Avaliação da Adequação da Plataforma Charm++ para Arquiteturas Multicore com Memória Hierárquica

Laércio L. Pilla[1,2], **Christiane P. Ribeiro**[2],
Philippe O. A. Navaux[1], **Jean-François Méhaut**[2]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Portal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{llpilla,navaux}@inf.ufrgs.br

[2]Laboratoire d'Informatique de Grenoble – INRIA – Universitè de Grenoble
Grenoble – France

{christiane.pousa, jean-francois.mehaut}@imag.fr

Abstract. *Multi-core nodes with Non-Uniform Memory Access (NUMA) are now a common architecture for high performance computing. On such NUMA nodes, the shared memory is physically distributed, making memory access costs vary depending on their distance. Therefore, there is a need to control the memory affinity to provide satisfactory performance. In this context, this paper presents an evaluation of the Charm++ runtime system on these architectures to verify its adequacy to the architecture on communications and load balancing. The results highlight the need of information about the memory hierarchy and machine topology to improve Charm++'s efficiency.*

Resumo. *Máquinas multicore com organização de memória NUMA servem atualmente como base arquitetural para o processamento de alto desempenho. Nestes nós NUMA, a memória compartilhada é fisicamente distribuída, de forma que o custo de acesso à memória pode variar conforme sua distância. Assim, há a necessidade do controle de afinidade de memória para garantir desempenhos satisfatórios. Nesse contexto, este artigo apresenta uma avaliação da plataforma Charm++ nestas arquiteturas para verificar a sua adequação à arquitetura nos quesitos de comunicação e balanceamento de carga. Os resultados ressaltam a necessidade de informações da hierarquia de memória e topologia da máquina para o aumento da eficiência da plataforma Charm++.*

1. Introdução

A importância das arquiteturas com acesso não uniforme à memória (*Non-Uniform Memory Access*, NUMA) vem crescendo como uma solução escalável para reduzir a discrepância de desempenho entre processador e memória (*memory wall* [Lenoski et al. 1993, Wulf and McKee 1995]) e, assim, prover escalabilidade em máquinas multicore. Agregados baseados em processadores AMD Opteron e Intel Nehalem são exemplos de máquinas multicore com organização de memória NUMA. Uma plataforma NUMA é um sistema multiprocessador onde os elementos de processamento compartilham uma memória global que é fisicamente distribuída em bancos de memória. Esses bancos são interconectados através de uma rede especializada. Devido a tal interconexão, o custo de acesso à memória pode variar dependendo da distância (latência)

entre elementos de processamento e bancos de memória, e dependendo do número de elementos de processamento acessando o mesmo banco de memória (largura de banda). Como essas plataformas vêm se tornando cada vez mais presentes em sistemas de Processamento de Alto Desempenho (PAD), é importante reduzir a latência e aumentar a largura de banda disponível para o acesso aos dados. Assim, controlar a afinidade de memória torna-se um elemento crucial para melhorar o desempenho de tais máquinas.

A afinidade de memória é aprimorada quando o mapeamento de threads e dados é feito de forma a reduzir a latência de acesso e contenção de memória no acesso aos dados [Ribeiro et al. 2009]. Essa melhora pode acontecer através de diferentes abordagens, como o uso de mecanismos de alocação eficiente de memória ou através do balanceamento de carga entre os diferentes elementos de processamento. A primeira abordagem foca na distribuição dos dados e em trazê-los para próximo de seus usuários, de forma a reduzir a latência e a contenção de memória. A segunda abordagem trata com uma melhor distribuição de trabalho entre elementos de processamento de forma a evitar pontos de concentração de trabalho e melhorar a comunicação entre threads. A implementação dessas abordagens está usualmente ligada as características do ambiente de programação paralela utilizado. Um exemplo de ambiente é o Charm++ [Kale and Krishnan 1993].

Charm++ é uma plataforma e modelo de programação paralela baseado em C++ desenvolvido com o objetivo de melhorar a produtividade de programação através de uma abstração de alto nível da computação paralela enquanto provendo bom desempenho. Programas em Charm++ são decompostos em objetos comunicantes chamados de *chares*, os quais trocam dados através de chamadas remotas de métodos. Uma vantagem da plataforma Charm++ é que ela captura estatísticas dos objetos durante a execução da aplicação, as quais podem ser utilizadas posteriormente para aprimorar o balanceamento de carga [Bhatele et al. 2009] e, consequentemente, melhorar a afinidade de memória. Porém, apesar da existência de otimizações feitas para ambientes multicore [Mei et al. 2010], elas não levam em consideração as características de uma organização de memória NUMA. Isso pode levar a uma perda de desempenho devido, por exemplo, ao excesso de comunicações entre objetos distantes (em diferentes nós NUMA), os quais poderiam ser aproximados.

Nesse contexto, este artigo apresenta uma avaliação da plataforma Charm++ para arquiteturas multicore com organização NUMA, sendo focados os desempenhos obtidos com diferentes versões de comunicação e através do uso de balanceamento de carga. Dessa forma, torna-se possível avaliar a adequação da plataforma e, consequentemente, desenvolver formas de melhorar a afinidade de memória com a plataforma Charm++.

O restante do artigo é organizado da seguinte forma: a Seção 2 descreve as características das arquiteturas NUMA. A Seção 3 detalha a plataforma Charm++. A Seção 4 apresenta o ambiente e a Seção 5 a avaliação experimental. Trabalhos relacionados são tratados na Seção 6. Por fim, a Seção 7 apresenta as conclusões e trabalhos futuros.

2. Arquiteturas Multicore com Organização NUMA

O uso de arquiteturas multicore segue crescendo para a computação científica, especialmente em PAD. Uma máquina multicore consiste de múltiplos núcleos agrupados em sockets e que compartilham diferentes níveis de hierarquias de cache e memória. Isto é feito de forma a reduzir alguns problemas ligados ao paralelismo no nível de instrução em

um chip e problemas com limites de dissipação de calor [Liu et al. 2009]. Arquiteturas multicore são utilizadas para a construção de poderosas máquinas de memória compartilhada com dezenas a centenas de núcleos. Entretanto, o aumento no número de núcleos requer uma solução de hierarquia de memória eficiente, pois múltiplos núcleos podem utilizar a mesma interconexão para acessar a memória compartilhada, gerando o problema da *memory wall* [Lenoski et al. 1993, Wulf and McKee 1995].

Para dar suporte ao grande número de núcleos e reduzir as limitações de desempenho ligadas à memória, arquiteturas multicore com organização NUMA são utilizadas. Nestas máquinas, múltiplos núcleos acessam a mesma memória compartilhada. Adicionalmente, ela é distribuída em diversos bancos de memória interconectados. Dessa forma, os problemas com o desempenho da memória são reduzidos porque os núcleos podem utilizar diferentes caminhos e bancos de memória para acessar os dados. Entretanto, essa organização gera uma assimetria na latência de acesso aos dados [Ribeiro et al. 2009, Awasthi et al. 2010], levando ao conceito de acessos locais e remotos. Um acesso local é feito quando um núcleo acessa a memória em seu nó, enquanto um acesso remoto ocorre através da requisição de dados alocados em outro nó.

A Figura 1 ilustra uma máquina NUMA possuindo dezesseis núcleos e quatro nós NUMA. A memória compartilhada é distribuída em quatro bancos. Nessa arquitetura, cada quatro núcleos possui seu banco de memória local e os outros bancos são acessados através da rede de interconexão. Adicionalmente, essa máquina possui múltiplos níveis de cache compartilhada para reduzir a latência de acesso aos dados. Nesse caso, cada par de núcleos compartilhada uma cache L2 e dois pares compartilham uma cache L3.

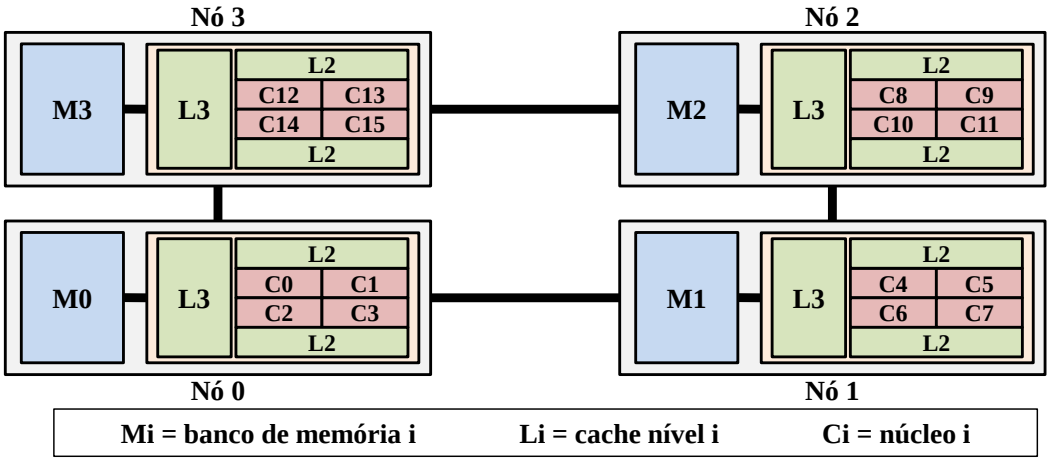

Figura 1. Exemplo de arquitetura multicore com organização NUMA.

Em máquinas NUMA, é particularmente importante garantir um uso eficiente dos bancos de memória para reduzir os custos de acessos remotos nas aplicações. Para tal, mecanismos como escalonamento de threads, alocação de memória e balanceamento de carga podem ser utilizados, dependendo das características da aplicação e da plataforma paralela [Ribeiro et al. 2009, Broquedis et al. 2010, Awasthi et al. 2010].

3. Sistema de Programação Paralela Charm++

Charm++ é uma plataforma paralela que provê uma linguagem de programação paralela orientada a objetos. Ela possui como objetivo prover produtividade ao programa-

dor. Charm++ abstrai características arquiteturais para o desenvolvedor e provê portabilidade sobre arquiteturas baseadas em memória compartilhada e sistemas distribuídos. As aplicações paralelas de Charm++ são escritas em C++ usando uma linguagem de descrição de interface para seus objetos [Kale and Krishnan 1993, Kale et al. 2008].

O processamento nas aplicações em Charm++ é decomposto em objetos chamados *chares*. O programador implementa as computações e comunicações descrevendo como esses objetos vão interagir e o ambiente de Charm++ gerencia as mensagens geradas por essas interações. Os objetos se comunicam através de chamadas remotas de métodos. Ainda, o ambiente é responsável pelo gerenciamento dos recursos arquiteturais. A Figura 2 ilustra a abstração de objetos comunicantes e seu mapeamento físico.

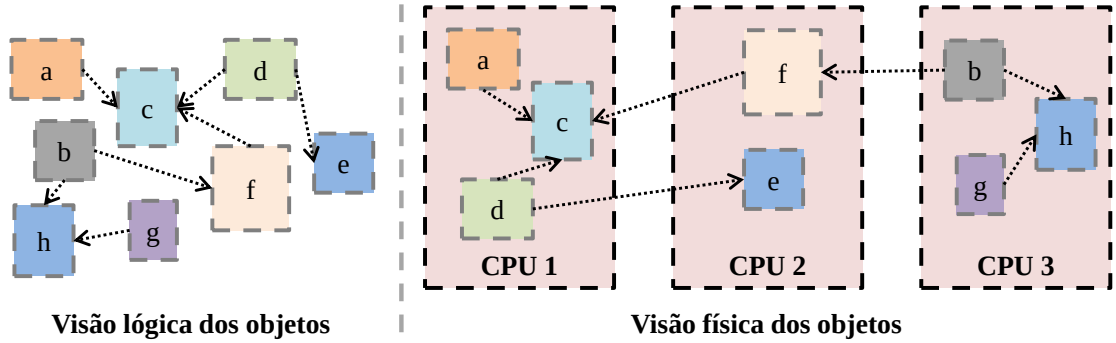

Figura 2. Abstração de aplicação na plataforma Charm++.

Quando utilizada a versão de Charm++ para memória compartilhada, todas as comunicações são feitas através da memória pela troca de ponteiros. Esse mecanismo permite ao ambiente reduzir os sobrecustos com o envio e recebimento de mensagens. Entretanto, no caso de máquinas NUMA, isto pode ser afetado pela assimetria nas latências e largura de banda de memória. Charm++ utiliza o esquema de afinidade de memória do sistemas operacional e não controla explicitamente o posicionamento de dados em memória. Isso pode levar a uma situação onde, por exemplo, mensagens são originalmente geradas e alocadas no nó NUMA 0. Após isto, essas mensagens são enviadas para um chare no nó 1, e encaminhadas sucessivamente até alcançarem o nó NUMA N. Nesta situação, diversos acessos remotos seriam gerados para cada comunicação.

Particularmente em alguns sistemas operacionais como Linux e Windows, a política padrão de gerenciamento de afinidade de memória em máquinas NUMA é a *first-touch*. Essa política posiciona os dados no nó NUMA que primeiro acessá-los [Joseph et al. 2006]. No caso do mecanismo de comunicação de Charm++, uma vez que os dados (por exemplo, uma mensagem) são tocados, essa política de memória não os migrará para aprimorar a afinidade de memória. Isso pode resultar em um posicionamento de dados subótimo em aplicações em Charm++. Dessa forma, torna-se importante avaliar o impacto de tais decisões no desempenho final em máquinas NUMA.

4. Ambiente de Experimentação

Nesta seção são apresentadas as máquinas utilizadas para a avaliação da adequação da plataforma Charm++ a arquiteturas multicore com organização NUMA. Duas máquinas NUMA representativas foram selecionadas para tais experimentos:

- **NUMA16**: baseada em oito processadores AMD Opteron dual-cores. Cada núcleo possui caches L1 e L2 privadas.
- **NUMA32**: baseada em quatro processadores Intel Xeon X7560 com oito núcleos cada. Cada núcleo possui caches L1 e L2 privadas e compartilha uma cache L3 no processador.

A Tabela 1 apresenta as características dessas máquinas, como a largura de banda de memória (obtida através da operação Triad do benchmark Stream [Mccalpin 1995]) e o fator NUMA (razão entre a latências de acesso remoto e local). O fatores NUMA são apresentados em intervalos das penalidades de acesso remoto mínimas e máximas.

Tabela 1. Descrição das arquiteturas multicore com organização NUMA.

Característica	NUMA16	NUMA32
Número de núcleos	16	32
Número de processadores	8	4
Nós NUMA	8	4
Frequência de relógio (GHz)	2.22	2.27
Maior nível de cache (MB)	1 (L2)	24 (L3)
Tamanho da memória (GB)	32	64
Largura de banda da memória (GB/s)	9.77	35.54
Fator NUMA (Min;Max)	[1, 1; 1, 5]	[1, 36; 3, 6]

Para a avaliação experimental foi utilizada a versão $6.2.1$ da plataforma Charm++. Os testes foram baseados na execução de três diferentes benchmarks disponibilizados com Charm++: (i) *kNeighbor*, um aplicação sintética iterativa onde cada chare se comunica com outros k à cada etapa; (ii) *jacobi2D*, uma computação stencil em 5 pontos sobre uma grade bidimensional; e (iii) *lb_test*, um benchmark sintético onde é possível escolher diferentes cargas computacionais e padrões de comunicação. Os resultados apresentados são as médias de desempenho obtidas sobre um mínimo de 20 execuções, com uma confiança estatística de 95% e erro relativo de 10% segundo a Distribuição T de Student.

5. Avaliação Experimental

Esta seção apresenta os experimentos realizados sobre as versões de comunicação e balanceadores de carga de Charm++ em máquinas NUMA.

5.1. Versões de comunicação

Charm++ permite a utilização de diferentes implementações para a troca de mensagens. No caso das máquinas NUMA utilizadas, duas versões de comunicação são de maior interesse: comunicação via troca de datagramas UDP e via troca de ponteiros em memória compartilhada. Apesar da segunda ser favorecida pela arquitetura, é importante verificar os sobrecustos da comunicação via datagramas tendo em vista que máquinas NUMA são utilizadas como nós em clusters de PAD.

A Figura 3 ilustra o desempenho obtido para o benchmark *jacobi2D* executando 100 chares e utilizando as diferentes formas de comunicação nas máquinas NUMA. O eixo vertical representa o tempo em segundos e o eixo horizontal representa diferentes

números de núcleos utilizados na execução do benchmark. As Figuras 3(a) e 3(b) apresentam os tempos obtidos nas máquinas NUMA16 e NUMA32, respectivamente. Como o desempenho do benchmark está mais ligado ao uso intensivo dos recursos de processamento do que pela comunicação, as diferentes versões de comunicação possuem impacto insignificante sobre o desempenho final da aplicação. Além disso, pode-se ver que quanto maior o número de núcleos utilizados, menor o tempo de iteração, demonstrando escalabilidade em ambas máquinas e versões.

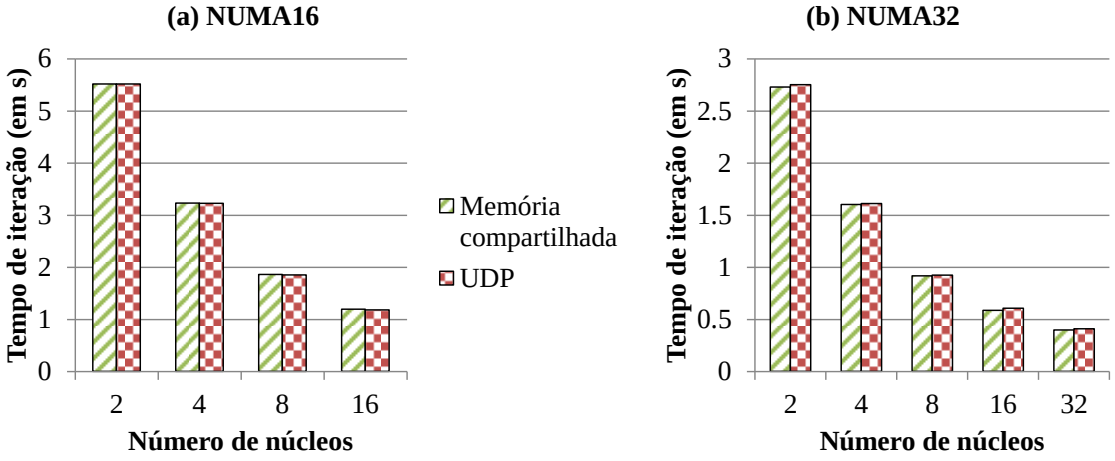

Figura 3. Tempo de iteração médio para as diferentes versões de comunicação com o benchmark *jacobi2D*.

Um caso mais extremo é ilustrado na Figura 4, a qual apresenta o desempenho obtido com o benchmark *kNeighbor* para as diferentes formas de comunicação. Neste experimento, foram utilizados 200 chares e um número de vizinhos comunicantes $k = 3$. Nesta figura, o eixo vertical representa o tempo de execução em milissegundos. Como visto na Figura 4(a), enquanto o tempo de execução é reduzido com o aumento de núcleos para a versão de comunicação via troca de ponteiros, o contrário acontece para a versão via UDP. Essa diferença chega ao seu máximo com 8 núcleos, onde a versão via UDP leva $9\times$ o tempo da versão via memória compartilhada. A redução no tempo de execução quando passando de 8 para 16 núcleos acontece devido a distribuição da comunicação através do aumento de vias de comunicação e distribuição da carga de trabalho. O mesmo não acontece para a máquina NUMA32, como ilustrado na Figura 4, devido à contenção na memória, pois há um maior número de núcleos acessando o mesmo banco de memória nessa máquina – oito núcleos por nó NUMA, enquanto a máquina NUMA16 possui apenas dois núcleos por nó.

Estes resultados mostram que a utilização das diferentes implementações de comunicação possuem um impacto pequeno sobre aplicações com computações intensivas. Entretanto, no caso de aplicações com muitas comunicações em clusters de máquinas NUMA, seria de interesse aplicar um esquema híbrido de comunicação, utilizando a troca de datagramas entre máquinas mas comunicação por memória compartilhada internamente à máquina NUMA. Ainda mais importante, os resultados com o benchmark *kNeighbor* mostram como o problema de contenção em memória pode prejudicar a escalabilidade de aplicações mesmo em arquiteturas com organização NUMA. Dessa forma, torna-se necessário assegurar uma melhor afinidade de memória através de outros meca-

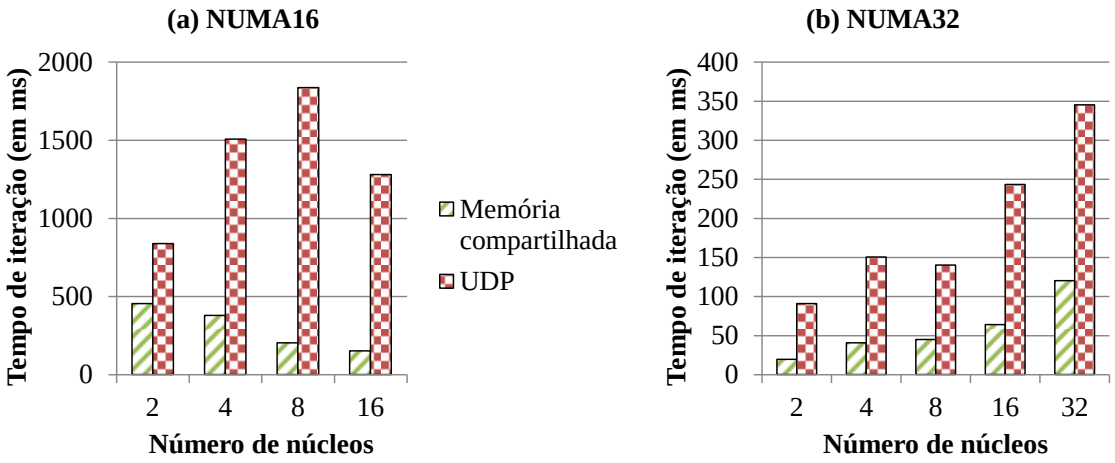

Figura 4. Tempo de iteração médio para as diferentes versões de comunicação com o benchmark *kNeighbor*.

nismos, como o balanceamento de carga.

5.2. Balanceamento de Carga

Através das estatísticas capturadas durante a execução de uma aplicação pela plataforma Charm++ é possível combater o desbalanceamento de carga, contenção de memória e reduzir as latências ligadas à comunicação remota. Para os experimentos com máquinas NUMA, foram considerados três balanceadores de carga disponibilizados com a plataforma: GREEDYLB, METISLB e SCOTCHLB. GREEDYLB reatribui os chares de forma gulosa. O algoritmo iterativamente mapeia o chare com maior tempo de execução para o núcleo com menor carga de trabalho. Sendo assim, o algoritmo não considera as comunicações entre chares. Apesar disso, esta estratégia possui um bom desempenho devido a sua simplicidade e velocidade. Já os balanceadores METISLB e SCOTCHLB são baseados em algoritmos de particionamento de grafos implementados nas bibliotecas METIS [Karypis and Kumar 1995] e SCOTCH [Pellegrini and Roman 1996], respectivamente. Estas estratégias consideram ambos tempos de execução e grafo de comunicação para aprimorar o desempenho das aplicações.

A Figura 5 apresenta os desempenhos obtidos para os diferentes balanceadores de carga aplicados aos benchmarks *lb_test* e *jacobi2D*. Nela, a coluna Base representa o tempo médio das iterações sem o balanceamento de carga. Para o benchmark *lb_test*, foram utilizados 200 chares, com tempos mínimos e máximos de computação de 50 e 200 ms, respectivamente, e um grafo de comunicações aleatório. Como pode ser visto na Figura 5(a), o uso de balanceamento de carga leva a melhoras em ambas máquinas NUMA. Entretanto, em ambos casos levam vantagem os balanceadores que consideram o grafo de comunicação. Por exemplo, na máquina NUMA32, o uso do SCOTCHLB leva a *speedups* de 1.4 sobre o tempo base e de 1.18 sobre o balanceador GREEDYLB.

Resultados similares foram obtidos para o o benchmark *jacobi2D*, como ilustrado na Figura 5(b). Porém, apesar dos *speedups* de até 1.57 sobre o tempo de execução base, os balanceadores de carga obtiveram uma eficiência final por volta de 90% ou menos. Isso acontece porque, apesar dos balanceadores de carga como o SCOTCHLB reduzirem as comunicações externas aos núcleos, eles não possuem conhecimento sobre a organização

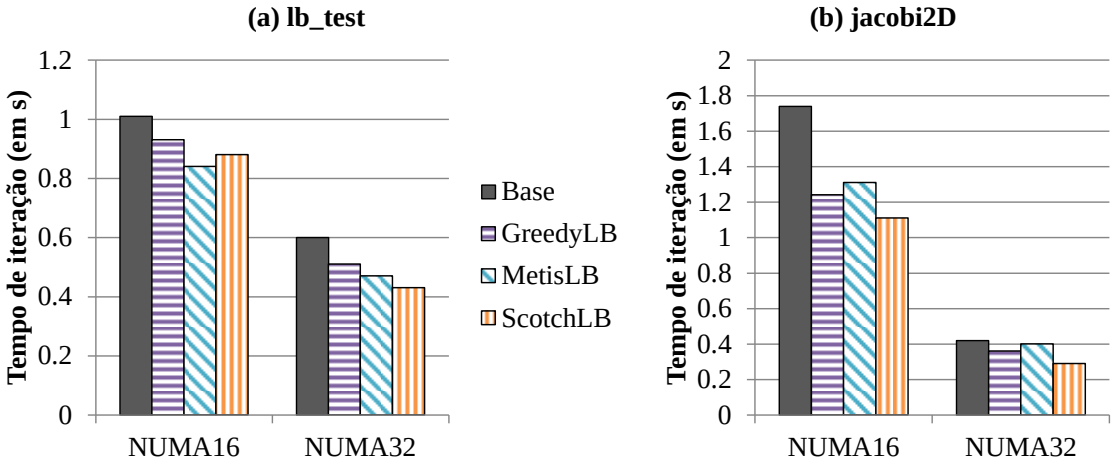

Figura 5. Tempo de iteração médio para diferentes balanceadores de carga e benchmarks.

Tabela 2. Número médio de migrações por chamada de balanceamento de carga.

Benchmark	Máquina	Balanceador de carga		
		GREEDYLB	METISLB	SCOTCHLB
lb_test	NUMA16	188	187	184
	NUMA32	194	194	192
jacobi2D	NUMA16	94	94	93
	NUMA32	97	96	98

e hierarquias de cache e memória. Dessa forma, não são aproveitados os núcleos próximos para reduzir as comunicações remotas de alta latência.

Adicionalmente a estes resultados, a Tabela 2 apresenta as quantidades médias de migrações feitas pelos balanceadores de carga. Como pode ser visto, tais balanceadores de carga não consideram o mapeamento inicial dos chares e, por isso, acabam migrando entre 92% e 98% dos chares. Dessa forma, mostra-se importante a avaliação do impacto que essas migrações afligem no desempenho final da aplicação, tendo em vista que elas resultados em cópias de memória através dos nós NUMA.

A Figura 6(a) apresenta os custos ligados à migração de 20 chares entre diferentes nós NUMA para diferentes tamanhos de chares na máquina NUMA16. Na figura, o eixo vertical representa o tempo em segundos para efetuar tais migrações e é apresentado em escala logarítmica. O eixo horizontal representa o número de bytes de cada chare migrado (também em escala logarítmica). Cada linha representa migrações para nós remotos. A linha intra-socket representa migrações para núcleos dentro do mesmo processador. Como pode ser visto na Figura 6(a), há uma grande diferença entre os custos de migrações para núcleos próximos e distantes para chares pequenos. Entretanto, elas são na ordem de milissegundos e acabam diminuindo com o crescimento dos chares. Ainda assim, há uma diferença de 1.1 segundos entre migrações locais para as mais distantes, o que representa 20% do tempo total de migração. Essa diferença pequena nos tempos de migração encontrados na máquina NUMA16 são explicados pela pequena diferença nas latências

de acesso à memória local e remota (baixo fator NUMA) e pela ausência de uma cache compartilhada entre os núcleos em um mesmo nó NUMA.

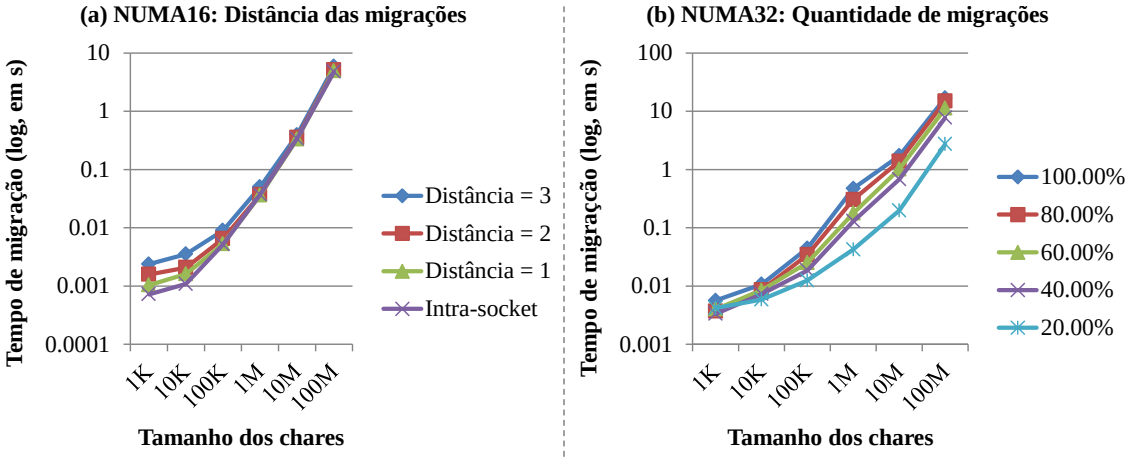

Figura 6. Tempo de migração para diferentes distribuições de chares.

Por fim, a Figura 6(b) ilustra os tempos de migração para diferentes percentuais de migrações aleatórias na máquina NUMA32. Não há praticamente diferença no custo de migração quando utilizando chares de até 10 KB. Porém, tal diferença cresce rapidamente com o tamanho dos chares, chegando ao ponto em que migrar apenas 20% dos chares leva apenas 16% do tempo de migrar todos os chares. Dessa forma, para aumentar o desempenho de uma aplicação em Charm++ em uma máquina multicore com organização NUMA, não basta apenas aprimorar a afinidade de memória, sendo também importante evitar migrações desnecessárias e, consequentemente, cópias remotas de dados.

6. Trabalhos Relacionados

Arquiteturas multicore com organização NUMA tem se tornado uma tendência em arquiteturas paralelas de computação de alto desempenho. Entretanto, poucos sistemas paralelos, linguagens de programação e interfaces oferecem algum suporte para esse tipo de arquitetura. A complexidade da hierarquia de memória dessas arquiteturas traz consigo custos associados aos acessos à esta memória. Nesse contexto, uma adequação dos sistemas paralelos, linguagens de programação e interfaces torna-se necessária.

Uma forma de evitar os custos de acesso à memória em arquiteturas NUMA é usar o mapeamento de threads. O mapeamento pode ser usado para reduzir a distância entre threads que realizam muita comunicação entre si. No caso específico de arquiteturas NUMA, esse mapeamento pode favorecer o uso de memórias caches e reduzir o uso da memória principal. Essa abordagem é utilizada em [Mercier and Clet-Ortega 2009] e [Jeannot and Mercier 2010]. Porém, os autores avaliam apenas a melhora no desempenho das aplicações, e não as características das plataformas e arquiteturas envolvidas.

Em [Mei et al. 2010], os autores apresentam algumas otimizações no sistema Charm++ para máquinas multicore. Uma das otimizações apresentadas é o suporte à afinidade de threads, que permite ao programador mapear threads nos núcleos da máquina e assim, reduzir os custos de comunicação. Resultados apresentados no trabalho mostram ganhos de até 15% quando o suporte para mapeamento de threads foi utilizado. Apesar

dessa otimização não ter sido proposta para arquiteturas NUMA, ela pode ser facilmente aplicada nessas plataformas de forma a reduzir os custos de acesso à memória.

Muitos outros ambientes de programação paralela também sofrem com a assimetria de acessos à memória em máquinas NUMA. Como exemplo, podemos citar ambientes de memória transacional por software (STM) que têm sido estudados nos últimos anos. STM é uma proposta que visa reduzir a complexidade do controle de seções críticas e o custo associado aos *locks*. Apesar de muitos estudos de memória transacional terem considerados máquinas multicore, poucos trabalhos foram propostos para máquinas multicore com organização NUMA. Em [Lu et al. 2010], são propostos mecanismos para STM que consideram os custos não uniformes de acesso à memória. A ideia principal é separar a máquina NUMA em clusters, dependendo da distância entre os componentes da máquina. Assim, a STM poderia reagir as necessidades da aplicação considerando os custos da máquina NUMA. Diferente do nosso trabalho, esse estudo não apresenta uma avaliação de desempenho de STMs em máquinas NUMA e como consequência, ele não trata dos diferentes impactos que uma arquitetura NUMA pode ter sobre STM.

OpenMP é uma interface de programação paralela que surgiu de um consórcio entre universidades e empresas [OpenMP 2010]. A principal proposta do consórcio é a definição de um padrão que simplifique o desenvolvimento de aplicações paralelas. Em OpenMP, aplicações são paralelizadas usando diretivas que informam ao compilador as regiões do código que devem ser paralelizadas. Entretanto, OpenMP não foi desenvolvido para arquiteturas NUMA. Desta forma, não existe nenhum suporte na interface para gerenciar a afinidade de memória. Em [Ribeiro et al. 2008], os autores apresentam uma avaliação de benchmarks e aplicações OpenMP em arquiteturas NUMA. Eles também apresentam o suporte NUMA existente em alguns sistemas operacionais para gerenciar a alocação de dados e o mapeamento de threads para reduzir as penalidades NUMA nas aplicações. Os resultados mostram a importância de gerenciar a afinidade de memória em arquiteturas NUMA e como usuários podem fazer esse controle em aplicações OpenMP usando o suporte do sistema operacional. Porém, os autores não avaliam o suporte de execução OpenMP, focando seus estudos nas aplicações e benchmarks. Ao contrário, a avaliação apresentada deste trabalho foca em compreender e investigar a adequação de um sistema paralelo em arquiteturas NUMA.

7. Conclusões

O aumento na popularidade de arquiteturas multicore com organização NUMA requer a adequação das plataformas de programação paralela para elas, de forma a prover desempenhos satisfatórios e reduzir sobrecustos ligados à comunicações remotas e contenção de memória. Entretanto, esta não é uma tarefa simples, pois a complexidade da hierarquia de memória pode influenciar aplicações de formas diferentes. Assim, faz-se importante estudar como estas plataformas são influenciadas pelos ambientes NUMA.

Os resultados desta avaliação da plataforma Charm++ em máquinas NUMA mostram que as comunicações entre nós NUMA devem ser mantidas através da troca de ponteiros em memória compartilhada, pois outros mecanismos podem acabar com a escalabilidade das aplicações, principalmente no caso de aplicações com comunicação intensiva. Além disso, os resultados com balanceamento de carga apresentam melhoras de desempenho, mas chegam a eficiências limitadas a 90%. Isso acontece devido a ausência de co-

nhecimento sobre a organização NUMA e das hierarquias de memória e cache, o que leva a mapeamentos que não favorecem comunicações através de acessos locais à memória. Por fim, o desenvolvimento de um balanceador de carga com conhecimento sobre a arquitetura deve evitar migrações desnecessárias, devido ao custos de cópias remotas de memória.

Trabalhos futuros incluem o desenvolvimento de um balanceador de carga consciente da organização NUMA e da hierarquia de cache das arquiteturas multicore. Isto requer a medição das diferentes latências de comunicação entre núcleos e nós NUMA. Através da organização destas informações, pretende-se adicioná-las à plataforma paralela Charm++, de forma a aprimorar sua adequação a tais arquiteturas de forma transparente ao programador.

Referências

Awasthi, M., Nellans, D. W., Sudan, K., Balasubramonian, R., and Davis, A. (2010). Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT 2010)*, pages 319–330, New York, NY, USA. ACM.

Bhatele, A., Kale, L. V., and Kumar, S. (2009). Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proceedings of the 23rd international Conference on Supercomputing (ICS 2009)*, pages 110–116, New York, NY, USA. ACM.

Broquedis, F., Aumage, O., Goglin, B., Thibault, S., Wacrenier, P. A., and Namyst, R. (2010). Structuring the execution of OpenMP applications for multicore architectures. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2010)*, pages 1–10. IEEE Computer Society.

Jeannot, E. and Mercier, G. (2010). Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In D'Ambra, P., Guarracino, M., and Talia, D., editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 199–210. Springer Berlin / Heidelberg.

Joseph, A., Pete, J., and Alistair, R. (2006). Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *International Conference on High Performance Computing (HiPC 2006)*, pages 338–352.

Kale, L. V., Bohm, E., Mendes, C. L., Wilmarth, T., and Zheng, G. (2008). Programming Petascale Applications with Charm++ and AMPI. In Bader, D., editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press.

Kale, L. V. and Krishnan, S. (1993). Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)*, pages 91–108. ACM.

Karypis, G. and Kumar, V. (1995). METIS: Unstructured graph partitioning and sparse matrix ordering system. *The University of Minnesota*, 2.

Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J. (1993). The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61.

Liu, M., Ji, W., Wang, Z., and Pu, X. (2009). A memory access scheduling method for multi-core processor. *International Workshop on Computer Science and Engineering (WCSE 2009)*, 1:367–371.

Lu, K., Wang, R., and Lu, X. (2010). Brief announcement: Numa-aware transactional memory. PODC, pages 69–70.

Mccalpin, J. D. (1995). STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia.

Mei, C., Zheng, G., Gioachin, F., and Kale, L. V. (2010). Optimizing a parallel runtime system for multicore clusters: a case study. In *Proceedings of the 2010 TeraGrid Conference (TG 2010)*, New York, NY, USA. ACM.

Mercier, G. and Clet-Ortega, J. (2009). Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In Ropo, M., Westerholm, J., and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 104–115. Springer Berlin / Heidelberg.

OpenMP (2010). The OpenMP API Specification for Parallel Programming.

Pellegrini, F. and Roman, J. (1996). Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking (HPCN 1996)*, pages 493–498. Springer.

Ribeiro, C. P., Dupros, F., Carissimi, A., Marangozova-Martin, V., Méhaut, J.-F., and de Aguiar, M. S. (2008). Explorando Afinidade de Memória em Arquiteturas NUMA. In *WSCAD '08: Proceedings of the 9th Workshop em Sistemas Computacionais de Alto Desempenho - SBAC-PAD*, Campo Grande, Brazil. SBC.

Ribeiro, C. P., Mehaut, J.-F., Carissimi, A., Castro, M., and Fernandes, L. G. (2009). Memory Affinity for Hierarchical Shared Memory Multiprocessors. In *21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009)*, pages 59–66.

Wulf, W. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23:20–24.