# Explorando Afinidade de Memória em Arquiteturas NUMA

Christiane Pousa Ribeiro Vania Marangozova-Martin
Jean-François Méhaut
Laboratoire d'Informatique Grenoble - Grenoble - França
(Christiane.Pousa, Vania.Marangozova-Martin, Jean-Francois.Mehaut)@imag.fr

Fabrice Dupros
BRGM - Orléans, França
f.dupros@brgm.fr

Alexandre Carissimi
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil
asc@inf.ufrgs.br

## Resumo

*Arquiteturas NUMA possuem latência e largura de banda assimétricas devido a existência de múltiplos níveis hierárquicos de memória no sistema. Para garantir desempenho neste tipo de arquitetura torna-se necessário garantir a afinidade de memória nas aplicações. Os sistemas operacionais, com suporte para arquiteturas NUMA, possuem políticas para alocação e escalonamento de memória e threads que visam a afinidade de memória. Entretanto, essas políticas não apresentam sempre o melhor desempenho para todos os tipos de aplicações. Ferramentas e APIs, presentes nestes sistemas operacionais, permitem gerenciar explicitamente a afinidade de memória nas aplicações. Neste trabalho será apresentado a avaliação de desempenho de diferentes estratégias para gerenciamento explícito de afinidade de memória, implementadas com APIs do sistema operacional em aplicações paralelas. Essas estratégias foram implementadas em uma aplicação sísmica e em kernels do Benchmark NAS e executadas em diferentes arquiteturas NUMA. Os resultados mostram a importância de garantir a afinidade de memória em arquiteturas NUMA (ganho médio de até 80%) e que isso pode ser obtido através de APIs do sistema operacional.*

## 1. Introdução

Arquiteturas NUMA (*Non Uniform Memory Access*) estão se tornando arquiteturas comuns em processamento paralelo. Isto pode ser explicado pelo surgimento das novas gerações de arquiteturas *multicore*, SMPs (*Symetric Multiprocessor*) e processadores *multithread* com múltiplos níveis de memória. Essas novas gerações são exemplos reais de arquitetura NUMA e têm sido amplamente utilizadas como máquinas para computação intensiva de aplicações paralelas [1]. A sua principal característica é a presença de vários níveis de hierarquia de memória distribuída compartilhados pelos elementos de processamento. Esta característica permite o desenvolvimento de aplicações usando o modelo de programação de memória compartilhada [2, 3]. O principal problema nesse tipo de arquitetura é o tempo de acesso aos dados compartilhados. Como os elementos de processamento compartilham a memória distribuída, o tempo de acesso aos dados locais e remotos não são uniformes e isso pode diminuir o desempenho da aplicação [2, 4, 5].

O desempenho das aplicações paralelas nessas arquiteturas é importante, uma vez que elas estão sendo usadas como máquinas para aplicações que demandam muito processamento e um baixo tempo de execução. Para otimizar o desempenho nas arquiteturas NUMA torna-se necessário minimizar o número de acessos remotos realizados pelas unidades de processamento, garantindo a afinidade de memória. A afinidade de memória está relacionada com a redução da distância entre os processos e dados usados por eles, ou seja, minimização do número de acessos remotos aos dados [2, 4, 6].

Para garantir a afinidade de memória e assim aumentar o desempenho da aplicação, o sistema operacional provê políticas para gerenciar alocação de memória e escalonamento dos processos nas arquiteturas NUMA. Entretanto, estas políticas não apresentam sempre o desempenho ótimo para todos os tipos de aplicação. Então, nos últimos anos, os sistemas operacionais (Linux, Solaris, etc) têm provido ferramentas em nível de usuário e APIs com chamadas de sistemas que permitem aos programadores gerenciar explicita-

mente a distribuição/alocação de memória e processos das suas aplicações, garantindo a afinidade de memória [5, 7].

Neste trabalho será apresentado a avaliação de desempenho de diferentes estratégias, implementadas com APIs do sistema operacional, que têm como objetivo garantir a afinidade de memória. Essas estratégias foram implementadas usando gerenciamento explícito de memória através de chamadas de sistema no código fonte das aplicações. As aplicações consideradas são *kernels* do *benchmark* NAS [8] e uma aplicação real que realiza simulação de sismos (Ondes 3D) [9]. Essas aplicações foram desenvolvidas em C e usando diretivas OpenMP [10] para o paralelismo do código. O principal objetivo deste trabalho é mostrar a possibilidade de obter-se ganhos importantes com gerenciamento explícito da afinidade de memória usando APIs do sistema operacional.

O restante do artigo está organizado da seguinte forma: Seção 2 apresenta os principais conceitos sobre arquiteturas NUMA. A Seção 3 discute os principais trabalhos relacionados. A seção 4 apresenta o método experimental, a aplicação sísmica, os *kernels* do *benchmark* NAS, as estratégias para gerenciamento de memória implementadas e as características das arquiteturas NUMA utilizadas. A Seção 5 apresenta os resultados obtidos e a Seção 6 as conclusões e possíveis trabalhos futuros.

## 2. Arquiteturas NUMA

Uma arquitetura NUMA pode ser definida como um conjunto de nodos que compartilham diferentes níveis de memória (caches e memória principal). Nessa arquitetura os tempos de acessos à memória não são uniformes. A assimetria no tempo de acesso acontece porque a memória global está fisicamente distribuída entre os nodos que compõem a arquitetura. Como esses nodos estão conectados através de uma rede, observamos o aparecimento do que chamamos de fator NUMA. O fator NUMA é a razão entre o tempo de acesso a um dado remoto (fisicamente alocado em outro nodo) e o tempo de acesso a um dado local (fisicamente alocado no bloco de memória presente no nodo). Nessas arquiteturas NUMA torna-se importante reduzir o impacto do fator NUMA sobre a aplicação [5, 3].

A redução do impacto do fator NUMA pode ser feita considerando dois tipos de otimizações: largura de banda e latência. Otimizar a largura de banda da rede de interconexão evita problemas de contenção de memória, permitindo muitos acessos concorrentes ao mesmo bloco de memória. A otimização da latênca evita grande número de acessos remotos, *threads* e dados são alocados próximos.

Além do fator NUMA, características como cache e falso compartilhamento devem ser consideradas quando desenvolve-se aplicações para essas arquiteturas. Nas arquiteturas NUMA deve-se otimizar o uso de memória cache e favorecer o princípio da localidade nas aplicações: maior número de acessos à memória cache significa menos acessos à memória principal, que no caso das arquiteturas NUMA pode ser local ou remota. Muitas vezes, nessas arquiteturas o desempenho pode ser prejudicado pelo falso compartilhamento. No falso compartilhamento processos/*threads* acessam dados que estão fisicamente localizados em uma mesma página de memória. Então, a página pode ser bloqueada por qualquer processo/*thread* que use um dos dados contidos nela, não permitindo que outro processo/*thread* tenha acesso aos outros dados contidos na página.

### 2.1. Modelo de Programação

Uma das principais vantagens de se utilizar arquiteturas NUMA é a possibilidade de desenvolver aplicações usando o modelo de programação de memória compartilhada. Nesse modelo de programação, as aplicações paralelas são desenvolvidas considerando que a memória é acessada por múltiplos fluxos de execução. Para desenvolver as aplicações considerando esse modelo o desenvolvedor pode utilizar APIs do sistema operacional ou de linguagens de programação. Como exemplos de APIs podemos citar, POSIX *threads* [11] e OpenMP [10].

OpenMP é um padrão muito utilizado para o desenvolvimento de aplicações paralelas no modelo de programação de memória compartilhada. As implementações deste padrão permitem, através de diretivas, paralelizar o código da aplicação de maneira simples e com um bom desempenho. Entretanto, como OpenMP foi proposto para arquiteturas UMA (*Uniform Memory Access*), não existe nenhum suporte para arquiteturas NUMA [10, 12]. Ou seja, nesse padrão não existe nenhuma otimização de memória para arquiteturas com múltiplos níveis de memória compartilhada.

### 2.2. Suporte do Sistema Operacional

O suporte para gerenciamento de memória em arquiteturas NUMA está atualmente presente em grande parte dos sistemas operacionais. Esse suporte pode ser tanto no nível de usuário (através de ferramentas/comandos) quanto no nível de chamadas de sistema (através de APIs). Podemos citar como exemplos desses sistemas operacionais, o Linux e o Solaris.

O suporte no nível de usuário usando ferramentas ou comandos permite ao desenvolvedor da aplicação especificar a política de memória e de escalonamento de *threads* para uma aplicação. A vantagem de usar esse tipo de suporte é que nenhuma modificação no código fonte da aplicação é necessário. Entretanto, essa solução afeta a aplicação

como um todo, não é permitido alterar as políticas durante a execução da aplicação [5].

A API NUMA é uma interface que define um conjunto de chamadas de sistema que afetam a alocação de memória e *threads*. Neste caso, o desenvolvedor deve ter acesso ao código fonte da aplicação para gerenciar a memória e a alocação de *threads*. A principal vantagem de usar APIs é a possibilidade de um controle mais fino da alocação de memória/*threads* [5, 13].

## 3. Trabalhos Relacionados

Em [7], os autores apresentam diferentes estratégias para alocação de memória e processos/*threads* em duas plataformas diferentes (UltraSparc-Solaris e Opteron-Linux) em nível de usuário. As estratégias usam ferramentas do sistema operacional para alocação e não APIs de chamadas do sistema. Além disso, os autores fazem uma avaliação da latência e largura de banda das duas plataformas e propõem um *framework* para realizar experimentos de alocação de memória e processos/*threads* nas duas plataformas. Os autores mostram em seus resultados que o uso do *framework* proposto pode ajudar os desenvolvedores a otimizar a afinidade de memória nas suas aplicações.

No trabalho [14] é apresentado a avaliação do impacto da migração de páginas de memória no sistema. Os autores avaliaram diferentes aplicações OpenMP (disponibilizadas nos *benchmarks* NAS e SPEC) em diferentes condições: política de escalonamento e carga do sistema. Os resultados mostram que ganhos de desempenho podem ser obtidos com o uso de migração de páginas se a política de escalonamento considera as características de memória.

Em [15], os autores apresentam um estudo sobre o desempenho alcançado por aplicações reais que usam o suporte do sistema operacional para migrar/replicar os dados. Com o suporte oferecido pelo sistema operacional os autores minimizam os efeitos NUMA sobre as aplicações e apresentam ganhos de até 30%.

No trabalho [3], os autores apresentam uma nova estratégia de alocação de memória, *on-next-touch*. Essa estratégia permite que os dados sejam migrados para os nodos onde as *threads* que os utilizam estão sendo executadas. A avaliação desta nova estratégia é feita considerando uma aplicação real que possui diferentes padrões de acesso. Os autores apresentam ganhos de até 69% com 22 *threads*.

Os trabalhos apresentados não consideram uma avaliação de desempenho de diferentes aplicações reais e *benchmarks* com padrões de acessos regulares e irregulares aos dados. Além disso, os trabalhos não avaliaram aplicações desenvolvidas com OpenMP e chamadas de sistema que visam minimizar o impacto dos efeitos NUMA e garantir a afinidade de memória.

## 4. Avaliação de Estratégias para Afinidade de Memória

Nesta seção é apresentado o método experimental, as diferentes estratégias implementadas, as aplicações e as plataformas NUMA utilizadas.

Os experimentos foram realizados considerando duas arquiteturas NUMA e diferentes implementações das aplicações. A métrica selecionada para avaliação de desempenho foi o tempo de execução da aplicação, que é uma das métricas mais utilizadas e representativas dentro de processamento paralelo.

### 4.1. Chamadas de Sistema

Neste trabalho, as estratégias para otimizar a afinidade de memória foram desenvolvidas utilizando o suporte do sistema operacional Linux. A escolha do Linux foi principalmente por ele ser um dos sistemas operacionais mais utilizados em arquiteturas NUMA.

Desde a versão 2.6 do kernel o Linux oferece suporte para afinidade de memória em NUMA [5, 7]. A política padrão de alocação de memória usada no Linux é a *First-Touch*, que consiste em alocar o dado no bloco de memória do nodo que fez o acesso primeiro. Em máquinas NUMA a política padrão do Linux para alocar as *threads* tem como objetivo minimizar a distância entre as *threads* e os dados [16, 5].

Aplicações paralelas OpenMP são geralmente desenvolvidas usando uma de duas estratégias de gerenciamento de afinidade de memória:

- `First-Touch`: diretivas OpenMP são usadas apenas para paralelizar as etapas de cálculo da aplicação. Não existe otimização para aspectos de memória (alocação física dos dados) e a política padrão do Linux (*First-touch*) é usada para alocação dos dados. Ou seja, *thread* mestre aloca e inicializa todas as estruturas de dados, mantendo-as em um único nodo da máquina NUMA.

- `Parallel-Init`: nesta versão são adicionadas diretivas OpenMP na etapa de inicialização das estruturas de dados. Desta forma, a inicialização das estruturas de dados também é realizada de forma paralela e dados são alocados fisicamente no mesmo nodo da *thread* que os utiliza. Essa solução é amplamente utilizada pelos desenvolvedores de aplicações paralelas OpenMP. Entretanto, ela apresenta bons resultados apenas quando o padrão de acesso aos dados é regular e a carga de trabalho de cada *thread* é definida estaticamente.

O desenvolvimento de aplicações paralelas usando OpenMP pode ser realizado como apresentado acima. Entretanto, não existe nenhuma garantia de afinidade de memória e o melhor desempenho provavelmente não será alcançado.

Em arquiteturas NUMA, a afinidade de memória pode ser garantida através do controle explícito da alocação de memória e *threads* da aplicação. Esse controle pode ser obtido através do uso da APIs do sistema operacional que permitem um controle fino da alocação de memória e *threads* na aplicação.

Este trabalho, foca-se na API NUMA de chamadas de sistema e as duas funções usadas para garantir a afinidade de memória são: `mbind` e `sched_setaffinity` [17, 13]. Estas duas funções permitem que o programador especifique a política de memória para uma seqüência de dados e onde os processos/*threads* serão executados. Então, o programador pode minimizar os efeitos NUMA em uma aplicação na arquitetura NUMA escolhida.

A função `mbind` usa alguns parâmetros para especificar a política de memória a ser adotada, os nodos que serão usados para alocação, a sequência de páginas e um *flag* que permite realizar a migração de páginas. As políticas de memória permitidas para o `mbind` são: *bind* (memória será apenas alocada em um conjunto pré-definido de nodos), *interleave* (memória será alocada ciclicamente em um conjunto de nodos, o nodo inicial é aquele que primeiro causou a falta de página) e *preferred* (mémoria será preferencialmente alocada em um conjunto espedífico de nodos) [18]. A função `sched_setaffinity` utiliza três parâmetros: o identificador do processo/*thread*, uma máscara que indica o processador/núcleo a ser utilizado e um último parâmetro que informa o tamanho da máscara [17].

Abaixo apresentamos duas possíveis estratégias de afinidade de memória que usam as funções citadas.

- `Round-Robin`: todos os dados da aplicação são alocados fisicamente na máquina usando a política de memória *interleave* da chamada de sistema `mbind`. As estruturas de dados são alocadas dinamicamente usando a chamada de sistema `mmap` que retorna um ponteiro para a primeira página de memória que contém os dados da estrutura. Usando esse ponteiro na função `mbind` informamos a política de memória que deve ser usada nesse conjunto de páginas. A implementação com *interleave*, favorece a largura de banda pois os dados são distribuídos ciclicamente nos nodos utilizados durante a execução da aplicação. Essa estratégia minimiza o número de acessos concorrentes no mesmo bloco de memória. As *threads* são alocadas nos processadores/núcleos usando a chamada de sistema `sched_setaffinity` e considerando o seu identificador OpenMP como número do processador/núcleo. A alocação das *threads* permite controlar as migrações realizadas pelo sistema operacional garantindo a afinidade de memória.
- `Memory-Bind`: utiliza chamadas de sistema para alocar dados e *threads* na arquitetura NUMA. As funções usadas são: `mbind` e `sched_setaffinity`. Considerando a função `mbind` e a política *bind*, apenas os dados mais importantes da aplicação (maior consumo de memória e número de acessos) são alocados com esta política. As estruturas de dados nessa implementação também são alocadas dinamicamente e o ponteiro retornado é usado para definir a política de memória para a seqüencia de páginas. Na implementação com *bind*, a latência é favorecida pois, os dados são alocados no mesmo nodo que a *thread* que vai utilizá-los. As *threads* são alocadas nos processadores/núcleos considerando os seus identificadores OpenMP como número do processador/núcleo, isso facilita o controle da seqüência de dados que deve ser alocada no nodo.

Essas estratégias foram empregadas na implementação das aplicações apresentadas na seção 4.2.

## 4.2. Aplicações

### 4.2.1 Ondes 3D

A aplicação Ondes 3D é uma aplicação real que simula a propagação de ondas em uma região. Esta aplicação é utilizada na previsão de sismos e geotérmicos. A sua principal característica é o alto consumo de memória (grandes matrizes de três dimensões) e de processamento. A aplicação possui três grandes etapas: a primeira é responsável pela inicialização das estruturas de dados, a segunda por calcular a velocidade da propagação da onda e a terceira pelo cálculo do estresse na região. O padrão de acessos aos dados durante as três etapas da aplicação é regular (*threads* sempre acessam o mesmo conjunto de páginas). A aplicação foi desenvolvida em C utilizando diretivas OpenMP para a paralelização do código [9].

### 4.2.2 Benchmark NAS

O *Benchmark* NAS é um conjunto de aplicações derivadas de aplicações espaciais (dinâmica de fluídos) usado amplamente para avaliação de desempenho em processamento paralelo. Esse *benchmark* é composto de cinco *kernels* e três aplicações que simulam a dinâmica de fluídos [8]. A principal diferença entre um *kernel* e uma aplicação é que o primeiro representa o núcleo de computação de métodos numéricos e a segunda simula a movimentação de dados e computação que existe em dinâmica de fluídos.

Para os experimentos foram selecionados os *kernels* CG e MG por eles terem como principais características um

padrão de acesso irregular aos dados (*threads* não trabalham sempre com o mesmo conjunto de páginas da memória), grandes conjuntos de estruturas de dados (alto consumo de memória) e grande número de acesso a eles. O CG é um *kernel* que utiliza o método *Conjugate Gradient* para calcular o menor auto-valor em uma matriz de grande dimensão e não estruturada. O *kernel* MG utiliza o método *V-cycle MultiGrid* para calcular a solução 3D para equação de Poisson.

### 4.3. Plataforma Experimental

Nos experimentos foram utilizados duas máquinas NUMA diferentes. A primeira, é um Itanium2 composta por 16 processadores de 1,6 GHz de freqüência e memória principal total de 64 Gbytes. A máquina está organizada em quatro nodos com quatro processadores com cache L3 compartilhada de 9 Mbytes. Esta memória principal está dividida em 4 blocos de 16 Gbytes, um em cada nodo da arquitetura. A largura de banda para acesso à memória principal é de 1.157,05 Mbytes/sec para leituras e 780,33 Mbytes/sec para escritas (resultados obtidos com Stream Benchmark [19]). Os nodos são conectados usando FAME Scalability Switch (FSS) que é um *backplane* proprietário desenvolvido pela BULL (www.bull.fr). Esta conexão gera diferentes latências na arquitetura, fator NUMA de 2 a 2,5. A distribuição Linux usada nesta máquina é a Bull Linux AS4 V5.1 (versão Kernel 2.6.18-intel-64). Os compiladores usados são: Intel C Compiler (ICC) e GNU Compiler Collection (GCC). A Figura 1 apresenta o esquema desta arquitetura.

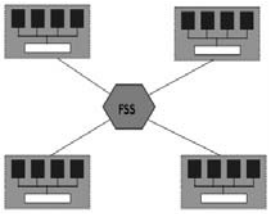


**Figura 1. Arquitetura NUMA - Itanium2**

A segunda arquitetura é um AMD Opteron de 8 processadores *dual core* com 2,2 GHz e cache L2 de 2 Mbytes para cada processador. A máquina está organizada em 8 nodos, uma capacidade total de 32Gbytes de memória, dividida igualmente entre os nodos. A largura de banda para acesso a memória principal é de 1.617,08 Mbytes/sec para leituras e 1.458,89 Mbytes/sec para escritas (resultados obtidos com Stream Benchmark [19]). Cada nodo possui três conexões que são usadas para conectar com outros nodos ou com cartões de entrada/saída. Essas conexões geram diferentes latências na arquitetura, fator NUMA de 1,2 a 1,5. A distribuição Linux usada nesta máquina é a Debian (versão Kernel 2.6.23-1amd64). Os compiladores usados são: Portland Group Inc. (PGI) e GCC. A Figura 2 apresenta o seu esquema.

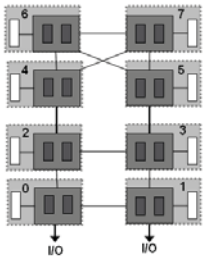


**Figura 2. Arquitetura NUMA - AMD Opteron**

## 5. Resultados

Esta seção apresenta a avaliação de desempenho das diferentes implementações das aplicações (`First-Touch`, `Parallel-Init`, `Round-Robin` e `Memory-Bind`) nas duas arquiteturas selecionadas. Os testes foram realizados com 2, 4, 8 e 16 *threads* e instância do problema de tamanho 2,4 Gbytes para aplicação real Ondes 3D e classe A ($256^2$ elementos e 20 iterações) para as aplicações do *benchmark* NAS. Os tempos de execução apresentados, para cada implementação e número de *threads*, são valores médios com validade estatística.

### 5.1. Arquitetura Itanium

A Figura 3 apresenta o tempo de execução obtido com a aplicação Ondes 3D (Compilador ICC) quando executada na arquitetura Itanium. Como podemos observar os resultados obtidos com a estratégia `Memory-Bind` foram na média 22% melhores que os resultados obtidos com a solução `Round-Robin`. Essa máquina possui fator NUMA alto, dados e *threads* devem ser alocados considerando uma distância mínima entre eles quando a aplicação tem como principal característica um padrão de acessos regular aos dados. Nesse caso, distribuir os dados ciclicamente gera um número maior de acessos remotos. Considerando a implementação `First-touch`, os resultados obtidos foram 5% piores que os resultados obtidos com `Memory-Bind`. Nesta implementação os dados são alocados fisicamente no nodo do processo mestre e as *threads* fazem acessos remotos e concorrentes para acessá-los.

Outro resultado interessante é que os resultados obtidos com a implementação `Memory-Bind` são próximos aos resultados obtidos com `Parallel-Init`. Como a implementação `Parallel-Init` usa diretivas OpenMP

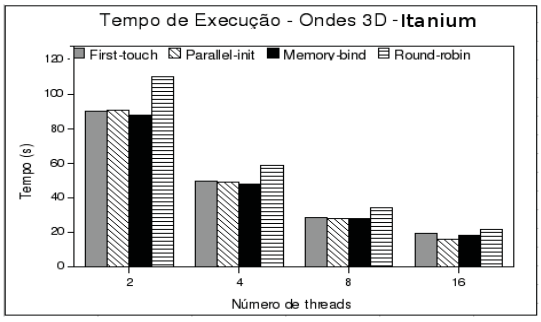


**Figura 3. Tempo Execução - Ondes 3D - Itanium**

na etapa de inicialização e a aplicação tem padrão de acesso regular, os dados são alocados fisicamente próximos das *threads* que os utilizam. Entretanto, é importante ressaltar que os resultados obtidos com `Memory-Bind` foram 2% melhores que os obtidos com `Parallel-Init`.

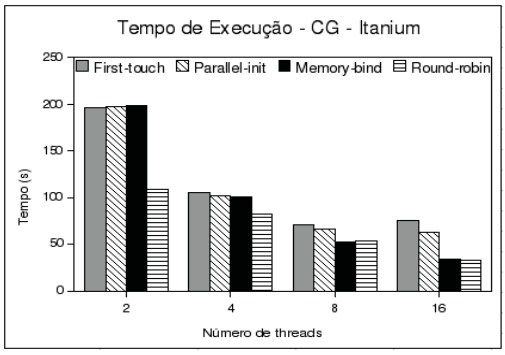


**Figura 4. Tempo Execução - CG - Itanium**

A Figura 4 fornece o tempo de execução obtido com o *kernel* CG (Compilador ICC) quando executado na arquitetura Itanium. Os resultados obtidos com `Round-Robin` foram na média 80% melhores que a versão `First-Touch` e 50% melhores que `Parallel-Init`. O *kernel* possui padrão de acesso irregular aos dados e distribuir as páginas de memória que contêm os dados ciclicamente entre os nodos da máquina NUMA, maximiza o número de acessos locais. A estratégia `Memory-Bind` apresentou resultados melhores que as estratégias `First-Touch` e `Parallel-Init` (27% e 39%), porém não melhores que a estratégia `Round-Robin`. Essa estratégia realiza muitas migrações de páginas da memória para minimizar o número de acessos remotos. Entretanto, o custo das migrações é alto pois, o padrão de acesso aos dados muda muito ao longo da aplicação (*threads* acessam diferentes conjuntos de páginas ao longo da execucão).

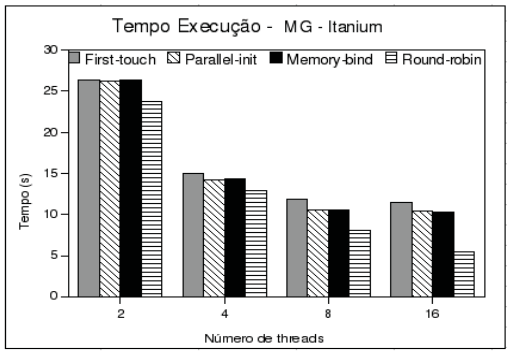


**Figura 5. Tempo Execução - MG - Itanium**

A Figura 5 mostra o tempo de execução obtido com o *kernel* MG (Compilador ICC) quando executado na arquitetura Itanium. Pode-se observar que a estratégia `Round-Robin`, na média, apresenta tempos de execução 45% melhores que `First-Touch` e 35% melhores que `Parallel-Init`. Assim, como o *kernel* CG, este *kernel* possui grandes estruturas de dados com padrão de acesso irregular. Então, distribuir os dados ciclicamente entre os nodos da máquina minimiza o número de acessos remotos e o número de migrações de páginas da memória. Como a máquina apresenta um custo alto para acessos remotos, minimizar o número de acessos remotos produz o melhor desempenho. Outro resultado importante, é a proximidade dos tempos de execução obtidos com as estratégias `Parallel-Init` e `Memory-Bind`. Nesse caso, o custo das migrações das páginas memória torna-se comparável ao custos dos acessos remotos que as *threads* realizam com a estratégia `Parallel-Init`.

A arquitetura Itanium possui um fator NUMA alto e uma largura de banda para memória local pequena. Portanto, neste tipo de arquitetura torna-se interessante trabalhar na latência para minimizar o número de acessos remotos e migração de páginas da memória.

## 5.2. Arquitetura Opteron

A Figura 6 mostra o tempo de execução obtido com a aplicação real Ondes 3D (Compilador PGI) executada na arquitetura Opteron. Como podemos observar, os resultados obtidos com a estratégia `Round-Roubin` foram, na média, 7% melhores que os resultados obtidos com a solução `Parallel-Init`. Essa máquina possui fator NUMA baixo, ou seja, a latência para acessos remotos é baixa. Portanto, a melhor solução é distribuir os dados entre os nodos minimizando o número de acessos concorrentes ao mesmo bloco de memória (contenção de memória).

Considerando ainda a Figura 6, observa-se que os piores

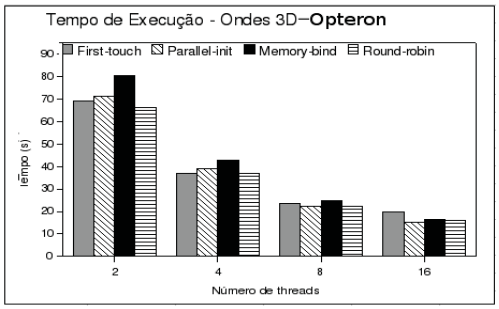


**Figura 6. Tempo Execução - Ondes 3D - Opteron**

resultados foram obtidos com a estratégia `Memory-Bind`. Nesta estratégia apenas as estruturas mais importantes foram usadas na alocação explícita com `mbind`. Na versão `Round-Robin`, a política *interleave* foi usada em toda a memória de dados da aplicação, ou seja, afinidade de memória garantida para todos os dados.

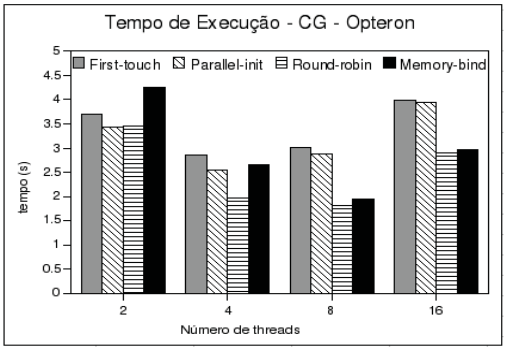


**Figura 7. Tempo Execução - CG - Opteron**

A Figura 7 apresenta o tempo de execução obtido com o *kernel* GC do NAS quando executado na arquitetura Opteron. Os resultados obtidos com a estratégia `Round-Roubin` foram, na média, 30% melhores que os resultados obtidos com as outras soluções. Essa máquina possui um fator NUMA pequeno e priorizar a largura de banda à latência gera melhores resultados. Além disso,este *kernel* possui como principal característica acessos irregulares à matriz. Então, distribuir os dados ciclicamente entre os nodos da máquina, como na estratégia `Round-Roubin`, favorece a largura de banda e o padrão de acesso.

Outro resultado importante é a degradação do desempenho (escalabilidade) com o aumento do número de *threads* utilizadas. Essa degradação esta principalmente ligada aos aspectos de cache e a forma como a aplicação foi paralelizada com as diretivas OpenMP. A paralelização do código foi realizada considerando laços de grão fino e um número maior de *threads* gera uma sobrecarga maior nestes laços. O custo de criar/lançar as *threads* é grande se comparado ao trabalho a ser realizado.

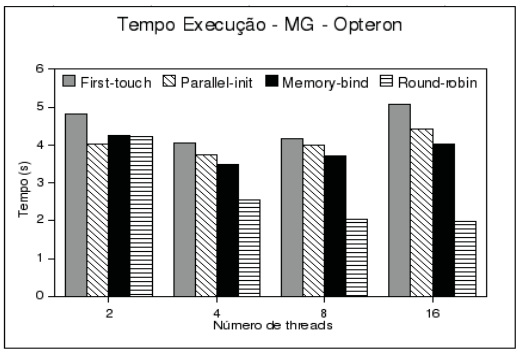


**Figura 8. Tempo Execução - MG - Opteron**

Por fim, a Figura 8 ilustra o tempo de execução obtido com o *kernel* MG (compilada com GCC) executado na arquitetura Opteron. Os resultados apresentados nessa figura, assim como os das duas figuras anteriores, mostram que os melhores tempos de execução são obtidos com a estratégia `Round-Roubin`. Esta solução favorece a largura de banda e, conseqüentemente, gera os melhores resultados. Considerando os resultados obtidos com a estratégia `Memory-Bind`, pode-se observar que, na média, são melhores que os resultados da estratégia `Parallel-Init`. Na estratégia `Memory-Bind` são realizadas migrações de páginas quando as *threads* acessam outro conjunto de dados.

Ainda na Figura 8, podemos observar que a solução `Round-Roubin` é a única que não apresenta degradação da aceleração (*speedup*). As demais soluções apresentam diminuição no ganho obtido quando aumentamos o número de *threads*. Esse resultado pode ser explicado pelas características da máquina e da aplicação. Esta máquina NUMA possui problema de contenção, logo, alocar as páginas considerando quem primeiro fez acesso a elas (`First-Touch` e `Parallel-Init`) aumenta o número de acessos concorrentes ao mesmo bloco de memória. Além disso, a aplicação possui padrão de acessos irregular; *threads* não acessam sempre o mesmo conjunto de dados.

A arquitetura Opteron possui um fator NUMA pequeno que torna o custo dos acessos remotos próximos ao custo dos acessos locais. Além disso, a largura de banda da memória local é alta, ou seja, tempo de acesso à memória local é comparável ao tempo de acesso a memória cache L2. Conseqüentemente, otimizar a largura de banda produz melhores resultados que otimizar a latência. Nossos resultados confirmam isso, pois a estratégia de afinidade de memória `Round-Robin`, que distribui *threads* e dados nos

nodos da máquina NUMA produziu os melhores resultados para as aplicações estudadas.

## 6. Conclusões

Neste artigo, foi apresentado a avaliação de desempenho de diferentes estratégias para afinidade de memória em uma aplicação real e em *kernels* do *benchmark* NAS. Essas estratégias foram implementadas com chamadas de sistema do Linux e permitem um controle explícito da alocação de memória e *threads* em máquinas NUMA.

Os resultados apresentados mostram a importância de gerenciar a memória das aplicações que são executadas em máquinas NUMA. Observamos ganhos de até 80% com as estratégias que usam chamadas do sistema operacional para otimizar afinidade de memória. O trabalho realizado justifica a importância de escolher a estratégia a ser usada na aplicação e que esta, depende de características da máquina NUMA e da própria aplicação. As chamadas do sistema permitem ao desenvolvedor implementar aplicações considerando as suas características e as da arquitetura NUMA podendo gerar ganho médio entre 2% - 80% em relação às implementações que não consideram afinidade de memória.

A principal contribuição deste trabalho é a avaliação de desempenho de diferentes estratégias para afinidade de memória em arquiteturas NUMA implementadas com APIs do sistema operacional. Como trabalhos futuros nós destacamos: estudo de mecanismos para a gerência de memória em outros sistemas operacionais, identificação de características da arquitetura e da aplicação que influenciam a afinidade de memória e proposta de um *framework* para controle da afinidade de memória em arquiteturas NUMA.

## 7. Agradecimentos

Agradecemos o apoio financeiro dos órgãos e projetos NUMASIS, ANR, Bull e CAPES (Processo 4874-06-4) na realização deste trabalho.

## Referências

[1] Top500 supercomputing sites - http://www.top500.org, 2008.

[2] T. Mu, J. Tao, M. Schulz, and S. A. Mckee. Interactive Locality Optimization on NUMA Architectures. In *Software Visualization*, 2003.

[3] H. Löf and S. Holmgren. Affinity-on-next-touch: Increasing the performance of an industrial PDE solver on a cc-NUMA system. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 387–392, New York, NY, USA, 2005. ACM.

[4] J. Marathe and F. Mueller. Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–99, New York, NY, USA, 2006. ACM.

[5] A. Carissimi, F. Dupros, J.-F. Mehaut, and R. V. Polanczyk. Aspectos de Programacão Paralela em arquiteturas NUMA. In *VIII Workshop em Sistemas Computacionais de Alto Desempenho*, 2007.

[6] F. Bellosa and M. Steckermeier. The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors. *J. Parallel Distrib. Comput.*, 37(1):113–121, August 1996.

[7] A. Joseph, J. Pete, and R. Alistair. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. *High Performance Computing - HiPC 2006*, pages 338–352, 2006.

[8] J. Y. Haoqiang Jin, Michael Frumkin. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report 99-011/1999, NAS System Division - NASA Ames Research Center, 1999.

[9] F. Dupros, H. Aochi, A. Ducellier, D. Komatitsch, and J. Roman. Exploiting intensive multithreading for the efficient simulation of seismic wave propagation. In *11th International Conference on Computational Science and Engineering*, Sao Paulo, Brazil, July 2008.

[10] The openmp specification for parallel programming - http://www.openmp.org, 2008.

[11] F. Garcia and J. Fernandez. Posix thread libraries. *Linux J.*, page 36, 2000.

[12] C. Terboven, An, and S. Sarholz. Openmp on multicore architectures. In *A Practical Programming Model for the Multi-Core Era*, pages 54–64. Springer, 2008.

[13] A. Kleen. A NUMA API for LINUX. Technical report, Novell, April 2005.

[14] J. Corbalan, X. Martorell, and J. Labarta. Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 121–129, New York, NY, USA, 2003. ACM.

[15] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 279–289, New York, NY, USA, 1996. ACM.

[16] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA machines. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, 2000.

[17] R. Love. Kernel korner: CPU affinity. *Linux Journal*, 2003(111):8, 2003.

[18] L. T. Schermerhorn. Automatic Page Migration for Linux. *Linux*, 2007.

[19] J. D. Mccalpin. STREAM: Sustainable memory bandwidth in high performance computers, 1995.