Implementação e Avaliação do Algoritmo de Leilão nas Arquiteturas Xeon Phi

Alexandre C. Sena[1], Aline P. Nascimento[2] e Leandro A. J. Marzulo[1]

[1]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

[2]Instituto de Computação
Universidade Federal Fluminense (UFF), Rio de Janeiro, Brasil

{asena,leandro}@ime.uerj.br,aline@ic.uff.br

Resumo. *O algoritmo de leilão tem sido amplamente utilizado para resolver problemas de várias áreas. Com seus vários núcleos de processamento e instruções vetorizadas de 512* bits*, arquiteturas Xeon Phi tem potencial para aumentar consideravelmente o desempenho desse algoritmo. O objetivo deste trabalho é executar eficientemente o algoritmo de leilão nessas arquiteturas. As principais contribuições são: implementação de uma versão vetorizada; Análise de desempenho da versões vetorizada e paralela; comparação do desempenho entre Xeon e Xeon Phi. Resultados mostram que a versão vetorizada paralela é capaz de aproveitar todo o potencial das arquiteturas Xeon Phi, atingindo um desempenho até* 750 *vezes melhor do que a versão sequencial.*

1. Introdução

Existem diversos problemas que podem ser modelados através de grafos bipartidos, onde um emparelhamento ótimo deve ser encontrado. O problema de emparelhamento de grafos bipartidos é explorado em diversas áreas tais como a bioinformática, para verificar semelhanças entre proteínas [Kollias et al. 2013] e Visão Computacional, onde o objetivo é o emparelhamento de duas imagens, avaliando as semelhanças entre seus pontos [Shokoufandeh and Dickinson 1999]. Por ser capaz de produzir a solução ótima e sua natureza distributiva simplificar a adoção de implementações paralelas, o algoritmo de leilão [Bertsekas 1979] tem sido amplamente usado para solucionar esse problema.

Os processadores Xeon (*multicore*) e Xeon Phi (*many-core*) disponibilizam um conjunto de instruções SIMD e vários núcleos de processamento para aumentar o desempenho das aplicações. Quando comparada com sistemas *multicore*, a arquitetura *many-core* tem uma quantidade maior de núcleos com frequência de *clock* menor para manter o consumo de energia gerenciável. Ainda assim, os processadores Xeon Phi possuem registradores de 512 *bits*, podendo oferecer maior poder computacional por *watt*.

O objetivo deste trabalho é executar eficientemente o algoritmo de leilão nas arquiteturas *many-core*. Para isso, uma versão vetorizada completa foi implementada e avaliada. Além disso, foi analisado o impacto do tamanho das matrizes e quantidade de iterações no desempenho da aplicação. Além disso, uma implementação paralela OpenMP em conjunto com a vetorização mostrou que é possível explorar todo o potencial dessas arquiteturas, inclusive obtendo desempenho maior do que sistemas *multicore*, conseguindo um desempenho até 750 vezes melhor do que a versão original sequencial.

Este trabalho está dividido da seguinte maneira: Trabalhos relacionados são descritos na Seção 2. A Seção 3 descreve o algoritmo de leilão. Em seguida, a Seção 4 apresenta a versão vetorizada proposta. A implementação paralela OpenMP é apresentada na Seção 5. Uma análise detalhada da versão vetorizada proposta é apresentada na Seção 6. Por fim, as conclusões são apresentadas na Seção 7.

2. Trabalhos Relacionados

A natureza distribuída do algoritmo de leilão permitiu que surgissem diversas implementações em diferentes arquiteturas. No início da década de 90, implementações paralelas síncronas e assíncronas do algoritmo de leilão foram propostas e executadas em uma máquina paralela real [Bertsekas and Castañon 1991]. Em [Vasconcelos and Rosenhahn 2009] foi apresentada uma implementação do algoritmo de leilão para GPU.

Versões paralelas capazes de emparelhar grandes grafos bipartidos densos e esparsos foram apresentadas em [Kollias et al. 2014]. Resultados experimentais em um supercomputador Cray XE6 mostraram que a implementação híbrida MPI−OpenMP reduziu drasticamente o tempo de execução, embora nenhuma análise do desempenho tenha sido apresentada. Por outro lado, o trabalho apresentado em [Nascimento et al. 2016] analisa como o tamanho da matriz e quantidade de iterações influenciam no tempo de execução de uma versão híbrida (OpenMP/MPI) do algoritmo de leilão. Versões vetorizadas do algoritmo de leilão para arquiteturas *multicore* (XEON) foram implementadas e avaliadas em [Sena et al. 2017]. O trabalho apresenta didaticamente apenas uma parte da vetorização do algoritmo de leilão.

Diferentemente de todos os trabalhos descritos nesta seção, este trabalho implementa e avalia uma versão COMPLETA vetorizada do algoritmo de leilão para XEON PHI. Além disso, uma análise detalhada do desempenho para arquitetura Xeon PHI (*many-core*) é apresentada. Por fim, uma comparação do desempenho do algoritmo de leilão nas arquiteturas *many-core* e *multicore* é apresentada.

3. Algoritmo de Leilão

Esta seção apresenta o Algoritmo de Leilão e sua implementação. Originalmente, ele foi 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 pode ser dividido em duas etapas principais: (1) LANCE (Algoritmo 1), na qual as pessoas dão lances para os objetos que elas desejam se associar, e (2) ASSOCIAÇÃO (Algoritmo 2), na qual o melhor lance dado para cada objeto é selecionado individualmente, determinando suas associações e novos preços. O preço atua como um regulador desse processo dinâmico, aumentado de acordo com o valor do melhor lance recebido pelo objeto correspondente.

O algoritmo itera em rodadas repetindo as duas etapas principais. Na fase de LANCE, cada pessoa livre i ofertará um novo lance para um objeto j, onde os lances são calculados como a diferença entre o objeto de maior e segundo maior valor para uma determinada pessoa (Algoritmo 3, linhas 9 a 17). É importante ressaltar que uma constante infinitesimal ϵ sempre é adicionada ao valor do maior lance para garantir a convergência

do algoritmo em casos de empate (Algoritmo 3, linha 17). Após a oferta de lances, na fase de ASSOCIAÇÃO, um ou mais objetos são então associados as pessoas que ofereceram o maior valor associado (Algoritmo 4, linhas 9 a 16).

Ao fim de cada rodada é produzido um conjunto de preços e associações (Algoritmo 2, linhas 4 e 5). Se todas as pessoas estão satisfeitas com isso, ou seja, cada pessoa está associada a um objeto, o algoritmo termina. Isso porque necessariamente cada pessoa está associada ao objeto de maior valor de acordo com a Equação $l_i = \max_{j=1,\ldots,n} \{a_{ij} - p_j\}$, e não há pessoas livres para fazerem novas ofertas. Caso contrário, uma nova rodada é iniciada. Desta forma, enquanto existirem pessoas sem objetos associados, o algoritmo de leilão continua trocando as pessoas associadas aos objetos, e ajustando o preço de cada objeto j com os valores dos maiores lances atribuídos a eles.

Algoritmo 1 lance(A, l, p, m, n)

```
1: for i = 0; i<m; i++ do
2:     if pessoa i não possui objeto associado then
3:         (l[i].obj, l[i].val) = max2Max(A[i], p, n)
4:     else
5:         (l[i].obj, l[i].val) = (-1,-1)
6:     end if
7: end for
```

Algoritmo 2 associacao(l, p, m, n)

```
1: for j = 0; j<n; j++ do
2:     (mVal, mInd) = maxValObj(l,j,m)
3:     if existe um novo venceder para obj j then
4:         p[j] = p[j] + mVal
5:         atualizaAssociacao(mVal,mInd)
6:     end if
7: end for
```

Algoritmo 3 max2Max(Ai, p, n)

```
1: (mInd,mVal,smVal) = (0,Ai[0]-p[0],-1)
2: for j = 1; j<n; j++ do
3:     aux = Ai[j]-p[j]
4:     if mVal < aux then
5:         (smVal,mVal,mInd) = (mVal,aux,j)
6:     else
7:         if aux > smVal then
8:             smVal = aux
9:         end if
10:    end if
11: end for
12: return (mInd, mVal - smVal + ϵ)
```

Algoritmo 4 maxValObj(l, j, m)

```
1: (mVal,mObj) = (-1,-1)
2: for i = 1; i<m; i++ do
3:     if l[i].obj == j then
4:         if l[i].val > mVal then
5:             (mVal,mInd) = (l[i].val,i)
6:         end if
7:     end if
8: end for
9: return (mVal, mInd)
```

4. Vetorização do Algoritmo de Leilão para Arquitetura Xeon Phi

As arquiteturas Xeon e Xeon Phi, além de vários núcleos de processamento, possuem instruções vetorizadas que podem ser usadas para aumentar substancialmente o desempenho da aplicação quando a mesma operação pode ser realizada em um conjunto de dados. A vetorização é o processo de conversão de uma implementação escalar de um programa para um processo vetorial [Mark-Sabahi 2012]. Esse processo é fundamental para extrair desempenho nessas arquiteturas, especialmente nos processadores Xeon Phi que possuem registradores de 512 *bits* e possuem frequência menor que o Xeon.

O próprio compilador é capaz de vetorizar o programa do usuário, mas apenas para códigos simples ou bem otimizados. Essa dificuldade acontece pois, para vetorizar um programa, o compilador tem que ser capaz de identificar a ausência de dependência de dados nas operações e que o acesso a memória seja contíguo [Mark-Sabahi 2012].

Quando o compilador não é capaz de vetorizar o código existem duas alternativas: vetorização explicita (diretivas OpenMP) ou programação com instruções SIMD (*intrinsics*) [Wende et al. 2016]. Enquanto que a vetorização explícita consiste em fornecer informações adicionais para ajudar o compilador a vetorizar o código, *intrinsics* é uma programação de baixo nível que utiliza as instruções SIMD. Apesar da vetorização explícita ser menos complexa, o desempenho depende no uso preciso das diretivas e está limitado a maneira como elas foram implementadas. Por outro lado, a programação com instruções SIMD é mais complexa, mas permite extrair o desempenho máximo.

Para extrair o desempenho máximo das máquinas Xeon PHI, foi implementada uma versão vetorizada do algoritmo de leilão utilizando a programação *intrinsics*. Enquanto que a implementação para *multicore* proposta em [Sena et al. 2017] apresenta apenas uma parte do algoritmo de leilão, este trabalho apresenta a vetorização completa do algoritmo. Foram vetorizados o procedimento MAX2MAX (Algoritmo 3) na etapa de LANCE e o procedimento MAIORVALPARAOBJ (Algoritmo 4) na etapa de ASSOCIAÇÃO.

4.1. Vetorização da etapa de LANCE

A vetorização da etapa de LANCE consistiu em vetorizar a função MAX2MAX (Alg. 3), que encontra o lance que a pessoa irá ofertar por um objeto. Basicamente, é necessário encontrar os objetos de maior e segundo maior valores para a pessoa que está sendo calculado o lance, e o índice do objeto de maior valor. O código vetorizado, que foi chamado de MAX2MAXVET, pode ser visto no Algoritmo 5.

Inicialmente, as variáveis vetorizadas mv e mi que armazenam o maior valor e o índice do objeto de maior valor são inicializadas. O comando ISET1(0) (linha 3), inicializa todas as 16 posições de mi com 0, enquanto o comando LOAD(v) inicializa as 16 posições de mv com as 16 primeiras posições de v.

O processo de vetorização acontece da linha 4 a 20. Repare que o **for** (linha 4) percorre os objetos de 16 em 16, uma vez que as variáveis vetorizadas são capazes de armazenar 16 valores pois os registradores da arquitetura Xeon PHI são de 512 *bits* e os números inteiros e com precisão simples utilizam 32 *bits*. Nas linhas 5 a 7, variáveis vetorizadas são inicializadas. As variáveis $vaux$ e $vaux2$ são inicializadas com as próximas 16 posições de v e de p, respectivamente, através da instrução LOAD, enquanto que vi é inicializado com o valor de i através do comando ISET. Em seguida, as 16 posições de $vaux$ recebem a diferença entre $vaux$ e $vaux2$ através do comando SUB. Esta diferença é necessária, pois o valor do lance a ser ofertado deve subtrair o preço corrente que está sendo oferecido para o objeto (inicialmente 0), conforme visto na Seção 3. Na linha 9, a instrução CMP compara quais valores de $vaux$ são maiores que mv e armazena na máscara de bits $vflag$. Caso todos os valores de $vaux$ sejam menores então os comandos dentro do **if** serão executados. Neste caso, apenas a variável que guarda o segundo maior valor deve ser atualizada. Para isso, a instrução CMP armazena na máscara de bits $vflag2$ todas as posições de $vaux$ que são maiores que smv (linha 11). Em seguida, o comando MBLEND atualiza apenas as posições de smv onde o valor de $vaux$ for maior (linha 12).

Algoritmo 5 max2MaxVet (v, p, n)

```
1: // v = A[i] - Linha da matriz de associação com os lances para a pessoa sendo processada
2: mi = ISET(0)
3: mv = LOAD(v)
4: for i = 0; i<n; i+=16 do
5:     vaux = LOAD(v+i)
6:     vaux2 = LOAD(p+i)
7:     vi = ISET(i)
8:     vaux = SUB(vaux, vaux2)
9:     vflag = CMP(vaux, mv, >)
10:    if !vflag then
11:        vflag2 = CMP(vaux, smv, >)
12:        smv = MBLEND(vflag2, smv, vaux)
13:    else
14:        smtmp = MBLEND(vflag, vaux, mv)
15:        mv = MBLEND(vflag, mv, vaux)
16:        mi = IMBLEND(vflag, mi, vi)
17:        vflag2 = CMP(smtmp, smv, >)
18:        smv = MBLEND(vflag2, smv, smtmp)
19:    end if
20: end for
21: STORE(max,mv)
22: ISTORE(imax,mi)
23: STORE(smax,smv)
24: mVal = LOW_VALUE
25: for i = 0; i<16; i++ do
26:    if max[i]≥ mVal then
27:        mVal = max[i]
28:        mInd = imax[i]
29:    end if
30: end for
31: max[mInd] = LOW_VALUE
32: smVal = LOW_VALUE
33: for i = 0; i<16; i++ do
34:    if max[i]≥ smVal then
35:        smVal = max[i]
36:    end if
37: end for
38: for i = 0; i<16; i++ do
39:    if smax[i]≥ smVal then
40:        smVal = smax[i]
41:    end if
42: end for
43: return (mInd, mVal - smVal + ϵ)
```

Algoritmo 6 maxValObjVet (l, j, m)

```
1: // Essa versão utiliza dois vetores lObj e lVal separados ao invés do vetor l com os campos obj e val
2: vetmVal = SET(LOW_VALUE)
3: vetmInd = ISET(-1)
4: vObj = ISET(j)
5: for i = 0; i<m; i+=16 do
6:    auxObj = ILOAD(lObj+i)
7:    mask = CMPEQ(auxObj, vObj)
8:    if mask then
9:        auxPreco = LOAD(lVal+i)
10:       mask2 = CMP(auxPreco, vetmVal, >)
11:       mask = mask & mask2
12:       vetmVal = MBLEND(mask, vetmVal, auxPreco)
13:       auxLeiloeiro = ISET(i)
14:       vetmInd = IMBLEND(mask, vetmInd, auxLeiloeiro)
15:   end if
16: end for
17: // Redução
18: // vMax, iMax (Vetores de 16 posições)
19: STORE(vMax,vetmVal)
20: ISTORE(iMax,vetmInd)
21: mVal = LOW_VALUE
22: for i = 0; i<16; i++ do
23:    if iMax[i]≥ 0 and vMax[i]> maiorLance then
24:       mVal = vMax[i]
25:       mInd = iMax[i]
26:    end if
27: end for
28: return (mVal, mInd)
```

Caso contrário, os comandos do **else** são executados. Neste caso, as variáveis que guardam o maior e o segundo maior valores devem ser atualizadas. Para isso, a variável $smtmp$ recebe os valores de $vaux$ ou mv através da instrução MBLEND respeitando a máscara de bits $vflag$ (linha 14). Ou seja, $smtmp$ recebe os valores de $vaux$ que não são maiores que os de mv (*bits* com valor 0 em $vflag$) e recebe os valores de mv que são menores que $vaux$ (*bits* com valor 1). Por outro lado, linha 15, a variável vetorizada mv recebe os maiores valores de mv e $vaux$ também através do comando MBLEND. O índice do objeto para o qual está sendo dado o maior lance também tem que ser atualizado. Logo, na linha 16, mi é atualizada através do comando IMBLEND. Apenas as posições de mi

onde o valor de $vaux$ é maior são atualizadas, conforme a máscara $vflag$ determina. Por fim, para atualizar a variável smv que contém o segundo maior valor é necessário criar a máscara de *bits* $vflag2$ que recebe 1 para as posições onde o valor de $smtmp$ for maior do smv e 0 caso contrário, através do comando CMP (linha 17). Em seguida, na linha 18, a variável smv é atualizada com os maiores valores de $smtmp$ e smv.

Após o termino do **for** inicia-se a redução (linha 21). Ou seja, reduzir as variáveis vetorizadas mv, mi e smv que contém 16 valores para apenas um valor. Inicialmente, essas três variáveis são armazenadas em vetores de 16 posições através dos comandos STORE e ISTORE (linhas 21 a 23) e a variável $mVal$ que irá armazenar o maior valor é inicializada com LOW_VALUE (linha 24). O processo para achar o maior lance dado e o índice para qual a o lance foi dado é muito simples. Basta percorrer as 16 posições do vetor que contém os maiores valores e achar o maior valor e seu índice (linhas 25 a 30).

Por sua vez, para achar o segundo maior lance é necessário percorrer tanto o vetor que contém os maiores valores como também o que contém os segundos maiores valores, uma vez que o segundo maior valor geral pode estar em um desses vetores. Para isso, a posição do vetor que contém o maior valor é atribuída para o valor LOW_VALUE (linha 31). Em seguida, basta percorrer as 16 posições do vetor max e as 16 posições do vetor $smax$ para encontrar o segundo maior valor, como pode ser visto nas linhas 33 a 42· Por fim, na linha 43, o algoritmo retorna o índice do objeto para qual será dado o lance e o valor do lance que é a diferença entre o maior e segundo maior valores, mais a constante ϵ que trata a convergência em casos de empate, conforme explicado na Seção 3.

4.2. Vetorização da etapa de ASSOCIAÇÃO

Por sua vez, a função vetorizada MAXVALOBJVET associa o objeto a pessoa que ofereceu o maior lance. O código dessa função pode ser visto no Algoritmo 6. Inicialmente, nas linhas 2 a 4, três variáveis vetorizadas auxiliares são inicializadas. O comando SET($valor$) armazena em cada posição da variável vetorizada o resultado de $valor$ em precisão simples, enquanto ISET($valor$) armazena um valor inteiro.

O processo de vetorização acontece da linha 5 a 16. Assim como na vetorização anterior, o **for** (linha 5) percorre as pessoas de 16 em 16. Na linha 6, o comando ILOAD armazena na variável vetorizada $auxObj$ os objetos que as 16 pessoas da iteração atual deram lances. Já, a linha 7, compara esses 16 objetos com o objeto que se deseja encontrar o maior lance j, através do comando CPEQM, e armazena na variável $mask$. O comando CPEQM vai colocar na variável vetorizada 1 se a comparação for verdadeira e 0 se for falsa. Caso todos os 16 objetos comparados sejam diferentes do objeto j, então a variável $mask$ vai ser igual a 0. Logo, o comando **if** da linha 8 não será executado. Caso contrário, a execução dos comandos dentro do **if** armazena na variável vetorizada $vetmVal$ os 16 maiores valores oferecidos ao objeto j e na variável vetorizada $vetmInd$ as 16 pessoas que deram esse lance. Para isso, na linha 9, os 16 valores correntes do vetor $lVal$ são armazenados na variável auxiliar vetorizada $auxPreco$ através do comando LOAD. Em seguida, linha 10, o comando CMP compara quais 16 valores armazenados em $auxPreco$ são maiores que os valores armazenados na variável $vetmVal$, armazenando na máscara de bits $mask2$ o valor 1 se for maior e 0 caso contrário. Como só interessa os lances dados ao objeto j, na linha 11 é realizado uma operação AND entre as máscaras de bits $mask$ e $mask2$, de maneira que no final a variável $mask$ só tenha o valor 1 na posição que for para o objeto j e o lance dado seja maior que o anterior. O comando MBLEND, linha

12, permite uma junção entre as variáveis $vetmVal$ e $auxPreco$, apenas das posições que tiverem o valor 1 em $mask$. Ou seja, só serão atualizadas as posições da variável vetorizada $vetmVal$ onde o bit correspondente na variável $mask$ tiver valor 1. Por fim, é necessário também armazenar as pessoas que deram os lances. Para isso, as pessoas correntes são armazenadas na variável $auxLeiloeiro$, através do comando ISET (linha 13). Em seguida, linha 14, o comando IMBLEND atualiza apenas as posições de $vetmInd$ que o bit correspondente na variável $mask$ tiver valor 1.

Ao final de todas as iterações do comando **for**, a variável vetorizada $vetmVal$ contém os 16 maiores valores ofertado ao objeto j, assim como a variável $vetmInd$ o índice das 16 pessoas que ofertaram esses valores. Para finalizar a vetorização basta reduzir esses 16 valores para apenas o maior valor ofertado e o índice da pessoa que ofertou. Para isso, basta armazenar os valores das variáveis $vetmVal$ e $vetmInd$ nos vetores $vMax$ e $iMax$ através dos comandos STORE e ISTORE (linhas 19 e 20). Para finalizar basta percorrer os 16 elementos do vetor para retornar o maior lance e o índice da pessoa (linhas 22 a 28).

Nos códigos vetorizados apresentados nos Algoritmos 5 e 6 optou-se por usar mnemônicos (alias) ao invés do nome real da instrução *Intrinsics* para facilitar a leitura do código. Assim, a Tabela 1 apresentas as instruções *Intrinsics* que foram realmente utilizadas na implementação.

Tabela 1. Instruções Intrinsics usadas na vetorização para Xeon Phi.

Alias	Intrinsics Xeon Phi (512 bits)	Alias	Intrinsics Xeon Phi (512 bits)
LOAD	_mm512_load_ps	ILOAD	_mm512_load_si512
SET	_mm512_set1_ps	ISET	_mm512_set1_epi32
STORE	_mm512_store_ps	ISTORE	_mm512_store_si512
MBLEND	_mm512_mask_blend_ps	IMBLEND	_mm512_mask_blend_epi32
CMP	_mm512_cmp_ps_mask	CMPEQ	_mm512_cmpeq_epi32_mask
SUB	_mm512_sub_ps		

5. Paralelização do Algoritmo de Leilão para Arquiteturas Xeon Phi

Para aproveitar todo potencial das arquiteturas Xeon Phi, além da vetorização que maximiza o uso das instruções SIMD, é necessário utilizar os vários núcleos de processamento de maneira eficiente, através de técnicas de paralelização para memória compartilhada. Para isso, foi utilizado o modelo de programação paralela *OpenMP* [van der Pas 2017].

Como descrito na Seção 3, o algoritmo de leilão é basicamente uma sucessão de iterações, onde em cada uma dessas iterações são executadas as fases de LANCE e ASSOCIAÇÃO. Na fase de LANCE, o cálculo do lance dado por cada pessoa para um objeto não depende do lance de outra pessoa. Assim, cada lance pode ser realizado em paralelo através de uma diretiva `parallel for` do openMP. Por sua vez, na fase de ASSOCIAÇÃO cada objeto seleciona o melhor lance recebido e determina sua associação e novo preço sem depender das seleções dos outros objetos. Assim, cada associação objeto-pessoa também pode ser realizado em paralelo, através de uma diretiva `parallel for` do openMP. Mais detalhes sobre a implementação paralela do algoritmo de leilão podem ser obtidos em [Nascimento et al. 2016, Sena et al. 2017].

6. Análise Experimental

Esta seção analisa o desempenho do algoritmo de leilão. Inicialmente, é apresentada uma análise detalhada da vetorização para Xeon Phi. Em seguida, o comportamento da versão

paralela vetorizada é analisado. Por fim, é apresentado uma comparação de desempenho do algoritmo de leilão nas arquiteturas Xeon e Xeon Phi.

6.1. Análise Experimental da Vetorização

Os experimentos foram realizados em uma máquina Xeon Phi 7250 com um total de 68 núcleos (*clock* de 1.40GHz) e 204 GB de memória. O processador conta com instruções SIMD do tipo AVX de 512 bits. Os programas foram compilados utilizando o compilador icc da Intel com otimização -O3 -xavx. Para o primeiro experimento foram utilizadas 20 matrizes reais 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.

Os tempos de execução do algoritmo de leilão original e da versão vetorizada proposta, assim como o speedup ($Speedup_{=}\frac{Tempo_{original}}{Tempo_{vetorizada}}$), podem ser visto na Figura 1. Cada matriz foi executada 30 vezes e a média aritmética do tempo de execução calculada, os resultados obtidos são bastante confiáveis com coeficiente de variação para todos os experimentos abaixo de 1%. Como mostrado na Figura 1, o tempo de execução (segundos) variou bastante. Isso porque, o tempo para emparelhar duas imagens é proporcional ao tamanho da matriz e a quantidade de iterações para achar o emparelhamento ótimo. O desempenho da versão vetorizada é significativamente melhor do que o da versão original. A média aritmética do tempo de execução das 20 matrizes usando a versão vetorizada foi ≈ 19 vezes menor do que o obtido através da versão original. A principal razão para o excelente desempenho da versão vetorizada foi o uso eficiente das instruções SIMD de 512 bits disponíveis, uma vez que apenas um núcleo da máquina é usado.

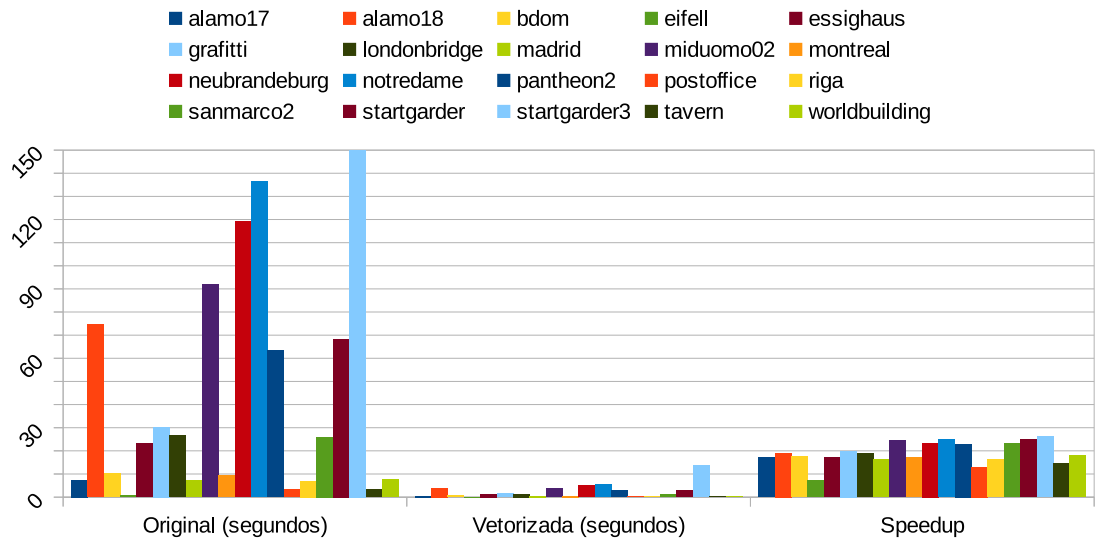


Figura 1. Tempo de execução (segundos) das versões original e vetorizada

Embora a média dos *speedups* obtidos tenha sido ≈ 19, houve uma variação, onde o pior *speedup* foi de 7.07 (eifell) e o melhor foi de $26,11$ (startgarder3). O motivo para essa variação de desempenho está diretamente ligado a grande variação nos tamanhos das matrizes e sua quantidade de iterações. Por exemplo, as duas matrizes com os menores tempos de execução (eifell e postoffice) obtiveram os dois piores *speedups* ($7,07$ e $13,05$).

Por outro lado, os melhores desempenhos foram obtidos pelas matrizes com os maiores tempos de execução (*speedups* > 20). Nas matrizes pequenas o tempo de execução das instruções que não foram vetorizadas tem um peso maior no tempo total de execução, afetando o desempenho. É importante destacar que o excelente desempenho proporcionado pela vetorização do algoritmo de leilão tem duas razões principais: execução das principais instruções em blocos em função da remoção de instruções de desvios e não execução de blocos de instruções em função do uso das instruções condicionais vetorizadas.

Assim, o experimento a seguir analisa o impacto do tamanho das matrizes e do número de iterações no desempenho da versão vetorizada. Para isso foi usada uma aplicação que simula o movimento de partículas, ao longo do tempo, dentro de um ambiente 3D com diferentes velocidades, perturbadas por ruídos aleatórios. Essa aplicação permite criar casos de estudo com maior concorrência ou maior esparsidade pela escolha do número de partículas e definição da caixa envolvente. Foram geradas matrizes de tamanhos 2000×2000, 4000×4000 e 8000×8000 com diferentes quantidades de iterações e o emparelhamento dessas partículas, entre dois instantes de tempo, foi realizado utilizando as versões sequenciais original e vetorizada do algoritmo de leilão. Cada instância foi executada 30 vezes e a média aritmética do tempo de execução calculada, os resultados obtidos são bastante confiáveis com coeficiente de variação $< 1\%$.

Tabela 2. Qtd de iterações, Tempo das versões Orig. e Vet. (segundos) e *speedup*

2000×2000				4000×4000				8000×8000			
Iter.	Orig.	Vet.	Speedup	Iter.	Orig.	Vet.	Speedup	Iter.	Orig.	Vet.	Speedup
3100	87,10	4,75	18,35	4479	497,71	22,27	22,35	7447	3177,16	117,04	27,15
4341	116,16	5,44	21,37	8760	920,24	35,46	25,95	12757	5381,02	192,29	27,98
12735	332,16	14,00	23,73	24652	2549,36	93,24	27,34	44227	15780,76	570,91	29,48

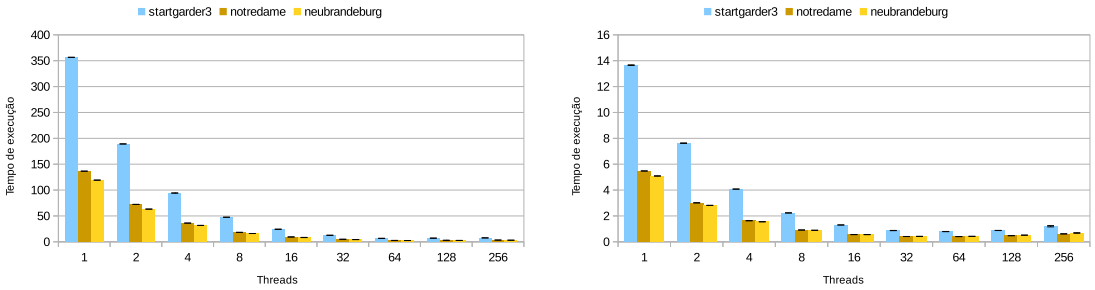
Os dados da Tabela 2 mostram que o desempenho da vetorização melhora com o aumento do tamanho da matriz e com o aumento no número de iterações. Por exemplo, para a matriz 2000×2000, os *speedups* das matrizes com 4341 e 12735 iterações foram $16,5\%$ e $29,3\%$ maiores do que os da matriz com 3100 iterações, respectivamente. Similarmente, os *speedups* cresceram de acordo com o aumento das matrizes. A razão para tal comportamento é que o crescimento do tamanho da matriz aumenta a quantidade de dados a ser vetorizada em relação a parte que não pode ser vetorizada, o que aumenta o desempenho em relação a versão original. Por sua vez, o aumento na quantidade de iterações melhora o desempenho das instruções condicionais vetorizadas, uma vez que o número de associações decresce monotonicamente.

6.2. Análise Experimental da Versão Vetorizada Paralela

O mesmo ambiente descrito na Subseção 6.1 foi usado para avaliar as versões paralelas. Porém, o objetivo agora é avaliar o desempenho do algoritmo de leilão ao executar com múltiplas *threads* e, com isso, aproveitar todos os núcleos disponíveis nas arquiteturas Xeon PHI. Para isso, foram utilizadas as três matrizes com os maiores tempo de execução da Figura 1: neubrandeburg, notredame e startgarder3. As versões original e vetorizada foram executadas 30 vezes para cada uma das matrizes, variando a quantidade de *threads* OpenMP (2, 4, 8, 16, 32, 64, 128 e 256). Apenas a política de escalonamento de laços *static* do OpenMP foi utilizada. A baixa sobrecarga dessa política foi a principal razão para a escolha.

A média aritmética dos tempos de execução (em segundos) e o intervalo de confiança de 95% foram calculados e podem ser vistos na Figura 2. Analisando os tempos

de execução, é possível observar que houve aumento de desempenho, tanto para versão original como também para a versão vetorizada, a medida que a quantidade de *threads* aumentava até o limite de 64. Por sua vez, a execução com 128 e 256 *threads* não obteve ganho, apesar do sistema utilizado possuir *quad hyper-threading* (68 núcleos reais + 4 hiperprocessamento por núcleo). As principais razões para a perda de desempenho do hiperprocessamento para o algoritmo de leilão são a ausência de operações de entrada/saída e uso intensivo do barramento.



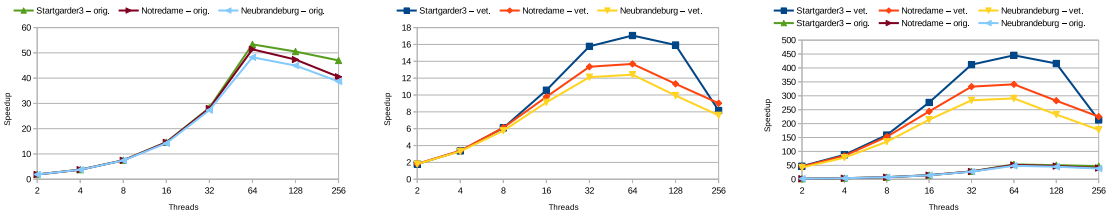
(a) Original (b) Vetorizada

Figura 2. Tempo de exec. (segundos) das versões paralelas original e vetorizada

A confiança nos resultados pode ser observada na Figura 2, onde o intervalo de dados foi muito próximo da média. A figura também destaca o desempenho das versões paralelas, original e vetorizada. Enquanto que para uma quantidade pequena de *threads* (≤ 16) a escalabilidade das duas versões é similar, para uma quantidade maior de *threads* (≥ 32) a escalabilidade da versão paralela original é bem superior. O principal motivo para diminuição da escalabilidade da versão paralela vetorizada nestes casos é a granularidade fina das tarefas em função do excelente desempenho da vetorização. Por exemplo, o tempo total de execução da versão original sequencial para a matriz *neubrandeburg* foi de $119,17$ segundos, enquanto que o tempo da versão vetorizada sequencial é de apenas $5,09$. Por sua vez, o tempo da execução paralela vetorizada com 64 cores foi de apenas $0,41$ segundos para executar as 2380 iterações, ou seja, um tempo de $\approx 0,000172$ segundos por iteração. Assim, como o paralelismo ocorre a cada iteração do algoritmo, fica evidente a granularidade fina das tarefas nesse ponto, limitando sua escalabilidade.

Uma melhor análise do desempenho das versões paralelas pode ser feita através da Figura 3. Os *speedups* para as versões paralelas original e vetorizada foram calculados através das equações, $Speedup_{original} = \frac{Tempo_{original}(1)}{Tempo_{original}(T)}$ e $Speedup_{vetorizado} = \frac{Tempo_{vetorizado}(1)}{Tempo_{vetorizado}(T)}$ onde T é o número de *threads*. Os gráficos das Figuras 3(a) e 3(b) mostram claramente a melhor escalabilidade da versão paralela original, especialmente quando $T \geq 32$. Por exemplo, considerando a matriz *startgarder3*, a versão paralela original alcançou um *speedup* máximo de $53,3$, utilizando 64 *threads*, enquanto que a versão paralela vetorizada alcançou apenas um *speedup* de $17,1$. Como explicado anteriormente a razão para esse baixo desempenho é a granularidade baixa das tarefas da versão paralela vetorizada em função do excelente desempenho da vetorização. Porém, quando se considera o desempenho total das versões paralelas original e vetorizada (SIMD + Paralelismo), o desempenho da versão paralela vetorizada é substancialmente melhor, como pode ser visto na Figura 3(c). Neste gráfico, o *speedup* da versão paralela vetorizada usa como base o tempo sequencial da versão original, sendo calculado através da

equação $Speedup_{SIMD+Paralelismo} = \frac{Tempo_{original}(1)}{Tempo_{vetorizado}(T)}$. Considerando o desempenho total é possível verificar que a versão paralela vetorizada foi capaz de aproveitar eficientemente o potencial das arquiteturas Xeon Phi. Por exemplo, no melhor caso, matriz *startgarder3*, o *speedup* máximo foi de 445.39, utilizando 64 *threads*. Muito superior ao desempenho máximo da versão paralela original que foi de $53,3$.



(a) Original (b) Vetorizada (c) Vetorizada/original

Figura 3. *Speedup* das versões paralelas vetorizada e original

6.3. Comparação de desempenho Xeon × Xeon Phi

Esta seção compara o desempenho do algoritmo de leilão nas arquiteturas *multicore* e *many-core*. A vetorização implementada para arquitetura *multicore* é similar a vetorização apresentada neste trabalho, porém apenas instruções AVX2 de 256 *bits* foram usadas. Enquanto que a versão para Xeon Phi foi executada na máquina descrita na Seção 6.1, todos os experimentos para *multicore* foram realizados em uma máquina NUMA (*Non-Uniform Memory Access*) com 36 *cores* (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. Os programas foram compilados utilizando o compilador icc da Intel com otimização -O3 -xavx. A execução de algumas matrizes já descritas anteriormente pode ser vista na Tabela 3, que mostra os tempos das versões sequenciais Original e Vetorizada, além de mostrar o melhor tempo da versão paralela vetorizada, a quantidade de *threads* que obteve o melhor tempo e também o *speedup* total.

Tabela 3. Comparação entre Xeon × Xeon Phi

Matriz	Iter.	Xeon					Xeon Phi				
		Tempo (segundos)			Threads	Speedup	Tempo (segundos)			Threads	Speedup
		Orig.	Vet.	Vet. Par.		Total	Orig.	Vet.	Vet. Par.		Total
2000×2000	3100	17,66	1,81	1,08	4	16,35	87,10	4,75	0,63	32	138,25
4000×4000	4479	101,43	5,99	2,66	6	38,13	497,70	22,27	2,04	32	243,97
8000×8000	7447	658,28	56,16	9,24	12	71,24	3177,15	117,04	5,23	64	607,48
16000×16000	4479	3209,20	332,60	32,59	18	98,47	15783,80	570,97	20,99	128	751,96

Os resultados da tabela mostram claramente as vantagens de cada arquitetura. A maior frequência de *clock* e a maior quantidade de memória *cache* por núcleo do Xeon resultou em um tempo de execução da versão sequencial Original quase 5 vezes mais rápido do que a versão sequencial Original do Xeon Phi. Em contrapartida, por possuir instruções SIMD de 512 *bits* (o dobro do Xeon), a versão sequencial vetorizada do Xeon Phi produziu resultados, na média, duas vezes melhores do que o Xeon. Por fim, os tempos obtidos com as versões paralelas vetorizadas e a quantidade de *threads* necessária para obter esses tempos evidenciam a melhor escalabilidade do processador Xeon Phi, que mesmo partindo de tempos sequenciais vetorizados, em média, $2,5$ vezes mais lentos do que os produzidos pela versão sequencial vetorizada do Xeon, produziu melhores resultados para todas as matrizes. Em média, os tempos da versão paralela vetorizada do Xeon foi $\approx 58\%$ maior do que os produzidos pela versão paralela vetorizada do Xeon Phi.

Este resultado é fruto do melhor *speedup* total (calculado em relação ao tempo sequencial da versão original) da arquitetura Xeon Phi que para a matriz de tamanho 16000×16000, por exemplo, foi de ≈ 750, enquanto que o da arquitetura Xeon foi de $98,47$.

7. Conclusões

Este trabalho implementou e avaliou uma versão vetorizada do algoritmo de leilão para Xeon Phi, assim como, sua implementação paralela em OpenMP. Os resultados mostram que é possível aproveitar o potencial das arquiteturas *many-core*, utilizando eficientemente tanto os registradores de 512 *bits*, como também os vários núcleos disponíveis. Enquanto que a versão vetorizada conseguiu ≈ 19 de *speedup* médio, a versão paralela vetorizada atingiu 750 de *speedup* total. Além disso, a execução em Xeon Phi obteve um desempenho superior ao Xeon.

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

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

Bertsekas, D. P. and Castañon, D. A. (1991). Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Comput.*, 17(6-7):707–732.

Kollias, G., Sathe, M., Mohammadi, S., and Grama, A. (2013). A fast approach to global alignment of protein-protein interaction networks. *BMC Research Notes*, 6(1):1–11.

Kollias, G., Sathe, M., Schenk, O., and Grama, A. (2014). Fast parallel algorithms for graph similarity and matching. *Journal of Par. and Dist. Comp.*, 74(5):2400 – 2410.

Mark-Sabahi (2012). A guide to auto-vectorization with intel c++ compilers. Technical report, Intel.

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.

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).

Shokoufandeh, A. and Dickinson, S. (1999). Applications of bipartite matching to problems in object recognition. In *In Proceedings, ICCV Workshop on Graph Algorithms and Computer Vision*, page http://www.cs.cornel.

van der Pas, R., S. E. S. E. T. C. (2017). *Using OpenMP – The Next Step:Affinity, Accelerators, Tasking, and SIMD*. The MIT Press.

Vasconcelos, C. N. and Rosenhahn, B. (2009). *Bipartite Graph Matching Computation on GPU*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Wende, F., Noack, M., Steinke, T., Klemm, M., Newburn, C. J., and Zitzlsberger, G. (2016). Portable simd performance with openmp* 4.x compiler directives. In *Proc. of the Inter. European Conference on Parallel Processing*, pages 264–277.