Ajuste Dinâmico de Threads para Execução Eficiente de Aplicações Iterativas OpenMP

Marcio N. P. Silva[1], **Aline P. Nascimento**[2], **Alexandre C. Sena**[1]

[1]Instituto de Matemática e estatística – Universidade do Estado do Rio de Janeiro
Rio de Janeiro – RJ – Brasil

[2]Instituto de Computação – Universidade Federal Fluminense
Niterói – RJ – Brasil

marcionps@gmail.com, asena@ime.uerj.br, aline@ic.uff.br

Resumo. *Duas abordagens distintas podem ser adotadas para aumentar o desempenho das aplicações paralelas em uma arquitetura de alto desempenho: (i) ferramentas de auto-tuning; (ii) tornar ou criar aplicações que consigam se adaptar ao ambiente disponível. Enquanto a primeira solução requer que o programador adapte sua aplicação a ferramenta e execute um profiling do ambiente, a segunda estratégia é mais complexa, pois necessita que o cientista seja capaz de criar aplicações com algum grau de autonomia. Assim, este trabalho propõe e avalia uma estrategia para ajustar dinamicamente a quantidade de theads de aplicações iterativas OpenMP. Os resultados obtidos mostram a viabilidade da estratégia proposta, que foi capaz de executar a aplicação para o problema de leilão eficientemente.*

1. Introdução

Achar soluções computacionais para problemas importantes em muitas áreas da ciência é uma tarefa árdua. Em razão da complexidade desses problemas, com frequência são necessários anos para desenvolver aplicações que consigam resolver o problema com precisão e, muitas vezes, serem adaptadas para executarem em ambientes de alto desempenho. Após a aplicação ser desenvolvida, ou mesmo durante seu desenvolvimento, o ambiente computacional de alto desempenho disponível pode sofrer mudanças, de maneira que a aplicação tenha que ser adaptada para este novo ambiente.

Uma solução para aumentar o desempenho de uma aplicação nas arquiteturas de alto desempenho disponíveis é utilizar ferramentas de *auto-tuning* para encontrar valores para parâmetros críticos para a execução como, por exemplo, quantidade de *threads* [Ansel et al. 2014, Rasch et al. 2017]. Porém, neste caso, os programadores tem que adaptar a aplicação e utilizar uma ferramenta de *auto-tuning* antes da execução real da aplicação. Além disso, esta ferramenta terá que ser utilizada toda vez que a aplicação tiver que ser executada em um novo ambiente computacional ou sofrer alguma modificação.

Uma segunda estratégia seria criar aplicações que consigam se adaptar as características dos ambientes disponíveis. Neste caso, segundo a classificação de aplicações paralelas definidas por Feitelson and Rudolph [Feitelson and Rudolph 1996], seria tornar as aplicações, originalmente RÍGIDAS ou MOLDÁVEIS, em aplicações MALEÁVEIS.

Assim, este trabalho apresenta uma estratégia para ajustar dinamicamente a quantidade de *threads* de uma aplicação iterativa OpenMP, de maneira que ela consiga executar eficientemente no ambiente disponível. Ou seja, tornar essas aplicações MALEÁVEIS.

Um estudo de caso com a aplicação paralela para o problema de leilão mostra que a estratégia proposta é eficiente, conseguindo resultados melhores do que os obtidos com a melhor execução paralela estática.

O restante deste trabalho está organizado da seguinte forma: (*i*) na Seção 2 são apresentados os trabalhos relacionados; a Seção 3 apresenta a estratégia para ajuste dinâmico de *threads* proposta; a análise experimental é apresentada na Seção 4.2; por fim, conclusões e trabalhos futuros são apresentados na Seção 5.

2. Trabalhos Relacionados

Uma abordagem para aumentar o desempenho de uma aplicação nas arquiteturas de alto desempenho disponíveis é o uso de ferramentas de *auto-tuning* para encontrar valores para parâmetros críticos para a execução, como por exemplo, quantidade de *threads* ou tamanho de bloco (*tiling*) [Hartono et al. 2009, Ansel et al. 2014, Rasch et al. 2017]. Entretanto, o uso dessas ferramentas requer a realização das três etapas a seguir: (i) geração de um espaço de busca; (ii) implementação de uma função de custo; (iii) explorar o espaço de busca [Rasch et al. 2017]. O problema dessas soluções que elas são específicas para cada aplicação e têm que ser refeitas toda vez que se mudar a arquitetura.

Uma segunda estratégia pode ser o ajuste dinâmico do número de processos/*threads* durante a execução. Por exemplo, um escalonador de *jobs* para aplicações capazes de ajustar a sua execução a quantidade de processadores disponíveis pode ser visto em [Sudarsan and Ribbens 2010]. Para melhorar a vazão dos *jobs* e o desempenho de cada *job* MPI, o escalonador é capaz de expandir ou diminuir a quantidade de processadores alocados a cada *job* dinamicamente. Lorenzon em [Lorenzon et al. 2019] realizou trabalho análogo utilizando um escalador de *threads* em OpenMP.

Baseados nesta mesma estratégia, duas abordagens para transformar aplicações iterativas MPI em aplicações maleáveis podem ser vistas em [El Maghraoui et al. 2009] e em [Sena et al. 2013]. Enquanto na primeira proposta é necessário chamar procedimentos para dividir/juntar processos para diminuir/aumentar a granularidade dos processos e, em seguida, migrar esses processos de acordo com os recursos disponíveis, na segunda proposta a aplicação MPI é transformada em uma aplicação autônoma, capaz de auto-ajustar a quantidade e granularidade de seus processos.

3. Ajuste Dinâmico de Threads

Com o objetivo de aumentar o desempenho e/ou diminuir o consumo de energia, aplicações devem ser otimizadas de acordo com a arquitetura disponível. Esta seção apresenta uma estratégia para ajustar dinamicamente o número de *threads* de aplicações iterativas OpenMP. É importante destacar que, diferentemente das ferramentas de *auto-tuning* (Seção 2), nas quais os programadores tem que adaptar a aplicação e usar a ferramenta antes da execução real da aplicação, a estratégia proposta pode ser usada em qualquer aplicação iterativa OpenMP e é capaz de ajustar a execução real da aplicação dinamicamente e transparentemente, sem que seja necessário qualquer ajuste do programador.

Feitelson and Rudolph [Feitelson and Rudolph 1996] classificaram as aplicações paralelas em 4 categorias de acordo com a sua capacidade de ajustar sua execução a quantidade de processadores disponíveis. Uma aplicação RÍGIDA só pode ser executada em

uma quantidade fixa de recursos específicos. Por sua vez, uma aplicação MOLDÁVEL pode ser executada em uma quantidade variável de processadores, onde o número de processadores deve ser decidido na submissão da aplicação. Ao contrário, as aplicações EVOLUTIVAS e MALEÁVEIS são capazes de adaptar sua execução de acordo com a mudança na quantidade de processadores disponíveis durante a execução.

Assumindo que as arquiteturas *Multicore* disponíveis atualmente apresentam uma grande variedade tanto na quantidade de recursos disponíveis (núcleos de processamento) como também na capacidade de processamento dos recursos e no tipo de arquitetura disponível (e.g. NUMA), as características das aplicações MALEÁVEIS apresentam um grande potencial para serem executadas eficientemente nestes ambientes.

Neste trabalho uma aplicação iterativa é definida como uma sequência de instruções que são executadas repetidamente, através da utilização de um ou mais laços. Assim, cada passo do laço principal é definido como uma iteração. Em uma típica aplicação OpenMP, é necessário definir a quantidade de *threads*, t, que será utilizada durante a execução do programa. O valor de t irá definir o grau de paralelismo da aplicação.

A estratégia proposta deveria definir o número de *threads* da iteração seguinte com base no tempo de execução de iterações anteriores. A princípio, cada iteração deveria ser executada com uma quantidade de *threads* que maximizasse seu desempenho, porém a sobrecarga imposta para executar uma heurística que calcule o grau de paralelismo ideal a cada iteração torna essa opção impraticável.

3.1. Estratégia Proposta

O objetivo da estratégia proposta é descobrir o grau de paralelismo ideal (quantidade de *threads* t) o mais rapidamente possível, ou seja, através do menor número de iterações. Para isso, o espaço de busca de soluções é definido como a quantidade máxima de *threads* disponível na arquitetura *multicore* disponível.

Como a heurística a ser usada deve ser executada entre iterações das aplicações iterativas OpenMP, os métodos não determinísticos não foram considerados por geralmente serem computacionalmente caros. Considerando que o número de valores a serem avaliados é relativamente pequeno, uma técnica de *hill climbing* [Russell and Norvig 2009] poderia ser uma boa escolha. Essa heurística gulosa calcula os pontos adjacentes do espaço de busca e se move na direção que otimiza o valor da função objetivo até que um valor pior seja encontrado, ressaltando que a heurística está sujeita a mínimos locais.

Inicialmente, para achar o número ideal de *threads*, t, a ser utilizado em uma aplicação iterativa OpenMP foram propostas quatro heurísticas baseadas em *hill climbing* e busca binária. O funcionamento das quatro heurísticas é baseado na comparação dos tempos de duas iterações consecutivas. Essas heurísticas foram denominadas *hillMax* (Figura 1(a)), *hillMin* (Figura 1(b)), *Bin* (Figura 2(a)) e *hillMinMax* (Figura 2(b)).

Sendo t_{max} o número máximo de cores disponível, T_i o número de *threads* usadas na iteração i e et_i o tempo de execução da iteração i, cada heurística pode ser descrita como a seguir:

• $hillMax$ (Figura 1(a)): A heurística é chamada ao final de cada iteração do programa OpenMP. Na primeira iteração, a quantidade de threads utilizada é $t = 1$, em seguida ao final de cada iteração i, o tempo da iteração atual et_i é comparado com o tempo da

```
01 | int hillMax(double etAtual, int
   |     ↪ tAtual){
02 |   static double etAnterior =
   |       ↪ 999999;
03 |   static int  tAnterior;
04 |   static int  estado = 0;
05 |
06 |   if(estado == 0){
07 |     if(etAtual < etAnterior){
08 |       tAnterior = tAtual;
09 |       tAtual++;
10 |     }
11 |     else {
12 |       tAtual = tAnterior;
13 |       estado = 1;
14 |     }
15 |     etAnterior = etAtual;
16 |   }
17 |   return tAtual;
18 | }
```

(a) hillMax

```
01 | int hillMin(double etAtual, int
   |     ↪ tAtual){
02 |   static double etAnterior =
   |       ↪ 999999;
03 |   static int  tAnterior;
04 |   static int  estado = 0;
05 |
06 |   if(estado == 0){
07 |     if(etAtual < etAnterior){
08 |       tAnterior = tAtual;
09 |       tAtual--;
10 |     }
11 |     else{
12 |       tAtual = tAnterior;
13 |       estado = 1;
14 |     }
15 |     etAnterior = etAtual;
16 |   }
17 |   return tAtual;
18 | }
```

(b) hillMin

Figura 1. Códigos das heurísticas hillMax e hillMin

iteração anterior et_{i-1} (linha 7). Caso o valor de $et_i < et_{i-1}$ então a quantidade de *threads* da próxima execução é acrescido de 1 (linha 9). Caso contrário, a heurística termina (a varíavel $estado$ recebe o valor 1) e as iterações seguintes são executadas com o valor constante igual a número de *threads* utilizado na iteração anterior T_{i-1} (linha 12).

• $hillMin$ (Figura 1(b)): O funcionamento desta heurística é basicamente o inverso da heurística anterior. A heurística é chamada ao final de cada iteração do programa OpenMP. Na primeira iteração, a quantidade de threads utilizada é igual ao valor de t_{max}. Em seguida ao final de cada iteração i, o tempo da iteração atual et_i é comparado com o tempo da iteração anterior et_{i-1} (linha 7). Caso o valor de $et_i < et_{i-1}$ então a quantidade de *threads* da próxima execução é decrementado de 1 (linha 9). Caso contrário, a heurística termina e as iterações seguintes são executadas com o valor constante igual ao número de *threads* utilizado na iteração anterior T_{i-1} (linha 12).

• Bin (Figura 2(a)): A heurística é chamada ao final de cada iteração do programa OpenMP. Na primeira iteração, a quantidade de threads utilizada é igual ao valor de $T_{min} = 1$. Na segunda iteração, a quantidade de threads utilizada é igual ao valor de $T_{max} = t_{max}$. Em seguida, para cada iteração, se $et_{min} < et_{max}$ então $T_{max} = T_{bin}$ (linha 23), caso contrário $T_{min} = T_{bin}$ (linha 28), sendo $T_{bin} = (T_{max} - T_{min})/2$. Quando $T_{max} \leq T_{min}$ a heurística termina (linha 32) e as iterações remanescentes serão executadas com T_{min} *threads*.

• $hillMinMax$ (Figura 2(b)): A heurística é chamada ao final de cada iteração do programa OpenMP. Assim como em Bin, inicialmente é comparado o tempo de execução utilizando-se T_{min} e T_{max} threads. Inicialmente $T_{min} = 1$ e $T_{max} = t_{max}$. Nas iterações seguintes, caso $et_{min} < et_{max}$ (linha 20) o número de *threads* T_{max} é decrementado de 1

```
01 | int bin(double etAtual, int
       ↪ tAtual){
02 |   static double etMin = 0;
03 |   static double etMax = 0;
04 |   static int tMin = 1;
05 |   static int tMax = 36;
06 |   static int estado = 0;
07 |   static int lado = 0;
08 |
09 |   if(estado == 0){
10 |     etMax = etAtual;
11 |     tAtual = tMin;
12 |     estado = 1;
13 |     lado = 0;
14 |   }
15 |   else {
16 |     if (estado == 1) {
17 |       if(lado == 0)
18 |         etMin = etAtual;
19 |       else
20 |         etDir = etAtual;
21 |
22 |       if (etMin < etMax){
23 |         tMax = (tMax + tMin)/2;
24 |         tAtual = tMax;
25 |         lado = 1;
26 |       }
27 |       else {
28 |         tMin = (tMax - tMin)/2;
29 |         tAtual = tMin;
30 |         lado = 0;
31 |       }
32 |       if (tMax <= tMin)
33 |         estado = 2;
34 |       else
35 |         estado = 1;
36 |     }
37 |   }
38 |   return tAtual;
39 | }
```

(a) Bin

```
01 | int hillMinMax(double etAtual,
       ↪ int tAtual) {
02 |   static double etMin = 0;
03 |   static double etMax = 0;
04 |   static int tMin = 1;
05 |   static int tMax = 36;
06 |   static int estado = 0;
07 |
08 |   if(estado == 0){
09 |       etMax = etAtual;
10 |       pAtual = tMin;
11 |       estado = 0;
12 |       return 36;
13 |   }
14 |   else {
15 |     if (estado == 1) {
16 |       etMin = etAtual;
17 |     else
18 |       etMax = etAtual;
19 |
20 |       if (etMin < etMax) {
21 |         tMax--;
22 |         tAtual = tMax;
23 |       }
24 |       else {
25 |         tMin++;
26 |         tAtual = tMin;
27 |         lado = 0;
28 |       }
29 |
30 |       if (tMin == tMax)
31 |         estado = 2;
32 |       else
33 |         estado = 0;
34 |     }
35 |   }
36 |   return tAtual;
37 | }
```

(b) hillMinMax

Figura 2. Código das heurísticas Bin e hillMinMax

```
01 | int checkNUMA(double tAtual, int iter){
02 |   static double t36 = 0;
03 |
04 |   if(iter == 0) {
05 |     t36 = tAtual;
06 |     return 18;
07 |   }
08 |   if (iter == 1) {
09 |     if (t36 > tAtual)
10 |         return 18;
11 |     else
12 |         return 36;
13 |   }
14 | }
```

Figura 3. Algoritmo para verificação do custo de acesso à memória NUMA

(linha 21) e a próxima iteração é executada com T_{max} *threads*, caso contrário T_{min} é incrementado de 1 (linha 25) e a próxima iteração é executada com T_{min} *threads*. A heurística termina quando $T_{min} = T_{max}$ (linha 30) e as iterações seguintes serão executadas com T_{min} *threads*.

Uma característica marcante da arquitetura de memória das novas máquinas *multicore* é ser do tipo NUMA (*Non-Uniform Memory Access*). Neste tipo de arquitetura o tempo de acesso à memória depende da localização da memória em relação ao processador. Assim, um processador pode acessar sua própria memória local mais rapidamente do que a memória não local (ou seja, memória local a outro processador ou memória compartilhada entre processadores). Essa característica pode influenciar consideravelmente o desempenho das aplicações e, consequentemente, o desempenho da estratégia proposta. Nesse contexto, este trabalho também propõe uma estratégia para avaliar o custo da sobrecarga do acesso nas arquiteturas do tipo NUMA, que pode ser vista na Figura 3.

Na estratégia $checkNUMA$ as duas primeiras iterações são usadas para verificar a sobrecarga do acesso a memória não local que existe em toda arquitetura NUMA. A primeira iteração é executada com todos os *threads* disponíveis (por exemplo, 36), fazendo com que a memória mais distante seja utilizada. A segunda iteração é executada com a metade das *threads*, de maneira que apenas a memória mais próxima seja utilizada. Ao fim desta etapa o parâmetro t_{max} é definido com base nos resultados, ou seja, $t_{max} = 36$ se $et_2 > et_1$, ou $t_{max} = 18$ caso contrário. É importante destacar que apesar da estratégia proposta ter assumido uma arquitetura NUMA de apenas 2 níveis, a generalização para N níveis é bastante simples, sendo necessário N iterações para definir a quantidade de processadores máximo t_{max} a ser utilizado nas heurísticas.

4. Estudo de Caso

Esta seção descreve o estudo de caso realizado com o algoritmo de leilão para avaliar a estratégia de ajuste dinâmico de *threads* proposta. O algoritmo de leilão foi escolhido por apresentar algumas características interessantes: ser uma aplicação iterativa; ser importante para várias áreas da ciência; o trabalho a ser realizado em cada iteração decresce monotonicamente; é composto por duas etapas que são paralelizadas separadamente.

```
01 | main leilao(){
02 |   while qtdObjAssociado <
       ↪ TOTALOBJ do
03 |     tinicio = GET(time)
04 |     lance(T)
05 |     associacao(T)
06 |     tfim = GET(time)
07 |     titer = tfim - tinicio
08 |     if (iter == 0 || iter == 1)
09 |         checkNuma(titer)
10 |     if (iter >= 2)
11 |         T = heuristica(titer)
12 |   end while
13 | }
```

(a) Estratégia com checkNUMA

```
01 | main leilao(){
02 |   while qtdObjAssociado <
       ↪ TOTALOBJ do
03 |     tinicio = GET(time)
04 |     lance(Tlance)
05 |     tfim = GET(time)
06 |     titer1 = tfim - tinicio
07 |     if (iter >= 2)
08 |         Tlance = heuristica(
               ↪ titer1)
09 |     tinicio = GET(time)
10 |     associacao(Tassoc)
11 |     tfim = GET(time)
12 |     titer2 = tfim - tinicio
13 |     if (iter >= 2)
14 |         Tassoc = heuristica(
               ↪ titer2)
15 |     if (iter == 0 || iter == 1)
16 |         titer = titer1 + titer2
17 |         checkNuma(titer)
18 |   end while
19 | }
```

(b) Estratégia de Ajuste para cada etapa

Figura 4. Ajuste dinâmico no Algoritmo de Leilão

4.1. Algoritmo de Leilão

O algoritmo de leilão foi originalmente proposto para atribuir m pessoas a n objetos distintos, onde cada par pessoa/objeto possui um custo associado que representa sua afinidade. O objetivo do algoritmo é atribuir uma pessoa ao objeto com maior afinidade, maximizando a soma total [Bertsekas 1979]. O Algoritmo de Leilão é basicamente um laço principal composto de duas fases principais: (1) LANCE, onde as pessoas dão lances para os objetos que elas desejam se associar, e (2) ASSOCIAÇÃO, na qual o melhor lance dado para cada objeto é selecionado individualmente, determinando suas associações e novos preços. O Algoritmo repete essas duas fases até que todos os objetos estejam associados, como pode ser visto na Figura 4(a). Explicações detalhadas sobre o algoritmo de leilão podem ser encontradas em [Bertsekas 1979, Nascimento et al. 2016]. Mais especificamente, para avaliar a estratégia de ajuste dinâmico proposta, neste trabalho é utilizada a implementação vetorizada do algoritmo de leilão apresentada em [Sena et al. 2017]. Esta versão se mostrou eficiente e foi paralelizada utilizando a biblioteca OpenMP.

Para aplicar a estratégia de ajuste dinâmico no algoritmo de leilão basta obter o tempo de cada iteração, como pode ser visto na Figura 4(a) (linha 7), e passá-lo como parâmetro para a rotina $checkNUMA$ (linha 9) e para a heurística que estiver sendo utilizada para o ajuste dinâmico ($HillMax$, $HillMin$, Bin ou $MinMax$) (linha 11).

4.2. Análise Experimental

Todos os experimentos deste trabalho foram realizados em uma máquina multicore NUMA (*Non-Uniform Memory Access*) com 36 núcleos de processamento (2 *chips* Intel®

Xeon® CPU E5-2699 v3 @ 2.30GHz) com 128 GB de RAM. Esse processador conta com instruções SIMD do tipo AVX2 de 256 bits. Todos os programas foram compilados utilizando o compilador icc da Intel (versão 18.0.2) com otimizacão -O3. A afinidade entre threads foi configurada de modo a ficar próxima da thread master pela variável de ambiente do OpenMP *OMP_PROC_BIND*.

Para todos os experimentos foram empregadas 5 matrizes. As matrizes $startgarder3$, $neubrandeburg$ e $notredame$ são usadas para o problema de emparelhamento de duas imagens, onde os pontos da primeira imagem são representados nas linhas, enquanto que os pontos da segunda imagem nas colunas. Cada elemento da matriz representa a afinidade de um ponto da primeira imagem com um ponto da segunda imagem. É importante destacar que o tempo de execução para emparelhar essas matrizes corresponde apenas ao tempo necessário para se emparelhar duas imagens. Assim, o emparelhamento de uma sequência de imagens de um filme, por exemplo, aumentaria consideravelmente o tempo de execução. Por sua vez, as matrizes $m8k\_7447$ e $m16k\_9115$ são usadas para realizar o pareamento entre partículas em determinados instantes de tempo.

No primeiro experimento foi executado o algoritmo de leilão sem utilizar a estratégia proposta. Cada instância foi executada 30 vezes e a média aritmética do tempo de execução calculada, assim como o intervalo de confiança de 95%. O objetivo deste experimento é encontrar a quantidade de *threads* que produz o melhor tempo de execução. Para isso, para cada uma das matrizes, o algoritmo de leilão foi executado variando a quantidade de *threads* de 2 a 36. O resultado pode ser visto na Figura 5. A primeira observação interessante sobre os resultados obtidos é que a quantidade de *threads* que proporcionou os melhores resultados para cada uma das matrizes variou bastante, sendo 4 *threads* para as matrizes $Notredame$ e $Neubrandeburg$, 8 para a matriz $Startgarder3$, 12 para a matriz $m8k7447$ e 18 para a matriz $m16k9115$. O motivo para tal variação é a quantidade do trabalho realizado em cada iteração que varia de acordo com o tamanho da matriz que para esses exemplos são: $Notredame$ (3026×3297); $Neubrandeburg$ (2142×3526); $Startgarder3$ (4028×4484); $m8k7447$ (8000×8000); $m16k9115$ (16000×16000).

Uma segunda observação interessante sobre todos os gráficos da Figura 5 é um aumento considerável no tempo de execução de 18 para 19 *threads*. Isto ocorreu em função da sobrecarga de acessar a memória não local na arquitetura NUMA. Ou seja, toda vez que mais do que 18 *threads* forem utilizadas essa sobrecarga terá impacto no tempo de execução. Por fim, este experimento mostra que até mesmo para matrizes com tamanhos parecidos a quantidade de *threads* ideal pode variar o que motiva bastante a adoção da estratégia proposta neste trabalho, que será avaliada nos experimentos a seguir.

No segundo experimento foi executado o algoritmo de leilão com as heurísticas propostas neste trabalho para o ajuste dinâmico de *threads* $HillMax$, $HillMin$, Bin e $MinMax$, com e sem a execução da rotina $checkNUMA$. Cada instância foi executada 30 vezes e a média aritmética do tempo de execução calculada, assim como o intervalo de confiança de 95%. Os resultados podem ser visto na Figura 6. De maneira geral, as heurísticas com a estratégia $checkNUMA$ produziram tempos de execução menores. A razão para tal desempenho das heurísticas com a estratégia $checkNUMA$ é a diminuição considerável da sobrecarga de se acessar a memória remota (não local ao processador) nas arquiteturas NUMA, conforme explicado na Subseção 3.1.

Quando comparamos os melhores tempos obtidos com as heurística propostas

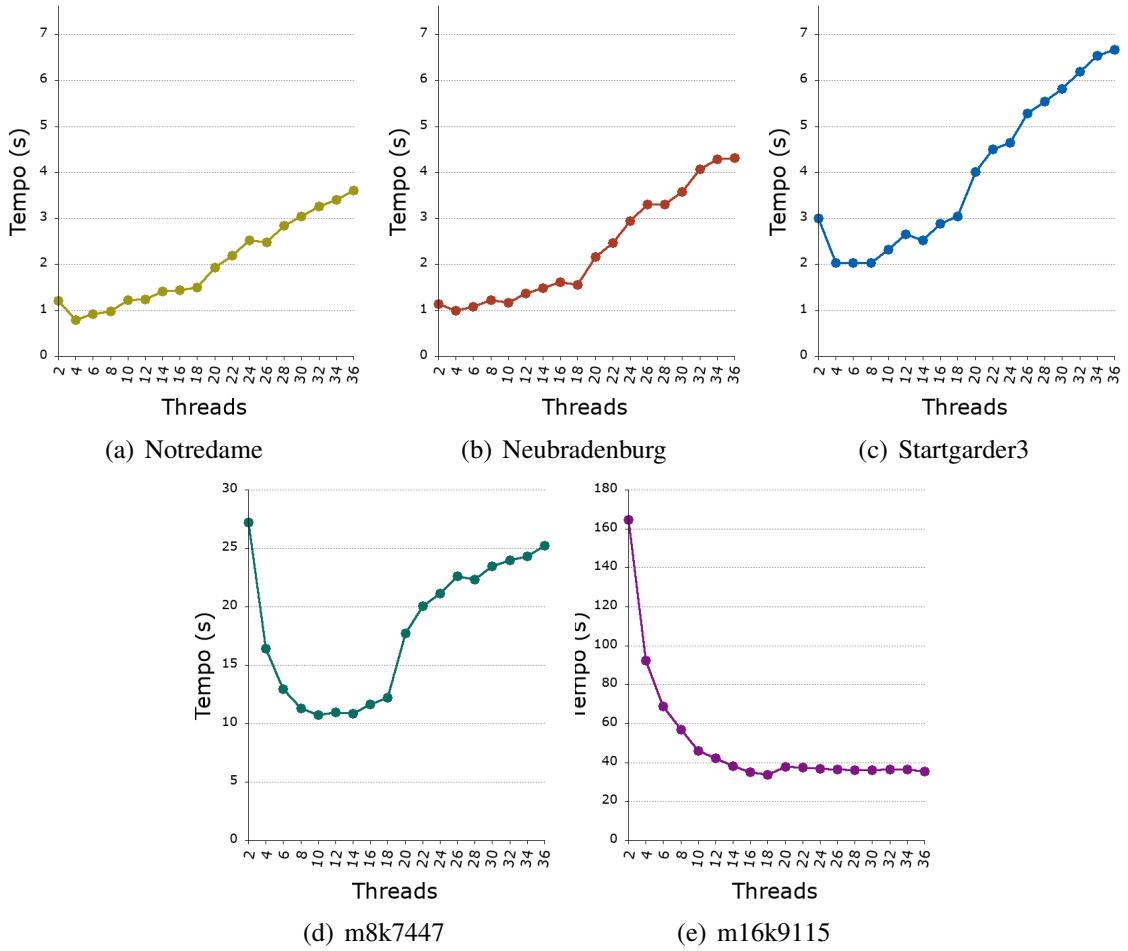
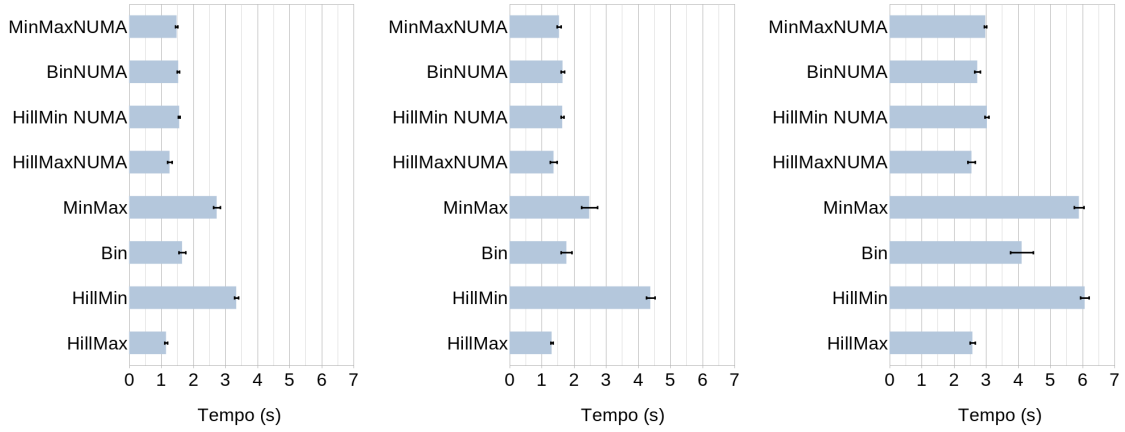


Figura 5. Tempo de execução do algoritmo de leilão com cada uma das matrizes de entrada e aumentando o número de *threads*

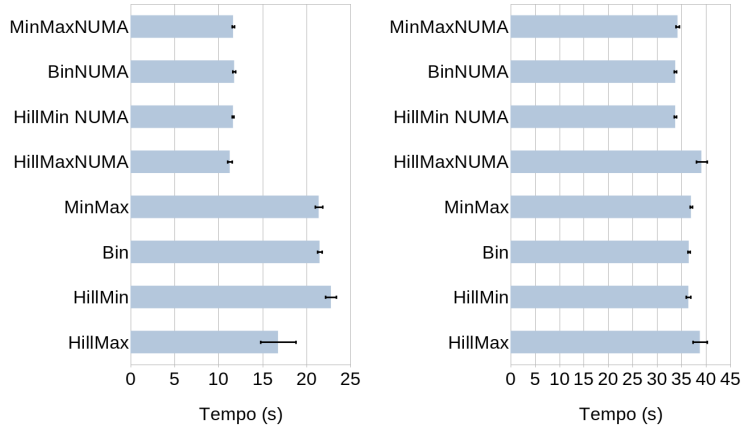
(com a estratégia $checkNUMA$) com os melhores tempos obtidos no experimento anterior, se verifica que para as matrizes menores a heurística proposta produz tempos $\approx 25\%$ mais lentos do que a MELHOR execução obtida no experimento anterior. Por outro lado, para as duas matrizes maiores, a perda é de menos de 5%, sendo que para a matriz $m16k9115$ praticamente não houve perda ($< 1\%$). A razão para a perda maior para as matrizes pequenas é que, mesmo com a baixa sobrecarga da estratégia proposta, o tempo para executar cada iteração é muito pequeno sofrendo uma influência maior do tempo de execução das heurísticas. Por fim, os resultados obtidos são confiáveis,como pode ser visto nos intervalos de confiança da Figura 6, sofrendo uma pequena variação em função das heurísticas estarem sujeitas a pararem em mínimos locais.

Devido, a duas características do algoritmo de leilão, (i) é composto por duas etapas que são paralelizadas separadamente e (ii) o trabalho a ser realizado em cada iteração decresce monotonicamente, duas outras estratégias foram testadas.

Como o algoritmo é composto de duas fases que são paralelizadas separadamente, ao invés de utilizar o tempo de execução de cada iteração total e calcular um valor t de *threads* único para as duas funções, como no experimento anterior, no próximo experimento cada heurística é aplicada para cada uma das etapas separadamente. Ou seja, são calcu-



(a) Notredame (b) Neubradenburg (c) Startgarder3



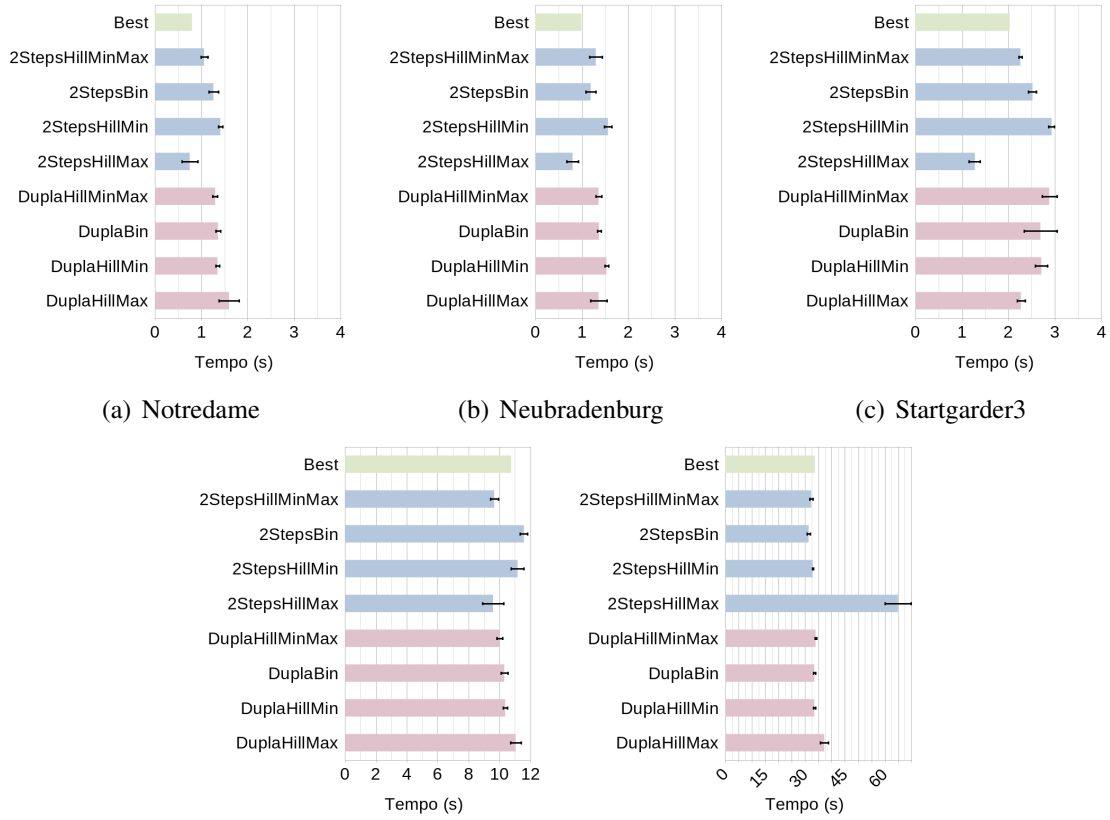
(d) m8k7447 (e) m16k9115

Figura 6. Tempo de execução com a estratégia de ajuste dinâmico

lados dois valores de *threads* distintos, um para a função LANCE (linha 8) e outro para a função ASSOCIAÇÃO (linha 14), como pode ser visto na Figura 4(b). Como a carga de trabalho realizado por cada uma das funções é diferente, a princípio, é possível que devam executar com uma quantidade de *threads* distinta, esta estratégia foi denominada Dupla.

Além disso, como a cada iteração do algoritmo de leilão o trabalho a ser realizado decresce monotonicamente e, mais do que isso, a maioria das associações ocorre nas primeiras iterações, foi testado também a estratégia Dupla com duas etapas (chamada de 2Steps). Ou seja, além de aplicar as heurísticas propostas para cada fase do algoritmo de leilão, as heurísticas são chamadas duas vezes, uma no início e outra após 400 iterações (melhor valor obtido após testes com 200, 400, 600 e 800). Assim, a estratégia proposta encontrará para cada função um valor de t que será usado do início até a iteração 400 e um outro valor de t que será usado da iteração 400 até o fim da execução. É esperado que o valor de t diminua após a iteração 400, produzindo uma execução mais eficiente.

A média de 30 execuções e o intervalo de confiança para as duas estratégias, Dupla e 2Steps, para cada uma das matrizes podem ser vistos na Figura 7, juntamente com a melhor execução paralela obtida no primeiro experimento. Como esperado os resultados para as duas estratégias foram melhores do que o experimento anterior, mostrando que



(a) Notredame (b) Neubradenburg (c) Startgarder3

(d) m8k7447 (e) m16k9115

Figura 7. Tempo de execução com as estratégias Dupla e 2Steps

realmente a quantidade de trabalho a ser computado é diferente, o que acarreta em quantidade de *threads* distintas para cada fase do algoritmo de leilão. Além disso, a estratégia de executar a heurística duas vezes (2Steps) se mostrou mais eficiente que a estratégia Dupla isoladamente. O que confirma a hipótese levantada que com a diminuição da carga de trabalho ao longo das iterações a variação na quantidade de *threads* necessárias para produzir o melhor resultado tende a variar.

Mais interessante, como o melhor resultado paralelo encontrado para cada matriz (Best) foi obtido com uma quantidade fixa de *threads*, vários resultados da estratégia 2Steps foram melhores do que o Best. Por exemplo, para matrizes pequenas a heurística $2StepsHillMax$ obteve resultados melhores que o Best, enquanto que para matrizes grandes a heurística $2StepsHillMinMax$ obteve resultados superiores ao Best.

5. Conclusões

Uma vez que para executar uma aplicação paralela eficientemente é necessário ajustar a aplicação paralela ao ambiente computacional disponível, este trabalho apresentou e avaliou uma estratégia para ajuste dinâmico de *threads* para aplicações iterativas OpenMP.

Os resultados são promissores, mostrando que quando o cientista não souber a quantidade ideal de *threads* a ser usada a estratégia proposta é bastante eficiente. Mais importante, quando a aplicação for composta de mais de uma etapa, o ajuste na quantidade de *threads* deve ser específico para cada etapa. Por fim, uma estratégia para aplicações

que a quantidade de trabalho diminui ao longo das iterações produziu resultados melhores do que a melhor execução paralela, o que mostra a relevância deste trabalho. Trabalhos futuros vão investigar técnicas para evitar que as heurísticas fiquem presas em mínimos locais, para aumentar a sua eficiência e diminuir a variação dos resultados.

Agradecimentos

Os autores agradecem o uso dos recursos computacionais *many-core* mantidos e operados pelo Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP), financiado parcialmente pela Intel, no contexto do projeto Intel/UNESP Modern Code.

Referências

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U. M., and Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. In *2014 Inter. Conf. on Par. Arch. and Compilation Techniques*, pages 303–315.

Bertsekas, D. P. (1979). A distributed algorithm for the assignment problem. Technical report, Lab. for Information and Decision Systems, M.I.T., Cambridge, MA.

El Maghraoui, K., Desell, T. J., Szymanski, B. K., and Varela, C. A. (2009). Malleable iterative mpi applications. *Concurr. Comput. : Pract. Exper.*, 21(3):393–413.

Feitelson, D. G. and Rudolph, L. (1996). Toward convergence in job schedulers for parallel supercomputers. In Feitelson, D. G. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer Berlin Heidelberg.

Hartono, A., Norris, B., and Sadayappan, P. (2009). Annotation-based empirical performance tuning using orio. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11.

Lorenzon, A. F., de Oliveira, C. C., Souza, J. D., and Beck, A. C. S. (2019). Aurora: Seamless optimization of openmp applications. *IEEE Transactions on Parallel and Distributed Systems*, 30(5):1007–1021.

Nascimento, A. P., Vasconcelos, C. N., Jamel, F. S., and Sena, A. C. (2016). A hybrid parallel algorithm for the auction algorithm in multicore systems. In *Inter. Symp. on Computer Architecture and High Perf. Comp. Workshops (SBAC-PADW)*, pages 73–78.

Rasch, A., Haidl, M., and Gorlatch, S. (2017). ATF: A generic auto-tuning framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications*, pages 64–71.

Russell, S. J. S. J. and Norvig, P. (2009). *Artificial intelligence : a modern approach*. Prentice Hall Press, 3 edition.

Sena, A., Ribeiro, F., Rebello, V., Nascimento, A., and Boeres, C. (2013). Autonomic malleability in iterative mpi applications. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 192–199.

Sena, A. C., Nascimento, A., Vasconcelos, C., and Marzulo, L. A. J. (2017). Execução eficiente do algoritmo de leilão nas novas arquiteturas multicore. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 18(1/2017).

Sudarsan, R. and Ribbens, C. J. (2010). Design and performance of a scheduling framework for resizable parallel applications. *Parallel Comp.*, 36(1):48 – 64.