Universidade Federal do Rio Grande do Sul
Instituto de Informática
Grupo de Processamento Paralelo e Distribuído

Proceedings

XIV Workshop de Processamento Paralelo e Distribuído

WSPPD 2016

2 de Setembro de 2016
Auditorio Centro de Eventos do Instituto de Informática
Porto Alegre, RS
ISSN: 2175-6848

List of Sessions

Performance Characterization of the Alya Fluid Dynamics Simulator

Guilherme Antonio Camelo, Lucas Mello Schnorr
Institute of Informatics – Federal University of Rio Grande do Sul
Caixa Postal 15.064 – CEP 91.501-970 – Porto Alegre – RS – Brazil

***Abstract*—This article presents preliminary results of a performance characterization of the *Alya* fluid dynamic simulator. Experiments are conducted in parallel using the MPI specification implemented by *OpenMPI*, and the application is traced using the *Extrae* tracing tool. By taking a closer look at the traces, it is possible to effectively see different performance patterns such as the communications among ranks, and the effective application load imbalance.**

I. Introduction

Numerical simulation is used to understand and model natural behaviors. Very often these simulations are carried out by techniques based on fluid dynamics, where a model of a real world scenario is implemented and solved using iterative numerical methods using time-steps. High performance computers are employed as execution platforms to run these models, as any sequential approach would make certain simulations executions impractical for the level of details required by the scientific community.

Distributed and parallel programming techniques provide high computational power. Interfaces such as OpenMPI enable portable implementations for the execution of complex programs in less time, and are ideal for dealing with physical problems for real world simulations. Alya[1] is an example of MPI-based fluid dynamic framework simulator. It is an open source project, part of the Prace Benchmark[2], from the Barcelona Supercomputing Center (BSC) to solve numerically physical problems.

Very often numerical simulation applications have irregular loads during execution. Such irregularity appears for many reasons, such as irregular control and data structures, adaptative mesh refinement (AMR)[3], or even irregular iteraction patterns among processes. These reasons ultimately lead to a load imbalance among both resources and time as the simulation advances. Finding out the actual application behavior on a particular platform is key to apply optimization techniques, such as better balancing algorithms or a more appropriate communication patterns. The most efficient way to obtain such information, when you do not know the application code is to apply tracing techniques. They use files to keep track of information regarding the application, which is saved under the form of events, for instance, to track MPI operations.

In this article, the Alya fluid dynamic simulation tool is executed on a 4-node platform with 64 cores, using the OpenMPI library[4]. The goal is to investigate whether Alya has a irregular execution behavior regarding the resources and the time for a representative input. We employ the Extrae tracing tool to record all MPI communication operations, and then present preliminary results of the performance analysis of Alya. A similar study has already been conducted[5] in a larger scale platform. Our secondary goal is to understand the behavior in a smaller scale platform.

The paper is structured as follows. Section II presents the platform used for the experiment. The methodology used in the performance characterization is detailed in section III. Results are presented in section IV. The main contributions and the future work are detailed in section V. A reproducible and open scientific research is essential [6]. This work is therefore the result of a effort to create a reproducible project, publicly available at http://github.com/guiacamelo/alya/.

II. Execution Environment

Experiments are conducted on computers that are part of the Draco cluster at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS). The cluster is formed by eight nodes, each one equipped with Intel (R) Xeon (R) E5-2640 v2 CPU @ 2.00GHz processors with 16 physical cores (32 with hyperthreading), 64 gigabytes of RAM, running Ubuntu 14.04.4 LTS. Only physical cores were used in the executions. In the experimentations different configurations have been tested, with various numbers of cores, number of nodes and of time-steps. Extrae version 3.3.0 is the tracing tool, and for the parallel execution, OpenMPI 1.10.2.

III. Methodology

We have used only four nodes of the Draco cluster for a total of 64 cores. The simulation ran only until the end of the third timestep due to the large size of the trace files. The trace files for this execution have size of 2,7 gigabytes. The tool *Extrae* creates individual traces for each processor, keeping information regarding the communication and execution. Multiple files are created in *.mpits* format containing information about what was executed by a given process. It is then necessary to convert and merge them using the tool *mpi2prv* that outputs the Paraver format (*.prv*). After that, a *perl* script is used to filter the relevant data creating a *Comma-Separated Values* (CSV) file used for the analysis scripts.

The resulting CSV output have four columns of data: the MPI rank (Rank), the time in which the process enter the state (Start), the time in which the process finished the state (End), and the MPI state name (State). With this information

it is possible to calculate the actual workload (running time in seconds) of each process, as well as create space/time graphics that tells us the order and what was the time that each state occurred in each case. All calculations are made from scripts written in the R language, making use of *ggplot2* and *dplyr* libraries. Figure 1 gives a summary of the steps to trace Alya and to conduct the performance analysis of the experiment.

In the schematics on figure 1 the first and second blocks represent the execution with *MPI* and trace with *Extrae* in 4 nodes each using 16 cores. The next two blocks in the flow are the output of the simulation results and the trace files in *.mpits* format. The following two blocks represent the conversion of the trace to *.prv* and then to *.CSV* using *mpi2prv* and a *Perl* script, respectively. Then a block representing the *CSV* format is presented followed by the last block that represents the creation of graphics and analysis of the experiment using language *R*.

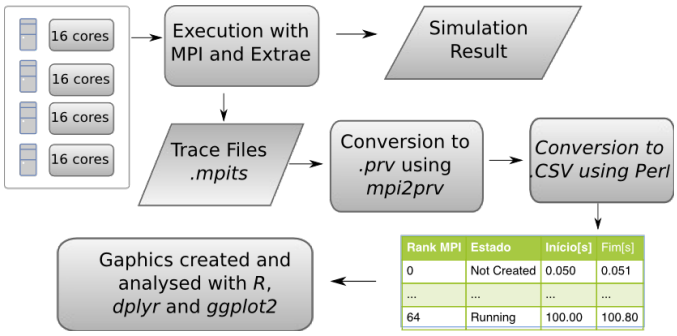


Fig. 1. Methodology used on the execution and analysis of the experiment.

IV. Preliminary Results

The goal of our performance analysis is to identify relevant characteristics of the application and evaluate whether it presents load imbalance among resources and along time. We also intend to refine our methodology to conduct larger scale experiments. We provide as follows an overview about the load imbalance, a detailed analysis using a traditional space/time view, and an attempt to explain the identified behavior using per-rank state statistics. We end the analysis by globally applying the percent imbalance metric to evaluate the load imbalance. All data presented in the graphics are acquired from one execution using 4 node and 64 cores.

A. Overview of Load Imbalance

Figure 2 shows the aggregate time of effective computation for each process. Computation is calculated by summing all time periods in which the process performs data processing. All periods of time on MPI communication, point-to-point synchronization and collective synchronization are not included in these measures. It is observed that most processes have a total computing time in the order of 150 seconds. In some cases, however, the times reaches up to 300 seconds, for instance, the process 60. This is an indicative that a load imbalance occurs considering this specific case study.

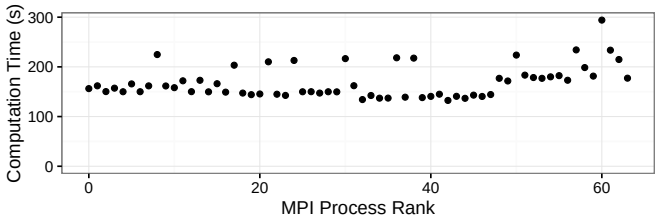


Fig. 2. Aggregated time of effective computation (Y axis) per rank (X).

B. Space/Time Execution

Figure 3 presents detailed information about the load balancing, showing the time that each process was in a state of effective computation. At the beginning of execution, process zero engages in distributing work among processes. This activity continues until the 150 seconds mark. After this time mark, remaining processes begin to work, and, the root process (zero) assumes the organizer role, becoming idle while waiting for the response of other processes.

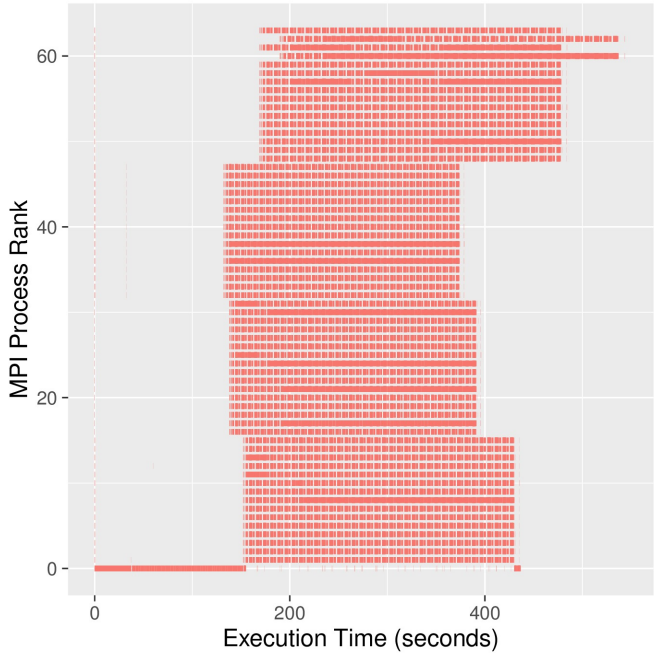


Fig. 3. Timeline (on X axis) showing computation states per process (on Y).

An interesting feature that can also be observed in Figure 3 is the existence of four groups of processes: the 0-15 of 16-31, and so on. This behavior correlates with the number of nodes used in the experiment, indicating that internally in a node all processes begin their computation roughly at the same time. This indicates that the initial load distribution from process zero is not scalable because visually the computations in each one of the four machines start sequentially: first the group between the processes 32-47, After the group of processes 16-31, then the group containing the zero process (0-15) and finally the group of processes 48-63. In the latter group, we

also observe an anomaly in the processes 60 and 62. They start and finish their computation after the other process from their group. Such anomaly is observed only within this group.

The process 60 has a peculiar behavior when compared to the other processes. Figure 2 features the time of effective computation: it runs for roughly 300 seconds, twice as many other processes. Figure 3 shows that process 60 has an anomalous behavior, beginning and ending its execution last. Moreover, the execution state (*Running*), where the computation is actually performed, does not seem to contain spaces as other states. This probably indicates that the time spent on communication functions for this particular case is much lower than in other processes.

C. Process Behavior by State

Figure 4 displays a summary of dedicated time (Y axis) by process (X axis) to each state (different facets). Only the five most important states are presented (*Blocking Send*, *Group Communication*, *Running*, *Send Receive* and *Waiting a message*). The other states present very little or inexistent time. The *Blocking Send* have less influence in load balance than the other states since they are somewhat similar in all processes (except for the 48-63 group, slightly higher). The *Group Communication* have quite different times among processes and a very long time to process zero, as detailed in previous sections. Load imbalance is shown in the facet *Running*, similar to the data presented in Figure 2. The sending and receiving times are relatively homogeneous between processes, except in some cases in which they are smaller. Therefore, it is possible to see in the rightmost facet of the graphic that the possible reason of the anomaly of the processes 60 and 62 is due to additional time spent in the state *Waiting a message*. Such fact contradicts our previous hypothesis drafted in previous section, where the timeline visually indicated a higher less communication time for process 60. This is probably due to drawing much more events than the screen space available [7] to draw them.

D. Percent Imbalance

According to Pearce [8], and then validated by Alles [9], the percent imbalance formula is used to formally calculate the load balance. It is described by the Equation 1 where λ is the load imbalance value to be calculate, L_{max} is the value of the process with the highest load, and $\overline{L}$ is the average computational load among processes. The metric may be calculated either for the entire execution or for different phases e.g. one measure for each time interval or timestep.

$$\lambda = (\frac{L_{max}}{\overline{L}} - 1) * 100 \quad (1)$$

In this work the calculation of the metric is performed considering the entire execution, thus being a global indicator of load imbalance. The metric characterizes the uneven distribution of work, and when applied to our measurements gives the value $\lambda = 74.25161$. That represents a workload imbalance of 75%. This indicates that if the load is more evenly distributed between the computational resources there would be room for a potential performance improvement.

Fig. 4. Time spend on each state by process.

V. Conclusion and Future Work

This paper presents preliminary results of the performance analysis of the Alya fluid dynamic simulation tool. Our goal is to better understand its behavior specially regarding the load balance. Therefore, a simulation is carried out on a platform of four computer nodes totaling 64 cores, leading to a execution of approximately 450 seconds. The execution is traced using the *Extrae*, enabling us to discover relevant information about the core operation of *Alya* in the addressed case. Among the results, we observe that a significant share of the time (about 34% in a run with three timesteps) is somewhat wasted in the beginning of the execution probably to divide the problem, creating considerable overhead since remanining processes are kept idle. Others results include the detection of anomalies in some processes and the perception that the root process (zero) sends the partitioned data sequentially to different nodes, making the start of application considerably slow and not scalable. The calculation of the percent imbalance indicates that there is a potential performance improvement. The reason for such imbalance is still being investigated.

As future work, we will study possible changes in the code that may provide a balanced distribution in a more egalitarian fashion. We intent to execute similar experiments with other configurations, varying the number of nodes, number of cores, and number of timesteps in order to confirm the behavior acquired in this experiment. We also hope to manually mark the beginning of each iteration of the algorithm (changing the application code) so that the balancing metrics may be calculated for each phase of computation/communication. This refinement will allow us to check the load balance along time.

VI. Acknowledgments

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law nº 8248, 1991, and its subsequent updates.

This investigation also receives funds from the H2020 program EU and MCTI / RNP-Brazil through HPC4E project with code 689772, the FAPERGS / Inria ExaSE design, universal design CNPq 447311 / 2014-0, and international CNRS / LICIA laboratory. We also thank Flavio Alles for the discussions regarding load balancing metrics.

References

[1] M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Aris, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, and J. M. Cela, "Alya: Towards exascale for engineering simulation codes," *arXiv.org*, 2014. [Online]. Available: http://arxiv.org/abs/1404.4881

[2] Prace, "Prace Research Infrastructure unified european applications benchmark suite - prace research infrastructure," http://www.prace-ri.eu/ueabs/, 2013, publicado: 2013-10-17, Acessado: 2016-07-15.

[3] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104.

[5] J. Rodríguez, "Performance analysis of alya on a tier-0 machine using extrae," Prace White Papers, Tech. Rep. 151, 2014.

[6] A. Legrand, "Scientific methodology and performance evaluation," https://github.com/alegrand/SMPE, 2015.

[7] L. M. Schnorr and A. Legrand, "Visualizing more performance data than what fits on your screen," in *Tools for High Performance Computing 2012*. Springer, 2013, pp. 149–162.

[8] O. Pearce, T. Gamblin, B. R. De Supinski, M. Schulz, and N. M. Amato, "Quantifying the effectiveness of load balance algorithms," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 185–194.

[9] F. A. Rodrigues, "Study of Load Distribution Measures for High-performance Applications," Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brasil, 2016.

Análise de Desempenho da Multiplicação de Matrizes por Strassen contra o Método Tradicional

Arthur Mittmann Krause, Gabriel Bronzatti Moro, Lucas Mello Schnorr
Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

***Resumo*—There are several strategies to do matrix multiplication using high-performance computers, each one with specific optimizations. This paper makes a comparison of two among them: the well known Strassen algorithm and the Conventional algorithm for matrix multiplication. We also investigate different levels of optimization to improve cache memory utilization. The results show that the best approach to be used is Strassen. From the cache optimizations point of view, the one that contributes more to the performance is the the algorithm that transposes the second input matrix.**

I. Introdução

A multiplicação de matrizes é uma operação da Álgebra Linear fundamental para a solução de problemas de uma ampla variedade de áreas do conhecimento. Por conta disso, o estudo de técnicas para uma execução mais eficiente da mesma é de grande importância para a ciência.

Diversos trabalhos estudam o algoritmo paralelizado utilizando OpenMP, tais como Andrade [1], que compara diferentes escalonadores OpenMP e uma versão implementada em Pthreads. Outro trabalho recente que também aborda estudo de otimizações em algoritmos de multiplicação de matrizes é Silva [2], que avalia o desempenho dos algoritmos utilizando diversos níveis de otimização fornecidos pelo compilador GCC. Porém, comparações entre os métodos de multiplicação de matrizes utilizando o método de Strassen [3] e o convencional levando em conta diferentes otimizações no uso da memória Cache não é um assunto muito explorado.

O objetivo principal deste trabalho é portanto apresentar uma análise de desempenho para dois algoritmos de multiplicação de matrizes, o tradicional e o algoritmo de Strassen, com as otimizações de Blocking, transposição e vetorização. Essas otimizações foram propostas para melhorar o uso da memória Cache dos processadores.

A Seção II apresenta os métodos Strassen e Tradicional para a multiplicação de matrizes, detalhando o funcionamento de cada algoritmo. A Seção III apresenta as abordagens de otimização no uso da memória Cache. A Seção IV detalha a metodologia utilizada no trabalho. Uma discussão dos resultados obtidos é realizada na Seção V. Em seguida, são apresentadas as considerações finais.

II. Métodos de Multiplicação de Matrizes

A. Algoritmo Tradicional

O algoritmo mais utilizado para a operação de multiplicação de matrizes possui três laços aninhados, com uma complexidade sequencial cúbica. O primeiro laço é responsável por iterar a matriz linha a linha, o segundo coluna a coluna e o mais interno permite o percorrimento da matriz (linha a coluna). O Algoritmo 1 apresenta a abordagem mais comum para se multiplicar duas matrizes. O exemplo apresenta a multiplicação de duas matrizes de entrada, as matrizes A e B. O resultado das operações de multiplicação do laço mais interno são somadas ao seu respectivo elemento da matriz R. Cada elemento $R_{i,j}$ é obtido através do somatório dos produtos dos elementos correspondentes de posição k na i-ésima linha de A pelos também de posição k na j-ésima coluna de B. Além disso, é possível visualizar que os laços definem n^3 operações [3].

```
Data: Matrizes de entrada: A e B. Matriz resultado: R.
for i a N do
    for j a N do
        for k a N do
            R[i][j] =+ A[i][k] * B[k][j]
        end
    end
end
```

Algorithm 1: Algoritmo Tradicional para Multiplicação de Matrizes.

B. Algoritmo de Strassen

O algoritmo de Strassen utiliza uma abordagem de divisão e conquista para realizar a multiplicação de matrizes. Diferente do algoritmo tradicional, este algoritmo realiza menos operações para obter o produto de duas matrizes, o que lhe concede uma complexidade de O($n^{2,8}$), menor do que a do algoritmo Tradicional, que possui a complexidade de O(n^3) [3].

Para implementar o algoritmo, as matrizes devem ser divididas em quatro submatrizes de mesmo tamanho, e sete novas matrizes temporárias P_n são calculadas a partir das equações representadas no Bloco de Equações 1, supondo uma multiplicação de matrizes de entrada A e B resultando numa matriz R.

$$
\begin{aligned}
P_1 &= A_{[i][j+1]} + A_{[i+1][j+1]}) * (B_{[i+1][j]} + B_{[i][j+1]}) \\
P_2 &= (A_{[j+1][i]} + A_{[i+1][j+1]}) * B_{[i][j]} \\
P_3 &= A_{[i][j]} * (B_{[i][j+1]} - B_{[i+1][j+1]}) \\
P_4 &= A_{[i+1][j+1]} * (B_{[i+1][j]} - B_{[i][j]}) \\
P_5 &= (A_{[i][j]} - A_{[i][j+1]}) * B_{[i+1][j+1]} \\
P_6 &= (A_{[i+1][j]} - A_{[i+1][j+1]}) * \\
&\quad (B_{[i][j]} + B_{[i][j+1]}) \\
P_7 &= (A_{[i][j+1]} - A_{[i+1][j+1]}) * \\
&\quad (B_{[i+1][j]} + B_{[i+1][j+1]})
\end{aligned}
\tag{1}
$$

Essas matrizes temporárias são então somadas como representado no Bloco de Equações 1 para obter o valor das submatrizes da matriz R.

$$
\begin{aligned}
R_{[i][j]} &= P_1 + P_4 - P_5 + P_7 \\
R_{[i][j+b]} &= P_3 + P_5 \\
R_{[i+b][j]} &= P_2 + P_4 \\
R_{[i+b][j+b]} &= P_1 + P_3 - P_2 + P_6
\end{aligned}
\tag{2}
$$

Essas divisões são efetuadas recursivamente até que as submatrizes sejam constutuídas de apenas um elemento.

A implementação do Algoritmo 2 contém três laços, similar ao utilizado no Algoritmo 1. No laço mais interno, no entanto, são efetuadas sete multiplicações 1 em vez de oito como no algoritmo Tradicional. É nesse fator que o algoritmo de Strassen se diferencia e obtém uma complexidade menor que implica numa melhor performance.

```
Data: Duas matrizes de entrada e uma matriz de
      resultado.
for i a N/2 do
    for j a N/2 do
        for k a N/2 do
            Calcular o produto dos blocos.
        end
        Somar os resultados adquiridos no laço k e
         redirecionar para a matriz resultado.
    end
end
```

Algorithm 2: Algoritmo de Strassen.

III. Otimização no Uso da Cache

Dentre as características essenciais a serem levadas em consideração quando se desenvolve um algoritmo para ser executado em uma máquina específica, o uso da memória Cache é fundamental quando se quer obter o máximo de desempenho em arquiteturas multiprocessadas. Kowarschik and Weib [4] apresentam várias técnicas que podem ser utilizadas para se otimizar o uso da memória Cache, dentre essas podemos destacar: Loop Interchange, Loop Fusion e Loop Blocking.

Loop Interchange. Esta técnica consiste basicamente em alterar a ordem de dois laços de interação aninhados. Uma situação muito comum ocorre quando existem dois laços de iteração, o mais externo tem um número de iterações menor que o mais interno [4]. Ambos podem ser trocados, dessa maneira a técnica seria implementada, o laço mais externo com mais interações permite utilizar melhor as posições da mesma linha da Cache. Uma otimização utilizada nesse trabalho foi realizar a transposta da segunda matriz da multiplicação, a fim de obter os mesmos benefícios da técnica Loop Interchange, fazendo com que para ambas matrizes sejam utilizados os mesmos índices de linha e coluna para o cálculo, dessa forma os dados são buscados mais rapidamente na memória Cache (menor caminho de busca na Cache).

Loop Fusion. A fusão de laços pode ser utilizada em situações em que dois laços de iteração possuem o mesmo número de iteração e nenhuma dependência de dados entre si [4]. Esses laços podem ser transformados em um único apenas, definindo as operações dos dois laços anteriores. O benefício adquirido com o uso dessa técnica é a diminuição considerável de acessos à memória Cache e *overhead* de branches, dois laços se transformam em um.

Loop Blocking. Enfim, esta técnica também conhecida por Loop Tiling consiste na organização do acesso aos dados da memória Cache por blocos. A cada iteração são inicializados laços que trabalham especificamente em blocos de iterações anteriores. A seguir pode ser visualizado um exemplo dessa abordagem.

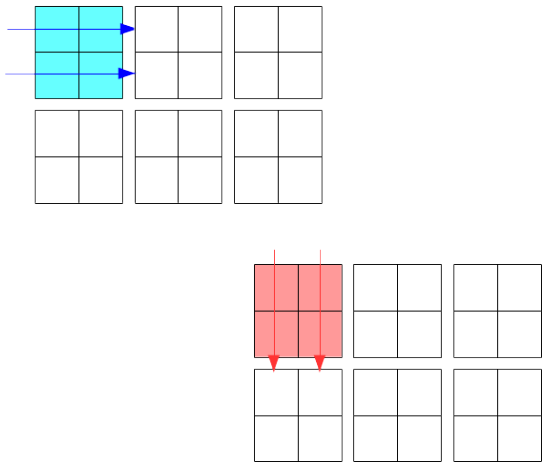


Figura 1. Técnica Loop Blocking ou Loop Tiling.

A Figura 1 demonstra a abordagem de multiplicação de matrizes convencional que implementa a técnica de Loop Blocking. A partir de cada iteração, a matriz é percorrida em blocos, os quais devem possuir o mesmo tamanho, permitindo o melhor uso da Cache, pois os dados acessados estarão próximos (dentro do limite do bloco). Para esse algoritmo é necessário utilizar além dos laços que percorrem os elementos da matriz, laços que disponibilizam esses blocos.

IV. Metodologia de Avaliação Experimental

A metodologia utilizada nesse trabalho consiste na definição de um Projeto de Experimento (*Design of Experiments*) [5].

O experimento foi realizado com matrizes quadradas, com dimensões de 256, 1024 e 2048, variando o número de threads em 1, 16, 32, 48 e 64. Cada configuração experimental foi replicada 15 vezes, de forma que podemos entender a variabilidade experimental, conduzida de maneira aleatória para que os resultados sejam mais confiáveis e que instabilidades no sistema não prejudiquem apenas uma configuração experimental.

Os algoritmos utilizados no experimento foram o algoritmo de Strassen e o Tradicional, ambos definidos anteriormente. Para cada algoritmo foram realizadas as seguintes otimizações: vetorizado, não vetorizado, com transposta e com Blocking. Na abordagem vetorizado, a matriz é alocada na memória como um grande vetor, essa técnica permite obter os mesmos benefícios da técnica Loop Interchange, visto que diminuirá o número de saltos para percorrer os dados na Cache, eles estarão armazenados propositalmente para beneficiar o acesso por linhas da Cache. Já na abordagem não vetorizada, a alocação é não contínua. A transposta consiste na transposição da segunda matriz do cálculo antes da multiplicação, já na abordagem Blocking a matriz é disponibilizada em blocos de tamanhos iguais.

O experimento foi executado na turing, uma máquina do Instituto de Informática que possui quatro processadores Intel(R) Xeon(R) CPU X7550 2.00GHz. Cada processador possui oito cores físicos, os quais possuem duas threads por core. Cada core conta com 32 KB de Cache de dados nível um e 256 KB de nível dois. O processador possui uma Cache L3 de 18 MB.

V. Resultados

Na Figura 2 é possível visualizar o gráfico de tempo de execução para todas as versões paralelas implementadas para os algoritmos de Strassen e o Tradicional. O eixo horizontal apresenta o número de threads utilizadas, já o eixo vertical apresenta a média do tempo de execução das 15 execuções aleatórias efetuadas para cada configuração.

O algoritmo de Strassen foi, como esperado, mais rápido que o normal em todas as combinações de fatores devido a sua menor complexidade. No entanto, utilizando a otimização de Blocking, o algoritmo de Strassen sequencial executou numa média de 114,3 segundos enquanto o normal sequencial apenas 88,5 segundos. Essa característica pode estar associada a granularidade do bloco, já que no método Strassen a matriz é dividida em blocos, somado a essa abordagem, o uso da técnica Blocking nesse cenário deixará a matriz mais subdividida, o que pode impactar negativamente no desempenho do algoritmo como um todo.

O melhor caso ocorreu quando ambos algoritmos fazem o uso da técnica Transpose. Para o algoritmo de Strassen, o melhor tempo com transposta foi 0,31s com 64 threads. Já para o algoritmo convencional, o melhor caso foi com a mesma combinação de fatores, levando um tempo médio de 0.52s para executar.

Na Figura 3 é possível visualizar o gráfico de Speedup para os algoritmos de Strassen e convencional, quando os mesmos

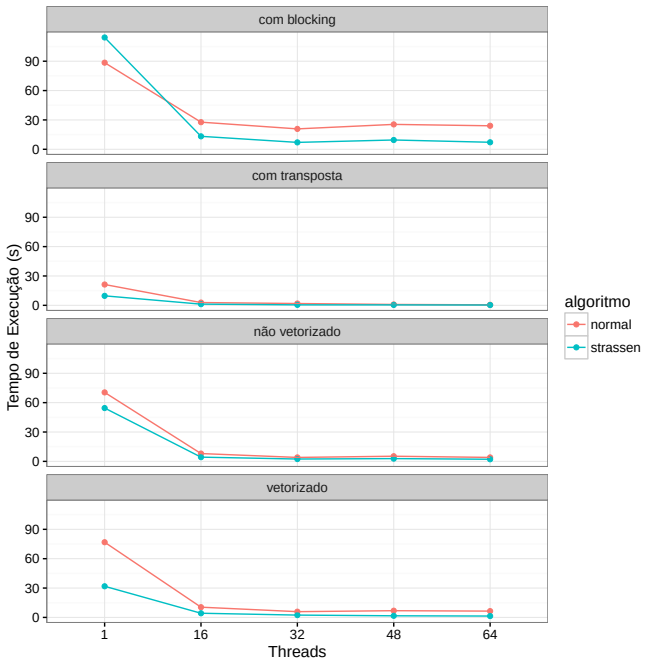


Figura 2. Tempo de Execução para diferentes versões paralelas para Multiplicação de Matrizes com dimensão de 2048.

utilizam a técnica da segunda matriz transposta (melhor caso para ambos). A partir do gráfico de Speedup é possível visualizar que o melhor Speedup é obtido com a versão Strassen. O melhor Speedup é obtido para ambos algoritmos com 64 threads, isso está relacionado a plataforma onde os experimentos foram executados (32 cores físicos e 64 lógicos).

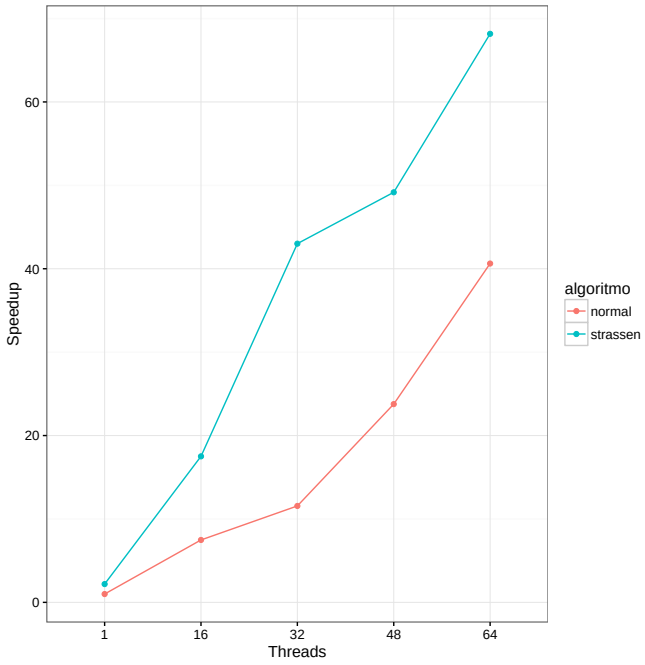


Figura 3. Speedup das duas melhores versões paralelas dos algoritmos de Strassen e Convencional para matrizes de dimensão 2048.

Uma observação interessante é o Speedup de ambos os algoritmos, utilizando a técnica transposta com 32 threads. Enquanto o algoritmo Tradicional demonstrou um Speedup praticamente linear, o de Strassen obteve um grande ganho com 32 threads. Sua medida de Eficiência, tendo como base o tempo médio de execução do algoritmo Tradicional com transposta sequencial foi de 1,34 enquanto com 16 threads foi 1,09, 48 threads 1,02 e 64 threads 1,07. Essa medida se deve à arquitetura da máquina de testes, que possui 32 cores físicos e à característica do algoritmo, que já conta com um alto nível de paralelismo em nível de instrução, portanto o *Hyper-Threading* não é eficiente nesse caso e o *overhead* da gerência das threads extras causa um impacto muito mais significante no resultado.

VI. Considerações Finais

O principal objetivo desse artigo é comparar os algoritmos de Strassen e o convencional para a multiplicação de matrizes com diferentes otimizações de Cache.

Os resultados apresentaram que a melhor configuração dos algoritmos testados foi Strassen com transposta, o qual possui a maior aceleração diante das demais versões.

A partir desse trabalho foi possível conhecer o impacto das diferentes otimizações para os algoritmos de multiplicação de matrizes testados. Foi possível visualizar que dependendo do tipo de aplicação a otimização pode ser insuficiente, um exemplo disso foi para o algoritmo de Strassen quando executado com Blocking.

Como trabalho futuro pretendemos rastrear a taxa de misses das diferentes versões dos algoritmos com as possíveis otimizações de Cache, a fim de compreender o momento da aplicação que a otimização vale a pena.

Agradecimentos

Esta investigação recebe fundos do programa H2020 da União Européia e do MCTI/RNP-Brasil através do projeto HPC4E com o código 689772, do projeto FAPERGS/Inria ExaSE, do projeto universal do CNPq 447311/2014-0, e do laboratório internacional CNRS/LICIA.

Referências

[1] G. L. Andrade and M. C. Cera, "Paralelização de uma multiplicação de matrizes utilizando openmp," *Anais do Salão Internacional de Ensino, Pesquisa e Extensão*, vol. 7, no. 2, 2016.

[2] S. A. da Silva and C. Schepke, "Análise de desempenho de aplicações paralelas em arquiteturas multi-core e many-core," *Anais da XVI Escola Regional de Alto Desempenho*, 2016.

[3] T. Cormen, *Introduction to Algorithms*. MIT Press, 2009.

[4] M. Kowarschik and C. Weiß, "An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms," *Algorithms for Memory Hierarchies*, vol. 2625, pp. 213–232, 2003.

[5] R. Jain, *Art of Computer Systems Performance Analysis: Techniques For Experimental Design Measurements Simulation and Modeling*. Wiley, 1991.

Frequency-based Overhead Compensation in HPC Application Traces

Alef Farah*,Lucas Mello Schnorr*†,Jean-Marc Vincent†
*Institute of Informatics, Federal University of Rio Grande do Sul – UFRGS
†Univ. Grenoble-Alpes, France

***Abstract*—Application tracing is widely used for performance analysis of parallel applications. Tracing perturbs the measured system, mainly because of the execution of additional instructions. The perturbations may be very small by themselves, but they accumulate along the application execution, altering its behavior, which is undesirable for performance analysis. This phenomenon is known as the probe effect, and can be reduced by registering less information, although the trade-off is usually undesirable. Alternatively, one can compensate for it by reducing an estimate of the registering overhead from the event timestamps. In this work we characterize the overhead caused by a tracing tool for MPI applications and compensate for it using a novel approach in which the event registering frequency is taken into consideration. Early results comparing the total execution time with that of the uninstrumented application show an encouraging improvement over traditional compensation methods, which do not consider event frequency.**

I. Introduction

Tracing comprises the register of significant events from an application execution, such as subroutine calls and change in variable values. The recorded data is used for future analysis of the application behavior, in a post-mortem fashion [1]. Unlike profiling techniques [1], every event selected for recording is registered individually, uniquely identified by type, timestamp and additional data according to its type. Tracing is generally used for performance analysis of parallel and distributed applications, in which the chronological order of events is important for identifying multiple performance bottlenecks [1].

Event registering can be done either via software or hardware, with passive counters [2]. Software-based tracing is much more common because of its flexibility. The simplest method is source code instrumentation, in which logging routines are inserted into the code, either manually [3] or automatically [4]. Perturbation caused by these methods can be both direct, by adding overhead to the point at which they are added, and indirect, by allowing or inhibiting compiler optimizations. These perturbations accumulate over the application run, which might register millions of events, so even the quickest routines may add significant overhead to the final result. This can lead to traces that unfaithfully represent the application run harming the performance analysis.

One way to avoid indirect perturbations is to instrument compiled code [5]. However, this can still affect hardware optimizations such as out of order execution and caching. Direct perturbations, on the other hand, cannot be avoided without the usage of a non-intrusive measurement technique such as passive hardware counters. One can, alternatively, minimize them. Either by reducing the amount of information recorded or the set of events to be logged, either way trading low intrusion for less information, which is not desirable. Another approach is to compensate for direct perturbations by adjusting the event timestamps to consider the overhead added by the logging routines. This doesn't require less information to be registered, and is the focus of this work.

In order to compensate, the overhead must first be measured. The usual approach is to isolate the logging routine and take enough measurements of its execution time to obtain a statistically significant expected value [6], [7], [8]. Obtaining this value, which is obviously architecture dependent, is not as trivial as it might seem. Some tools [9], [4], [10] try to minimize the logging overhead by using very fast routines, leading to high volatility in their execution times. Due to their small size, any interference, such as task scheduling by the operating system, induce high relative error. Attention to this error is of utmost importance: an incorrect measurement for the overhead might lead to over or undercompensation, possibly shifting the compensated trace even further away from the uninstrumented (real) execution it is trying to approximate.

We observed that the logging routine execution time is a function of the frequency with which it is called, i.e. of the trace event registering frequency. Previous authors [6], [7], [5], [11], [12] did not take this into consideration, which might lead to a bad approximation of the uninstrumented run. We employ two metrics to compare our method, which considers event frequency, with traditional ones which do not. We use the distance from the uninstrumented application execution time and a space/time comparison. Early results comparing total execution times show improvement from traditional methods, though finer grain comparisons are still inconclusive.

The rest of this paper is organized as follows. Section II presents state of the art. Section III outlines the perturbation and compensation models we are proposing. Results and discussion are presented in Section IV. Conclusions and future work are shown in Section V.

II. Related Work

Wolf et. al. [8] define a model for compensating event timestamps in message passing applications. They estimate the logging routine execution time by doing repetitive calls to it, measuring the execution times and calculating the average. De Kergommeaux et. al. [12] re-introduce the same models with slightly different – but semantically equivalent –

formulas. They also provide a methodology to deal with clock synchronization. More relevant to our work, they discuss the impossibility of compensating non-deterministic applications (see Section III). Kranzlmuller et. al. [11] use SKaMPI, a tool for benchmarking MPI implementations, to measure monitoring overhead of tracing applications that use PMPI (a library interposing interface for tracing MPI programs). The usage of SKaMPI standardizes the measurement process. It is one of the first tools to consider the standard error when benchmarking, and is thus variability-aware.

All previously described related work exclude event frequency and, except for the SKaMPI effort, they do not mention measurement variability in the compensation process. As far as our knowledge goes there hasn't been recent development in the field of overhead compensation in application traces. Thus, our work also shines a new light on the topic considering the latest processor architectures. Unlike previous authors we acknowledge the fact that measuring overhead is (largely) dependent on event frequency, and define a measuring routine which takes event frequency into account.

III. Perturbation and Compensation Models

As an implementation of our model, described in the subsections below, we characterized and measured the overhead of Akypuera [9], a tool developed by the authors for tracing OpenMPI [13] applications. It uses source code instrumentation via the PMPI interface. We also designed and implemented a compensation tool to automatically adjust the timestamps of execution traces generated by Akypuera prior to performance analysis.

A. Measurement of the Intrusion Overhead

We isolate the logging routine to measure the tracing overhead considering how frequent such routine is called. From now on, we identify the logging routine simply as `log`, in order to simplify the explanation. We measure its runtime with a user-defined call frequency f, equal to the event call frequency in the application trace. To do so the user inputs for how long he wants to measure `log` under f. After a warmup to avoid spurious observations, we do $t \times f$ calls to `log` and a function to sleep $1/f$ units of time, which we'll call `sleep`. If the `log` execution was instantaneous and `sleep` runtime was exactly the amount of time it is asked to sleep (i.e., if it had zero overhead), then the calls would take exactly t units of time. Thus, to get the mean execution time of `log` we reduce t from the measured execution time, and also reduce the overhead of `sleep` under f, which is measured in the exact same fashion. We repeat this process according to a user definition, obtaining n means which later on are used to obtain an estimation of the expected value for the overhead of `log` under f. Algorithm 1 details this routine. Notice that `sleeping` holds the time spent sleeping and also the overhead of both `sleep` and of the timer.

Ideally the `sleep` function should mimic the system load during the execution of the traced application. Currently we use a function which suspends the execution of the current thread (namely POSIX `nanosleep`), which is an imperfection of our tool. However, our methodology allows system load to be taken into account by using a more appropriate function instead. We also observe that in our implementation we assume an uniform event frequency throughout the application run. We therefore assume the application behavior is regular. Finally, we currently assume the same event frequency on every process of a parallel application.

Algorithm 1 The benchmarking routine

```
measurements ← ∅ ; period ← 1/f ; iters ← t × f
for i = 0; i <iters; i++ do                          ▷ Warmup
    sleep(period)
end for
sleeping ← 0
for i = 0; i <replications; i++ do
    start_timer()
    for j = 0; j <iters; j++ do
        sleep(period)
    end for
    end_timer()
    sleeping += elapsed_time
end for
sleeping = sleeping / replications
for i = 0; i <replications; i++ do
    start_timer()
    for j = 0; j <iters; j++ do
        log()
        sleep(period)
    end for
    end_timer()
    measurement_i ← (elapsed_time − sleeping) / iters
end for
```

This might seem overzealous, but as aforementioned the calls to `log` are usually very fast. In our case, they are smaller than the overhead of `sleep`. We observed that, for very high frequencies, the variability is very large and we often end up with negative values for execution times after reducing the overhead of `sleep`. There are two strategies to handle high measurement variability: taking more replications or considering the standard error. For the former, we recommend the user do a large number of replications, i.e., such that he obtains a (positive) estimation of the expected value that ceases to change with an increase in the amount of replications. The replications have to be used both when measuring the overhead of `sleep` as well as when measuring the overhead of `log`. For the later, set the number of replications that gives a sufficiently low value for the standard error of the mean [11]. The second strategy can also be implemented in an online fashion, during the execution of Algorithm 1.

B. Compensation Strategies using the Measured Overhead

After estimating the overhead as described in Section III-A, we subtract such estimation from the timestamps of recorded events. For events that are local to a process, i.e. independent

of events from other processes, this compensation is done directly. For non-local events, i.e. events that depend on events from other processes, we must compensate based also on the timestamp of any dependent event in order to respect and maintain the causality between them.

Given $event_m^i$, the ith measured timestamp of an event on a certain process, $event_a^i$, the approximated (compensated) timestamp of that event is given by Equation 1, where O is the overhead estimate.

$$event_a^i = event_a^{i-1} + (event_m^i - event_m^{i-1}) - O \quad (1)$$

Compensating non-local events is not as straightforward, and depends on the concurrency model. We adopt the formulas for message passing applications defined by Malony [8]. As noted by Vincent [12], compensation in the face of non-determinism may change the program behavior because it might break the observed causal relationships among processes. In the case of MPI applications, non-determinism is present when using `MPI_ANY_SOURCE` to receive a message from any process instead of a specific one. In this case, our tracing tool register which process actually sent the message in *that* execution, instead of registering `MPI_ANY_SOURCE`. Thus, the application might be non-deterministic, but the traced execution is not, and we can compensate it normally.

IV. Early Results and Discussion

In this section we present early results comparing the implementation of our method, which considers event frequency to calculate the mean execution time, with traditional methods which are base solely on averages. We traced a set of applications (described in section IV-B) in a shared memory environment (in IV-A) and compared the compensation techniques using several metrics (in IV-B).

A. Hardware/Software Configuration and Benchmarks

Besides using simple "ping pong" applications, we evaluate our model with two real world applications: Ondes3D v1.1, an earthquake simulator [14] and OSU Microbenchmarks v5.2 [15], more specifically the `osu_multi_lat` benchmark.

We use a server from INF/UFRGS with 32GB of memory and two Intel Xeon E5-2630 Sandy Bridge processors with six physical cores each (each core with two processing units), running Ubuntu 14.04.1, Linux 3.16.0-51. Everything was compiled using OpenMPI 1.6.5 implementation of the MPI standard and GCC 4.8, with the default optimization flags from each application. More specifically, all tests were done using the SM (shared memory) BTL (byte transfer layer) of OpenMPI, with the default MCA (Modular Component Architecture) parameters.

B. Results

We first do a visual space/time comparison of the compensated and uncompensated traces. Such fine grained comparison strategy looks for small changes in the compensated version. An example is shown in Figure 1, depicting a space/time view of the original execution trace of a run of Ondes3D. The processes are displayed on the vertical axis, while the runtime is in the horizontal axis. Each rectangle represents an event, and the arrows show message passing among them. The arrow color represents the message size in bytes. Early attempts with this method yielded too dissimilar views for a given time slice, rendering the comparison between the compensated (with our method) and original traces inconclusive. In other words, compensation shifted the event timestamps to such a degree that zooming in the same time slice renders two different regions of the trace, one in the original trace and another in the compensated one, at least at the end of the trace file, where the accumulated overhead is larger. Since we could not determine the fine grain impact of our method, we did not go any further with this metric to try to compare ours with traditional methods. As future work we intend to run Ondes3D on a networked environment and use the simpler metric described below to evaluate our method.

The main problem with the space/time view is the lack of a unique and simple metric that represent how far we are from the original uninstrumented version. Because of this, we consider also a simple metric to compare mean application execution times – that of the uninstrumented application run, that of the instrumented application, and that of the compensated version of that trace using both ours and traditional methods. In this case we say that a method is better than another if the execution time is closer to that of the uninstrumented run. For such comparison, we do thirty executions and compare the mean execution times.

Table I shows the comparison of the mean execution time of the OSU Microbenchmarks application. When considering event frequency, the execution times in the compensated trace are better than when not considering it. Although we observed similar results with other applications as well, we note that the difference in execution time is within the standard error. Furthermore, there was very little difference between the instrumented and uninstrumented execution times (i.e. little intrusion to be compensated in the first place). The measurements fit a somewhat normal distribution, with two modes towards the center.

TABLE I: Comparing the mean execution time of each trace (for a 99.7% confidence interval) of the OSU benchmark.

Execution	Mean	Standard error
Uninstrumented	12.958	0.2805
Instrumented	13.102	0.1766
Traditional	13.058	0.1765
Frequency (our approach)	12.945	0.1765

V. Conclusion and Future Work

In performance analysis we wish for the recorded application behavior to faithfully approximate the real application run. Compensation is one method to do this approximation, and a correct overhead estimation is paramount. In this work

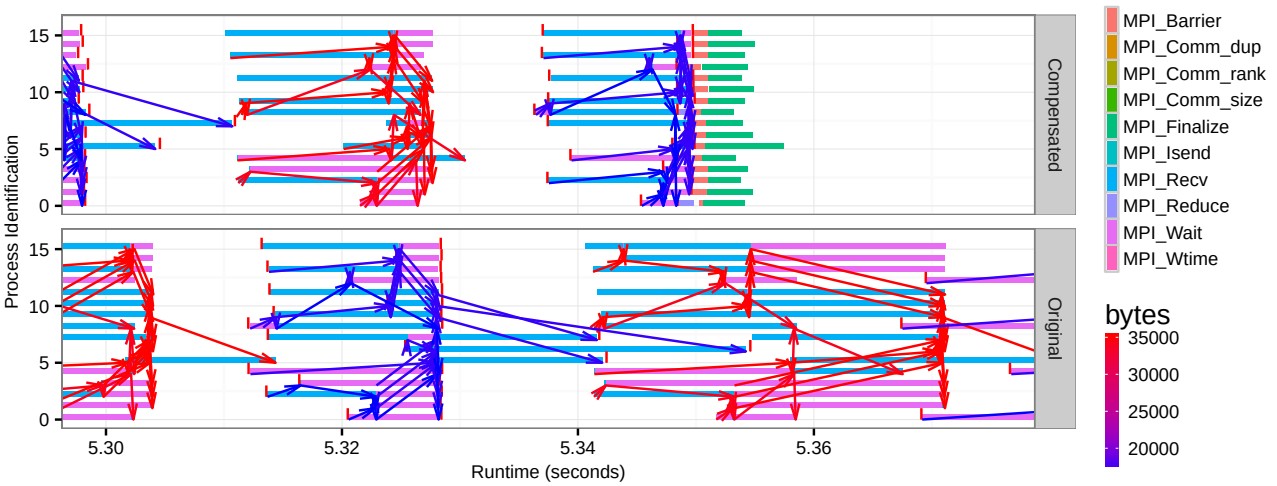


Fig. 1: Space/time comparison of original (bottom) and compensated version (top) for the Ondes3D trace with 16 ranks.

we observed that event frequency is a determinant factor of logging overhead. We also presented a novel approach to deal with this relation, and presented evidence from early experiments suggesting improvement over traditional methods.

Although we did not yet find a suitable way to do a fine grained comparison of our method with previous ones, mean execution time comparisons (as used by previous authors [8]) indicate improvements. Moreover, the mere observation of the dependency between event frequency and execution time should be enough to reduce the error in the compensation.

Future work include tests with applications with larger intrusion, and tests in a networked environment instead of a shared memory one. We also intend to improve our implementation of the method described in this article, for instance considering per process frequencies and comparing this to the current implementation. We also intend to study the validity of this approach when working with irregular applications. At the same time we shall keep searching for comparison metrics to better evaluate the impacts of our work.

Acknowledgements

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law nº 8248, 1991, and its subsequent updates. This investigation also receives funds from the H2020 program EU and MCTI / RNP-Brazil through HPC4E project with code 689772, the FAPERGS / Inria ExaSE design, universal design CNPq 447311 / 2014-0, and international CNRS / LICIA laboratory.

References

[1] B. de Oliveira Stein, "Depuração e visualização de programas paralelos," in *I Escola Regional de Alto Desempenho*, T. A. Divério and P. O. Navaux, Eds. Porto Alegre: Gráfica da PUCRS, 2001, pp. 151–175.

[2] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.

[3] G. J. da Silva, L. M. Schnorr, and B. Stein, "Jrastro: A trace agent for debugging multithreaded and distributed java programs," in *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*. Los Alamitos: IEEE Computer Society, 2003, pp. 46–54.

[4] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*. Springer Berlin Heidelberg, 2012, pp. 79–91.

[5] S. S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[6] A. Fagot and J. C. de Kergommeaux, "Systematic assessment of the overhead of tracing parallel programs," in *Parallel and Distributed Processing, 1996. PDP'96. Proceedings of the Fourth Euromicro Workshop on*. IEEE, 1996, pp. 179–186.

[7] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff, "Performance measurement intrusion and perturbation analysis," *IEEE Transactions on parallel and distributed systems*, vol. 3, no. 4, pp. 433–450, 1992.

[8] F. Wolf, A. D. Malony, S. Shende, and A. Morris, "Trace-based parallel performance overhead compensation," in *International Conference on High Performance Computing and Communications*. Springer, 2005, pp. 617–628.

[9] L. M. Schnorr, "Akypuera," http://github.com/schnorr/akypuera, 2016.

[10] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44. mar, 1995, pp. 17–31.

[11] D. Kranzlmüller, R. Reussner, and C. Schaubschläger, "Monitor overhead measurement with skampi," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 1999, pp. 43–50.

[12] J. C. De Kergommeaux, E. Maillet, and J. Vincent, "Monitoring parallel programs for performance tuning in cluster environments," *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments" book, P. Kacsuk and JC Cunha eds*, 2001.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[14] F. Dupros, C. P. Ribeiro, A. Carissimi, and J.-F. Méhaut, "Parallel simulations of seismic wave propagation on numa architectures." in *PARCO*, 2009, pp. 67–74.

[15] T. O. S. University, "Osu micro-benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks, 2016.

Energy Consumption and Performance analysis between ARM and Intel *

Ricardo Klein Lorenzoni[1], Edson L. Padoin[1,2],
Manuel Binelo[1], Philippe O. A. Navaux[2]

[1]Department of Exact Sciences and Engineering
Regional University of Northwest of Rio Grande do Sul (UNIJUI) – Ijui, RS – Brazil
[2]Institute of Informatics
Federal University of Rio Grande do Sul (UFRGS) – Porto Alegre, RS – Brazil
{ricardo.lorenzoni, padoin, binelo}@unijui.edu.br, navaux@inf.ufrgs.br

Abstract

Power demand and energy consumption is a primary constraint for the HPC community. In response, researchers aim to build alternatives to reduce power demand without reducing computing power. One approach to overcome power constraints is using low power processors. However, these ARM processors have low power demand and lower performance. In this paper we have compared the energy efficiency between a MPSoC with ARM processor and a conventional Intel XEON processor. We have used different benchmarks of NAS Parallel Benchmark (NPB) to analyze execution time and total energy consumption. The preliminary results show the ARM processors are up to 20.56 *times lower performance. Moreover, their power demand are up to* 27.65 *times lower for all benchmark tested. In this way, using ARM processor, we can reduce the total energy consumption in up to* 1.34 *times.*

1. Introduction

Power demand and energy consumption is a primary constraint for the HPC (High Performance Computing)community. According to DARPA report, the power limit for future exascale systems is 20MW [5]. However, the current topfive system, the Sunway TaihuLight, demand 15.4MW to compute 93PFlops. In response to high power demand, researchers aim to build alternatives to reduce power demand without reducing computing power. One approach to overcome power constraints is using MPSoCs with ARM processors. The Mont-Blanc Project is one of the first to introduce this idea [7].

Today, the main objective of the HPC community is the reduction of power consumption while maintaining or even improving the performance of supercomputers. One of the community approaches to increase performance without incurring the increase in energy consumption is the use of low-power processors, such as the ARM architecture. Newer processors this architecture have floating-point processing units, and possess a frequency of the reasonable clock, making it even more plausible the use of this HPC.

In this context, this paper question the feasibility to use low power processors in HPC. To address this question we compare the performance and energy consumption of a MPSoCs with ARM processors with a Intel XEON using NAS benchmark. The rest of the paper is structured as follows. In Section 2, we present related works on energy consumption and performance on ARM platforms. In Section 3, we discuss the methodology and the testbed evaluation. Section 4 describes the tests results obtained. Section 5 outlines our conclusions, contributions and future work perspectives.

2. Related Work

To build the next generation of supercomputers the HPC community have the power demand as a central question. Many papers have evaluated the runtime and power demand tradeof with different benchmark and real applications. Among them, different approaches have been used in the care of power consumption on the HPC.

In this context, power hungry CPUs are leaving space to new technologies in supercomputers design. Embedded processors, such as ARM are a possible alternative over CPUs that are the norm to supercomputers. Once, recent ARMv7 ISA have native support to single-precision FP and

* This research has received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E Project, grant agreement number 689772. Work developed on the context of the associated international laboratory between UFRGS and Université de Grenoble - LICIA.

double-precision. Also, they have support for SIMD instructions with the NEON unit [1]. Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam [3] reviewed the RISC vs. CISC debate considering the contemporary ARM and x86 processors running modern workloads to understand the role of ISA on performance, power and energy.

Jack Dongarra and Piotr Luszczek [4] present a landscape to analyze ARM. The authors make a comparison of energy efficiency with ARM and several Intel CPUs. Their results point that ARM has better efficiency (about 4 GFlops/W). Valero *et al.* [9] present similar results. The Cortex A9 have an efficiency of 4 GFlops/W and the future Cortex A15 have 8 GFlops/W. These values depict better efficiency than INTEL and IBM processors.

Rafael Aroca and Luiz Gonçalves [2] evaluated the power efficiency of several ARM and x86 devices while running typical server and number crunching tasks. They conclude that ARM processors have good performance to build servers and clusters, specially when considering the performance per watt and that processing clusters based on ARM processors are feasible to decrease power usage of several server applications.

Nikola Rajovic *et al.* [8] deploy and evaluate the first cluster for HPC with ARM mobile processors. The results show that the ARM Cortex-A9 is up to ten times slower than an intel i7 processor, but achieves a competitive energy efficiency compared to multicore systems in the Green500 list. Dominik GöDdeke *et al.* [6] employed a prototype ARM-based cluster to evaluate the performance trade-off between time and energy solving real-world problems.

Different to selected related works, our work analize the feasibility to use low power processors in HPC. In this we have used NPB benchmarks to compare the runtime and energy consumption of a MPSoC with ARM processor A20 model with a Intel processor XEON model.

3. Experimental Method

This section describes the methodology used in this study. We present the execution environment, and then discuss the Benchmark and Workload methodology.

3.1. Execution Environment

The execution environment is composed of two platforms. The first is a MPSoC Cubietruck equipped with ARM processor and the second is a server with Intel processor. Table 1 shows the main characteristics of each equipment.

The operating system on the MPSoC was provided by manufacturer, a modified version of GNU/Linux distribution with kernel version 3.4.106 compiles the application using gcc version 4.9.2. On Server, the operating system is the Ubuntu version 14.04 with kernel version 3.16.0-70 and gcc version 4.8.4

	Cubietruck	Server
Processor	A7	Xeon
Manufacturer	AllWinnerTech	Intel
Processor Model	A20	E5-2640v2
Processor Technology (nm)	40	22
Clock Frequency (GHz)	1	2
Cores/Processor (#)	2	8
Memory (GB)	2	32
Cache L1/Core (KB)	64	512
Cache L2 (KB)	256	2048
Cache L3 Shared (MB)	0	20

Table 1. Detailed configuration.

3.2. Benchmark and Workload

To measure the runtime and energy consumption we used NAS Parallel Benchmark (NPB). We run BT, FT, LU, MG and IS benchmarks sequentially and parallelly aim to analyze different workloads. In parallel version OMP was used with 2 threads in the Cubietruck and with 2, 4 and 8 threads in the Server.

We are measuring the power demand only of the processor, once this component determines the greater constraint of electrical power on the supplier.

4. Results

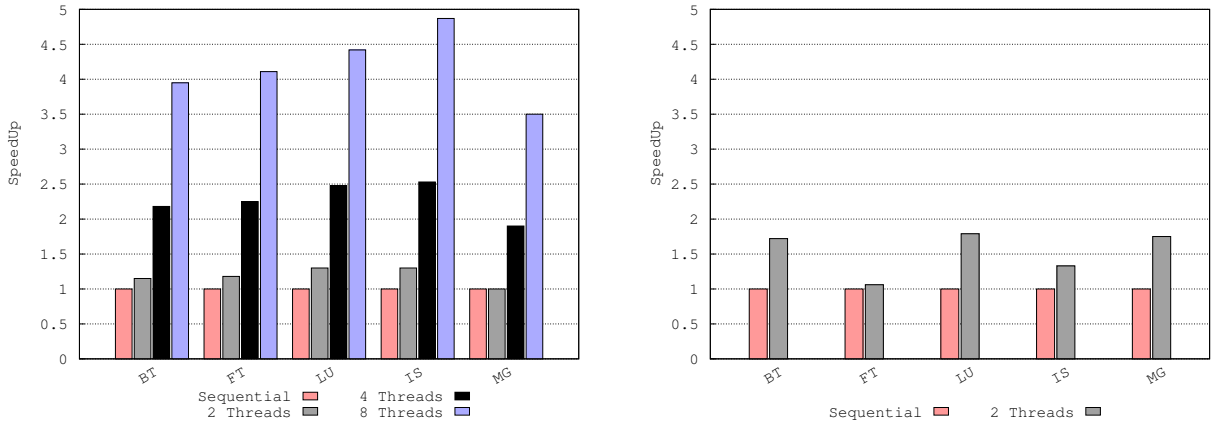
In this section the results will be presented. First we analyze the scalability of the devices with the parallel benchmarks and then the power consumption thereof.

4.1. Scalability Analysis

Scalability is one of the big challenges in HPC systems and is related to problems of all levels in the system. In the context of this work, the goal is to analyze scalability of an ARM processor and an Intel processor, making a runtime comparison between both.

Figure 1 presents the speedup achieved in the equipments with all benchmarks used. In the Server running the sequential version and paralleled with 2, 4 and 8 threads, and in the Cubietruck running in sequential version and with 2 threads, according the total amount of cores.

On the Server with 2 threads was obtained the small speedup, an average of 1.18. For 4 threads the average was 2.26. When using 8 threads, the speedup obtained were 3.50 and 4.87 for MG and IS respectively, as can be seen in the Figure 1(a). On the Cubietruck was obtained an average scalability of 1.52. The smallest speedup was 1.06 for



(a) Server

(b) Cubietruck

Figure 1. Speedup of each Equipment

FT and the highest was 1.79 for LU, as shown in the Figure 1(b).

It is interesting to note that, on Intel processor used in the Server (Figure 1(a)) has a very small speedup when 2 threads were used if compared with Cubietruck. Featuring a gain of speedup considerably with 4 and 8 threads. For example, for MG benchmark with 2 threads the speedup was of only 1.0039.

To better analyze this point, we show in the Figure 2 a speedup comparison between the two processors, while running the benchmarks with 2 threads.

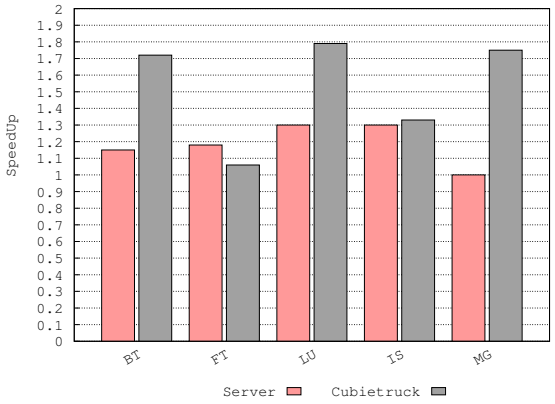


Figure 2. Speedup comparison for 2 Threads

Comparing the speedup between the two devices when used only 2 threads, the Cubietruck presents better performance in 4 benchmarks (BT, LU, IS and MG), with smaller speedup that the Server only on the FT benchmark.

4.2. Total Energy Consumption Analysis

Taking into account that the energy consumption is one of the current limitations of the HPC systems, in this section we analyze the total energy spent by devices to complete the execution of each benchmarks.

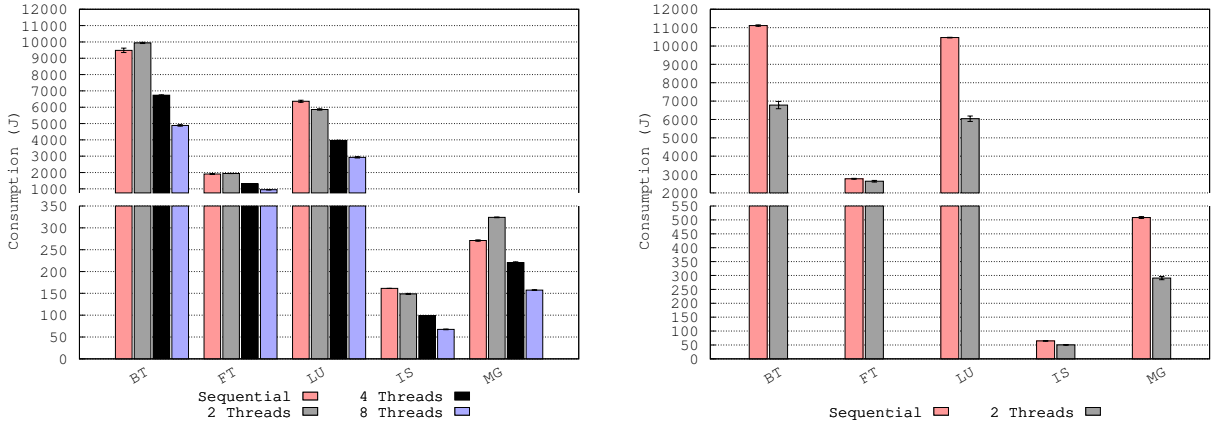
The Figure 3(a) shows the total energy consumption for the Server, running the sequential version and paralleled with 2, 4 and 8 threads. In the Figure 3(b) shows the total energy consumption for the Cubietruck running the sequential version and parallel with 2 threads.

When executed the benchmarks on the Server (Figure 3(a)) with 2 threads, it increases the total energy consumption for BT, FT and MG. On the other hand, the energy was reduced for IS and LU and for all other tested configurations.

Server presents a reduced of the total energy consumption of approximately half of the sequential version when paralleled with 8 threads (Figure 3(a)). The largest consumption reduction for this configuration was obtained in the IS achieving a reduction of 58.21% and the lowest was MG with reduction of 41.85%.

ARM processor provides a significant reduction in total energy consumption to some of the benchmarks when paralleled with 2 threads (Figure 3(b)). Were achieved consumption reductions between 4.80% (FT) and 42.86% (MG). The lower reduction in total energy spent for FT benchmark can be explained by the fact that the performance of the parallel version does not increase.

It is interesting to make a direct comparison between the devices when both run a parallel version of benchmarks with 2 threads. Cubietruck spents 66% less energy for IS benchmark and 35% more for FT benchmark when compared to energy spent on Server.



(a) Server

(b) Cubietruck

Figure 3. Total Energy Consumption of each Equipment

When compared the energy using all processors available in each platform (Cubietruck with 2 threads x Server with 8 threads), the diference increases. Cubietruck spent 25.62% less energy for IS benchmark, and Cubietruck spent between 38.85% (BT) and 179.02% (FT) more energy than Server.

5. Conclusion

This paper discusses the question of use low power processors in HPC. We analyse the scalability and the total energy consumption between a MPSoC with ARM processor and a Intel processor. Using NPB benchmark, our contributions include: (i) evaluation and comparison of performance and scalability of each processor; and (ii) comparison of total energy consumption of processors to finalize the execution of the benchmarks.

We evaluate 1 MPSoC with processor on ARM and 1 Server with processor Intel. In our experiments, we saw that the intel processor running the benchmarks in parallel is about 20.56 to 73.55 times faster than the ARM processor. On the other hand, the ratio of the total energy of the two processors is far lower, being between 0.74 and 2.79 times.

We find that ARM processors have good scalability compared to the intel processor, getting a speedup average 1.52 with 2 threads, while the intel processor, with the same number of threads obtained an average speedup of only 1.18.

Our future work will focus on several energy consumption reduction options for hpc environments. Focusing mainly on MPSoCs and GPUs.

References

[1] ARM NEON Tecnology, 2012. http://www.arm.com/products/processors/technologies/neon.php.

[2] R. Aroca and L. Garcia Gonçalves. Towards Green Data-Centers: A Comparison of x86 and ARM Architectures Power Efficiency. *Journal of Parallel and Distributed Computing*, 72(12):1770–1780, 2012.

[3] E. Blem, J. Menon, and K. Sankaralingam. A detailed analysis of contemporary arm and x86 architectures. *UW-Madison Technical Report*, 2013.

[4] J. Dongarra and P. Luszczek. Anatomy of a Globally Recursive Embedded Linpack Benchmark. In *16th IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2012.

[5] M. Dorier, O. Yildiz, S. Ibrahim, A.-C. Orgerie, and G. Antoniu. On the energy footprint of i/o management in exascale hpc systems. *Future Generation Computer Systems*, 62:17–28, 2016.

[6] D. GöDdeke, D. Komatitsch, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic, and A. Ramirez. Energy efficiency vs. performance of the numerical solution of pdes: An application study on a low-power arm-based cluster. *Journal of Computational Physics*, 237:132–150, 2013.

[7] Mont-Blanc. *Mont-Blanc Project Home Page*. Last visited January 2013. http://www.montblanc-project.eu/.

[8] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439–443, 2013.

[9] M. Valero. Towards Exaflop Supercomputers. In *High Perfomance Computing Academic Research Network (HPC-net)*, pages 1–117, Rio Patras, Greece, 2011.

Coordinating Data Access at I/O Forwarding Nodes

Jean Luca Bez*, Francieli Zanon Boito†, Lucas Mello Schnorr*, Philippe Olivier Alexandre Navaux*
* Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
† Department of Informatics and Statistics – Federal University of Santa Catarina (UFSC), Florianópolis, Brazil

***Abstract*—In High-Performance Computing, parallel file systems (PFS) are used by applications to obtain I/O performance even when handling large amounts of data. To alleviate the concurrency caused by thousands of nodes accessing a smaller number of PFS servers, intermediate I/O nodes are deployed between processing nodes and the file system servers. Each intermediate I/O node forwards requests from multiple clients to the file system. This scenario is suitable to perform optimizations like I/O scheduling.**

In this paper, we present and evaluate a new scheduling algorithm for the forwarding layer that coordinates intermediate I/O nodes' accesses to servers. Our proposal works to decrease concurrency at the data servers, a factor previously shown to negatively affect performance. The proposed algorithm is able to improve read performance by up to 30% over other scheduling algorithms and by up to 51% over not using an I/O forwarding layer.

I. Introduction

Scientific applications demand increasing performance from the High-Performance Computing (HPC) field. These requirements justify the appearance of ever increasing large scale parallel platforms. It is common for such platforms to have a shared storage infrastructure over a dedicated set of nodes with a parallel file system (PFS) deployment. Due to the historical gap between processing and data access speeds, parallel I/O is a limiting factor for many applications' performance. Furthermore, the applications' performance could be impaired if all processing nodes were to concurrently access the file system servers.

I/O forwarding is a technique employed by several large-scale clusters and supercomputers aiming at reducing the number of clients concurrently accessing the file system servers. In this context, some dedicated nodes receive I/O requests and forward them to the file system [1], as illustrated in Fig. 1. Besides performance, this additional layer between application and file system provides the opportunity to apply optimizations like requests reordering and aggregation.

Write operations have been the main focus of parallel I/O research. However, scientific applications have been reading increasing amounts of data to leverage previous knowledge into their analysis. Argonne National Laboratory analyzed the top ten applications regarding its I/O operations on a supercomputer. It was revealed that large amounts of data were being read, with three of these applications exclusively performing read operations during their executions [2]. For this reason, we focus our study on the performance of read requests.

In this paper, we evaluate a new scheduling algorithm for the forwarding layer that aims at decreasing contention when accessing the parallel file system data servers. It coordinates accesses using time windows so that in each window each I/O node focus all its requests to one data server and different I/O nodes focus on different data servers. We show performance improvements with our algorithm over existing schedulers.

The rest of the paper is organized in the following way: the next section provides a background and discusses related work. Section III discusses our new scheduling algorithm for the forwarding layer, including its implementation and requirements. The experimental methodology and results are presented in Section IV. Final remarks and future work are presented in Section V.

II. Background and Related Work

IOFSL [3] is an open-source framework that implements the I/O forwarding technique as an attempt to bridge the increasing performance scalability gap between computing and I/O components. IOFSL ships I/O calls from compute nodes to dedicated I/O nodes, that perform operations on behalf of those compute nodes. It uses the stateless ZOIDFS I/O protocol, the API from the ZOID forwarding infrastructure [4], and the Buffered Message Interface (BMI) [5] network abstraction layer to provide request forwarding over multiple parallel file systems and interconnection networks. The IOFSL software stack consists of two main components: a ZOIDFS client library running on the compute nodes and an I/O forwarding daemon running on the intermediate I/O nodes. The client

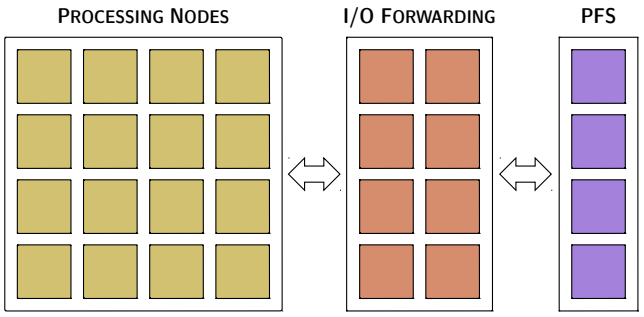


Fig. 1. I/O forwarding scheme on a large-scale cluster or supercomputer. The number of forwarding nodes is generally smaller than the number of processing nodes and larger than the number of parallel file system servers.

library transparently forwards I/O requests from the compute node to the corresponding I/O node.

Ohta et al. [6] improved the performance of the IOFSL framework by using I/O scheduling. They implemented two algorithms: a simple *First In, First Out* (FIFO) and a quantum-based algorithm they called *Handle-Based Round-Robin* (HBRR). The latter is based on an algorithm successfully applied to parallel file systems' data servers, and aims at performing requests reordering and aggregations to improve the generated access pattern.

We have chosen to use IOFSL in our research because it is the only available open-source tool already tested on large scale clusters and supercomputers. Furthermore, we could build on previous works and contributions to effectively compare our new solution with the state of the art.

AGIOS [7] is a scheduling library that can be used by I/O services to manage incoming I/O requests at the file level (file offsets). It implements five scheduling algorithms and it exposes a simple API to build new schedulers. Because we wanted our solution to be generic and possibly ported to other forwarding tools, or even used in the file system server context, we have integrated the AGIOS scheduling library in the IOFSL as a scheduling option and harness its API to prototype our new scheduler. With the AGIOS scheduling option, requests are added to the library's queues when they arrive at the forwarding nodes. When the algorithm applied by AGIOS decides it is time to process a request, a callback function inside IOFSL simply adds it to the dispatch queue. This ensures requests will be processed in the order dictated by the scheduling algorithm in use.

III. Coordinating PFS Server Access with TWINS

In this section, we present a new I/O scheduling algorithm for the I/O forwarding layer called *Time WINdows Scheduler* (TWINS). The main idea is coordinate intermediate I/O nodes accesses to the file system servers so that, at any given moment:

1) an I/O node is focusing all its accesses on one server;
2) different I/O nodes are focusing on different servers.

TWINS keeps multiple request queues, one per data server. During the execution, it iterates the queues in a round robin fashion, respecting the time window dedicated to each server. This implies in additional waiting time if there are no requests for the current server, even if there are incoming requests to other data servers. The pseudo-code for this scheduler is presented in Algorithm 1.

Algorithm 1 TWINS

Require: $Q[i]$ is the updated list of requests to server i

```
1:  i ← 0
2:  while true do
3:      resetTimer()
4:      while elapsedTime() < windowSize do
5:          if length(Q[i]) > 0 then
6:              processRequest(Q[i])
7:          else
8:              timeout ← windowSize − elapsedTime()
9:              timedWaitForRequests(Q[i], timeout)
10:         end if
11:     end while
12:     i ← nextServer(i)
13: end while
```

Our scheduling algorithm requires additional information to work, besides the typically available information - file handle, offset, type, and size - found I/O requests. It is necessary to know exactly where each stripe of a file is located so the requests could be grouped by data servers.

We have modified the IOFSL code to collect the file layout information from the file system when opening or creating a file. Since this information is requested only once and kept while the file is open, no significant overhead is expected. Furthermore, as the file distribution never changes during its lifetime and in several file systems it is possible to obtain the data file layout, our approach continues to be generic.

Using the file distribution information, the starting server for a request is obtained as a function of its starting offset and stripe size. An additional translation step is required so each IOFSL server focuses its window on a distinct server. This translation is done according to the IOFSL node identifier. The N_{th} I/O node will use the N_{th} permutation of the data servers list as a translation rule. Therefore, if the number of intermediate I/O nodes is larger than the number of data servers, more than one node may access the same server at the same time, but these concurrent accesses are minimized.

IV. Experimental Results

Experiments were conducted in clusters from the Nancy site of Grid'5000 [8]. Four machines from the Grimoire cluster were selected as PVFS2 servers and 32 machines from the Grisou cluster as clients. Four additional machines from Grisou were configured to act as the forwarding layer. Clients are equally distributed among the I/O nodes.

Grimoire's nodes have two 8-core Intel Xeon E5-2630 v3 and 126 GB of RAM. Grisou's nodes are identical to Grimoire's ones. A 558 GB hard disk is used for storage at the servers. Nodes are interconnected through a 10 Gbps Ethernet network, and there is a 10 Gbps link between the clusters.

PVFS version 2.8.2 was deployed with its default parameters. Data servers were configured to bypass buffer caches. IOFSL uses the PVFS2 client library to communicate directly with the file system. The maximum number of requests that can be aggregated from the dispatch queue was 16, the default.

The MPI-IO Test benchmark was executed by 128 processes. Each one generates 1024 requests of 32 KB (32 MB per process), a total of 4 GB per test. Processes read a shared file using a 1D strided access pattern. From each execution, we take the completion time of the slowest process (makespan).

Experiments were repeated 8 times in a random order, and error bars were calculated using a 99.7% confidence interval, i.e. there is a 99.7% probability that the true mean lies between the lower and upper bounds of the interval. These bounds are

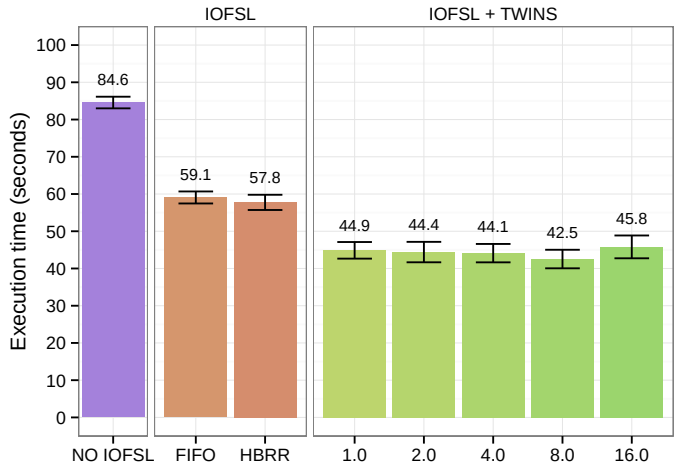


Fig. 2. Comparison of different window sizes for TWINS scheduler and the default IOFSL schedulers - FIFO and HBRR - for the single shared-file with 1D strided access pattern.

equivalent to three times the standard deviation divided by the square root of the number of measurements.

Figure 2 depicts the results obtained by TWINS and compares them with results obtained with other employed scheduling algorithms for the I/O forwarding layer and with not using IOFSL. For the 1D strided access pattern, TWINS provides a performance improvement of approximately 30% over the HBRR algorithm (with IOFSL) and of approximately 51% over not using IOFSL. This access pattern is ideal for TWINS because the scheduler always has requests for all servers since processes start their accesses at different ones. Therefore it has the opportunity to perform meaningful coordination, reducing competition for resources.

The time window duration also has an impact on the execution time. A small window does not allow an effective access coordination because it does not hold requests to other servers long enough for them to be aggregated. On the other hand, a large window imposes overhead as there are not enough requests to each data server to fill a whole window, so the scheduler spends too much time waiting. Figure 2 also shows that the window size that presented the best performance is of 8 milliseconds. This initial results demonstrate that TWINS is able to coordinate I/O nodes access by reducing contention when accessing the file system data servers.

V. Conclusions

In this paper, we studied read performance in the I/O forwarding layer. We evaluated two algorithms from related work - FIFO and HBRR - in the IOFSL framework. Our analysis has shown that, despite improving read performance, techniques to adjust the access patterns (requests aggregation and reordering) are only partially effective because the access pattern is not the main factor for read performance through the I/O nodes.

We have proposed a new scheduling algorithm for the I/O forwarding layer called TWINS. Our algorithm uses time windows to coordinate the I/O nodes' accesses to different data servers, working to decrease contention. Our performance evaluation has shown improvements for 1D strided access patterns of 30% over the FIFO and HBRR algorithms. Compared to not using I/O forwarding nodes this gains goes up to 51%. These initial results demonstrates that our new scheduler is able to reduce contention and improve performance.

Future work will focus on studying other access patterns and exploring additional I/O forwarding layer configurations such as the ratio of clients to I/O nodes and additional benchmarks. Furthermore, we also expect to focus our study on proposing an automatic mechanism to tune the window size based on the system configuration and on the application access pattern.

Acknowledgments

This research has received funding from the MCTI/RNP-Brazil under the HPC4E Project, grant agreement nº 689772.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr/).

References

[1] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[2] R. Ross, "Future HPC systems and some implications for storage software," http://www.opensfs.org/wp-content/uploads/2014/04/D2_S26_2020Panel_Ross.pdf, Accessed: August 2015.

[3] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *Proceedings...*, IEEE International Conference on Cluster Computing and Workshops. IEEE, Aug. 2009, pp. 1–10.

[4] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O forwarding infrastructure for petascale architectures," in *Proceedings...* 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008, pp. 153–162.

[5] P. Carns, R. Ross, W. L. III, and P. Wyckoff, "BMI: a network abstraction layer for parallel I/O," in *19th IEEE International Parallel and Distributed Processing Symposium*, April 2005, pp. 8 pp.–.

[6] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization techniques at the I/O forwarding layer," in *Proceedings...*, IEEE International Conference on Cluster Computing. IEEE, Sep. 2010, pp. 312–321.

[7] F. Z. Boito, R. V. Kassick, P. O. A. Navaux, and Y. Denneulin, "AGIOS: Application-guided I/O scheduling for parallel file systems," in *Proceedings...*, International Conference on Parallel and Distributed Systems (ICPADS). IEEE, Dec. 2013, pp. 43–50.

[8] D. Balouek, A. C. Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, *Adding Virtualization Capabilities to the Grid'5000 Testbed*, ser. Communications in Computer and Information Science. Springer International Publishing, 2013, vol. 367, pp. 3–20.

Viability of Low-Power Architectures as Parallel File Systems

Amanda B. Braga[1], Natália G. Rampon[1], Vinícius R. Machado[1], Jean L. Bez[1],
Francieli Z. Boito[2], Rodrigo V. Kassick[1], Edson L. Padoin[3], Philippe Navaux[1]

[1]Federal University of Rio Grande do Sul – Porto Alegre, Brazil
{amanda.binotto, natalia.rampon, vrmachado, jean.bez, rvkassick, navaux}@inf.ufrgs.br
[2]Federal University of Santa Catarina – Florianópolis, Brazil
francieli.boito@posgrad.ufsc.br
[3]Regional University of Northwest of Rio Grande do Sul (UNIJUI) – Ijuí, Brazil
elpadoin@inf.ufrgs.br

Abstract

This paper presents an I/O performance and energy efficiency analysis of low-power processors when compared to conventional architectures. This study aims at evaluating the viability of using such low-power architectures as servers to file systems in HPC environments. Results have shown that the low-power architecture could be an alternative to applications that perform many more read operations than write operations. The study also has shown that this architecture could lead to 93% of energy consumption decrease in read operations.

1. Introduction

The increase of processing power in architectures of High-Performance Computing (HPC) came along with the increase of a significant power demand on these systems. Great energetic demands like this are not the best scenario ecologically and economically speaking. In this regard, a DARPA report suggests that future HPC systems - from which is expected exaflops performance - should obey a 20MW limit on power demand [1].

Based on this premise, researchers have been seeking alternatives to respect such limits. A commonly used strategy is the use of low power architectures, changing regular processors to Advanced RISC Machines processors (ARM). Despite presenting less performance, these architectures give better power efficiency to some scientific applications [2].

However, processing is not the only one responsible for the big energetic consumption on High-Performance Computing Systems. Input and Output operations - known as I/O operations - also contributes to the high power demand, due to the crescent gap between processing and storage latencies. Therefore, it is common that applications spend a significant time on I/O operations and, for this reason, methods on how to decrease this power demand should be studied.

Considering the HPC scenario, I/O operations are processed by the parallel file system (PFS) and machines operate as servers for the data. These servers receive requests from the processing nodes and process them accessing the local storage system. Therefore, the processing capability of these machines is not well explored due to the elevated time spent with I/O operations. As an alternative, it is possible to consider that low demanding power architectures when used as storage servers could be an alternative that might lead to better energy efficiency.

With the possibility to use a low power architecture as a storage server in mind, the objective of this paper is to do a comparison between the mentioned architecture with a traditional computer, both working as parallel file system data servers. The rest of this paper is organized as follows. The next section discusses the experimental methodology involved and results are shown in Section 3. Lastly, the conclusion of this paper and future work are presented in Section 4.

2. Experimental Methodology

To achieve the objective of this research, two machines were used. The low power computer chosen was an A20 dual-core ARM Cortex-A7 processor produced by AllWinner running at 1GHz frequency. Besides, the machine has 2GB RAM memory and 8GB NAND storage. The storage device used was an SSD 840 Series MZ-7TD500BW by Samsung with a capacity of 500GB and a SATA 6Gb/s bus.

It was not possible to run a real file system on the ARM processor. The impossibility was due to the fact that the processor only accepts a modified Linux kernel as an operational system that does not support the file system. Attempts were made to install the module that supports it, but they were not successful. Instead, an MPI code was used to simulate the file servers' activity. The servers' simulation consists of them receiving requests that are staggered and there is no communication between the servers.

Requests are provided to the servers through trace files. Each one of them is organized as follows: there is one request per line and each line contains the application's timestamp, the request's size and the file's offset. The trace file approach was used because it was of this research's interests to isolate the network so a fair comparison could be made. By using the mentioned approach, it is possible to exclude the time needed to a client to make a request.

In addition, it would also have been unfair to compare the network used in our low power cluster with a regular cluster's one, because it is far slower and would harm performance. With that in mind, we decided to take the network off our experiments, because we wanted to focus on the processing capabilities of the low power architectures. Without the network in place, we were able to use a single desktop in the comparison, which allowed the usage of the same SSD memory in both architectures, simulating, the fact that in a conventional cluster we would not be able to change storage devices. By using the same storage device, a more fair comparison was made, taking into account particularities in our storage device and its characteristics.

The measurement of power was made with a P4460 Kill A Watt EZ power meter, which has an accuracy of 0.5% and a refresh rate of 1 second. However, since the equipment used does not have any means of communication with external devices, it was impossible to gather data directly from it. We then devised a method for data gathering, which consisted of filming the data being shown in the display of the power meter and then synchronizing it with the timestamps of beginning and end of each test. Thus, we were able to manually get the data and analyze it.

Trace files were created to emulate a single client contiguously accessing a 6GB file from the parallel file system server. This access is separated in small (32KB) or large (4MB) requests.

The size of all executed tests was 6GB. Tests were made considering the requests' size and the type of operation; they were executed in both worked architectures. In order to increase the reliability of the tests, each one of them was repeated five times. Afterward, the arithmetical mean of the five tests was made and the obtained value was used to analyze the data. Therefore, there were made forty experiments and there were eight different configurations of tests.

3. Results

In this section, the results obtained will be presented. Firstly, we will analyze how the size of requests and operation influence the power demand. Afterward, in Section 3.2 the tests' runtimes in both architectures will be compared.

3.1. Power demand analysis

In order to investigate the influence of requests on power demand, the average power will be used. This value is obtained by the arithmetic mean of instant power measurements.

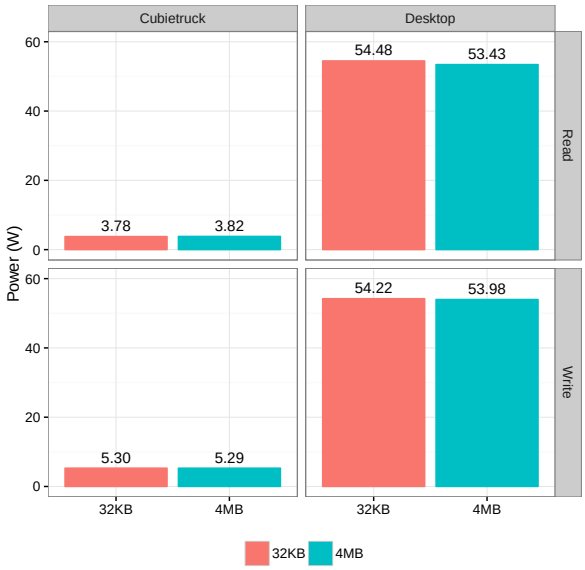


Figure 1. Power demand

Figure 1 shows the power demand on each architecture, separated by size and type of operation (write or read). The results for each architecture are in different columns and the rows discriminate between the operation performed.

From this data, it is clear that the requests' size does not have any impact whatsoever on the demand for power, neither in the Cubietruck nor in the Desktop configuration. However, in the Cubietruck device the type of operation performed alters the demand: write operations consume 40.2% more power than read operations in this architecture. The Cubietruck alternative also decreases power demand in 90.2% in comparison to the Desktop architecture in write operations and in 92.8% in readings.

3.2. Runtime analysis

Another important measure of this research was the runtime analysis of the conducted tests. The measures were made by running the same tests on the low power and the regular one architectures.

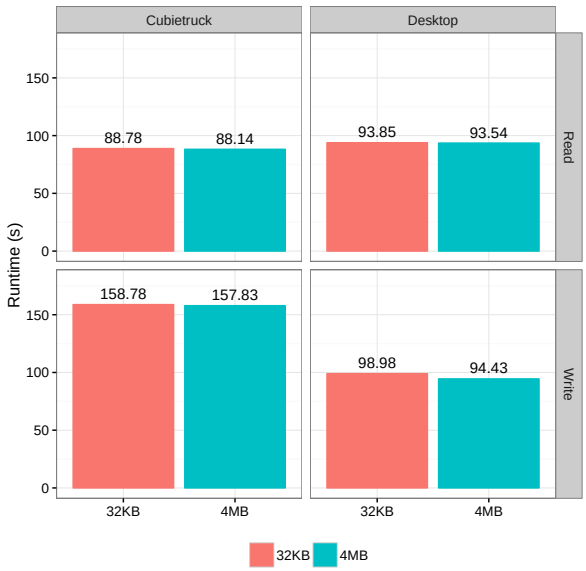


Figure 2. Execution time

Figure 2 shows the runtime analysis on each architecture, separated by size and type of operation (write or read).

By the results' analysis, it is possible to see, in the first place, that the requests' size does not affect the tests' runtime. Also, it is clear to note that on read operations the ARM processor and the regular processor present very similar performance. Yet, we can see that write operations on the low power architecture present a worse performance than on the regular architecture. With 32KB request size, by instance, the Cubietruck had a 68,8% increase of time by comparing with the Desktop. Therefore, the ARM processor could be an alternative to regular processors when dealing with an application that performs a lot more read operations than write ones.

3.3. Energy consumption analysis

The energy consumed in Joules by the experiments is obtained multiplying the median power by the execution time. Moreover, besides the impact caused by the type of operation because of the change in runtime, it is also expected to be found differences in energy consumption by the tests between the different architectures. This happens because the devices have different power demands.

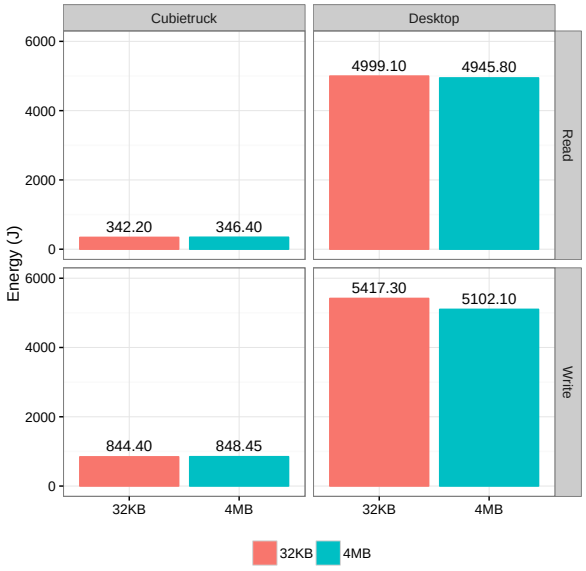


Figure 3. Energy consumption

Figure 3 shows the energy analysis on each architecture, separated by size and type of operation (write or read). It is again seen that the requests size have little to no influence in the results. By the graphics, it's clearly observed that the Cubietruck has great energy efficiency. The low-power architecture consumes 93% less energy in read operations and 83.4% in write ones in comparison to the usual one.

Considering the fact that both architectures have similar runtimes in read operations, the ARM processor is a feasible low-power alternative to regular processors in this aspect. Even when taking into account the 68.8% increase in execution time seen during write operations in Section 3.2, the alternative still stands as a possibility, since the energy consumption decline in this scenario is of 83.4%. Hence, the Cubietruck device would be a good substitute for regular architectures in applications that use more read operations than write ones.

4. Conclusions and Future Work

This study showed that low power processors could be an alternative to applications that perform many more read operations than write ones. The substitution would lead to an energy economy of up to 83.4% in write operations and 93% in read ones.

As future work, we plan to emulate real scientific applications and include more workloads in the study, such as non-contiguous accesses to files, more clients accessing concurrently the file system and different access patterns. Moreover, we will include the network in the future analysis.

Acknowledgements

The authors of this paper would like to thank Rodrigo Kassick from the Federal University of Rio Grande do Sul for the code used to simulate the file system activities.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, M. Denneau, P. Franzon, W. Harrod, K. Hill, and et. al. Exascale computing study: Technology challenges in achieving exascale systems. pages 1–297, 2008.

[2] E. L. Padoin, F. Z. Boito, L. L. Pilla, M. B. Castro, P. O. A. Navaux, and J.-F. Mehaut. Performance/energy trade-off in scientific computing: The case of arm big.little and intel sandy bridge. *IET Computers & Digital Techniques*, pages 1–14, 2014.

Análise da Eficiência Energética de Operações de E/S em Arquiteturas de Baixa Potência *

Pablo J. Pavan [1], Ricardo K. Lorenzoni [1], Jean L. Bez[2],
Edson L. Padoin[1,2], Francieli Z. Boito[3], Philippe O. A. Navaux[2], Jean-François Méhaut[4]

[1] Universidade Reg. do Noroeste do Estado do Rio G. do Sul (UNIJUI) – Ijui, RS – Brazil
[2] Universidade Federal do Rio Grande do Sul (UFRGS) – Porto Alegre, RS – Brazil
[3] Universidade Federal de Santa Catarina (UFSC) – Florianópolis - SC – Brazil
[4] Universidade de Grenoble – Grenoble – France

{pablo.pavan, ricardo.lorenzoni, padoin}@unijui.edu.br,
francieli.boito@posgrad.ufsc.br, {jean.bez, navaux}@inf.ufrgs.br, jean-francois.mehaut@imag.fr

Resumo

This work presents a performance and energy efficiency analysis of I/O operations when using ARM processors with SSD and HDD storage. We aim to evaluate the viability of using such low-power architectures as file systems servers in HPC environments. To address this question we used the IOzone benchmark to analyze the execution time and total energy consumption. Prelimiry results point that ARM can achieve energy savings of up to 136 *times when compared to conventional architectures.*

1. Introdução

Em decorrência do aumento significativo da demanda de potência dos sistemas computacionais, inúmeras pesquisas vem sendo desenvolvidas pela comunidade de computação de alto desempenho em busca de alternativas que reduzam demanda de potência dos sistemas. Nesta premissa, pesquisadores sugerem no relatório DARPA um limite de 20MW para os futuros sistemas exascale [4].

Para atender este limite pesquisas têm sido realizadas com diferentes alternativas. Uma delas é o uso de processadores *Advanced RISC Machine* (ARM). Estes, apesar de possuírem menos desempenho, propiciam uma melhor eficiência energética para algumas aplicações científicas [9].

Os processadores representam um percentual significativo na demanda de potência dos sistemas de HPC. Da mesma forma, sistemas de arquivos também apresentam impacto na demanda de potência. Nesses sistemas, as operações de E/S geralmente são feitas em sistemas de arquivos paralelos. Entretanto todo o poder de processamento do sistema geralmente não é utilizado durante operações de E/S. Nesses cenários, o uso de arquiteturas de baixa demanda de potência, como servidores de armazenamento, poderia apresentar uma melhor eficiência energética [10].

A fim de avaliar a viabilidade dessa ideia, o presente artigo apresenta um estudo comparativo de eficiência energética entre um computador com processador tradicional e um *Multi-Processor System-on-Chip* (MPSoC) com um processador ARM. Essa avaliação inclui discos rígidos (HDDs) e dispositivos de estado sólido (SSDs) para o armazenamento.

O restante do trabalho está assim organizado. A Seção 2 discute os trabalhos relacionados Na Seção 3 descreve o método experimental e os ambientes utilizados nos testes. Resultados são discutidos na Seção 4 seguidos das conclusões e trabalhos futuros.

2. Trabalhos Relacionados

Diversos trabalhos da literatura focam no consumo energético de operações de E/S. Tais operações representam grande parte do tempo de execução de muitas aplicações. Pesquisas menos recentes exploravam a

* Trabalho parcialmente apoiado por CNPq, CAPES, FAPERGS. Possui recursos do projeto Europeu EU H2020 Programme e do MCTI/RNP-Brasil no Projeto HPC4E, sob número 689772. É realizada no contexto do Laboratório Internacional Associado entre a UFRGS e Université de Grenoble-LICIA.

utilização de discos de múltiplas velocidades para servidores de armazenamento [1, 3, 13]. Outras, mais atuais, empregam *Dynamic Voltage and Frequency Scaling* (DVFS) para reduzir o consumo do processador durante as operações de E/S.

Neste contexto, Ge *et al.* [2] propõem uma estratégia para arquiteturas de HPC que envolve a realização de DVFS nos *nós* de processamento durante as suas operações de E/S. Esta abordagem leva em consideração as características dos acessos das aplicações para decidir a frequência mais apropriada. Uma técnica semelhante havia, anteriormente, sido aplicada para aplicações sequenciais [11].

O consumo de energia é um dos principais problemas para o desenvolvimento da próxima geração de supercomputadores. Pesquisas buscam avaliar o consumo de energia de processadores de baixo consumo. McKenney *et al.* [5] atingiu um ganho de eficiência energética de 10% em sistemas Mobile utilizando Cortex-A15.

Ou *et al.* [7] comparou ARM e clusters baseados em Intel Nehalem para web services, e concluiu que ARM proporciona na media uma eficiência energética de 1.3 vezes melhor e é capaz de ter uma performance melhor do que o Intel em alguns cenários.

Por outra lado, Nijim *et al.* [6] combinam dispositivos de armazenamento baseados em flash (SSDs) com discos HDD para prover armazenamento com menor consumo energético. Isso é alcançado utilizando os dispositivos mais rápidos como uma cache para os discos rígidos. Essa estratégia híbrida (SSD+HDD) de armazenamento é explorada em diversos trabalhos para prover alto desempenho para servidores de E/S [12]. Nesses casos, o SSD é utilizado como uma cache por causa do seu alto custo por byte, que inviabilizaria a substituição total dos discos rígidos.

Desta forma, independentemente das técnicas discutidas para redução do consumo, em muitas aplicações, o poder de processamento as vezes subutilizado durante os períodos de E/S. Nestes casos, o uso de processadores ARM poderia ser uma alternativa para redução da demanda de potência nos sistemas de armazenamento.

Apesar da viabilidade do uso dessas arquiteturas para a computação de alto desempenho ter sido objeto de trabalhos, não foram encontradas pesquisas que investiguem o consumo energético de operações de E/S em arquiteturas de baixa potência.

3. Método experimental

Dois ambientes foram utilizados para esse trabalho. O primeiro, chamado de PC, é um computador tradicional com um processador Intel Core2Duo modelo E8400. Esse processador é da arquitetura *Wolfdale* e possui pipeline de 14 estágios com execução de até 4 instruções por ciclo. O equipamento possui 6 GB de RAM com frequência de 800 MHz.

O segundo ambiente é um MPSoC CubieTruck com um SoC A20 fabricado pela AllWinnerTech e uma dual GPU MALI400 MP2, chamado de MPSoC. O processador é um Dual Core ARM Cortex-A7. Esse processador possui uma arquitetura superescalar, duas unidades de execução parcial, *pipeline* com 8 estágios e execução em ordem. Ele é baseado na arquitetura ARMv7-A e permite escalabilidade e controle sobre o consumo de energia, uma vez que possibilita o desligamento de qualquer um dos cores quando estiverem ociosos. O equipamento possui 2 GB de RAM com frequência de 480 MHz.

O sistema operacional instalado em ambos equipamentos é GNU/Linux. No PC utilizou-se Ubuntu com kernel $3.16.0-38$ e no MPSoC Debian com kernel $3.4.106$. O sistema de arquivos utilizado para os experimentos foi o *ext4*. A Tabela 1 apresenta as principais características dos ambientes experimentais.

	PC	MPSoC
Processador	Intel Core2Duo	ARM Cortex A7
Modelo Processador	E8400	AllWinnerTech SoC A20
Técnica de Fabricação (nm)	45	40
Frequência de Clock	3.0GHz	960MHz
Número de processadores	1	1
Cores/Processador (#)	2	2
TDP do processador (W)	65	0,25
Cache L1/Core (KB)	64 x 2	64
Cache L2/Core (KB)	6144	1024
Memória (GB)	6 (DDR2)	2 (LP DDR3)

Tabela 1: Configuração dos Ambientes

Também foi utilizado quatro dispositivos de armazenamento, dois SSDs e dois HDDs, a fim de cobrir diferentes características. Eles são apresentados na Tabela 2. Os nomes apresentados na primeira coluna da tabela, serão utilizados no restante do texto para referenciá-los. Em todos os experimentos foi utilizada a interface SATA II, suportada pelas duas arquiteturas e por todos os dispositivos. Foram realizados testes com os quatro dispositivos de armazenamento nos dois equipamentos, totalizando oito configurações.

	Tipo	Fabricante	Capacidade (GB)	RPM	Especificações Fabricante	
					Tensão (VDC)	Corrente (A)
HDD1	HDD	Western Digital	160	5400	5	0,55
HDD2	HDD	Seagate	500	7200	5	0,45
SSD1	SSD	Samsung	240	-	5	0,50
SSD2	SSD	Kingston	120	-	5	1,00

Tabela 2: Dispositivos de Armazenamento Utilizados

O benchmark utilizado para os testes foi o IOzone[1], escolhido por ser amplamente aplicado e por permitir a descrição de diversos padrões de acesso. Os experimentos foram realizados em cada uma das configurações *com* e *sem* o uso da buffer cache. Foram testados os seguintes padrões de acesso: escrita sequencial; escrita randômica; leitura sequencial; e leitura randômica.

Outro parâmetro avaliado nos testes foi o tamanho das requisições. Neste caso, foram utilizados 32 KB ou 4 MB. Portanto, em cada configuração, foram realizados 16 testes, totalizando 128 experimentos, sendo que cada experimento foi repetido 10 vezes para o cálculo da média. Para medição de potência foi utilizado o equipamento Dranetz PP-4300, que mensura tensão e corrente alternada (CA) de todo o equipamento [8].

4. Resultados

Nesta seção, serão apresentados os resultados obtidos durante a execução do benchmark, bem como da medição da potência instantânea consumida para a realização dos testes.

A Tabela 3 demostra a potência média dos equipamentos durantes os testes do benchmark.

Equipamento	Requisição	Cache	HDD1	HDD2	SSD1	SSD2
PC	32KB	off	60,071	56,240	59,247	59,987
PC	32KB	on	59,214	57,205	64,752	69,503
PC	4096KB	off	58,153	57,084	58,306	57,962
PC	4096KB	on	60,537	59,301	66,432	65,500
MPSoC	32KB	off	15,269	12,883	8,473	10,265
MPSoC	32KB	on	16,443	12,979	9,233	11,235
MPSoC	4096KB	off	16,931	13,659	8,661	10,954
MPSoC	4096KB	on	17,405	14,479	9,257	11,268

Tabela 3: Potência Média (W) dos Equipamentos

Estes valores foram obtidos através da média aritmética das medições de potência instantânea providas pelo equipamento de medição durante cada teste. A potência é analisada, uma vez que ela não depende da duração dos testes, ou seja, independe do tempo dos testes realizados pelo benchmark.

Os dados de tempo foram alcançados de modo que o benchmark acessava 2 GB no dispositivo de armazenamento teste, estes acessos aconteciam com vários padrões, já discutidos na Seção 3, assim a Tabela 4 traz uma média aritmética do tempo de execução de todos os padrões. O tempo de execução do benchmark entre os dispositivos difere consideravelmente entre os HDDs em relação aos SSDs. No PC as diferenças de tempo de execução são mais

1 Disponível em http://www.iozone.org/

Equipamento	Requisição	Cache	HDD1	HDD2	SSD1	SSD2
PC	32KB	off	253,768	237,647	221,071	16,694
PC	32KB	on	17,449	18,037	16,500	3,432
PC	4096KB	off	36,756	28,541	8,306	8,743
PC	4096KB	on	13,400	8,752	4,125	4,018
MPSoC	32KB	off	275,091	252,950	44,589	58,651
MPSoC	32KB	on	101,812	104,275	33,622	34,881
MPSoC	4096KB	off	51,316	43,863	34,142	36,666
MPSoC	4096KB	on	37,004	36,043	32,124	32,006

Tabela 4: Tempo Médio (s) de Execução do Benchmark

evidentes, apresentando reduções de tempo entre 2 e 20 vezes. Já no MPSoC, estas diferenças ficam menos evidentes, apresentando tempos de execução muito semelhantes na maioria dos testes.

A Figura 1 apresenta valores obtidos quando os dispositivos estavam ociosos e executando o benchmark. A partir destes dados, notou-se que ao acessar os HDDs e SSDs no PC, a demanda de potência aumenta $3,7\%$ e $6,4\%$ respectivamente. No MPSoC, a potência aumenta $2,6\%$ e $68,4\%$ ao acessar HDDs e SSDs. Comparado ao PC, o MPSoC apresenta demanda de potência $74,2\%$ menor com HDDs e $83,7\%$ com SSDs.

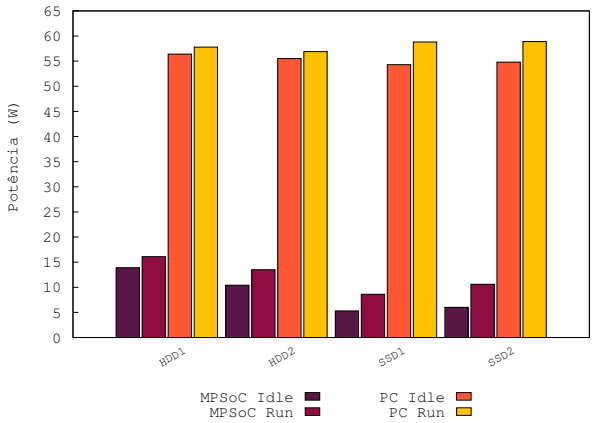


Figura 1: Potência Média em *idle* e Durante os Testes.

A energia consumida (J) de cada experimento foi calculada pela multiplicação da potência média mensurada (W) pelo tempo de execução (s), como mostra a Equação 1.

$$E = P_{avg} * t \quad (1)$$

Percebeu-se que o padrão de acesso não interfere na potência do equipamento, porém o padrão de acesso resulta em tempos de execução diferentes. Desta forma, almejando relacionar a energia consumida dos sistemas, dividiu-se o consumo do PC pelo consumo do MPSoC. Esta relação é apresentada na Figura 2.

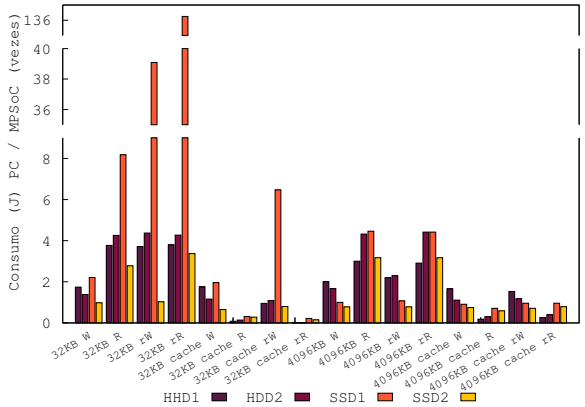


Figura 2: Relação da Energia Consumida pelos Sistemas

Foi observado que, utilizando discos rígidos para armazenamento, o uso do MPSoC leva a uma eficiência energética até $4,4$ vezes maior do que o observado usando o PC. Para leitura utilizando SSDs, o MPSoC leva a uma eficiência energética até 136 vezes maior. Essa diferença acontece porque, apesar do maior tempo de execução medido nesse ambiente, a demanda de potência é até $6,7$ vezes menor.

5. Conclusões e Trabalhos Futuros

Esse artigo apresentou uma análise de desempenho e eficiência energética de uma arquitetura de baixa demanda de potência - um MPSoC usando um processador ARM - para operações de E/S. Esses resultados foram comparados com um computador tradicional a fim de estudar a viabilidade do uso de arquiteturas de baixa potência como servidores de armazenamento. Esse estudo incluiu diferentes padrões de acesso e dispositivos de armazenamento a fim de cobrir diferentes situações e características.

Os resultados mostraram que a demanda de potência não é afetada pelo padrão de acesso. No entanto, este possui um impacto no tempo de execução, o que afeta o consumo de energia.

Concluiu-se que a substituição de um servidor de configuração PC + HDD por múltiplos servidores de baixa potência com SSDs seria viável e manteria um desempenho semelhante. Dependendo da carga de trabalho esperada, essa substituição diminuiria a demanda de potência, e consequentemente o consumo de energia, em até 85%.

Como trabalho futuro pretende-se expandir a análise apresentada a outros modelos de equipamentos de dispositivos de armazenamento. Além disso, serão avaliados ambientes distribuídos em que múltiplos servidores de baixa potência oferecem armazenamento.

Referências

[1] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. *Proceedings of the 17th annual international conference on Supercomputing - ICS '03*, page 86, 2003.

[2] R. Ge, X. Feng, and X. H. Sun. SERA-IO: Integrating energy consciousness into parallel I/O middleware. In *Proceedings - 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, pages 204–211, 2012.

[3] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003.

[4] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, et al. Exascale computing study: Technology challenges in achieving exascale systems. pages 1–297, 2008.

[5] P. E. McKenney, D. Eggeman, and R. Randhawa. Improving energy efficiency on asymmetric multiprocessing systems. Technical report, 2013.

[6] M. Nijim, A. Manzanares, X. Ruan, and X. Qin. Hybud: An energy-efficient architecture for hybrid parallel disk systems. *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, 0845257(2005), 2009.

[7] Z. Ou, B. Pang, Y. Deng, J. Nurminen, A. Yla-Jaaski, and P. Hui. Energy-and Cost-Efficiency Analysis of ARM-based Clusters. In *12th IEEE/ACM Cluster, Cloud and Grid Computing (CCGrid)*, Ottawa,Canada, 2012.

[8] E. L. Padoin, L. L. Pilla, F. Z. Boito, R. V. Kassick, P. Velho, and P. O. A. Navaux. Evaluating application performance and energy consumption on hybrid CPU+GPU architecture. *Cluster Computing*, 16(3):511–525, 2013.

[9] E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. A. Navaux, and J.-F. Mehaut. Performance/energy trade-off in scientific computing: The case of ARM big.LITTLE and Intel Sandy Bridge. *IET Computers & Digital Techniques*, 2(3):1–14, 2014.

[10] P. J. Pavan, R. K. Lorenzoni, J. L. Bez, E. L. Padoin, F. Z. Boito, P. O. A. Navaux, and J.-F. Méhaut. Análise da eficiência energética de operações de e/s com arquiteturas de baixa potência - (artigo submetido). In *XVII Simpósio de Sistemas Computacionais (WSCAD-SSC)*, pages 1–8, Aracaju, SE, 2016.

[11] P. Shang and J. Wang. A novel power management for CMP systems in data-intensive environment. *Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011*, pages 92–103, 2011.

[12] B. Welch and G. Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2013.

[13] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. *Proceedings of the twentieth ACM symposium on Operating systems principles - SOSP '05*, 39:177, 2005.

Memory Performance Comparison of Heap and Data Segments with Different Compiler Optimizations

Bruno Oliveira Cattelan, Lucas Mello Schnorr
Institute of Informatics
Federal University of Rio Grande do Sul
Caixa Postal 15.064 – CEP 91.501-970 – Porto Alegre – RS – Brazil

Abstract

It is usual for programmers to use at least the O1 optimization level when compiling C programs. However, sometimes this optimization works better with one kind of memory allocation than others. In this paper we show the execution time differences for two programs that rely heavily in memory access. For both programs were made two versions, one that allocated very big arrays in the data segment and the other in the heap. Using the assembly code produced by GCC we then compared the program versions, to try and understand the time execution differences.

1. Introduction

The GCC compiler is widely used in C programming. For this reason, it is imperative that we better understand it's behavior for different memory allocations when using the optimization levels.

This work has been inspired by reading a section in [3], in which was shown that the heap had a bigger cache miss rate than the other kinds of memory. If this was true, execution times would also show this.

To test this differences we created two programs, one that used big arrays and one that used big matrices. For each one we also made a version that allocated the variables in the data segment and one that used the heap. To prevent bias a full factorial approach was used, with the help of a R script. To understand the results, we studied the assembler code generated by GCC.

In methodology we explain how the experiments were executed, along with some details about the machine used and the programs created. In Results we show the resultant behavior of the programs and make some considerations. Finally, in the Conclusion we explain what the results showed us, along with future work.

2. Related Work

Many works have already been done in the study of the GCC optimization levels. A similar work has been done by [1]. Although they used a well known benchmark for their experiments, the work has no distinction between the different memory allocation types that a programmer may use, and their focus was on Integer variables. Our work uses double precision variables, and a closer inspection is made in the resultant program, with a study in the generated assembly code.

3. Methodology

All tests were executed in the following machine:

Parameter	Value
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	32
On-line CPU(s) list:	0-31
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	45
Stepping:	7
CPU MHz:	1202.968
BogoMIPS:	4001.13
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	20480K
NUMA node0 CPU(s):	0-7,16-23
NUMA node1 CPU(s):	8-15,24-31

To prevent bias, we used a full factorial with the size and the kind of memory allocation as factors. This was created using a R script with the DoE package, thoroughly explained in [2]. Also, to be sure that no NUMA effect was interfering with our experiment the following command was used:

```
GOMP_CPU_AFFINITY=0-31 numactl -i all
```

It prevents the OS from moving the threads between the processors and make all allocations following a round-robin policy, rather than by proximity to the processor that is going to use it. In both tests we created two programs, one that used the data segment to store the variables and one that used the heap. They were both compiled using from O0 to O3 optimization levels. Differences in the assembly code were verified using the program "Kompare".

3.1. Array

This very simple program adds the current index divided by the array size to the variable pointed by the index. This is done for the array size times. It is a sequential program, as shown below.

Data Segment code:

```
#include <stdlib.h>
#include "../../lib/hpcelo.h"

double bigArray[100000];

int main(int argc, char *argv[]){
 size_t SIZE = (size_t)atoi(argv[1]);
 HPCELO_DECLARE_TIMER;
 size_t i,j;
 HPCELO_START_TIMER;
 for(j=0;j<SIZE;j++)
  for(i=0;i<SIZE-1;i++)
   bigArray[i] = bigArray[i] + i/SIZE;
 HPCELO_END_TIMER;
 HPCELO_REPORT_TIMER;
}
```

Heap Code:

```
#include <stdlib.h>
#include "../../lib/hpcelo.h"

double *bigArray;

int main(int argc, char *argv[]){
 HPCELO_DECLARE_TIMER;
 size_t SIZE = (size_t)atoi(argv[1]);
 bigArray = (double*)calloc(SIZE,sizeof(double));
 size_t i,j;
 HPCELO_START_TIMER;
 for(j=0;j<SIZE;j++)
  for(i=0;i<SIZE-1;i++)
   bigArray[i] = bigArray[i] + i/SIZE;
 HPCELO_END_TIMER;
 HPCELO_REPORT_TIMER;
}
```

3.2. Matrix Multiplication

To better use the cache, our program multiplied the matrix by first calculating it's transpose. The execution time showed later however is only the multiplication, without the transposing time.

It is a parallel program that uses the openmp library. Since it's code is much more complicated than the Array, we decided to omit it.

4. Results

4.1. Array

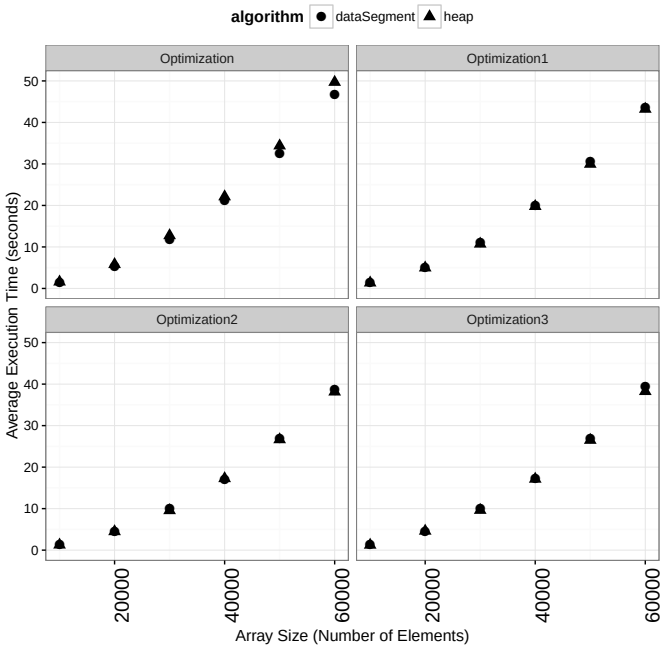


Figure 1. Array Execution Time

The data segment program for O0 shows a better execution time, as shown in 1. However, for O3 the heap is actually the fastest. For simplicity, we compared the assembler code only for O0 and O1. The differences are shown below.

Data Segment:

```
.L6:
  movq   -40(%rbp), %rax
  movsd   bigArray(,%rax,8), %xmm1
```

Heap:

```
.L6:
  movq   bigArray(%rip), %rax
  movq   -40(%rbp), %rdx
  salq   $3, %rdx
  leaq   (%rax,%rdx), %rcx
  movq   bigArray(%rip), %rax
  movq   -40(%rbp), %rdx
  salq   $3, %rdx
  addq   %rdx, %rax
  movsd   (%rax), %xmm1
```

Above we show the main differences in the main loop of the data segment program and the heap program using the O0 optimization level. The later clearly has not only more instructions, but also more memory accesses, which are the parentheses in the code.

Data Segment:

```
.L8:
  movq   %rcx, %rax
  movl   $0, %edx
  divq   %rbx
  testq   %rax, %rax
  js   .L4
  cvtsi2sdq   %rax, %xmm0
  jmp   .L5
```

Heap:

```
.L8:
  movq   bigArray(%rip), %rax
  leaq   (%rax,%rcx,8), %rsi
  movq   %rcx, %rax
  movl   $0, %edx
  divq   %rbx
  testq   %rax, %rax
  js   .L4
  cvtsi2sdq   %rax, %xmm0
  jmp   .L5
```

Above we show the main differences in the main loop of the data segment program and the heap program using the O1 optimization level. Although the heap program still has more instructions than the data segment, in 1 it showed a lower execution time. This might be explained by other differences in the rest of the code:

Data Segment:

```
.L5:
  addsd   bigArray(,%rcx,8), %xmm0
  movsd   %xmm0, bigArray(,%rcx,8)
  addq   $1, %rcx
  cmpq   %rsi, %rcx
  jne   .L8
.L7:
  addq   $1, %rdi
  cmpq   %rdi, %rbx
  jbe   .L2
.L3:
  testq   %rsi, %rsi
  je   .L7
  movl   $0, %ecx
  jmp   .L8
```

Heap:

```
.L5:
  addsd   (%rsi), %xmm0
  movsd   %xmm0, (%rsi)
  addq   $1, %rcx
  cmpq   %rdi, %rcx
  jb   .L8
.L7:
  addq   $1, %r8
  cmpq   %r8, %rbx
  jbe   .L2
.L3:
  testq   %rdi, %rdi
  je   .L7
  movl   $0, %ecx
  jmp   .L8
```

Even though both codes have the same number of instructions, and that except for the ”jne” and ”jb” ones they are all the same, the data segment program has some much more complex addresses to calculate than the heap. This may explain the slightly better execution time of the heap. Also, this shows that the GCC compiler can optimize the heap in a more efficient way than the data segment.

4.2. Matrix Multiplication

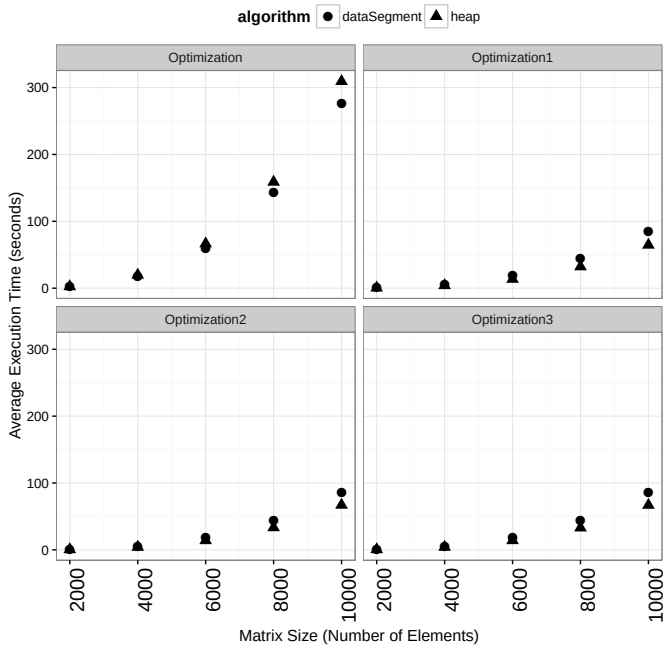


Figure 2. Matrix Execution Time

The program that allocates the matrix in the data segment clearly has a lower execution time than the one that allocates in the heap for O0, as shown in 2. However, as the optimization level is increased the difference becomes the opposite, with the heap being faster than the data segment.
Although this experiment showed the same behavior as the Array, it was made using a much more complex program. Even more, we used such large matrices that they did not even fit inside the L3 cache. With this we intended to rule out any possibility that the cache was responsible for the difference in the execution time of the programs.

5. Conclusions

For higher optimization levels, there seems to be better to use heap allocation. Not only it gives a better flexibility to the program, but also it gives smaller executables and higher efficiency. However, in cases that optimization may not be used, data segment allocation can have a positive impact in the program execution time.
For future work we would like to do a more thorough inspection in both programs, and also in new ones, that not only access the memory in different ways but also study the stack optimization efficiency. To better understand the effects, it would also be interesting to use trace tools like score-p or the intel pcm.

6. Acknowledgements

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law n 8248, 1991, and its subsequent updates.

References

[1] R. D. Escobar, A. R. Angula, and M. Corsi. Evaluation of gcc optimization parameters. *Ing. USBMed*, July 2012.

[2] U. Grmping. R package doe.base for factorial experiments. *Reports in Mathematics, Physics and Chemistry*, February 2016.

[3] H. Hadimioglu, D. Kaeli, J. Kuskin, A. Nanda, and J. Torrellas”. *High Performance Memory Systems*. Springer, 2004.

Tuning space optimization for stencil-based applications on multi-core

Víctor Martínez, Philippe Navaux
Informatics Institute (INF)
Federal University of Rio Grande do Sul (UFRGS),
Av. Bento Gonçalves, 9500, Campus do Vale,
91501-970, Porto Alegre, Brazil
{victor.martinez,navaux}@inf.ufrgs.br

Fabrice Dupros, Hideo Aochi
BRGM, 3 Av. Claude Guillemin, Orléans, France
{f.dupros,h.aochi}@brgm.fr

Márcio Castro
Department of Informatics and Statistics (INE)
Federal University of Santa Catarina (UFSC),
Campus Reitor João David Ferreira Lima, Trindade,
88040-970, Florianópolis, Brazil
marcio.castro@ufsc.br

Abstract

Stencil computations are fundamental to solve Partial Differential Equations (PDEs) used in numerical simulations. However, their performance is very sensitive to many optimization parameters such as architectural features, compiler flags, memory contention and multi-threading. Fine-tuning these parameters relies on finding the best configuration that results in optimal performance for a given stencil application. In this work, we improve performance of OpenMP stencil programs on multicore architectures through tuning input parameters. We create a large configuration set of input values (problem size, threads number, chunk size, scheduling algorithm), until we reach the peak of best performance, and analyze how these parameters affect the performance.

1. Introduction

The performance of HPC applications depends on many factors: architecture, code optimization, compiler and runtime frameworks. An example of HPC applications are stencil-based applications (nearest-neighbor), which are used to solve many problems related to Partial Differential Equations (PDE). Then, optimizing these computations improve performance in many simulations.

One extended methodology to get best performance is application tuning, in which several parameters are adjusted to achieve the best performance. But it depends on several variables of a large set: machine architecture, parallelization strategy, domain decomposition, compiler flags, scheduling and load balancing algorithms at runtime, etc. Finding best space of input parameters requires to search on this large set of configurations. In [3] the authors optimize stencil computations for multiple architectures (multicore and accelerators).

Our work is oriented to obtain the best performance on multicore architectures for stencil computations, that are implemented in diverse areas as electromagnetics, fluid dynamics and wave propagation (i.e., seismic simulations). This paper presents our approach to find best tuning space of input parameters for stencil computations. The paper is organized as follows. Section 2 discusses the fundamentals of our stencil based model on single seven-point Jacobi. Then, Section 3 presents the testbed used and explains the methodology of experiments. Section 4 discusses the performance of simulations and how we can obtain the best performance. Finally, Section 5 concludes this paper.

2. Stencil Model

2.1. Stencil Equation

The stencil model is given by the explicit 3D heat equation [4]:

$$\begin{aligned} B_{i,j,k} =& \alpha A_{i,j,k} \\ &+ \beta(A_{i-1,j,k} + A_{i,j-1,k} + A_{i,j,k-1} \\ &+ A_{i+1,j,k} + A_{i,j+1,k} + A_{i,j,k+1}) \end{aligned} \quad (1)$$

This seven-point stencil performs a single Jacobi (out-of-place) iteration. Thus, reads and writes occur in two distinct arrays (A, B), where each subscript represent the 3D index into array A or B. For each grid point, this stencil will execute 8 floating point operations [3]. The stencil sweep can be expressed as a triply nested loop presented in algorithm 1.

Algorithm 1 Pseudocode for stencil algorithm

```
1:  for each timestep do
2:      Compute in parallel
3:      for each block in X-direction do
4:          for each block in Y-direction do
5:              for each block in Z-direction do
6:                  Compute stencil(3D tile)
7:              end for
8:          end for
9:      end for
10: end for
```

3. Experimental Setup

In this section we present the configurations considered in this work: stencil algorithms, the multicore architectures and the input and performance vectors.

3.1. Stencil algorithms

In order to obtain best performance we used three algorithms implemented in [5] with optimizations explained in [1].

3.1.1. Naive: First algorithm is the standard implementation of the triple nested loops coming from the three spatial dimensions. This allows a very straightforward use of OpenMP directives. Unfortunately, this standard implementation offers a poor cache reuse.

3.1.2. Blocking: Second algorithm uses the cache blocking technique. The main idea is to exploit the inherent data reuse available in the triple nested loop of the elastodynamic kernel by ensuring that data remains in cache across multiple uses. Dependency between the velocity and the stress components is exploited to implement a space-time decomposition.

3.1.3. Skew: Third algorithm is also based on cache misses reduction. These new approaches decompose the stencil using both the space and the time directions. Indeed, the spacetime domain is computed in a specific order, which means that the computations begin with the subdomains closest to the left boundary and then extend to the right boundary to honor the data dependencies [2].

The advantage on Blocking and Skew algorithms is to improve the computational intensity by keeping a relatively small amount of data in cache memory and by performing many more floating-point operations on them.

3.2. Experimental testbed

We used two multicore architectures to run our experiments: one machine with a single Intel Haswell processor (single socket) to avoid cache transferences and one NUMA platform composed of 4 Intel Nehalem processors. Configurations of our testbed are listed in Table 1.

	Node 1	*Node 2*
Processor	i5-4570	Xeon X7550
Clock (GHz)	3.20	2.0
Cores	4	8
Sockets	1	4
Threads	4	64
L3 cache size (MB)	6	18
Compiler	gcc-5.3.1	gcc-4.6.4

Table 1: Experimental testbed configurations.

3.3. Input and performance space

For each stencil experiment, we created one configuration vector and obtained one performance vector. Input parameters are related to code optimization and execution runtime. Fore code optimization we use one parameter for parallel looping in OpenMP (`omp task` or `omp parallel for`). For the runtime, we used two parameters for thread counting (total available threads and used threads), one parameter for the chunk size and one parameter for scheduling policy used by OpenMP (static, guided and dynamic). Each one of input configurations were executed 15 times to compute the averages. We used PAPI library [6] to measure cache-related information.

The performance measures are obtained with following values: Total cache misses L3 (`PAPI_L3_TCM` event),Total

cache access L3 (`PAPI_L3_TCA` event), Time (which corresponds to the total execution time to solve the stencil), Performance in GFLOPS (obtained from the execution time and stencil size).

4. Experimental Results

4.1. Algorithms

We first analyze the impacts of algorithms on the performance. Each algorithm presents a different performance for each machine. Figure 1 shows that best performance on node 1 is achieved with Skew algorithm (left) whereas on node 2 the Naive algorithm achieves the best performance. Blocking and Skew algorithms showed performance losses.

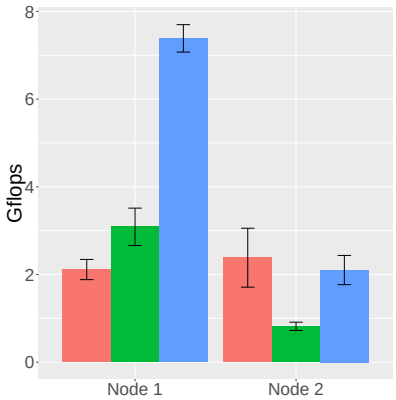


Figure 1: Performance (left) for each stencil algorithm: Naive (magenta), Blocking (green) and Skew(cyan).

As it can be observed in Figure 1, we obtained the expected behavior on node 2: better performance is quite related to the amount of L3 cache misses, as we can see low rate of cache misses is presented with high Gflops values and poor performance has high number of cache misses. We found that Skew and Blocking algorithm reduces cache misses significantly as it was reported in [5].

4.2. Scalability

We now analyze the scalability of the algorithms on node 2. The results presented in Figure 2 (left) show an expected behavior: when the number of threads is increased the performance of Naive algorithm also increases.

The Skew algorithm showed limited scalability on node 2, reaching a peak performance with 8 threads. The Blocking algorithm presented poor scalability on both platforms. We analyze this unexpected behavior with the other input parameters.

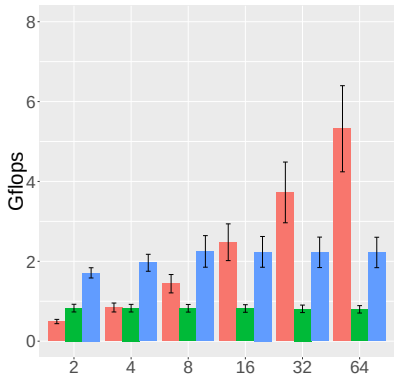


Figure 2: Performance for different number of threads on node 2. Algorithms: Naive (magenta), Blocking (green) and Skew(cyan).

4.3. Code optimization: parallel loop vs tasking

Our 3D stencil is calculated by three `for` instructions, as explained in Section 2. In this section, we compare two possible parallel implementations: i) we collapsed all `for` loops and performed the parallelization with `#pragma omp for`; and (ii) we used `#pragma omp task` in the inner `for` to create parallel tasks. Figure 3 presents the results for each node using the maximum number of available threads in each platform.

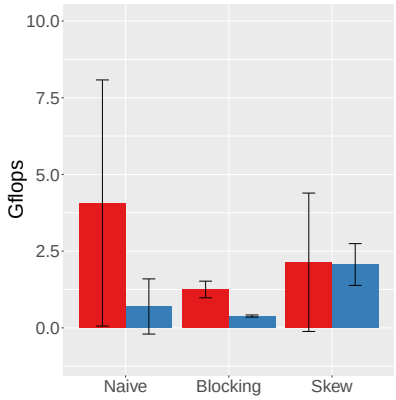


Figure 3: Performance for code optimization on Node 2. Method: parallefor (red) and Tasking (blue)

As it can be observed, the `parallel for` implementation achieved better performance than tasking in most of cases. The main reason is twofold: it creates a lot of tasks that need to access more cache levels and threads have to synchronize with each other in the `taskwait` clause. To

confirm this fact we traced the execution with pajeNG[1] and found that the OpenMP implicit barrier takes more time to synchronize all threads.

4.4. Scheduling

Loop scheduling on OpenMP is defined by two parameters: chunk size and policy [7]. Chunk size defines number of loop iterations to be assigned to each thread. Now, we analyze how this parameter influences the overall performance. For small chunk size we have more data communication between threads. We analyze scheduling on Node 2 (more cores, more threads, more cache levels and communications).

Figure 4 presents results of changing chunk size and scheduling for Naive algorithm. Three strategies are available in OpenMP: dynamic, guided and static. Then we found that the scheduling policy does not affect the performance for guided scheduling, while dynamic and static achieve better performance when the chunk size is increased.

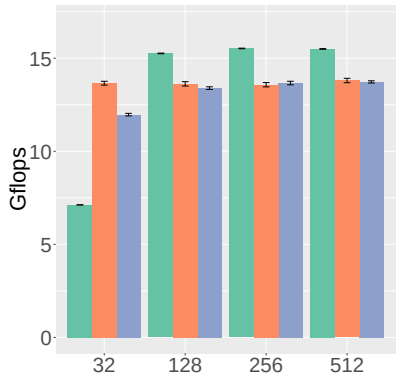


Figure 4: Performance for different scheduling with `omp parallel for` on Node 2: Dynamic (green), Guided (orange) and Static (gray). Algorithms: Blocking (left), Naive (center), Skew (right).

5. Conclusions and Future Work

In this work we studied the influence of several configurations and algorithms on the performance of stencil computations. We observed that two known algorithms (Blocking and Skew) may present poor scalability in several scenarios. Moreover, we observed that tasking achieves good performance when the algorithm does not use cache intensively (Skew). Chunk size and scheduling algorithms play an important role and can contribute to achieve a peak of performance if threads do not perform intensive data communications. As a future work, we intend to develop an auto-tuning approach to automatize the choice of the input parameters. One possibility is to use Machine Learning algorithms to perform that task.

1 https://github.com/schnorr/pajeng

6. Acknowledgments

This work have been granted by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and Bureau de Recherches Géologiques et minières (Institut Carnot BRGM). Research has received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E Project, grant agreement n° 689772.

References

[1] C. Andreolli. Eight optimizations for 3-dimensional finite difference (3dfd) code with an isotropic (iso). `https://software.intel.com/en-us/articles/openmp-loop-scheduling`. Accessed: 2016-01-01.

[2] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] K. Datta, S. W. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. *Scientific Computing with Multicore and Accelerators*, chapter Auto-Tuning Stencil Computations on Multicore and Accelerators. CRC Press, Taylor & Francis Group, 2010.

[5] F. Dupros, F. Boulahya, H. Aochi, and P. Thierry. Communication-avoiding seismic numerical kernels on multicore processors. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 330–335, Aug 2015.

[6] ICL. Papi reference. `http://icl.cs.utk.edu/projects/papi/wiki/Main_Page`. Accessed: 2016-01-01.

[7] Intel. Openmp loop scheduling. `https://software.intel.com/en-us/articles/openmp-loop-scheduling`. Accessed: 2016-01-01.

Measuring Hardware Counters for HPC Application Phase Detection

Gabriel Bronzatti Moro, Lucas Mello Schnorr
Institute of Informatics, Federal University of Rio Grande do Sul
Caixa Postal 15064 — CEP 91501-970 Porto Alegre – RS – Brazil
Email: *{gabriel.bmoro,schnorr}@inf.ufrgs.br*

***Abstract*—Besides reducing the execution time of parallel applications, the power consumption is an increasingly addressed problem in High-Performance Computing. A parallel program can be divided into regions, which can have particular characteristics, for instance, a behavior more CPU or memory bound. This paper presents a preliminary effort to measure performance counters along time to enable the automatic detection of memory-bound regions. Once attained, the automatic detection differs from the previous works since it does not require any code instrumentation, making it less intrusive. Three NAS Parallel Benchmark (NPB) applications are used in the experiments: the Discrete 3D fast Fourier Transform (FT), the Lower-Upper Gauss-Seidel solver (LU), and the Conjugate Gradient (CG). As hardware counters, we measure the L2 and L3 cache misses rate along time using the likwid's timeline mode, a tool to measure hardware counters from the user space. The experiments enable us to identify possible memory-bound code regions by correlating with changes in cache misses rates. We intend to configure a suitable processor frequency for each memory-bound parallel region of the application, reducing the energy consumption of the application with minimal performance loss.**

I. Introduction

Large HPC applications are composed of many parallel regions that are executed by different threads. For example, in an application that simulates the heat transfer through of a metal plate, we could define two parallel regions: the first to define the initial condition of the plate and another part would be responsible for calculating the heat transfer across different points of the plate for a number of timesteps. Each parallel region has its own characteristic. Some may be considered memory-bound, with a high rate of cache misses, while others may be considered more CPU-bound, with a high instruction execution rate, and even others might be considered IO-bound. In the previous example of the heat plate, one could measure the hardware counters at a given frequency to define the main characteristic of each parallel region. An automatic detection of memory-bound parallel regions enable one to adjust the processor frequency, possibly reducing energy consumption with minor performance penalties in execution time.

The main objective of this work is to measure hardware counters at every given time interval to discover memory-bound code regions. Once these regions have been detected, we intend to apply Design of Experiments screening techniques [1] to find the best processor frequency configuration for each region, pretty similar to what has been done already [2], but automatically. This paper presents our preliminary results by showing a time-oriented method to collect hardware counters, using likwid's timeline mode [3]. The preliminary results indicate that the collected data can possibly enable the automated identification of memory-bound code regions.

The paper is organized as follows. Section II presents related work regarding automatic phase detection for HPC applications. In Section III is presented our proposal and its corresponding methodology. The Section IV describes the platform used in the experiments and the preliminary results. Section V concludes the paper listing the main contributions and future works.

II. Related Work

There is no definitive solution to detect if a code region is more memory or CPU bound. Some works focus more on phase detection to sequential applications [4][5]. Spiliopoulos et al. [4] present in his work a tool that analyzes the behavior of a sequential application by detailed analysis of its execution phases, based on cache misses of the different levels of cache. The identified phase may be comprised of a set of program functions, which are grouped by having a similar behavior. The identification of the functions' behavior is performed in accordance with a prior history of execution, based on the trace generated by the application. The tool generated identifies the best processor frequency to be used in each phase to best performance and reduce energy consumption. This paper analyzes only sequential applications, in this perspective the identification of the memory-bounds regions areas can be obtained in a coarser granularity in the interval between samples timesteps. As for parallel applications running different flows may have different behaviors which may vary according to the application of load balancing, which causes the granularity of the samples is thinner for better understanding its behavior. In addition to this approach, Laurenzano et al.[5] present an approach finer granularity for identifying the most appropriate processor frequency for each application loop. Through multiple executions is set a model of various sizes, it allows to find for example the most appropriate setting for the given bond program, varying aspects that can influence the energy reduction and performance improvement for the application.

There is also investigation to consider MPI or OpenMP parallel applications. Freeh et al.[6] present an approach to define the most suitable frequency for each phase of an MPI application. Among the available frequencies, the approach

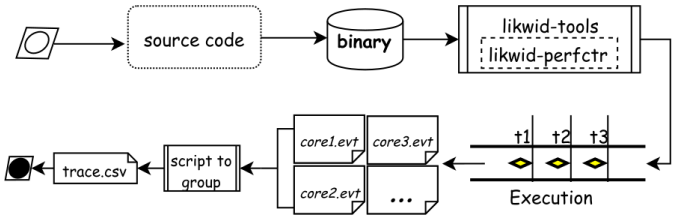
looks at what is the best frequency for a given node operate during the execution of the application. Another approach [2], focused in OpenMP applications, analyzes the parallel regions of a program using design of experiments and screening designs. According to their analysis based on seven benchmarks, it is possible to reach a considerable gain in energy reduction, eventually with no performance loss, depending on the characteristics of the benchmark. Their approach use manual instrumentation to identify the code regions that are going to be analyzed. We focus instead in the automatic detection of such regions based on hardware counters. In this paper we show our investigation in how these counters can be measured along the application execution.

The next section describes our measurement and evaluation methodology to collect hardware counters at a given frequency.

III. Measurement and Evaluation Methodology

The methodology used in the work first defines the compilation a source code into binary. The program is run under the likwid-perfctr tool that allows you to collect hardware counters for each processing core. The data is processed by a script tailored to generate a detailed application trace to carry out the data analysis. The Figure 1 shows an overview of the methodology with all such steps.

Fig. 1. Overview of the methodology.



We employ such methodology in three NAS Parallel applications: the 3D Discrete Fast Fourier Transform (FT), the Lower-Upper Gauss-Seidel Solver (LU) and the Conjugate Gradient (CG). The applications are executed with 32 threads using the input size (class B) of the NAS benchmark. The execution platform used was the beagle1, a Workstation with 2 processors Intel (R) Xeon (R) E5-2650 CPU 2.00 GHz, each with 8 physical cores and Hyper-Threading technology.

The hardware counters are collected using likwid's timeline mode [3], configured to measure L2 and L3 cache misses rate at a given time interval. Such interval between each metric recording is defined according to the total execution time of each application. For example, in the FT application, the measurement period is defined as 30 milliseconds, generating about 172 samples (for each of the 32 threads). For the LU application a 100 milliseconds is adopted, for 363 samples. For CG, a period of 50 milliseconds for 384 samples. According to the likwid authors, adopting a period less than 100 milliseconds might generate non-valid results. Even so, the global behavior is still valid if one aggregate such information along time. We report FT and CG results under this limitation in order to investigate how frequent measurements impact our analysis of phase detection.

IV. Preliminary Results

We present the L2 and L3 cache misses rate considering the aggregated metrics for all cores of the two processors where we conducted experiments. Points in the plots represented such aggregated values, while lines are there only to show the metric trend along time. Despite the fact that we aggregated values, we have looked to each core cache level miss and they are all similar and homogeneous (because of the regular nature of the applications we used), justifying such aggregation to simplify the analysis.

A. Discrete 3D Fast Fourier Transform (NPB-FT, B Class)

Figure 2 depicts the L2 and L3 cache misses rate of the Discrete 3D Fast Fourier Transform (NPB-FT, B Class) when measuring metrics every 100 milliseconds. It clearly shows phases – represented by the peaks at regular intervals – when taking into account the L2 cache misses rate. For the L3 cache misses we observe that after the initialization phase (where a peak of 37% L3 misses rate is measured), the rate decreases towards zero with minor outliers. The highest L2 cache misses rate found for FT is about 30% between 7.5 to 10 seconds late time execution, while generally we can see that the observed phases reach a level of 30% in L2 misses. The lowest L2 misses rate is measured at 10% during the initialization phase.

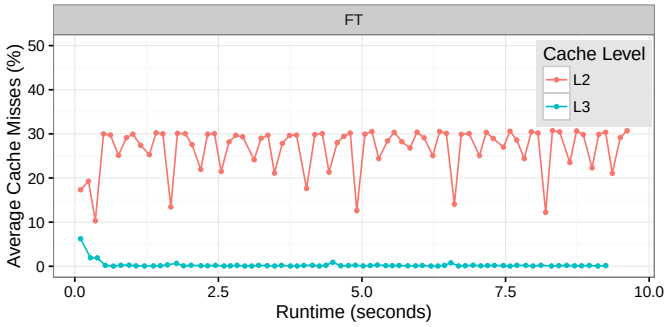


Fig. 2. The L2 and L3 cache misses rate of the Discrete 3D Fast Fourier Transform (NPB-FT, B Class) when measuring metrics every 100 milliseconds.

B. Lower-Upper Gauss-Seidel Solver (NPB-LU, B Class)

Figure 3 shows the L2 and L3 cache misses rate of the Lower-Upper Gauss-Seidel Solver (NPB-LU, B Class). Behavior is clearly different from the FT application, seen in Figure 2. We observe that the L2 cache misses rate fluctuates around 20%, while the L3 cache misses rate is about zero the whole execution, except during the initialization phase, where a 13% rate is observed.

C. Conjugate Gradient (NPB-CG, B Class)

Figure 4 shows the L2 and L3 caches misses rate for Conjugate Gradient (NPB-CG, B Class) application. After the initialization phase, behavior of both metrics becomes stable

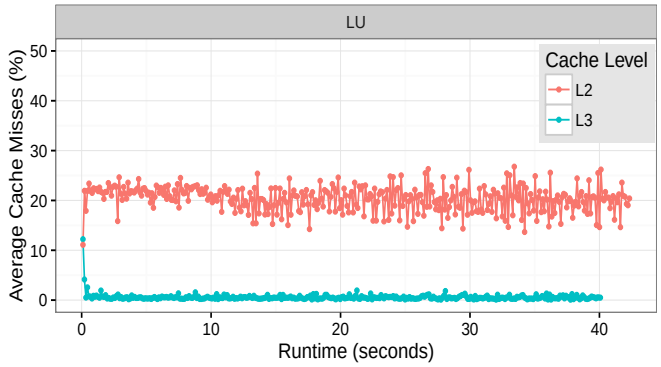


Fig. 3. The L2 and L3 cache misses rate of the Lower-Upper Gauss-Seidel Solver (NAS-LU, B Class) when measuring metrics every 100 milliseconds.

at about 38% of misses for L2, and around zero for L3. Comparing against the previous applications, we see that the L2 misses rate for CG is greater than the others, suggesting that it is more memory-bound. This could potentially lead to a more gains in energy reduction if a proper processor frequency is selected.

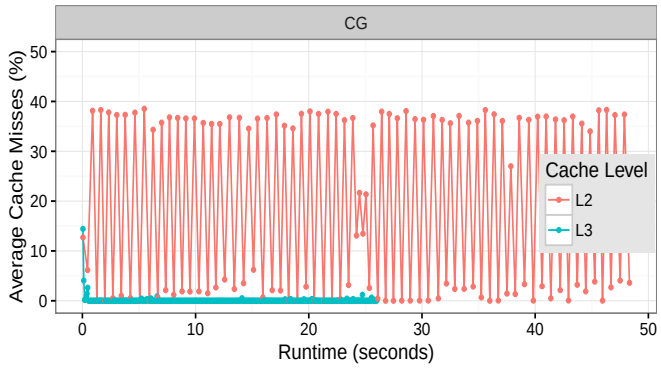


Fig. 4. The L2 and L3 cache misses rate of the Conjugate Gradient (NPB-CG, B Class) when measuring metrics every 50 milliseconds.

V. Conclusion

Parallel applications have many code regions, each one with unique performance characteristics. If one is capable to detect which regions are memory-bound, we might be able to reduce the processor frequency with DVFS-based techniques without affecting too much the execution time. The main goal of this study is to identify memory-bound code regions. The main contribution of this work is its methodology, which defines the steps and tools necessary that should be used to identify memory-bound portions of a parallel application. By applying such methodology, we have been able to measure L2 and L3 cache misses for three NPB applications, each one with different behavior. As future work, we plan to automatically identify those memory-bound regions based on the hardware counters we have measured so far.

Acknowledgements

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law nº 8248, 1991, and its subsequent updates.

References

[1] R. Jain, *Art of Computer Systems Performance Analysis: Techniques For Experimental Design Measurements Simulation and Modeling*. Wiley, 1991.

[2] L. F. Millani and L. M. Schnorr, "Computation-aware dynamic frequency scaling: Parsimonious evaluation of the time-energy trade-off using design of experiments," in *3rd International Workshop on Reproducibility in Parallel Computing (REPPAR)*, 2016.

[3] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.

[4] V. Spiliopoulos, A. Sembrant, and S. Kaxiras, "Power-sleuth: A tool for investigating your program's power behavior," in *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2012, pp. 241–250.

[5] M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. M. Tikir, and S. Poole, "Reducing energy usage with memory and computation-aware dynamic frequency scaling," in *European Conference on Parallel Processing*. Springer, 2011, pp. 79–90.

[6] V. W. Freeh, F. Pan, N. Kappiah, D. K. Lowenthal, and R. Springer, "Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 4a–4a.

A Comparative Study about Task Parallelism in OpenMP and Cilk

Guilherme Rezende Alles, Lucas Mello Schnorr
Institute of Informatics – Federal University of Rio Grande do Sul
Caixa Postal 15.064 – CEP 91.501-970 – Porto Alegre – RS – Brazil

***Abstract*—In this paper, we explore task parallelism as an approach to parallel programming. More specifically, two parallel frameworks - OpenMP and Cilk - are compared to one another with respect to their support regarding task parallelism. In order to do that, we will use an algorithm that provides not only task independency but also load imbalancing issues, to assess how well different parallel frameworks handle the resources available to them. The benchmark of choice for this study is the Mandelbrot set calculation, which is a classic problem for irregular workloads. By the end of this study, we seek to have a better understanding of the implementation of each runtime and also conclude which one offers the best performance of the two.**

I. Introduction

With the massive popularity of parallel processors and the stagnation of single core performance due to energy and heat dissipation constraints, the focus on parallel algorithms and parallel execution flow now demands programmers to worry about the parallel execution of algorithms, in order to increase performance and/or efficiency in modern chips. Consequently, the number of frameworks available that enable parallel software development has grown a lot, varying in many aspects such as complexity of development, structure, support for parallel programming paradigms, efficiency and performance. Thus, simpler parallel programming models are of major interest among software developers.

In this study, we compare two different parallel frameworks that support task parallelism: Intel Cilkplus and OpenMP with tasks. The emphasis of the comparison is on the performance of both frameworks, and we will be using the mandelbrot set calculation as the prime benchmarking algorithm. This algorithm was chosen based on it being a classic irregular workload problem. By implementing a problem of this nature, we are able to identify load balancing issues that may affect performance.

Considering that OpenMP and Cilk constructs for task parallelism present different semantics, we expect to produce different results when comparing them in the same conditions. At first glance, OpenMP presents itself as a lower overhead API (compared to Cilk) and, for that reason, should perform better than its counterpart.

The objective of this comparison is primarily to better understand the use cases in which Cilk and OpenMP differ from one another, and conclude which one offers a better tradeoff between performance and ease of use. Another goal is to understand the implementation decisions that were involved in developing each framework and what effects they have in overall performance.

II. Background and Experimental Context

A. Background

OpenMP[1] is a language extension that aims for providing parallel tools for programmers, in the form of compiler directives and a runtime library to command the creation and management of threads. Because OpenMP is fairly extense, it provides a large collection of directives and constructs that enable the programmer to express and explore both task and data parallelism. OpenMP supports a large variety of parallel programming paradigms, such as the SMPD pattern, loop parallelism, the divide-and-conquer paradigm and others.

Cilk[2] is a runtime library that extends the capabilities of programming languages (usually C or C++) to incorporate parallel programming constructs. Contrasting with OpenMP, Cilk's main objective is to provide parallel constructs in a simpler, more intuitive way for the programmer. On these grounds, Cilk is mainly focused on task parallelism and it does so by providing just a few keywords that allow the programmer to express parallelism.

For the sake of comparing the performance of both frameworks, it is important to understand how each one handles the given thread pool and what are the runtime implications of the thread management. While the OpenMP environment allows the programmer to explicitly control a fair amount of settings and runtime behaviours (such as the threads scheduler, shared variables and overall memory model), Cilk keeps this sort of control to its own runtime. Also, expressing parallel sections in Cilk code does not mean that the code will be certainly executed in parallel: the runtime will decide in execution time which parts of the code will be executed sequentially and which parts will not. This is done through a work-stealing routine that tries to avoid parallel execution of very small tasks, that would be executed much faster if done sequentially.

OpenMP, on the other hand, offers much less runtime support, forcing the programmer to explicitly write code that will create threads and a task pool to be executed in parallel. This approach theoretically allows for a lower overhead with the cost of a less intelligent runtime, which will execute everything that is determined by the programmer in parallel.

B. Experimental Context and Workload Details

The experimental design considers the mandelbrot set calculation as the benchmarking algorithm. This algorithm was chosen because we can leverage the irregular workload characteristic to assess how well the schedulers of the two frameworks can distribute work among worker threads, while also maintaining a somewhat closer to the reality scenario (as the majority of algorithms are not completely regular and with static tasks complexities). The Mandelbrot set algorithm was, thus, implemented in the C language, and the runtime settings for both Cilk and OpenMP are left in their default state. The number of threads created by the program is, thus, the amount of hardware threads available in the testing system. We used an R script to create a CSV file with a full factorial design, in which each experiment was replicated 20 times. The factors that are present in the design are the maximum number of iterations per point (which affects the irregularity of the workload) and the granularity of the problem (explained below).

With respect to the granularity of the problem, our Mandelbrot set calculation accepts a variable grain size as a parameter. This means that we can define an experiment with different grain sizes to also assess how well OpenMP and Cilk handle low, medium and high number of tasks in the task pool. This specific parameter is aimed at measuring the overhead involved in creating a task when compared to computing the Mandelbrot point in each framework.

The task creation is handled in the following way: a worker thread is assigned as responsible for creating the task pool. This thread then iterates through the discrete Mandelbrot space, creating tasks of equal grain size (using the constructs #pragma omp task in OpenMP and cilk_spawn in Cilk) and adding them to the pool. Idle threads are, then, able to execute those tasks. A synchronization construct (#pragma omp taskwait in OpenMP and cilk_sync in Cilk) ensures the threads finish the work properly.

The testing environment is a NUMA machine with two Intel Xeon E5-2650 processors, with a total of 32 hardware threads and 32 GB of RAM. No NUMA specific features were explored in this study.

Both implementations (using OpenMP Tasks and using Cilk) produce an image of dimensions 2000x2000, and the programs were kept as equal as possible. The estimated memory consumption for the mandelbrot calculation for this input is of 12 MB. The program was compiled using GCC version 6.1.0 and the optimization flag -O2.

III. Related Work and Motivation

While comparative studies between parallel frameworks have already been done in the past, the general focus of published papers is to present an overview about the performance on different use cases. For instance, [3] presents an analysis of different types of algorithms with different parallel programming paradigms such as divide and conquer and loop parallelization. The parallel frameworks studied in this paper are the Intel TBB, Cilk and OpenMP. Although the comparison between the three frameworks yielded solid results, the scope of this analysis does not allow for it to explore the runtime overhead of each approach, and explain which factors are able to impact performance the most.

[4], on the other hand, presents a comparison of task parallel frameworks using a highly unbalanced benchmark: the Unbalanced Tree Search. While this benchmark is more specific and the benchmark used is suitable regarding load imbalancing, the study lacks an explanation for the results that were observed. In the two papers, the general conclusion is that OpenMP is, in most cases, outperformed by Cilk.

[5] also presents an overall comparison between different task parallel frameworks, using the Mandelbrot set calculation as the benchmarking scheme. This paper provides an overview of the design and implementation of many frameworks (including Cilk and OpenMP), and cites many task parallelism issues such as synchronization and memory models, which were also observed in the current study.

IV. Obtained Results

By executing the Mandelbrot algorithm with a grain of size 1 (i.e. when for every point a task is created), we obtained the result shown in Figure 1.

This graphic presents a behaviour that was not expected at all: when using OpenMP, the execution time of the program was more than 10 times higher than its Cilk counterpart, and the standard error in the results is also very high (between 5 and 7 seconds, on average). Additionally, there seems to be no relation whatsoever between the number of maximum iterations per point. In fact, the execution time is lower when the number of iterations increases, which may indicate that the work is not being equally distributed among threads, and that the time of each individual thread is not being spent with calculation. These results clearly state that something in the OpenMP runtime is preventing the threads to work properly. In order to investigate this matter more closely, we used the Score-P profiling and analysis tools to trace the execution of the OpenMP program, which yielded the plot presented in Figure 2.

By analysing this trace, we can see that the threads spend most of the time blocked by an implicit barrier imposed by OpenMP, which prevents them from calculating the Mandelbrot set properly. Another observation is that the workload is not well balanced among all available threads (as we had assumed before, and this is caused probably by the threads being blocked most of the time.

In order to distribute work more equally between threads and reduce the irregularity of the workload, we benchmarked the program using a grain size equivalent to one line of the Mandelbrot fractal. The results are shown in Figure 3.

As we can see, by increasing the grain size, the behaviour of the program is somewhat normalized. This result can indicate two things at the same time: OpenMP is not able to efficiently handle tasks that are too small and possibly irregular, and also that the excessive amount of tasks can massively degrade the performance of an OpenMP application. This behaviour was

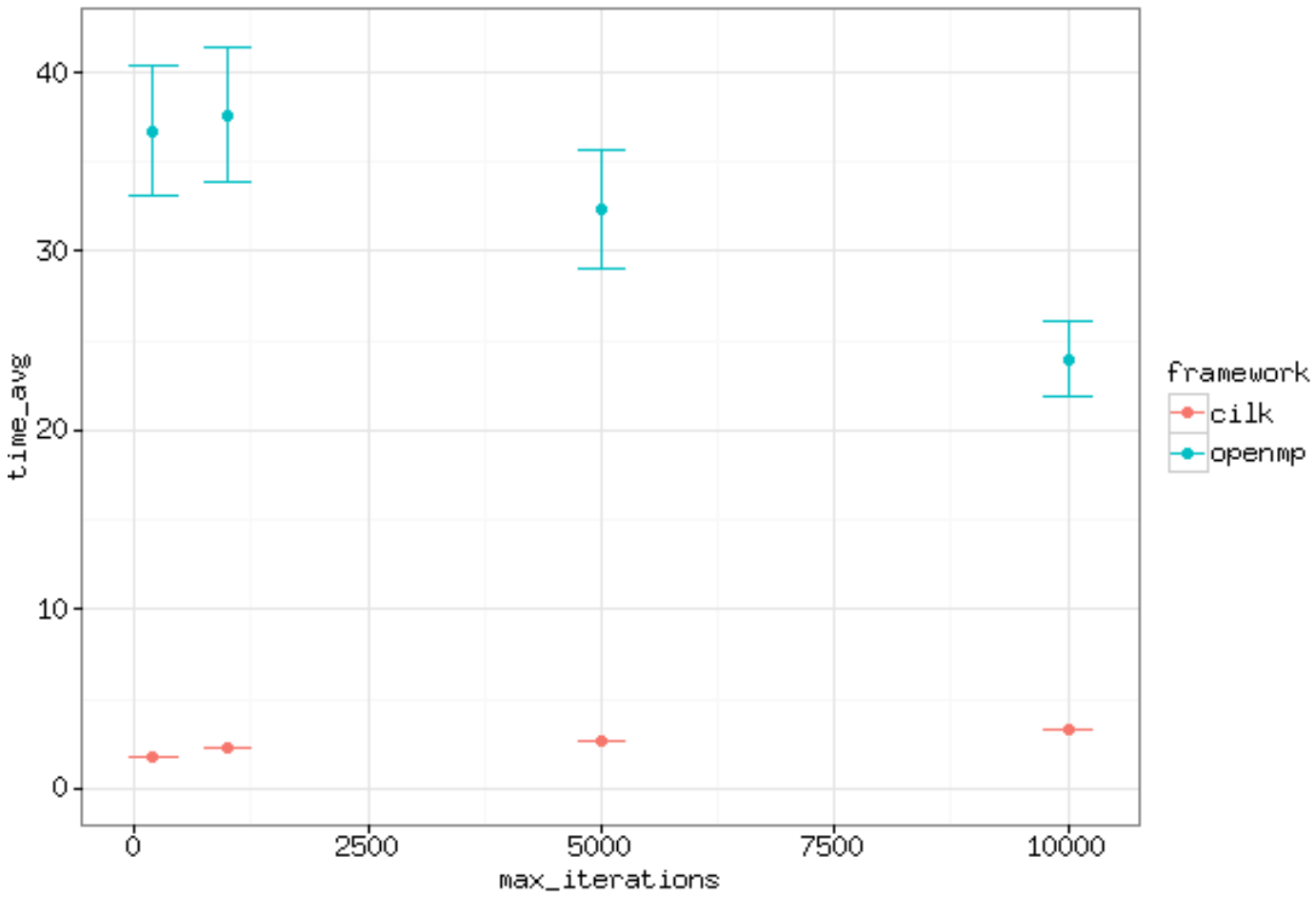


Fig. 1. Average time comparison considering a task for each mandelbrot point

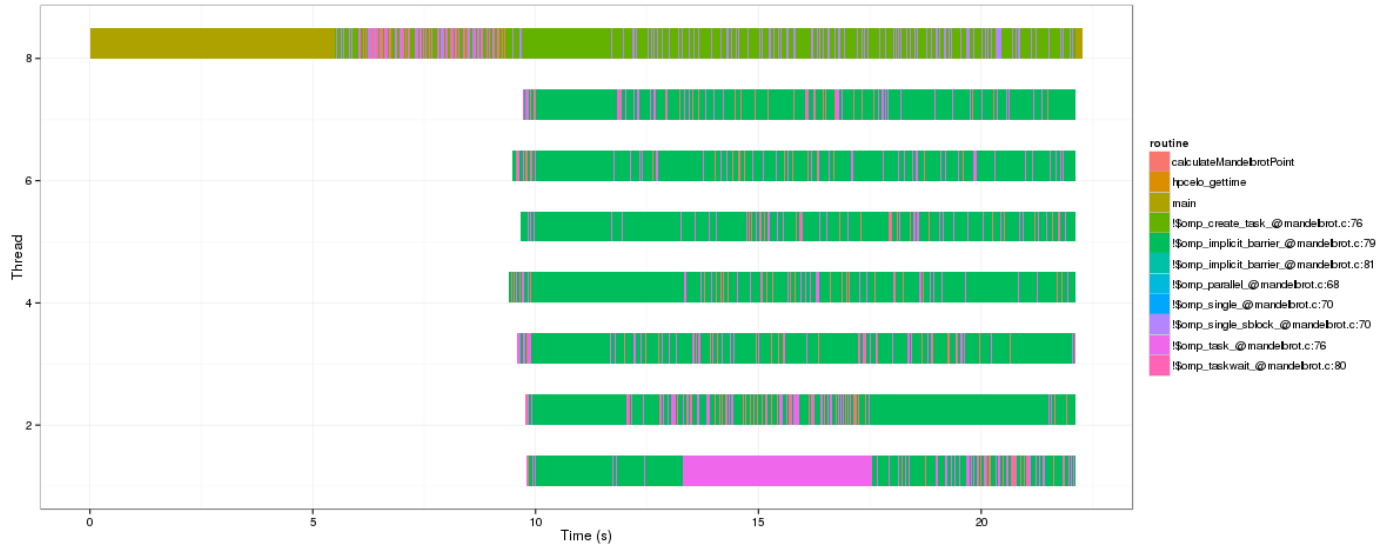


Fig. 2. Trace of the execution of the application using 8 threads

not observed in Cilk, though, probably because of the way that the runtime handles task creation and manages the task pool (i.e. by deciding in execution time whether the tasks will be executed in parallel or not). When the number of tasks created by the program is not too big, both OpenMP and Cilk offer similar performance.

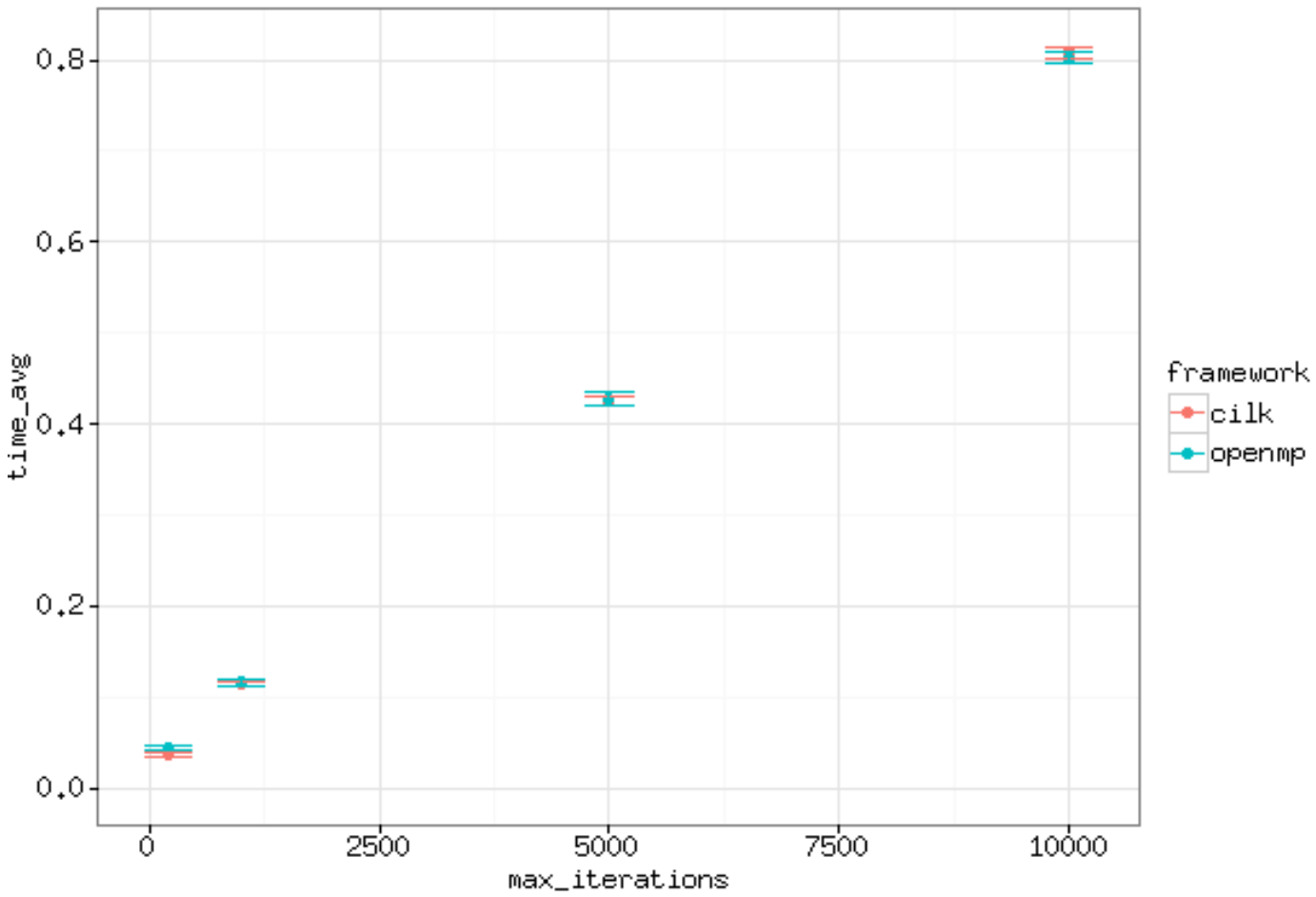


Fig. 3. Average time comparison considering a task for each line of the mandelbrot fractal

V. Conclusion and Future Work

By analysing the performance offered by OpenMP and Cilk when task parallelism is concerned, we were able to identify that the OpenMP runtime is very sensitive when handling the creation of tasks, that is, the programmer needs to be very careful with the task constructs that they insert on the code, because they might be introducing barriers that can ultimately be slowing down the execution of the program. When considering a controlled amount of tasks (which, in this study, was simulated by a coarser grain size), both the performances of OpenMP and Cilk are comparable to one another.

This comparison, however, has proved itself incomplete, especially with regards to the OpenMP barriers that are implicitly inserted in the code, which does not allow for a fair comparison between the two frameworks. As a future study, we seek to understand what is causing these barriers to occur and how to avoid them, in order to increase the overall quality of the experimental design.

Acknowledgements

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law nº 8248, 1991, and its subsequent updates.

References

[1] "Openmp specification," in *http://openmp.org/wp/*.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Journal of Parallel and Distributed Computing*, 1995, pp. 207–216.

[3] A. Leist and A. Gilman, "A comparative analysis of parallel programming models for c++," in *The Ninth International Multi-Conference on Computing in the Global Information Technology*. IARIA, 2014, pp. 121,127.

[4] S. L. Olivier and J. F. Prins, "Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs," in *Int J Parallel Prog*. Springer, 2010, pp. 341–360.

[5] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, "Task parallelism and data distribution: An overview of explicit parallel programming languages," in *25th International Workshop on Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2012, pp. 174–189.

Quick Introduction to Quality of Context

Hélio Brauner and Claudio Geyer

Universidade Federal do Rio Grande do Sul
GPPD
Instituto de Informática Av. Bento Gonçalves, 9500 Porto Alegre, RS, Brazil
Email: {hcbfilho, geyer}@inf.ufrgs.br

Abstract

With the increasing number of IoT capable sensors being deployed, the need for application programmers to be able to use easy selection methods in order to get adequate sets of sensored data providers to fulfill application requirements became a main concern. It's in this context that the concept of Quality of Context(QoC) arose. This paper aims to introduce some of the basic concepts of QoC, along with two methods used to calculate QoC rankings, as well as presenting open challenges in the area.

1. Introduction

In the last few years, the rise of IoT devices began to motivate new works and research. With estimations of up to 50 billion IoT devices connected by 2020 [3], it becomes especially important to understand the requirements for new applications taking IoT into account. In particular, the problem of selecting adequate sensors considering data sources amid billions of possibilities requiring the ranking of IoT sensors according to certain criteria and in reasonable time is one that must be faced in order to facilitate the implementation of applications capable of benefiting from IoT capabilities [8]. It was in this context that the concept of Quality of Context(QoC) was created.

This paper is organized according to the following structure: Section 2 introduces the basic concepts of QoC. Section 3 presents two methods for calculating QoC values. Section 4 discusses some of the open challenges in the area. Section 5 concludes the paper.

2. Quality of Context

Although the idea of evaluating the quality of information from context sensors has been discussed at least since 2000 [2], the first time it was presented as *Quality of Context* was in [1], where the ideas of how to describe the quality of context information provided by sensors and which criteria to use in order to make this description were introduced. According to [1], QoC should comprise information about criteria such as precision, probability of correctness, trust-worthiness etc. that allows the evaluation of information provided by sensors. It is also shown in [1] that the concept of QoC is related to Quality of Service (QoS) and Quality of Device(QoD), but differs from both.

In [4], a new criterion, *completeness*, was added, and a mathematical approach to calculating values for each of the criteria in order to enable ranking was introduced.

In [9], those definitions where revisited, and the definition that QoC should be assessed by obtaining a final value of confidence between 0 and 100% atrributed to the evaluated information provided by a given context provider was established.

In [6], QoC is also linked to the idea of not only having descriptive information about the quality of context data, but also a single value, obtained from the values of the criteria, that allows the ranking of context providers by quality of sensed data information.

Based on this concept, papers such as [7] and [5] implemented their methods to calculate QoC values. These methods are presented in the next chapter.

3. Calculating QoC

3.1. CASSARAM Comparative Priority-based Weighted Index(CPWI)

In [7], a method for calculating QoC based on the distance between a point given by the user and the sensors in a given set is presented. The sensors are plotted as points in a multidimensional space, where each space represents one of the context criteria and is defined in the space $\{0 \cdots 1\} \in R$). The user defined point is also plotted in this space, with values defined according to need. These values may vary in importance, given different weights through an interface where one can adjust sliding bars that vary from "least important" to "most important" in a gradual manner. The weights are then obtained by comparison of slider positions, the most important being given maximum weight and the others being valued proportionally according to this maximum.

That information is then calculated using the Weighted Euclidean Distance formula of CPWI introduced in [7]. The version introduced in that paper presented an error, omitting the first subtraction of the corrected version presented here:

$$(CPWI) = 1 - \sqrt{\sum_{i=1}^{n} \left[W_i \left(U_i^d - S_i^\alpha \right)^2 \right]}$$

Where *Wi* is the weight assigned through the sliders for criterion *i*, U_i^d is the user-defined ideal value for criterion *i* and S_i^α is the value attributed to sensor α (α being any sensor in the complete set of evaluated sensors) for criterion *i*.

This method is interesting because it has the possibility of weighting the criteria in a fashion that doesn't distort the importance of the criteria considered. However, it has the disadvantage of considering the closest matches as the best possible sensors, while sensors that exceed the required level of quality may be ranked lower.

3.2. General Weighted QoC Value Assignment (GWQoC) Method

In [5], a very simple method for calculating QoC is presented, based on [6]. It is not named in the original article, but will be referred to as GWQoC in this article. In the particular case presented by the paper, where QoC was used to indicate the best healthcare sensors for patients, only a few criteria were used, but the method might be extended to consider as many criteria as desired. The generalized version would be as follows:

$$QoC = \frac{\sum_{i=0}^{n} (C_i * W_i)}{\sum_{i=0}^{n} W_i}$$

Where *Ci* is the quality value for criterion *i* and *Wi* is the assigned weight for criterion *i*.

This method is simpler than the one presented in [7], but might allow distortions in the final results because the criteria are evaluated in a manner that allows criteria with smaller weight to compensate for criteria with higher weight. On the other hand, this method ranks sensors that exceed required values for given criteria higher than the ones which are found to be closest matches, which is an advantage when compared to [7].

4. Open Challenges

While the main aspects of QoC are well defined, and there are different solutions to calculate QoC, some problems haven't been solved yet. For instance, most of the literature presents solutions that use a given set of criteria and defines the criteria, but without a mathematical definition on how to value criteria, thus being an unadressed challenge.

Another important matter in this direction is to evaluate which of the criteria found in the literature are really important, and which ones might be presented as different criteria, but would be revealed as being the same thing if further analyzed.

Also, while there are some ways of calculating QoC, such as the two presented in this paper, there are no studies comparing these methods. It would also be beneficial to find a new method that combines the strengths and removes the weaknesses found in approaches such as [7] and [5].

5. Conclusion

In this paper, some of the fundamentals and a short history of QoC have been presented in Section 2, showing the need for methods that calculate QoC in a precise way.

Two methods, CPWI and GWQoC were presented in Section 3, and their main strengths and weaknesses were shown.

In Section 4 some of the open challenges were presented and briefly discussed. Those challenges include the mathematical definition of how to value the criteria presented in the literature, the analysis of these criteria in order to define which ones are important and which aren't, as well as finding redundant criteria to eliminate, and the comparison of methods to calculate QoC values, as well as the creation of new methods combining the strengths and removing the weaknesses of existing methods.

References

[1] T. Buchholz et al. Quality of context: What it is and why we need it. 2003.

[2] G. Chen, D. Kotz, et al. A survey of context-aware mobile computing research. Technical report, Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, 2000.

[3] D. Evans. The internet of things how the next evolution of the internet is changing everything. Technical report, Cisco, 2011.

[4] Y. Kim and K. Lee. A quality measurement method of context information in ubiquitous environments. In *2006 International Conference on Hybrid Information Technology*, volume 2, pages 576–581. IEEE, 2006.

[5] D. C. Nazário et al. Cuida: um modelo de conhecimento de qualidade de contexto aplicado aos ambientes ubíquos internos em domicílios assistidos. 2015.

[6] K. Paridel, D. Preuveneers, Y. Berbers, et al. When efficiency matters: Towards quality of context-aware peers for adaptive communication in vanets. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 1006–1012. IEEE, 2011.

[7] C. Perera, A. Zaslavsky, P. Christen, M. Compton, and D. Georgakopoulos. Context-aware sensor search, selection and ranking model for internet of things middleware. In *2013 IEEE 14th International Conference on Mobile Data Management*, volume 1, pages 314–322. IEEE, 2013.

[8] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE Communications Surveys & Tutorials*, 16(1):414–454, 2014.

[9] K. Sheikh, M. Wegdam, and M. Van Sinderen. Middleware support for quality of context in pervasive context-aware systems. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pages 461–466. IEEE, 2007.

Performance Evaluation of MPI Parallel Transfer in Microsoft Azure Cloud

Eduardo Roloff, Emmanuell Diaz Carreo, Jimmy Valverde Sanchez, Philippe Navaux
Federal University of Rio Grande do Sul - UFRGS
Informatics Institut
Porto Alegre - RS - Brazil
{eroloff,edcarreno,jkmvsanchez,navaux}@inf.ufrgs.br

Abstract

Since the firsts appearances of the cloud computing paradigm, it is attracting attention of the High-Performance Computing (HPC) community. Due to its characteristic of elasticity and unlimited amount of resources, combined with the pay-per-use charge model. These key concepts makes T̈he Cloudän interesting alternative as an environment for HPC applications. This work presents a first step of a project were we will investigate the capabilities of cloud computing for HPC. In this work we executed the MPI Exchange Benchmark among four different Azure Data Centers to study the behavior of the network in the same provider. Our results shown that, compared with a traditional HPC cluster, the cloud presents a 10 times performance loss, but with a predictable behavior.

1. Introduction

Cloud Computing has two main characteristics: the pay-per-use cost model and the elasticity [1]. These characteristics offer an interesting alternative for High Performance Computing (HPC) applications. Due to them, it is possible to have access to, virtually, any amount of resources in little time without upfront costs.

This work is one of the firsts steps towards a project that aims to explore opportunities for symbiotic, coordinated use of HPC and cloud computing infrastructures. The idea is, starting from an HPC infrastructure such as a traditional HPC cluster. The purpose is to identify cloud computing aspects that could be explored to provide a far-reaching, flexible (e.g., in terms of adaptation to fluctuating demands) and efficient environment for running HPC.

It is a common sense that the main bottleneck of Cloud Computing is the network performance, a very important aspect for HPC. Since MPI is an important standard for HPC communication [2] its performance need to be measured.

In this paper, we provide a evaluation of MPI Parallel Communication in the Microsoft Azure public Cloud.

We compared the same type of virtual machine (VM) instance among four different Azure Data Centers and used the machines during working hours and during the night.

Our intention was to verify the allocation time presents performance impact in the machines and if they have different performance levels among different Data Centers using the same instance. To have a baseline for comparison purposes we used the traditional cluster *econome* from the GRID5000 project.

Our results shown that the execution during the day and night has minimum performance difference and could be ignored. We also conclude that there a slightly difference in the network performance when compared the execution times between the different Data Centers.

This work is organized as follow. Section 2 presents the methodology used to conduct our work. In the Section 3 we present the results of the execution of the network benchmark among the Azure Data Centers. Finally, Section 4 presents a discussion of the work and talks about the future work.

2. Methodology

We performed experiments on one traditional cluster system as well as four Data Center locations of Microsoft Azure using the G5 VM instance. The G5 instance is a VM with 32 cores, composed of a E5-2698v3 CPU running at 2.3 GHz with 448 GB of RAM, there is no precise information about the network interconnection. The traditional cluster is the *econome* machine from *GRID 5000* and is composed of two 8-core processors, the network interconnection is 10 Gbit Ethernet.

In all environments, we create systems with 128 cores in total to maintain a comparable baseline. The total number of nodes were four, for the G5 machines, and 8 for the econome cluster. The locations of Microsoft Azure used were: West Europe (WEU), West USA (WUS), East USA

Machine name	Processor model	Cores per instance	Network	Location
Econome	E5-2660	16	10 Gbit/s	France
G5	E5-2698 v3	32	—	West EU East US Singapore West US

Table 1. Configuration of the environments used in the experiments.

(EUS) and Southeast Asia (SAS). To the best of our knowledge, all systems are running without Hyper-Threading. All environments use Intel processors of recent generations, at least the Sandy-Bridge family.

Table 1 contains an overview of the machines used in the evaluation. Although main memory sizes vary between different instance sizes, all amounts were sufficient for our experiments and are therefore not mentioned in the table.

We use the Intel MPI Benchmark Exchange test. According with the Intel MPI Benchmark manual *"Exchange is a communication pattern that often occurs in grid splitting algorithms (boundary exchanges). The group of processes is similar to a periodic chain, and each process exchanges data with both left and right neighbor in the chain."* This test measures both the bandwidth and the latency of the network.

We executed the tests five time on each environment and allocation. Each one of the experiments was performed with different message sizes, 0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, and 32768 bytes.

3. Results

Figure 1 shows the bandwidth results of the Exchange test in logarithmic scale. The cloud instances present the same behavior pattern of increasing the bandwidth when the package size increases. We could observe that we have a very predictable bandwidth capacity according with the increase of the package size.

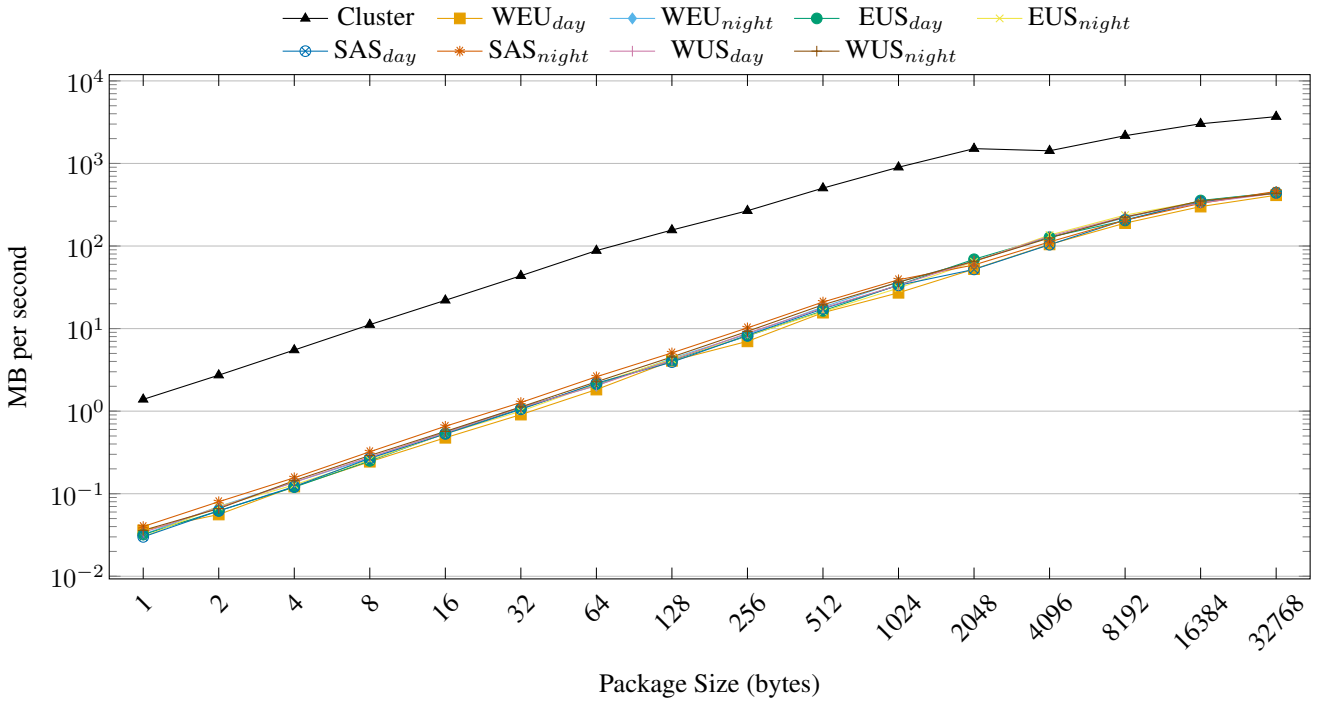


Figure 1. Bandwidth Results for Exchange benchmark.

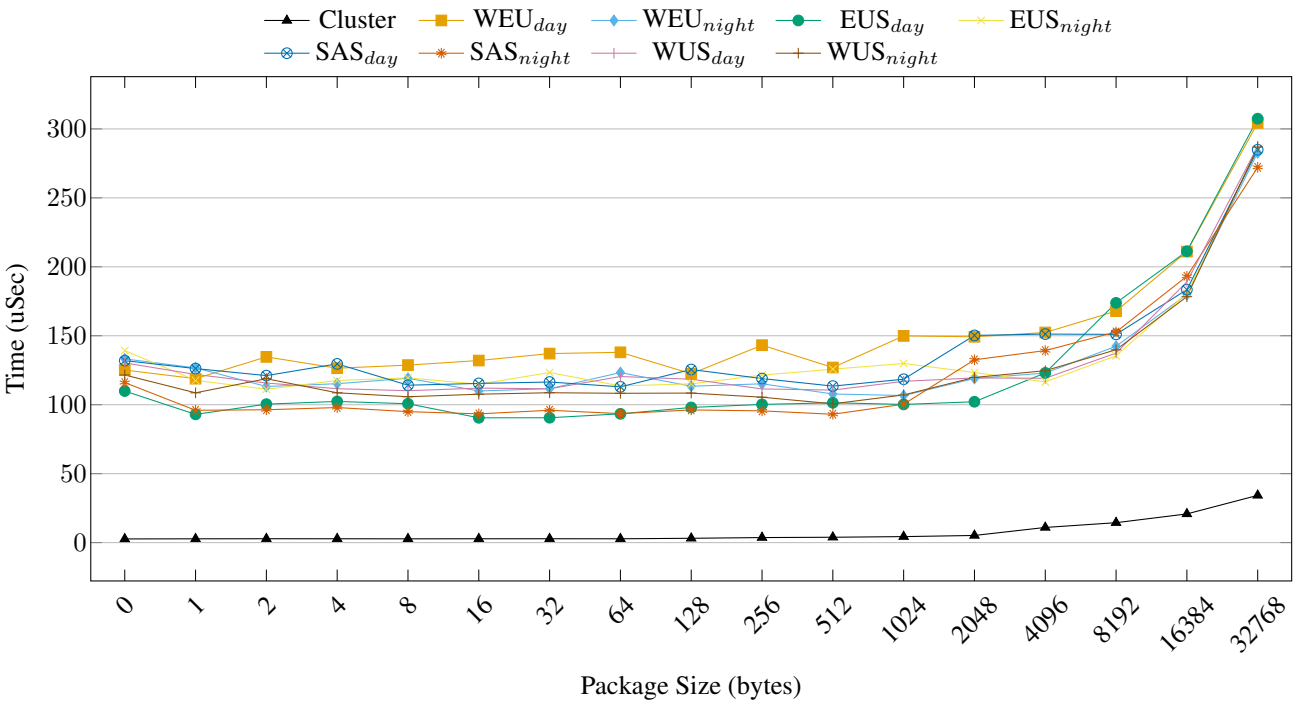


Figure 2. Latency Results for Exchange benchmark.

The bandwidth achieved by the cluster was 3 Gb/sec when the cloud allocations were around 350 Mb/sec for a 32 KB package. Comparing these numbers, we could observe that the cluster achieved a bandwidth around 10 times better than the cloud allocations. This indicates that the cloud network presents less capacitate at all as possible a certain level of contention, due to the virtualized and shared environment. However, the predictable behavior and the bandwidth increase of the cloud allocations could benefit the user when configuring his application to be executed in the cloud.

Figure 2 shows the latency results for the Exchange test. It is possible to verify that the cluster latency has a increase when the package size increases. This behavior could mean a level of network contention, the reason could be that this test performs a lot more concurrent communication in the network.

The cloud results showed a decent latency and a more predictable behavior, all the cloud instance allocations displayed the same pattern. It is interesting to note that all of the cloud allocations present a huge increase when the package size turns 8KB, and then all of them follow the standard pattern. This could be explained for a possible network optimization. The network is configured to handle a certain number of bytes at same time, for optimization, and when this number is reached the switches need to go to the controller to get a new configuration. This took some time, then the latency increases a little.

4. Conclusions and Future Work

In this works we performed an evaluation of the Exchange benchmark in the Microsoft Azure cloud. We shown that the cloud has a performance loss compared with a physical cluster and it is not ready for any HPC application. However for some kinds of applications, with a well defined communication pattern, it is suitable to be used.

As future work, we will expand this tests and provide a deep investigation of the cloud bottlenecks.

Acknowledgments.

This research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement no. 689772. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). Additional funding was provided by CAPES and Microsoft.

References

[1] R. d. R. Righi, V. F. Rodrigues, C. A. da Costa, G. Galante, L. C. E. de Bona, and T. Ferreto. Autoelastic: Automatic resource elasticity for high performance applications in the cloud. *IEEE Transactions on Cloud Computing*, 4(1):6–19, Jan 2016.

[2] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi. Using mpi in high-performance computing services. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 43–48, New York, NY, USA, 2013. ACM.

List of Authors