Análise Comparativa de MPI e OpenMP em Implementações de Transposição de Matrizes para Arquitetura com Memória Compartilhada

Lucas R. de Araujo, Crístian M. Weber
Fernando E. Puntel, Andrea S. Charão, João Vicente F. Lima

[1] Laboratório de Sistemas de Computação
Universidade Federal de Santa Maria

Abstract. *This work presents two implementations of a matrix transposition application, with MPI or OpenMP, to investigate the impact of these tools on performance in shared memory architecture. The results obtained from the analyzes show which variables influence the performance of the application and in what form they cause this impact.*

Resumo. *Este trabalho apresenta duas implementações de uma aplicação de transposição de matrizes, com MPI ou OpenMP, para investigar o impacto dessas ferramentas no desempenho em arquitetura de memória compartilhada. Os resultados obtidos a partir das análises realizadas mostram quais as variáveis influenciam no desempenho da aplicação e de que forma causam esse impacto.*

1. Introdução

A computação de alto desempenho motiva programadores a explorarem técnicas de paralelização de seus códigos para obtenção de execuções mais velozes. Um dos desafios da programação paralela se refere à maneira como os processos e/ou *threads* se comunicam. Esse fator é dependente da arquitetura em que o código a ser desenvolvido será executado. Arquiteturas de memória compartilhada entram nesse contexto e funcionam de maneira que todos os processos e *threads* possuem acesso direto aos dados.

No âmbito de programação paralela, duas ferramentas que se destacam são MPI e OpenMP. OpenMP é originalmente voltada para arquiteturas paralelas com memória compartilhada, enquanto MPI foi concebida para arquiteturas com memória distribuída, com possíveis otimizações em arquiteturas multiprocessadas. Em arquiteturas comuns em servidores atuais, com múltiplos processadores ou núcleos compartilhando memória, é possível executar aplicações desenvolvidas com OpenMP e/ou MPI.

Em um trabalho anterior [de Araujo et al. 2018], realizou-se experimentos com uma aplicação de computação científica denominada Incompact3d[1], que utiliza MPI e foi originalmente concebida para execução em *clusters*. Ao executá-la em arquitetura com memória compartilhada, que é atualmente mais acessível a alguns de seus usuários, observou-se *overhead* significativo na comunicação entre processos, durante sequências de operações de transposição de matrizes. Alguns autores (Seção 2) já compararam MPI e OpenMP e revelaram casos em que OpenMP ou MPI obtiveram melhor desempenho. Isso motivou uma investigação comparativa de desempenho entre MPI e OpenMP para a aplicação em questão, em arquitetura com memória compartilhada.

[1] https://www.incompact3d.com

Neste trabalho, busca-se explorar essa possibilidade por meio de dois programas que reproduzem as sequências de transposições realizadas no Incompact3d: um com MPI e outro com OpenMP. Por meio dessas implementações, objetiva-se obter métricas de desempenho sobre o uso de MPI e OpenMP, investigando causas de alguma implementação apresentar desempenho inferior a outra, principalmente no que se refere à comunicação de dados. Optou-se por externalizar as etapas do Incompact3d devido ao grande porte da aplicação (aproximadamente 30000 linhas de código) e, dessa forma, evitar inúmeras alterações sem ter indícios prévios de melhoria no desempenho ao implementar a aplicação de outra forma.

2. Trabalhos Relacionados

Diversos trabalhos na literatura abordam a análise de aplicações paralelas utilizando MPI e OpenMP de forma comparativa. Uma implementação bastante comum é a híbrida, que utiliza recursos das duas bibliotecas para paralelização. A implementação híbrida é a que tem aparecido em mais trabalhos ao lado da implementação em MPI. Um possível motivo para uso reduzido de OpenMP puro é devido a maneira como as *threads* acessam a memória nessa implementação (*fine-grained memory access*), então o desempenho acaba sendo afetado [Jin et al. 2011].

Em um dos trabalhos que exploram OpenMP puro [Krawezik and Cappello 2003], os autores avaliam alguns *benchmarks* implementados em MPI e em 3 estilos de implementação OpenMP: *loop level*, *loop level with large parallel sections* e SPMD (*Single Program Multiple Data*). Eles concluem que o estilo de implementação em OpenMP é determinante para um melhor desempenho da aplicação em relação a aplicação em MPI.

Já em um trabalho mais recente [Xu and Zhang 2015], os autores utilizaram uma arquitetura SMP (*Symmetric Multiprocessing*) e obtiveram êxito ao conseguir melhor desempenho usando uma abordagem híbrida. Esse caso mostra a maneira mais comum em que a implementação híbrida é utilizada: com MPI para comunicação entre os nós da máquina e OpenMP para comunicação dentro dos próprios nós. Basicamente, o MPI assume a comunicação em um nível de sistema mais distribuído e o OpenMP é utilizado na paralelização em nível compartilhado [Jin et al. 2011].

Apesar da predominância da implementação híbrida em trabalhos recentes, optou-se pela implementação em OpenMP puro (em comparação com MPI puro) pelo fato das execuções serem realizadas em uma arquitetura de memória compartilhada e possivelmente beneficiarem-se de tal arquitetura. Autores também citam implementação pelo compilador, implementação pouco intrusiva e a facilidade para controlar as diretivas como motivos para utilização do OpenMP [Bird et al. 2017].

3. Desenvolvimento

As duas implementações foram desenvolvidas na linguagem C e têm o mesmo esqueleto, que está representado em forma de fluxograma na Figura 1. Na etapa inicial, a aplicação cria uma estrutura de dados referente à distribuição das matrizes entre os processos (MPI) ou *threads* (OpenMP) e então aloca e inicializa as matrizes para sequência da execução. Na fase seguinte, ocorrem diversas iterações em que realizam-se etapas de cálculo e de transposição, alternadamente. É na etapa de transposição que se localizam as principais diferenças entre as implementações e que, devido às particularidades de cada uma, são

propagadas alterações nas outras partes do código, principalmente nas variáveis utilizadas e no compartilhamento de dados entre elas.

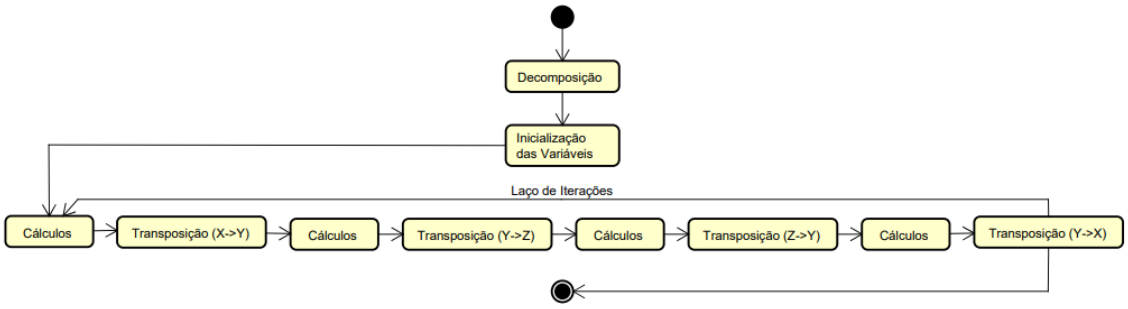


Figura 1. Fluxograma representando a execução da aplicação.

O desenvolvimento utilizando MPI teve como base o código do Incompact3d, passando por um processo de simplificação, mas sem perder a essência do código. Seu processo de transposição, assim como no Incompact3d, acaba tendo grande auxílio de estruturas de dados e de funções que o MPI disponibiliza. Os comunicadores criados durante a decomposição são estruturas que facilitam a identificação dos processos que devem compartilhar dados entre si em determinada transposição. A utilização de MPI_Alltoallv para o compartilhamento de dados acaba ocultando o funcionamento da troca de informações entre os processos, que se dá pelo método de passagem de mensagens.

Já a implementação desenvolvida em OpenMP teve o código baseado na aplicação de mesmo esqueleto em MPI. A transformação do código passou pela retirada e substituição de funções e estruturas de dados do MPI e a inserção de uma estrutura de dados compartilhada. O processo de troca de dados para realização da transposição fica bem mais claro no próprio código da aplicação e funciona de forma em que as *threads* inserem todos os seus dados em uma estrutura de dados compartilhada. Há a utilização de uma barreira para que todos os dados necessários estejam disponíveis e então as *threads* acessam a estrutura de dados em memória compartilhada para leitura dos dados necessários. Nesse processo de leitura dos dados, as *threads* utilizam índices calculados no particionamento original da matriz para acesso às posições corretas.

Ambas as implementações dispõem de um arquivo de parâmetros que facilita a instrumentação de cada execução. Nesse arquivo é possível definir o tamanho da matriz, o número de processos ou *threads*, a divisão da matriz, o número de iterações e o número de repetições dos laços de cálculos. Parte dessas variáveis serão exploradas nas execuções apresentadas na Seção 5.

4. Metodologia

O ambiente de execução da aplicação foi um servidor NUMA SGI UV2000 com 48 núcleos distribuídos em 8 processadores Intel® Xeon® CPU E5-4617 (2.90GHz de frequência) de 6 núcleos cada e 512GB de memória RAM. A memória cache da máquina se organiza da seguinte forma: cache L1 de dados e de instruções (32K cada), cache L2 de 256K e cache L3 de 15360K.

Os testes inicialmente visaram medidas de tempo obtidas através da própria

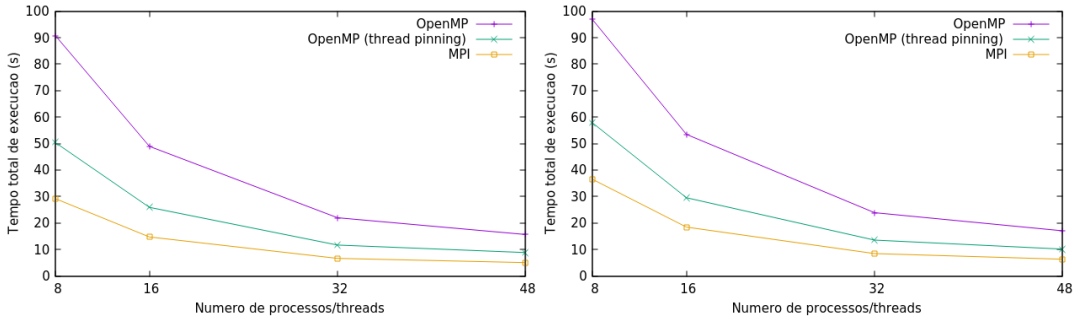
aplicação com a utilização de *clock_gettime* da biblioteca time.h. Posteriormente utilizou-se a ferramenta *perf*[2] (versão 4.9.30) para extração de dados de eventos de *software* e de *hardware* da máquina.

A ferramenta Score-P[3] (versão 4.0) foi utilizada para obter *logs* de informações do tempo ocupado pelas diretivas de barreira na implementação em OpenMP. Esses *logs* foram lidos na ferramenta Cube[4] (versão 4.4). Por fim, a aplicação LIKWID[5] (versão 4.3.0) foi utilizada para a tarefa de *thread pinning* da implementação OpenMP através de `likwid-pin`.

As configurações das variáveis do programa durante as execuções se dividem em 2 partes. Na primeira há uma malha fixa de 128*256*128 nós variando apenas o número de repetições do laço de cálculos e o número de processos ou *threads*. Já na segunda parte há uma diversidade maior de execuções, onde não há execução dos laços de cálculos e a grande variação encontra-se no tamanho da malha, que tem 11 tamanhos diferentes analisados, para diferentes quantidades de processos ou *threads*.

5. Resultados e Discussão

Após a implementação e revisão de ambos os códigos iniciou-se a fase de execuções. As execuções iniciais foram usadas para avaliar o desempenho das duas implementações, variando o número de cálculos e o número de processos ou *threads*. Os 2 parâmetros de cálculo utilizados foram 5 e 10, enquanto o número de processos ou *threads* variou nos valores 8, 16, 32 e 48. Já o número de iterações manteve-se fixo em 100, assim como tamanho da matriz manteve-se em 4.194.304 nós (128*256*128).



(a) Parâmetro de cálculos: 5 repetições (b) Parâmetro de cálculos: 10 repetições

Figura 2. Tempo de execução por número de processos (MPI) ou *threads* (OpenMP).

Os gráficos nas Figuras 2(a) e 2(b) representam as execuções da aplicação em MPI e OpenMP com as condições citadas acima. Além disso, os gráficos trazem um caso de OpenMP com *thread pinning* que será abordado a seguir. Os gráficos mostram que há um desempenho inferior da implementação em OpenMP em relação à desenvolvida anteriormente utilizando MPI. Na Figura 2(a), enquanto a aplicação em MPI varia de

[2]https://perf.wiki.kernel.org/index.php/Main_Page
[3]http://www.vi-hps.org/projects/score-p/
[4]http://www.scalasca.org/software/cube-4.x/
[5]https://github.com/RRZE-HPC/likwid

29,2 segundos com 8 processos para 5,02 segundos com 48 processos, a aplicação em OpenMP (sem *thread pinning*) varia de 90,8 segundos com 8 *threads* para 15,5 segundos com 48 *threads*. Apesar da diferença de tempo entre as implementações, o *speedup* com o aumento do número de processos/*threads* nas duas implementações e nos dois casos onde há variação da quantidade de cálculos é bem semelhante e varia entre 5,7 e 5,8.

Considerando o desempenho inferior da aplicação em OpenMP, utilizou-se a ferramenta *perf* para realizar o monitoramento de alguns eventos da máquina durante a execução da aplicação. Inicialmente, o objetivo de utilizar o *perf* era monitorar a memória cache durante a execução, devido ao grande volume de acesso e escrita em variáveis que ocorre no decorrer da execução. Porém, os resultados coletados em relação à memória cache não justificavam os tempos de execução obtidos anteriormente, visto que eram bastante semelhantes.

Dessa forma, outras duas variáveis observadas nos *logs* do *perf* apresentaram uma discrepância maior: trocas de contexto e migrações de CPU. A Tabela 1 apresenta as medidas para as trocas de contexto e a Tabela 2 apresenta medidas para migrações de CPU, ambas referentes ao caso que utiliza 5 cálculos. É possível observar, na maioria dos casos, um aumento de trocas de contexto e de migrações de CPU da aplicação em MPI para a aplicação em OpenMP.

# de processos/*threads*	MPI	OMP	OMP (*pin*)
8	6442	80876	96303
16	15995	100425	126223
32	49957	100003	124878
48	126206	104063	116888

Tabela 1. Trocas de contexto (Parâmetro de cálculos: 5).

# de processos/*threads*	MPI	OMP	OMP (*pin*)
8	80	15	59
16	127	771	68
32	251	5951	84
48	400	5404	100

Tabela 2. Migrações de CPU (Parâmetro de cálculos: 5).

Como possível solução para o problema de migrações de CPU elevado na aplicação em OpenMP, utilizou-se a ferramenta `likwid-pin` que permite ao usuário o *thread pinning* sem alterar o código da aplicação. Os resultados obtidos com a utilização de `likwid-pin` estão presentes na última coluna Tabela 2 e apresentam diminuição para casos com 16, 32 e 48 *threads*. Ainda que no caso de 8 *threads* haja aumento no número de migrações de CPU, na Figura 2(a) é possível observar que o tempo de execução utilizando `likwid-pin` também é inferior ao caso em que não se utiliza.

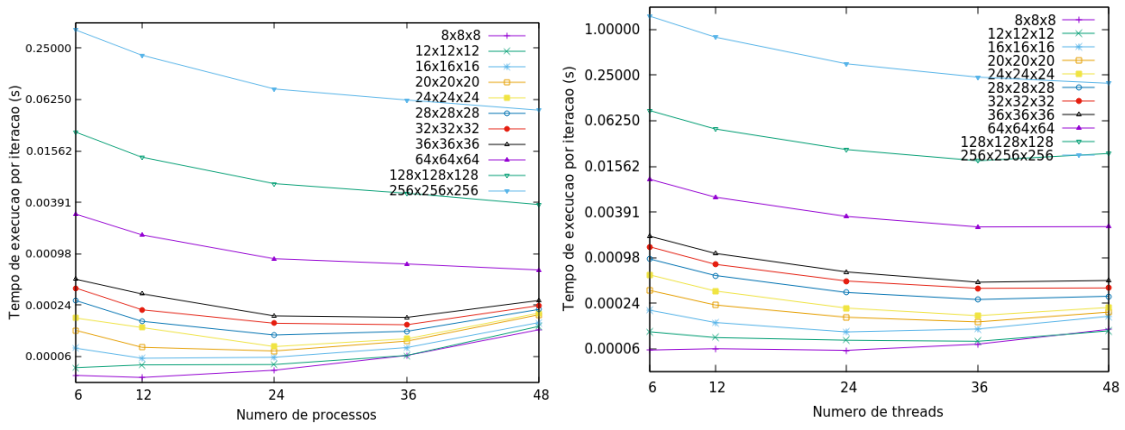
O melhor desempenho nas execuções utilizando `likwid-pin` é consequência da redução da troca de núcleo das *threads* do OpenMP. Essa redução aumenta o aproveitamento da memória cache, principalmente as caches L1 e L2, que são exclusivas para cada núcleo. Dessa forma, ao evitar as migrações de CPU, o desempenho geral da aplicação

é melhor e é consequência de uma maior reutilização dos dados presentes na cache. Os resultados alcançados representam uma melhora no tempo de execução da aplicação em OpenMP, que varia de 53% nos casos de 16 e 32 *threads* a 56,4% no caso de 48 *threads*. Essa comparação é em relação aos casos onde não se utilizou o *thread pinning*.

Outra variável que influencia no tempo da implementação em OpenMP são as barreiras necessárias para a sincronização das *threads*. As barreiras são chamadas através da diretiva `#pragma omp barrier` e se localizam após declarações de variáveis, durante o manuseio da estrutura de dados compartilhada na transposição e antes da liberação de memória. A visualização do tempo ocupado pelas barreiras, na ferramenta Cube, possibilita afirmar que nas execuções com *thread pinning* o tempo ocupado pelas barreiras corresponde ao *overhead* em relação as execuções da implementação em MPI. Sem o *thread pinning*, os tempos de execução em OpenMP são mais elevados e o *overhead* também é causado pelo número de migrações de CPU, como visto anteriormente.

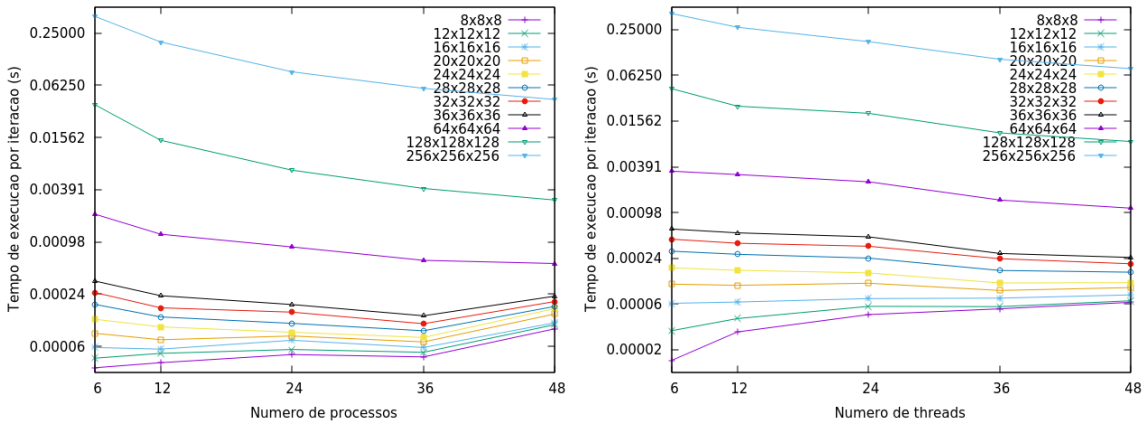
Após os resultados obtidos com os casos analisados anteriormente, o objetivo foi analisar o custo das transposições com maior granularidade, em cada implementação. Dessa forma, o código que anteriormente tinha 4 matrizes e ciclos de transposições para cada uma delas, passou a ter uma única matriz para realizar esse ciclo (como descrito na Figura 1). Além disso, o laço de cálculos foi zerado para essa análise. Devido aos resultados anteriores obtidos com a utilização de `likwid-pin` na implementação em OpenMP, as execuções seguiram envolvendo esse caso e, dessa vez, também utilizou-se o `likwid-pin` para execução da implementação em MPI.

As execuções envolveram casos com 6, 12, 24, 36 e 48 processos ou *threads*, de maneira a tentar abranger os nós NUMA inteiramente nas execuções, e os resultados serão apresentados com tempos médios de execução de cada iteração e não com tempos médios da execução total. Isso se deve ao fato de que a variação de tamanho nas malhas causa diferentes tempos totais de execução para um caso com o mesmo número de iterações e então as execuções com malhas maiores se estenderiam por um período de tempo muito longo. Logo, considerando a variação utilizada no número de iterações para os diversos casos abordados, a avaliação de desempenho se baseou no tempo médio de execução de cada iteração.



(a) MPI. (b) OpenMP.

Figura 3. Execuções sem utilização de `likwid-pin`.



(a) MPI. (b) OpenMP.

Figura 4. Execuções com utilização de `likwid-pin`.

Considerando o conteúdo das Figuras 3(a), 3(b), 4(a) e 4(b) é possível observar que os casos executados apresentam um comportamento padrão e bastante semelhante, onde existem poucos casos em que as execuções da implementação em OpenMP têm melhor desempenho. Normalmente, isso acontece com malhas menores ou com a combinação de uma malha pequena e um maior número de *threads*, o que reduz a carga de trabalho de cada *thread* e que torna a quantidade de dados compartilhados, a cada transposição, bastante reduzida.

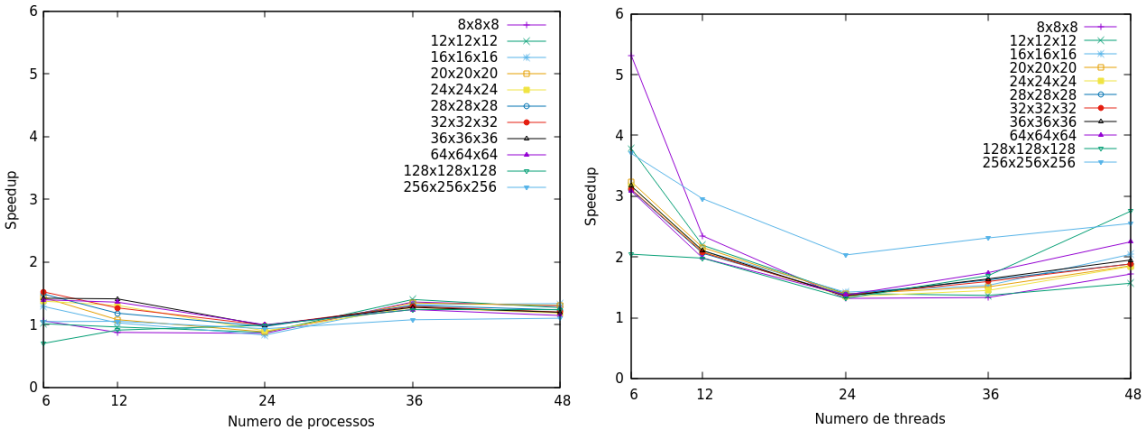
Quanto a isso, é possível observar que na maioria dos casos, as malhas até 36ˆ3 nós não se beneficiam do aumento para 48 processos ou *threads*, sendo esse um comportamento comum para ambas as implementações. Levando isso em consideração, alguns dos poucos casos onde a implementação em OpenMP tem melhor desempenho que a implementação em MPI são nas malhas de 8ˆ3 até 28ˆ3 para 48 *threads*. Porém, esse bom desempenho acaba sendo superado pelos números obtidos com as execuções da implementação em MPI para o caso de 36 processos e ratifica o fato de que a utilização de 48 processos ou *threads* será benéfica apenas para malhas maiores, sendo que para malhas menores a utilização de 48 processos ou *threads* reflete em desperdício de recursos. Esse comportamento é exemplificado na Tabela 3.

Tamanho da malha	OMP (48 *threads*)	MPI (48 processos)	MPI (36 processos)
8ˆ3	0,000064	0,000097	0,000046
12ˆ3	0,000067	0,000108	0,000052
16ˆ3	0,000080	0,000114	0,000059
20ˆ3	0,000100	0,000144	0,000068
24ˆ3	0,000116	0,000166	0,000077
28ˆ3	0,000159	0,000176	0,000092

Tabela 3. Casos em que a implementação MPI é melhor com menos processos. (Tempo de execução por iteração, em segundos)

Ao considerar diversos tamanho de malha, existem grupos de tamanhos que acabam beneficiando-se de determinadas condições de execução e, para cada um desses grupos,

existem determinadas condições que proporcionam um melhor desempenho. Sendo assim, o melhor desempenho para malhas pequenas (8^3 nós até 16^3 nós) aconteceu com 6 *threads* para a implementação em OpenMP com a utilização de `likwid-pin`. Já o melhor desempenho para tamanhos médios de malha (20^3 nós até 36^3 nós) ocorreu em 36 processos com a implementação em MPI utilizando `likwid-pin`. Para tamanhos de malha maiores (64^3 nós até 256^3 nós) a implementação em MPI com `likwid-pin` também apresenta os melhores resultados, mas com 48 processos.



(a) MPI (b) OpenMP

Figura 5. Speedup das implementações com a utilização de `likwid-pin`.

Além do que já havia sido testado anteriormente, nesse novo conjunto de testes também houveram execuções utilizando `likwid-pin` com o MPI. Em uma comparação com os ganhos da utilização dessa ferramenta na implementação em OpenMP, é possível observar na Figura 5 que os ganhos são menores e a utilização da ferramenta ocasionou perda de desempenho em algumas execuções. Nas execuções da implementação OpenMP o *speedup* variou de 1,3 até 5,3, com os maiores ganhos no caso de 6 *threads*, onde foi possível manter as *threads* no mesmo nó NUMA e, consequentemente, evitar acessos remotos na memória. Já nas execuções da implementação MPI, alguns casos tiveram um desempenho de cerca de 0,8 em relação ao inicial, enquanto outros chegaram a um *speedup* de no máximo 1,51.

6. Considerações Finais

Com base nos resultados obtidos é possível observar a influência das migrações de CPU no baixo desempenho inicial da implementação em OpenMP. A técnica de *thread pinning* usada com a ferramenta LIKWID foi efetiva ao diminuir esse número de migrações de CPU e consequentemente os tempos de execução.

Além disso, as barreiras utilizadas para a sincronização se tornam um problema necessário na implementação em OpenMP, principalmente no que se refere ao acesso a estrutura de dados compartilhada. O acesso a essa estrutura necessita de sincronização para que haja apenas a disponibilidade de dados corretos para a utilização.

Apesar disso, de maneira geral, a implementação em OpenMP teve um desempenho inferior em relação a implementação em MPI para os casos apresentados na arquitetura

utilizada para as execuções. Nos poucos casos em que a implementação em OpenMP obteve melhor desempenho, esse melhor desempenho era relativo, pois havia desperdício de recursos, visto que poderíamos obter um desempenho ainda melhor com a implementação em MPI e com menos processos, sendo assim, há um melhor desempenho absoluto da implementação em MPI para a maioria dos casos.

Os resultados obtidos nessa pesquisa possibilitam que em trabalhos futuros investiguem-se outras maneiras de melhorar o desempenho da aplicação em OpenMP, seja através da diminuição de trocas de contexto, que possivelmente influencia no desempenho, quanto em uma maneira de implementar barreiras mais seletivas dentro do código, onde as *threads* não necessitem aguardar todas as outras para seguir a execução, mas aguardem somente as *threads* com as quais compartilharão dados. Também espera-se retornar ao Incompact3d para realizar melhorias na questão da comunicação.

7. Agradecimentos

Este trabalho foi parcialmente financiado pelo projeto "GREEN-CLOUD: Computação em Cloud com Computação Sustentável" (#162551-0000 488-9), no programa FAPERGS-CNPq PRONEX 12/2014.

Referências

Bird, A., Long, D., and Dobson, G. (2017). Implementing shared memory parallelism in MCBEND. In *EPJ Web of Conferences*, volume 153, page 07042. EDP Sciences.

de Araujo, L. R., Weber, C. M., Puntel, F. E., Charão, A. S., and Lima, J. V. F. (2018). Análise de desempenho do software Incompact3D em uma arquitetura com múltiplos núcleos. *Fórum de Iniciação Científica da Escola Regional de Alto Desempenho do Rio Grande do Sul (ERAD RS)*, (2/2018).

Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., and Chapman, B. (2011). High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9):562–575.

Krawezik, G. and Cappello, F. (2003). Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127. ACM.

Xu, Y. and Zhang, T. (2015). A hybrid Open MP/MPI parallel computing model design on the SMP cluster. In *Power Electronics Systems and Applications (PESA), 2015 6th International Conference on*, pages 1–5. IEEE.