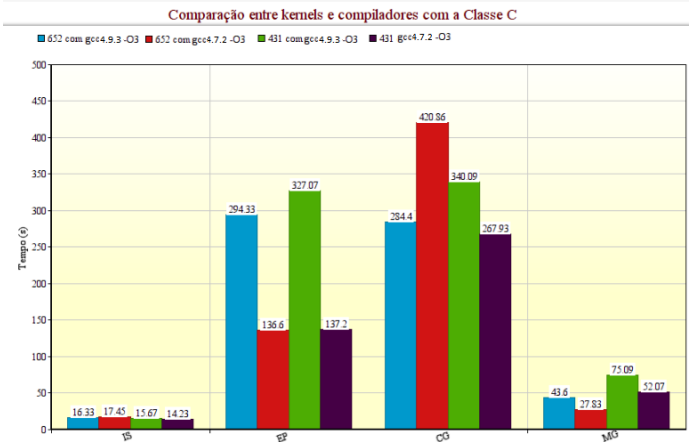
Vemos agora os mesmos gráficos mas para o segundo compilador:

Kernel IS com compilador gcc 4.7.2:

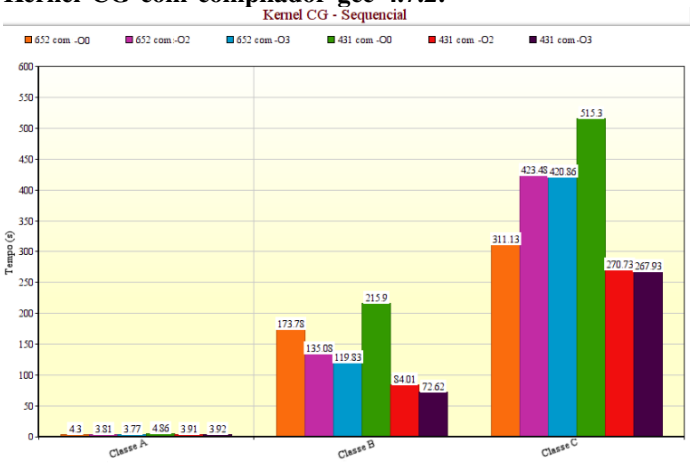


Kernel EP com compilador gcc 4.7.2:

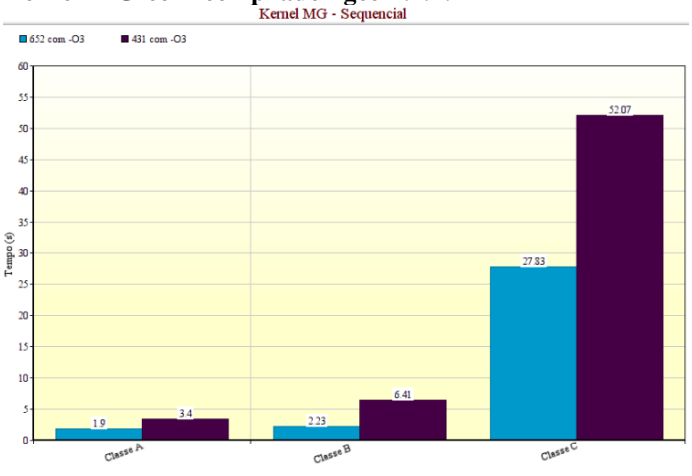




Kernel CG com compilador gcc 4.7.2:



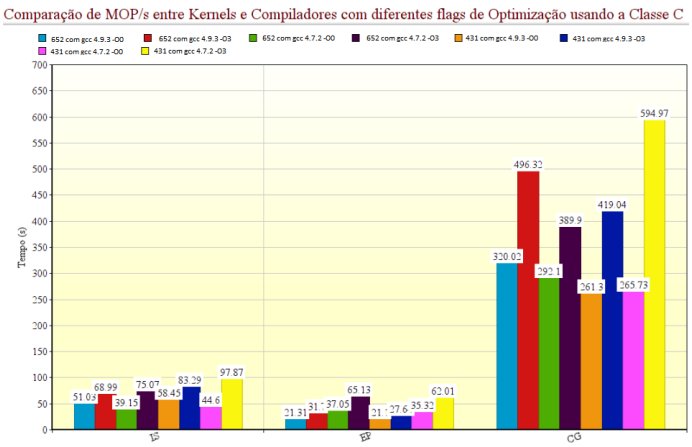
Kernel MG com compilador gcc 4.7.2:

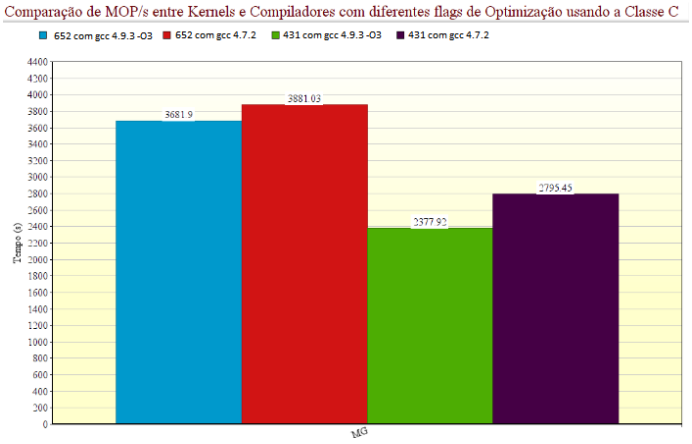


Relativamente aos nodos as mesmas conclusões que foram obtidas para o compilador anterior mantêm-se. De forma a poder comparar diretamente os resultados obtidos com os dois compiladores distintos apresenta-se um gráfico que mostra para cada nodo e cada kernel os resultados dos dois compiladores com a flag -O3:

Com este gráfico Observamos que o segundo compilador obtém melhores resultados com a flag -O3 do que o compilador 1, no entanto observando os gráficos anteriores vemos que para as restantes flags o compilador 1 tem melhores resultados. Isto leva a crer que o segundo compilador consegue ultrapassar mais facilmente as barreiras dos kernels no que diz respeito a otimizações.

Observamos agora, também para o paradigma sequencial, MOP/s (Milhões de Operações por segundo) para cada kernel de forma a entendermos melhor a diferença entre cada compilador sem flags de otimização e com a flag -O3 e entender também melhor as diferenças de performance obtidas num kernel em relação a outro já que seremos capazes de entender em que kernels foi possível obter mais optimizações com as flags de optimização. Note-se que agora apenas usamos a classe C, pois como observamos nos gráficos anteriores é a que causa maiores tempos de execução, logo é aquela que se torna mais relavante sendo a mais fácil de observar variações, mesmo que pequenas.



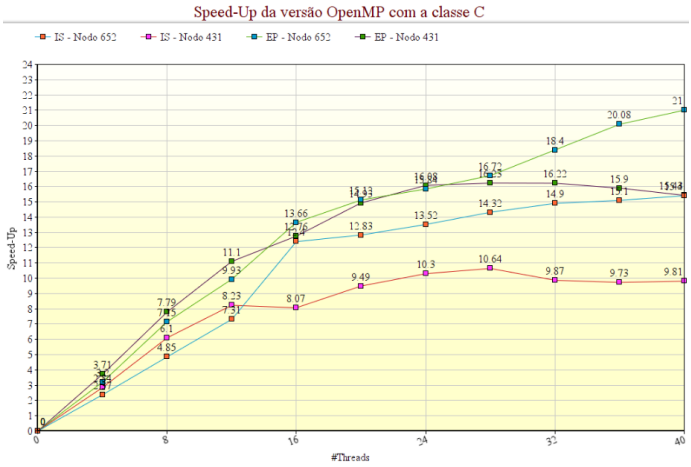


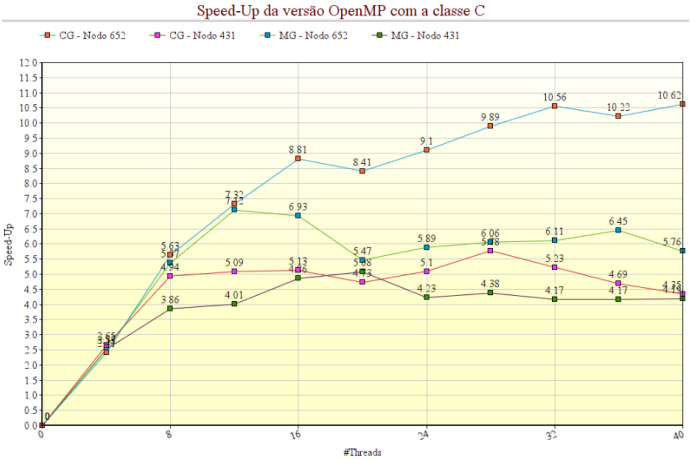
Aqui podemos concluir que o uso de flags de otimização superiores possibilitam um maior número de operações por segundo em ambos os compiladores, o que era expectável, mostrando assim realmente que níveis de otimização superiores podem alcançar melhores tempos de execução como vimos anteriormente já que no mesmo período de tempo é realizado um maior número de instruções.
Vemos também que em várias situações o nodo 431 se revela melhor a nível de MOPS/s com flag de otimização O3, no entanto para os kernels mais paralelizáveis como o EP esta diferença já não é tão acentuada. Conclui-se então que código pouco paralelizável é mais fácilmente otimizado no nodo 431, enquanto que código muito paralelizável como o EP obtém melhores otimizações no nodo 652 levando a crer que este último terá resultados melhores que o 431 para as versões paralelas quando otimizadas com a flag -O3.

Em suma, Se usada a flag de compilação -O3 então é vantajoso o uso do segundo compilador, no caso contrário o uso do primeiro é preferível. Se o código sequencial a executar tiver demasiadas dependências sendo então código muito pouco paralelizável então há vantagens em o usar na microarquitetura Nehalem como a que temos no nodo 431. Caso o código a executar seja muito paralelizável, até se for embaraçosamente paralelo como o kernel EP então será preferido o uso da microarquitectura Ivy Bridge como temos no nodo 652. Podemos ainda ter em conta que código que envolva operações de ordenação de inteiroso tem melhores resultados no nodo 431 sendo então justificável o uso do mesmo para executar programas que recorram a ordenação.

IV. Medições de Desempenho - Paradigma Paralelo: Memória Partilhada

Apresentam-se agora gráficos dos ganhos obtidos na versão paralela em memória distribuída com OpenMP em relação á versão sequencial. Note-se que, como referimos anteriormente, a classe C é a que se torna mais relevante e, por esse motivo, é a classe que usamos em todos os testes daqui para a frente. O compilador usado é o compilador 1 apenas com a flag -O3 já que esta foi a flag que proporcionou os melhores resultados sequenciais.





Infelizmente houve a necessidade de separar os kernels em 2 gráficos distintos para evitar que fosse confuso demais. Desta forma torna-se mais complicado entender para cada número de threads utilizado as diferenças de desempenho dos kernels em comparação uns com os outros. No entanto podemos fácilmente ver que o kernel EP é o que obtém os melhores ganhos relativamente á sua versão sequencial, isto já era de esperar já que tinhamos visto a sua natureza paralela explícita no facto de o número de MOP/s da versão sequencial ser o menor de entre todos os kernels. Como os MOP/s são tão reduzidos a divisão da computação entre threads torna-se fácil e sendo ele embaraçosamente paralelo, tendo poucas dependências, a divisão dos dados também se torna simples. Contrastando com o kernel EP temos o kernel MG que é o que obtém menos ganhos em ambos os nodos. O facto de ser o kernel que obteve o maior número de MOP/s na versão sequencial pode ser sinal de que a divisão da computação pelas threads é um processo complicado. Em norma, no nodo 652, os ganhos são superiores aos obtidos no nodo 431, mas apenas apartir de cerca das 12 threads. Como tinhamos visto no capítulo anterior era expectável que o nodo 652 paralelizasse mais facilmente. Era também já esperado que o nodo 431 após as 12 threads comece a ter uma curva de speed up menos acentuada já que apenas tem 12 cores físicos e, apartir das 12 threads estaremos a recorrer a hyperthreading o que terá algum impacto no desempenho. Entendemos de imediato que 12 cores físicos nunca poderão competir com os 20 cores físicos presentes no outro nodo onde só se começa

a ver uma quebra na curva por volta das 20 threads. Para além disso, o facto de o nodo 652 ter uma cache de nível 3 maior que o nodo 431 também é um aspeto relevante que leva a um melhor desempenho por parte do 652. O facto de, em algumas situações, antes das 12 threads o nodo 431 obter melhores resultados pode ter sido influenciado pela sua maior frequência de clock que, embora seja uma diferença não muito significativa, poderá ser mais explícita em código em execução paralela já que temos vários cores a operar com uma frequência superior aos cores do outro nodo. Além disso, o facto de o nodo 431 ter tido resultados muito mais baixos na versão sequencial poderá fazer com que o aumento do desempenho com a versão paralela seja mais significativo.

No kernel EP usando o nodo 652 notamos também que a quebra na curva dos ganhos mencionada anteriormente é praticamente inexistente. Uma possível razão poderá ser a natureza embaraçosamente paralela do kernel em questão que apenas tem a beneficiar com um maior número de threads e a razão pela qual isto não acontece tanto para o nodo 431 poderá ser precisamente por o nodo 652 ser capaz de melhores otimizações a código paralelizável, como vimos no capítulo anterior.

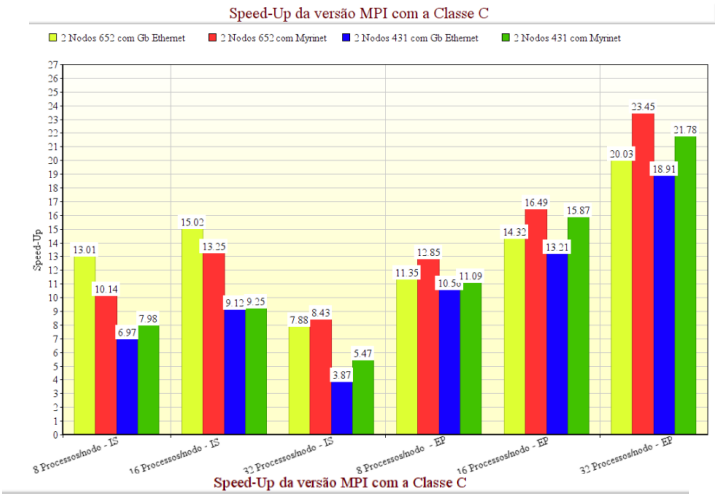
Notemos ainda que, para um número de threads superior a 24, o nodo 431 passa a não ter threads suficientes para que todas possam agir concurrentemente, nem mesmo com recurso a hyperthreading. Deste modo incorre em vários overheads que poderão ser a razão pela qual para um grande número de threads os resultados do nodo 431 são tão inferiores aos resultados do nodo 652. Já que no nodo 431 um certo número de threads terá que estar em espera sendo que, quando finalmente forem escalonadas terão que copiar tudo o que foi feito pelas outras para a sua cache de forma a manter a coerência de cache já que os dados que tinham poderão agora estar obsoletos ao terem sido modificados pelas restantes threads.

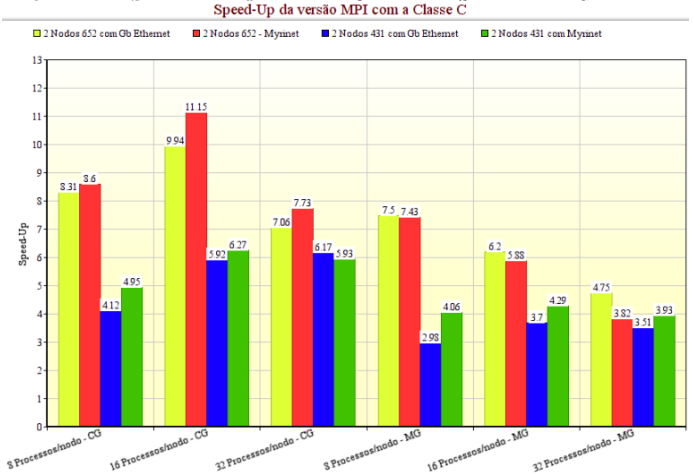
Em suma, para grandes números de threads o nodo 652 mostra vantagens significativas, como já seria de esperar, no entanto, se o objetivo for usar um algoritmo que apenas escale com um pequeno número de threads (12 ou menos) o nodo 431 já será uma melhor opção mesmo tendo uma menor cache de nível 3.

V. Medições de Desempenho - Paradigma Paralelo: Memória Distribuída

Com o intuito de fazer testes em ambiente paralelo com memória distribuída surgiu uma questão a solucionar. Não tem propósito o teste em ambiente de memória distribuída se for feito num só nó como foi feito nos capítulos anteriores, pois assim os processos irão comunicar com recurso a memória partilhada e não através de rede Ethernet ou Myrinet impedindo-nos de avaliar as diferenças de desempenho com estas diferentes maneiras de comunicação, portanto há que decidir quantos nodos usar. Bem sabemos que quantos mais nodos usarmos mais relevante será o nosso estudo já que a memória e a computação estarão muito mais distribuídas entre diferentes nodos sendo possível avaliar melhor o impacto no desempenho. No entanto, embora tenhamos 10 nodos 431, apenas temos 2 do tipo 652[4]. Por este motivo apenas foram usados 2 nodos iguais em cada teste já que se entende que deve ser usado o mesmo número de nós para os testes em cada tipo de nó distinto de forma a manter o máximo de parcialidade e 'justiça' nos resultados obtidos. Foram então realizados testes para 2 nós 652 comunicando entre si usando Gigabit Ehternet, 2 nós 652 comunicando entre si usando Myrinet com 10Gbps e o mesmo para o tipo de nodo 431.

Seguem-se gráficos de barras demonstrativos dos ganhos obtidos nesta versão, usando openmpi 1.6.2, em relação à versão sequencial.





Mais uma vez foi necessário dividir o gráfico em dois para facilitar a visualização da informação e, já que agora temos ainda mais informação a mostrar (testes com Ethernet e testes com Myrinet), optou-se pelo uso de gráficos de barras aos invés de gráficos de linhas.

Olhando ainda num panorama muito geral notamos logo que o uso de Myrinet para comunicação entre processos de diferentes máquinas permite a obtenção de maiores ganhos do que o uso de Ethernet. No entanto seria de esperar que o kernel EP obtivesse resultados muito parecidos com Ethernet e Myrinet, pois sendo um kernel embaraçosamente paralelo cada processo irá ser praticamente independente, logo a comunicação será extremamente reduzida. No entanto nota-se nos gráficos uma variação significativa entre EP com Ethernet e com Myrinet, a razão para isto é desconhecida, talvez o facto de terem sido feitas medições em momentos diferentes pode ter tido influência,

pois embora se tente ao máximo que todos os nodos estejam reservados somente para estes testes há sempre processos que correm em 'background' que em certos momentos poderão ser mais pesados que outros adulterando um pouco os resultados finais. O próprio hardware pode ter estado numa situação de maior calor aquando de algum teste o que pode ter impedido a frequência do clock de algum nodo de subir tão alto como poderia ter subido. As possibilidades são muitas, no entanto não é possível ter a certeza da razão pela qual o resultado para o kernel EP não foi o expectável.

Note-se agora que embora realmente fosse esperado que comunicação com recurso a Myrinet com 10Gbps obtesse melhor desempenho do que o uso de Gigabit Ethernet, este desempenho ficou aquem das expetativas. Isto porque a latência de comunicação por Myrinet será à volta de 10 microsegundos equanto que por Ethernet será à volta de 100 microsegundos o que significa que, teóricamente deveriamos ter um ganho de 10 vezes com Myrinet em relação a Ethernet[5]. No entanto uma possível explicação para isto não acontecer é que estes valores de latência encontrados em várias fontes distintas normalmente apenas têm em consideração a própria comunicação em si e não têm em consideração a latência que a própria aplicação em execução pode causar. Isto porque, mesmo que tenhamos um método de comunicação praticamente instantâneo a comunicação terá sempre que obedecer aos 'caprichos' da aplicação, se é necessário fazer algum cálculo importante este pode causar que a 'bandwith' da rede não esteja a ser ocupada sempre a cem por cento já que, por vezes, a comunicação acontece somente em certos pontos do programa e não constantemente. Por isso mesmo que tenhamos uma rede com muito mais latência o impacto no desempenho causado por essa latência diminui se o programa for mais intensivo a nível de computação independente e tiver menos dependências de dados. Podendo até certos cálculos serem realizados ao mesmo tempo que é efetuada a comunicação mascarando, em certas situações, a latência praticamente por completo. Esta afirmação comprova-se quando se repara que nos gráficos os ganhos obtidos com Myrinet nos kernels com pouca dependência de dados são mais próximos dos ganhos obtidos com Ethernet, embora que mesmo assim sejam ganhos maiores.

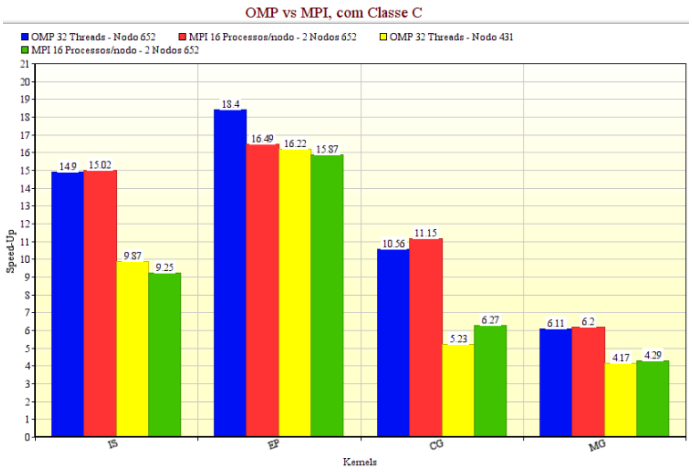
Por fim notamos mais uma vez que os ganhos obtidos com nodos 652 são significantemente maiores do que os ganhos obtidos com nodos 431. Apenas no kernel EP vemos o desempenho nos dois tipos de nodos ser mais semelhante do que com os outros kernels.

Quanto à escalabilidade a nível de processos nota-se mais uma vez que após a utilização de mais processos do que o número de cores físicos temos uma quebra no desempenho, o que seria de esperar pelas razões já mencionadas no capítulo anterior. Esta quebra mais uma vez surge mais cedo com nodos 431 já que, usando dois nodos de uma vez temos somente 24 cores físicos por isso, a não ser no kernel EP, temos uma quebra quando passamos de 16 processos por nodo para 32 por nodo. E até mesmo com 16 por nodo, já que cada nodo apenas tem 12 cores físicos estamos também já a recorrer a hyperthreading. Para os nodos 652 o impacto é menor pois passa a ter 40 cores físicos, mas mesmo assim sente-se o impacto ao usar 32 processos por nodo já que cada nodo tem somente 20 cores físicos. Notamos que tanto os nodos 652 como os nodos 431 obtêm resultados piores com 32 processos por nodo do que com 8 por nodo (excepto no kernel EP). Isto significa que é alcançado o pico da escalabilidade entre 16 processos por nodo e 32 por nodo. Por razões óbvias o ponto de escalabilidade dos nodos 652 é alcançado mais tarde do que os 431 já que alcançam os 32 processos por nodo com melhores ganhos do que os nodos 431. O que significa que os nodos 652 escalam melhor a nível de threads do que os nodos 431, sendo esta a mesma conclusão a que se chegou ao olhar os gráficos do caítulo anterior.

Em suma, mais uma vez os nodos 652 mostram vantagem em serem usados se for necessário um número elevado de processos. No entanto a não ser que o algoritmo seja embaraçosamente paralelo não se deve usar mais do que cerca de 20 processos por nodo que é o ponto de pico de speed up estimado. Ao contrário da versão anterior, não há vantagens no uso de nodos 431 para um pequeno número de processos. Em todos os casos o uso de Myrinet mostra-se vantajoso.

VI. Conclusões

Em jeito de conclusão olhamos agora para um gráfico que nos permite relacionar diretamente os melhores resultados obtidos com OpenMP e MPI:



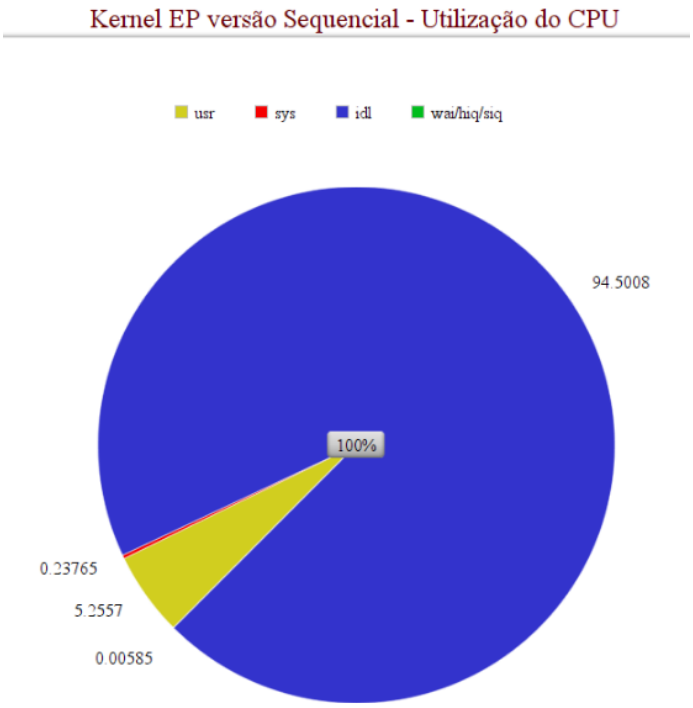
Com este gráfico vemos que para os kernels MG e IS a Versão MPI obtém praticamente os mesmos resultados do que a versão OMP. No entanto note-se que a versão OMP tem menos recursos à sua disposição. Isto porque estamos a comparar OMP com 32 threads OMP com MPI com 16 processos por nodo (e dois nodos). Desta forma percebemos que na versão MPI temos mais cores físicos do que na versão OMP, no entanto para estes dois kernels mencionados o resultado foi mesmo assim praticamente igual. Com isto conclui-se que os kernels MG e IS têm vantagens em ser aplicados com num só nodo com OMP já que assim poupam-se recursos. No entanto estima-se que se os recursos não forem um problema então qualquer uma das opções é viável.

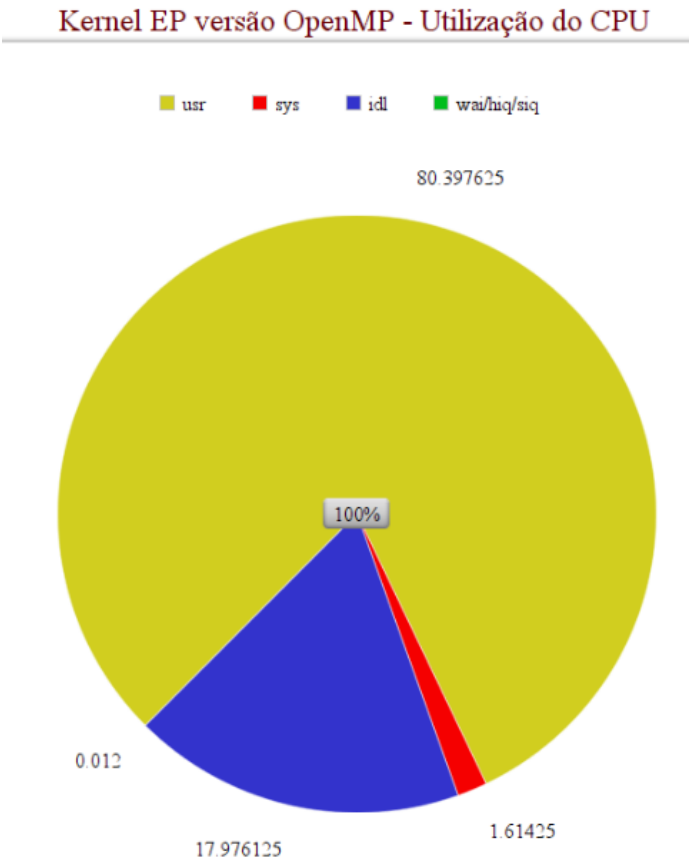
Para o kernel CG notamos que a versão MPI obtém melhores resultados do que a versão OMP. Isto é um indicador que o tipo de algoritmos no qual o kernel CG se insere beneficia do

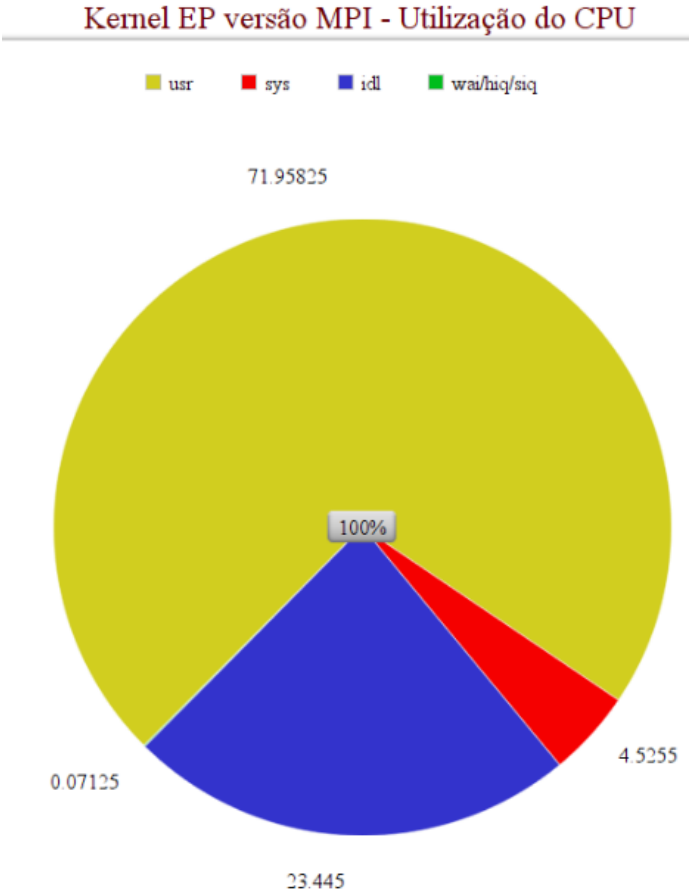
paradigma de memória distribuída mais do que o de memória partilhada. No entanto estes melhores resultados também podem ser fruto do maior número de cores físicos presentes ao usar MPI como referido anteriormente. Mesmo assim, o uso de MPI é vantajoso, pois com OMP não é possível obter mais ganhos significativos, pois podemos ver no gráfico do capítulo da versão OMP que com 32 threads o kernel CG encontra-se já a converger sendo que se adicionassemos mais threads veriamos os ganhos a alcançarem um pico máximo rapidamente. Como este pico máximo seria alcançado através de HyperThreading seria, em princípio, mais baixo do que o pico máximo do que com MPI, contando que o overhead da comunicação por Myrinet não mascare os ganhos do maior número de cores físicos, já que na versão com OMP a comunicação entre threads será bastante rápida pois será feita com recurso a memória partilhada.
Para o kernel EP a versão OMP é superior e tal já seria de esperar pelas razões já referidas nos capítulos anteriores. Seria então lógico escolher uma implementação em OMP para executar algoritmos embaraçosamente paralelos já que estes são os que mais beneficiam do paradigma de memória partilhada.
Por último voltamos a verificar que o nodo 652 é o que permite obter melhores resultados para todos os kernels quando são usados números elevados de threads. Como ambos os nodos 652 e 431 têm microarquitecturas desenvolvidas pela Intel e como 652 é uma microarquitectura mais recente concluímos também que a Intel está a seguir a filosofia de que ter um maior número de cores físicos um pouco mais fracos é melhor do que ter menos cores com uma maior frequência. Estes dois nodos descrevem esta situação na perfeição já que o nodo 431 é mais antigo, tem menor número de cores, mas uma frequência mais elevada, no entanto o nodo 652 tem maior número de cores e uma menor frequência. E realmente verificamos que esta filosofia adoptada no nodo 652 traz vantagens quando o nosso interesse é a execução de programas paralelizando com um grande número de threads.

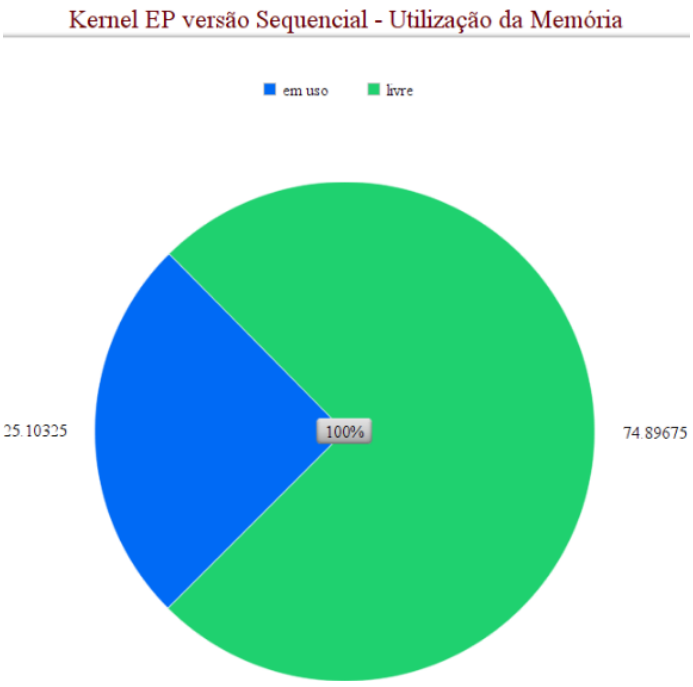
Tendo considerado o nodo 652 como o melhor na generalidade dos casos, já que mais vezes é a escolha mais acertada do que o outro nodo, achou-se relevante observar alguns testes que indiquem como se comportou ao serem efetuados os testes para a melhor medição obtida do kernel EP com o intuito de entender como os algoritmos a testar nesta máquina podem ser modificados de forma a tirar melhor partido da mesma. Os graficos que iremos agora observar foram obtidos com recurso à ferramenta **Dstat** como referido no capítulo 1.D:

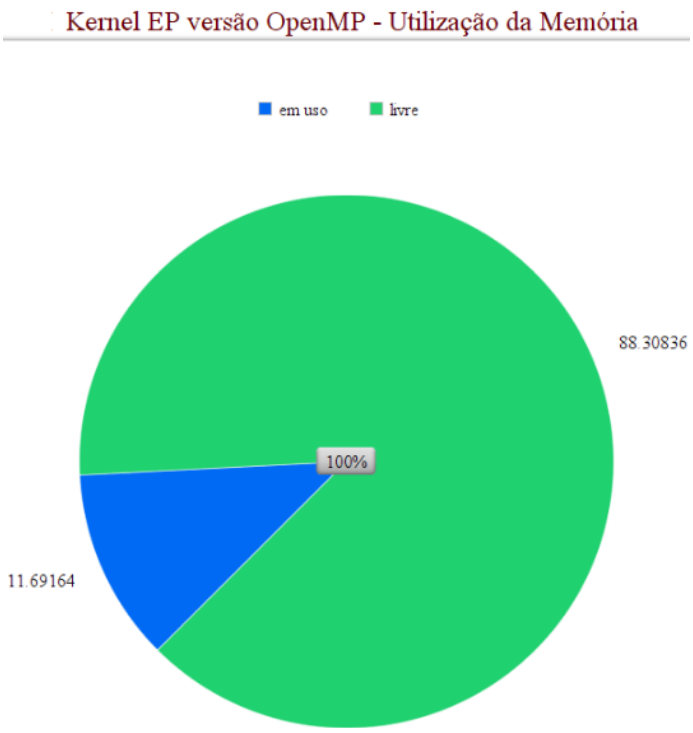
Utilização do CPU:





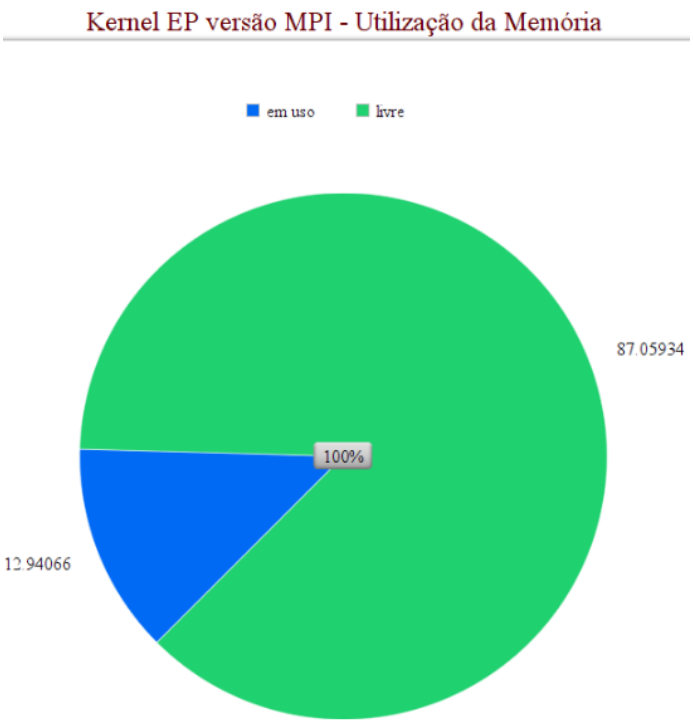


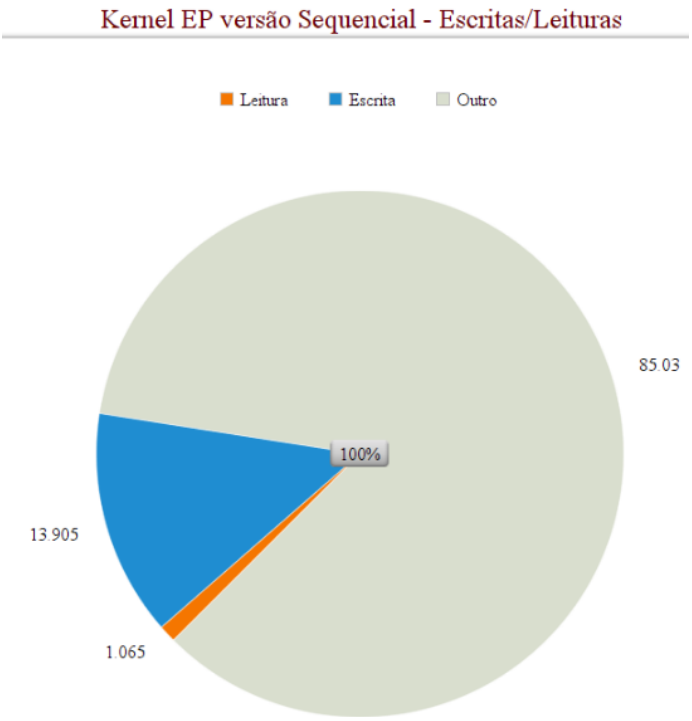




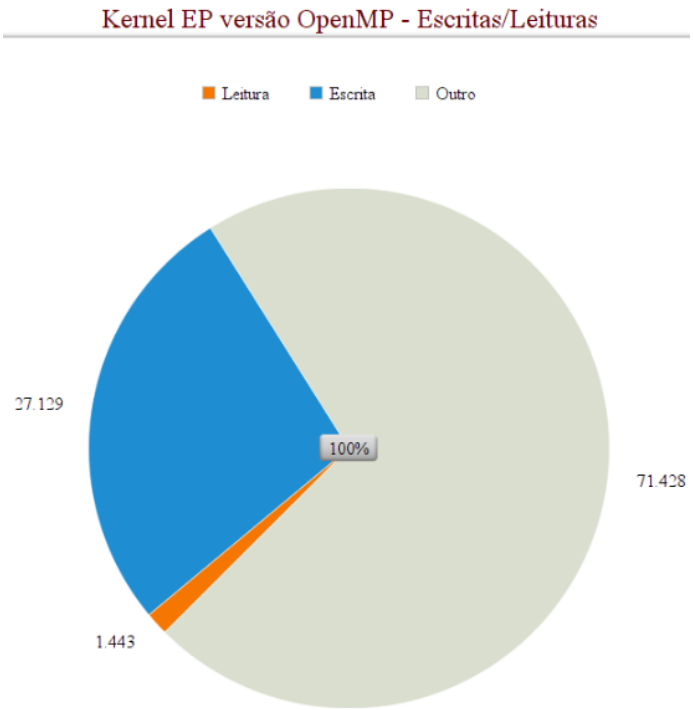
Podemos observar que na versão sequencial o CPU encontra-se na maior parte do tempo parado (percentagem representada pela ssigla **idl**). Isto deve-se à natureza embaraçosamente paralela do kernel EP e prova-se isto ao se notar nos outros dois gráficos das versões paralelas como agora o CPU é muito mais utilizado, principalmente a executar em user-mode (representado pela sigla **usr**). No entanto o gráfico relativo à versão MPI mostra uma menor ocupação do CPU, provavelmente permanece idl uma maior percentagem de tempo devido à latência de comunicação entre processos de máquina distintas que têm que esperar pela comunicação uns dos outros. Daqui concluimos algo que já era expectável, mas que fica agora provado. Nesta máquina com este kernel o fator limitante principal surge na versão MPI com a comunicação entre processos de máquinas distintas. De forma a melhorar as versões MPI de forma a igualar o OMP ou até superar, devemos focar-nos nos custos de comunicação de forma a reduzir a percentagem de tempo **idl** do CPU.

Utilização da Memória:

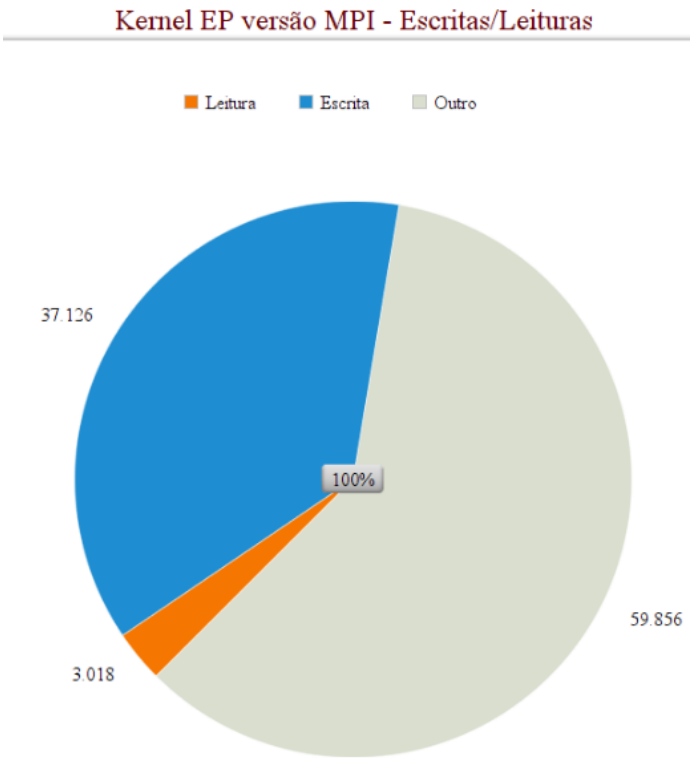




Estes gráficos indicam-nos a percentagem de memória em uso e livre ao longo do teste efectuado. Os resultados são o esperado já que os valores de memória livre da versão sequencial são um pouco maiores e os valores da versão OMP e MPI são bastante parecidos, pois ambos têm necessidades extra ligadas à memória, como por exemplo o acesso de parte de uma thread para atualizar valores escritos por outra na cache do seu próprio core de forma a manter a coerência de cache. Isto é algo que não acontece na versão sequencial e poderá ser uma limitação para as versões paralelas em casos extremos. Por esse motivo poderá ser um ponto a focar de forma a melhorar o desempenho das versões paralelas.



Balanceamento Escritas/Leituras:



Aqui vemos para este tipo de algoritmos (embaraçosamente paralelos) as leituras são mínimas comparadas com as escritas, sendo que as escritas têm muito maior impacto na versão sequencial. as versões paralelas resolvem um pouco a questão, no entanto a versão MPI tem ainda assim muito mais escritas do que a versão OMP. Por esta razão as escritas são um factor mais limitante do que as leituras e, se isso for uma preocupação, deve ser usada a versão OMP já que é a que sofre menos com as escritas. Se se tiver o objetivo de melhorar a versão MPI de forma a igualar ou a superar a versão OMP então as escritas poderá ser um ponto interessante a tentar optimizar.

VII. Trabalho Futuro

Tendo estudado pormenorizadamente nodos representativos da maior parte das máquinas pertencentes ao cluster utilizado, chegando à conclusão dos pontos fortes e fracos de cada uma e avaliando fatores limitantes e pontos a focar de forma a otimizar código para o nodo que se considerou o melhor na maioria dos casos, entende-se que o projeto terá sido concluído com sucesso. No entanto muita coisa ainda poderia ser feita, por exemplo, foi avaliado o impacto das diferentes classes para a versão sequencial, no entanto nas versões paralelas apenas se usou a maior classe. Se se testasse os ganhos para as restantes classes seria possível também compreender a escalabilidade ao nível dos dados, um assunto que foi pouco abordado neste relatório mas que é de todo relevante. O uso de mais do que um compilador para teste foi interessante, no entanto os dois compiladores usados são ambos do mesmo tipo (gcc) apenas variando na versão. Seria mais interessante talvez a comparação entre compiladores bastante distintos com o gcc e o icc. Noe ntanto não foi possível usar o icc pelas razões já indicadas no capítulo 2.
Estes extras revelam-se como algo de interessante a ponderar no futuro e como possíveis adições ao relatório final aquando da entrega do portefólio.

VIII. Bibliografia

[1] https://www.nas.nasa.gov/publications/npb.html
[2] https://en.wikipedia.org/wiki/NAS_Parallel_Benchmarks
[3] https://www.nas.nasa.gov/publications/npb_problem_sizes.html
[4] http://search6.di.uminho.pt/wordpress/?page_id=55
[5] https://pdfs.semanticscholar.org/3994/7b2fbbc5d25d5a827edf1c234099