Soluções Paralelas para o Problema de Roteamento Usando o Algoritmo de Lee

William Felipe C. Tavares[1], **Nahri Moreano**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Campo Grande – MS – Brasil

Abstract. *Lee's algorithm is a popular method for routing wires on a circuit board. This task, in the VLSI context, requires intense computing and high memory comsumption. This paper evaluates optimizations described in the literature that reduce the time and memory consumption of the algorithm, and constructively proposes techniques for its parallelization. The final result presented a speedup of* $2,25$ *with* 2 *threads and* $3,70$ *with* 4 *threads.*

Resumo. *O algoritmo de Lee é uma técnica popular para realizar o roteamento de trilhas em uma placa de circuito. No âmbito de VLSI, essa tarefa se torna computacionalmente intensa e exige grande quantidade de memória. Este artigo avalia otimizações descritas na literatura que reduzem o consumo de tempo e memória do algoritmo, e propõe, de maneira construtiva, técnicas para a paralelização do mesmo. O resultado final apresentou speedup de* $2,25$ *com* 2 *threads e* $3,70$ *com* 4 *threads.*

1. Introdução

A partir do momento em que a tecnologia de circuitos integrados evoluiu para uma escala de milhões de transistores (*VLSI*), a produção se tornou mais complexa. Dentre as diversas tarefas da fabricação de um *chip*, há o roteamento, que consiste em conectar os componentes de um *chip*, definindo a localização das trilhas [Chen and Chang 2009]. O roteamento, além de evitar a intercepção de trilhas, também deve minimizar o comprimento das trilhas – visando o desempenho e o custo de produção. O problema pode ser aplicado em diferentes configurações de *chip*, como a presença de mais de uma camada, roteamentos de várias trilhas em uma mesma placa ou uma trilha com múltiplas conexões.

Uma das técnicas utilizadas no roteamento é o algoritmo de Lee [Lee 1961], que encontra o menor caminho conectando dois pontos (se ele existir) em um labirinto. O algoritmo apresenta uma complexidade de tempo e de memória $O(m \times n)$, sendo m e n as dimensões do *chip*. O algoritmo de Lee torna-se inviável no âmbito de *VLSI*, e é utilizado como último recurso em casos que outros algoritmos não encontrem uma solução.

Este trabalho avalia otimizações propostas na literatura e propõe soluções paralelas para o algoritmo de Lee, com intuito de atingir resultados melhores que aqueles da solução sequencial básica, em relação ao tempo de execução e à memória utilizada.

Este artigo estrutura-se da maneira que se segue. A Seção 2 descreve o algoritmo de roteamento de Lee e algumas otimizações. A Seção 3 analisa os resultados das soluções sequenciais implementadas. A Seção 4 propõe soluções paralelas e apresenta seus resultados preliminares. A Seção 5 analisa os resultados obtidos com as implementações paralelas. A Seção 6 conclui o artigo e descreve trabalhos futuros.

2. Algoritmo de Lee e Otimizações

O contexto deste trabalho, um *chip* com apenas uma camada, permite que se trabalhe apenas em espaços bidimensionais, os *grids*, representados por matrizes. Portanto, um *grid* G, de comprimento m e largura n, possui $m \times n$ células. Cada célula c, com as coordenadas $c.i$ e $c.j$, possui no máximo quatro vizinhos, um para cada orientação.

Sendo s e t os pontos de origem (fonte) e de chegada (terminal), respectivamente, deseja-se encontrar um caminho de s a t de comprimento mínimo, através de uma sequência de células vizinhas. Cada célula pode estar em um de dois estados: bloqueado, representando os obstáculos, ou livre, representando a região de roteamento. Cada célula livre possui um valor da distância de s calculado pelo algoritmo. Duas células com o mesmo valor inteiro positivo estão no mesmo nível de expansão.

O algoritmo de Lee possui duas fases: expansão e *backtracking*. A primeira fase realiza uma onda de expansão que inicia na fonte em busca do terminal, onde os vizinhos de cada célula são tratados (apenas se estão livres e não tenham sido visitados) e suas distâncias são calculadas com uma unidade a mais da distância da célula. Ao iniciar na fonte com distância 0, seus vizinhos então terão distância 1, em seguida, os vizinhos dessas células terão distância 2, e assim por diante, até o terminal ser encontrado (Figura 1a). O *backtracking* é iniciado a partir do terminal, consultando os seus vizinhos e avaliando qual distância calculada sinaliza o menor caminho para a fonte. Efetuando essa operação, encontra-se o caminho de volta para a fonte (Figura 1b). Caso não exista um caminho que conecte ambos os pontos, a fase de expansão termina quando não houver mais células a serem tratadas, sem que o terminal tenha sido alcançado. No Algoritmo 1 é descrita a técnica. A fila Q armazena as células a serem tratadas, organizadas em ordem FIFO. A fila P armazena as células do caminho encontrado de s a t, se ele existir.

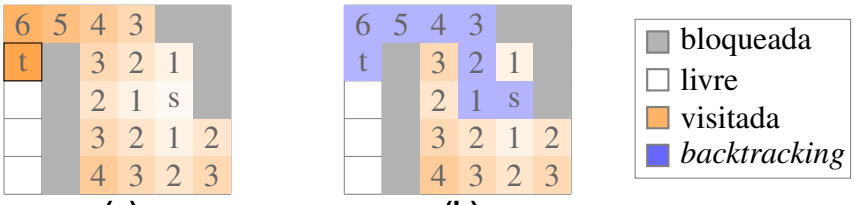

Figura 1. Fases de (a) expansão e (b) *backtracking* do algoritmo de Lee

Algumas otimizações foram propostas para o algoritmo em questão. Acker verificou que uma célula de distância k possui vizinhos de distância $k-1$, de onde chegou, e vizinhos de distância $k+1$, para onde alcançou. Dessa forma, é possível, ao invés de calcular as distâncias, utilizar uma rotulação com apenas um bit $0, 1, 1, 0, 0, 1, 1, 0, ...$ [Akers 1967]. Assim, cada célula é representada com apenas dois bits, um representando o rótulo e outro o estado (bloqueado ou livre), reduzindo o consumo de memória da implementação do algoritmo. Sait propôs otimizações para reduzir o número de células tratadas [Sait and Youssef 1999], com a expansão a partir do ponto (fonte ou terminal) mais próximo à borda do *grid*, ou ainda, a expansão a partir dos dois pontos (até que um ponto intermediário seja alcançado).

3. Resultados Preliminares e Análise

O algoritmo foi implementado utilizando a linguagem de programação C++. Os testes foram conduzidos em um computador com processador Intel $i7$ de 3.6GHz com 32GB de

Algoritmo 1: Algoritmo de Lee

```
Entrada: Grid G, fonte s e terminal t
Saída: Caminho P
Q ← ∅ // Células a serem tratadas
G[s.i][s.j] ← 0
encontrado ← false
// Expansão
insere(Q,s)
enquanto Q não está vazio e encontrado = false
 faça
    c ← remove(Q)
    se c = t então
       encontrado ← true
    senão
       para cada vizinho v de c em G faça
          se G[v.i][v.j] = ∞ então
             G[v.i][v.j] ← G[c.i][c.j] + 1
             insere(Q,v)
// Backtracking
P ← ∅ // Caminho de s a t
se encontrado = true então
    c ← t
    insere(P,t)
    repita
       v ← vizinho de c em G, tal que
         G[v.i][v.j] = G[c.i][c.j] − 1
       c ← v
       insere(P,v)
    até c = s
retorna P
```

memória RAM. Foram gerados aleatoriamente nove *grids* de dimensão 70.000×70.000 a 90.000×90.000. Além do algoritmo original, em que a expansão é iniciada na **fonte**, foram implementadas as expansões iniciadas no ponto mais ao **extremo** e nos **dois pontos** (fonte e terminal). Para cada uma dessas variações, foram desenvolvidos programas **inteiros** e **binários** (em que o *grid* é representado como uma matriz de inteiros ou uma matriz de bits).

A Figura 2a apresenta o tempo médio de execução para cada programa implementado. O *grid* inteiro, por utilizar uma maior quantidade de memória, exige muita paginação durante a execução, o que afeta negativamente seu desempenho em relação ao *grid* binário. A quantidade de memória utilizada em cada um dos programas é mostrada na Figura 2b. A Tabela 1 mostra o número médio de células visitadas em cada forma de expansão. Esses valores são iguais para *grids* inteiros e binários. Como esperado, a cada otimização feita, a quantidade de células visitadas diminui.

A expansão a partir do ponto mais extremo obteve o melhor desempenho em relação a tempo de execução, para os dois tipos de *grid*, com *speedup* de $1,56$ no inteiro e $1,37$ no binário, em relação à expansão a partir da fonte. A expansão de dois pontos obteve *speedup* de $1,37$ em relação à expansão a partir da fonte no *grid* inteiro. Era esperado que essa última otimização tivesse um desempenho melhor que a expansão do extremo, porém, sua implementação exige mais estruturas de dados e controle, o que impactou negativamente no desempenho. Esse impacto é mais evidente no *grid* binário.

Dimensão do *grid*	70.000×70.000	80.000×80.000	90.000×90.000	Média
Fonte	$0,52$	$3,98$	$4,39$	$2,90$
Extremo	$0,47$	$2,61$	$3,60$	$2,23$
Dois Pontos	$0,31$	$2,17$	$2,95$	$1,81$

Tabela 1. Número médio, em bilhões, de células do *grid* visitadas, para cada forma de expansão

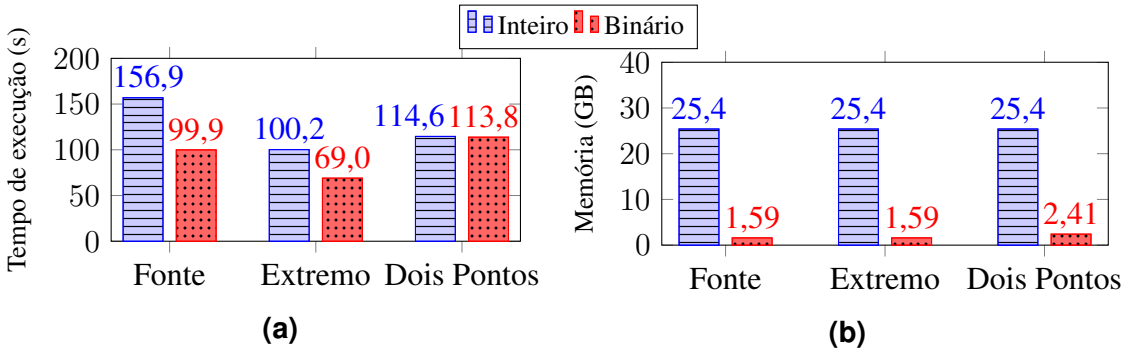

Figura 2. (a) Tempo de execução médio e (b) quantidade média de memória utilizada para cada forma de expansão para *grids* inteiros e binários

4. Soluções Paralelas

Alguns trabalhos da literatura propõem a paralelização do algoritmo de Lee utilizando ambientes em grade [Yen et al. 1993] ou hipercubo [Olukotun and Mudge 1987], onde cada processador é responsável por uma área específica do *grid*. Há trabalhos que paralelizam o roteamento de várias trilhas em uma mesma placa [Seaton et al. 2012]. Neste trabalho é explorado o paralelismo na visita das células de um mesmo nível da expansão, roteando uma única trilha no *grid*. Para isto foi utilizado o modelo de programação OpenMP [OpenMP Architecture Review Board 2018]. As execuções foram realizadas na mesma máquina usada para as implementações sequenciais, que possui quatro núcleos.

4.1. Paralelismo em *grids* inteiros

Para garantir que o menor caminho entre os pontos s e t seja encontrado, é necessário tratar todas as células de um nível da expansão, antes das células do nível seguinte. Assim, a proposta de paralelização é tratar em paralelo todas as células de um mesmo nível da expansão. Para isso, a fase de expansão do Algoritmo 1 é modificada, com a inclusão de um novo laço realizado de forma concorrente, na qual cada *thread* é responsável por um conjunto de células do mesmo nível (com o uso da construção *parallel for* do OpenMP). Também são utilizadas duas filas, Q, que compreende as células do nível atual de expansão, e Q_{aux}, contendo os vizinhos das células em Q. Para evitar a necessidade de exclusão mútua (e consequente sincronização) na inserção de células na fila Q_{aux}, essa estrutura é replicada para cada *thread*, tornando-as exclusivas de cada *thread*, e concatenando-as para construir a fila Q ao fim de cada nível de expansão. O Algoritmo 2 apresenta a modificação da fase de expansão.

O *grid* é uma estrutura de dados compartilhada entre as *threads* e, portanto, é necessária a exclusão mútua na visita a uma célula, para evitar que a mesma seja visitada ao mesmo tempo por várias *threads* e tenha seu valor sobrescrito. Essa sincronização é realizada usando a construção *critical* do OpenMP e cada visita é tratada como uma única seção crítica. Essa primeira solução cria sincronizações desnecessárias, pois células diferentes podem ser visitadas em paralelo sem necessidade de exclusão mútua. Para restringir a seção crítica para uma célula específica são utilizadas variáveis *lock* do OpenMP. Idealmente, cada célula teria um *lock*, o que exigiria $n \times m$ *locks*. Porém, como nem todas as células são visitadas em um nível de expansão, é viável reduzir o número de *locks* para $n + m$. No entanto, como as células visitadas em paralelo estão no mesmo nível, a

Algoritmo 2: Expansão: tratamento em paralelo das células de mesmo nível

```
// Expansão
enquanto Q não está vazio e encontrado = false faça
    para cada c ∈ Q faça em paralelo                    // Laço paralelizado
        c ← remove(Q)
        se c = t então
            encontrado ← true
        senão
            para cada vizinho v de c em G faça
                se G[v.i][v.j] = ∞ então
                    G[v.i][v.j] ← G[c.i][c.j] + 1
                    insere(Q_aux,v)
    Q ← Q_aux                                   // Construção de Q a partir de Q_aux
    Q_aux ← ∅
```

sobrescrita do valor não causa inconsistência, pois todas as *threads* escreveriam o mesmo valor.

Essas três abordagens iniciais foram aplicadas para os *grids* inteiros, com a expansão iniciada no ponto mais extremo, que foi a solução com melhor desempenho das implementações sequenciais. Os programas foram executados com 2 *threads*. A Figura 3a mostra os tempos de execução obtidos, que indicam que a sincronização causa um *overhead* desnecessário. O programa com *critical* tem o pior desempenho por exigir que cada visita seja realizada com exclusão mútua. O programa com *lock*, em que a visita é realizada a depender apenas da célula, ainda possui *overhead*. O programa sem sincronização possui o melhor desempenho, com *speedup* de $1,67$ em relação ao programa sequencial equivalente.

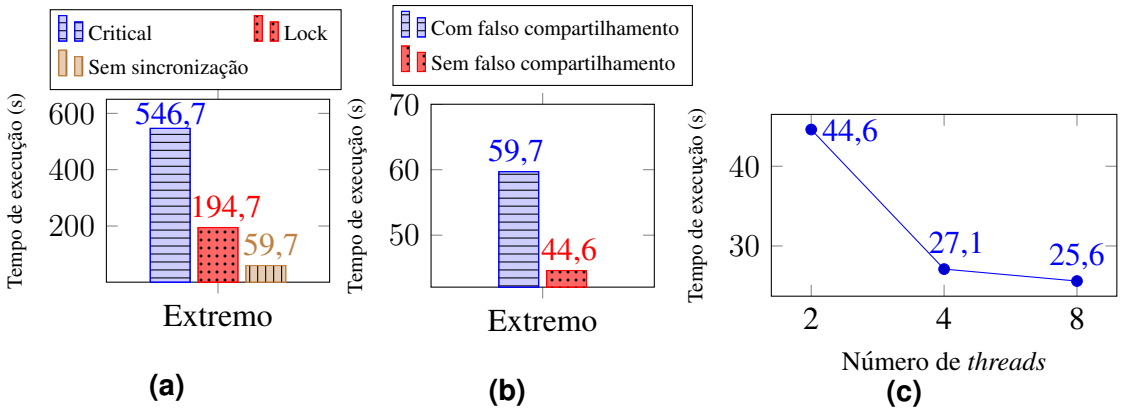

Figura 3. Tempo de execução médio dos programas paralelos, (a,b) usando 2 *threads* e (c) variando o número de *threads*, para *grids* inteiros partindo do ponto mais extremo

As estruturas utilizadas para a fila auxiliar Q_{aux} geram um falso compartilhamento entre as *threads*, e por consequência, entre as memórias caches dos diferentes núcleos. Conceitualmente, cada *thread* possui uma fila Q_{aux} independente das demais, porém essa estrutura foi implementada como um vetor de filas compartilhado, em que cada *thread* acessa uma posição diferente do vetor. Quando uma *thread* modifica a sua fila (na cache local do seu núcleo), ocorre invalidação ou atualização nas caches dos núcleos das outras *threads*, gerando transferências pelo barramento. As estruturas foram rearranjadas para

evitar que o falso compartilhamento aconteça. A Figura 3b apresenta o tempo médio de execução dos programas paralelos com *grids* inteiros partindo do ponto mais extremo, com e sem falso compartilhamento. O falso compartilhamento entre as *threads* causa grande perda de desempenho, e sua eliminação proporcionou *speedup* de $1,34$.

A Figura 3c mostra o tempo de execução desse programa que obteve o melhor desempenho, variando a quantidade de *threads*. O melhor resultado foi obtido com 8 *threads*, com *speedup* de $3,91$ em relação ao sequencial de *grids* inteiros partindo do ponto mais extremo.

4.2. Paralelismo em *grids* binários

Como o *grid* binário é implementado como uma matriz de inteiros, um único elemento na matriz representa várias células do *grid*. Exige-se, então, sincronização para acessar cada elemento da matriz, para evitar que duas *threads*, atualizando diferentes células do *grid* em paralelo, modifiquem um mesmo elemento da matriz. Uma primeira solução foi implementar novamente seções críticas com o uso de *locks*. Entretanto, como a sincronização é necessária apenas na escrita no elemento da matriz, pode-se modificar a sincronização para utilizar a construção *atomic*, que permite a atualização de uma posição na memória de forma atômica. Manteve-se a ideia de evitar o falso compartilhamento.

Na Figura 4a fica evidente que a sincronização com *atomic* é muito mais eficiente que a realizada por *locks* em termos de tempo de execução. Essa eficiência dá-se pelo fato das variáveis *locks* apresentarem *overhead* de criação e destruição e por serem destinadas para mais de uma célula, enquanto *atomic* é uma construção que realiza uma operação atômica em uma posição da memória específica.

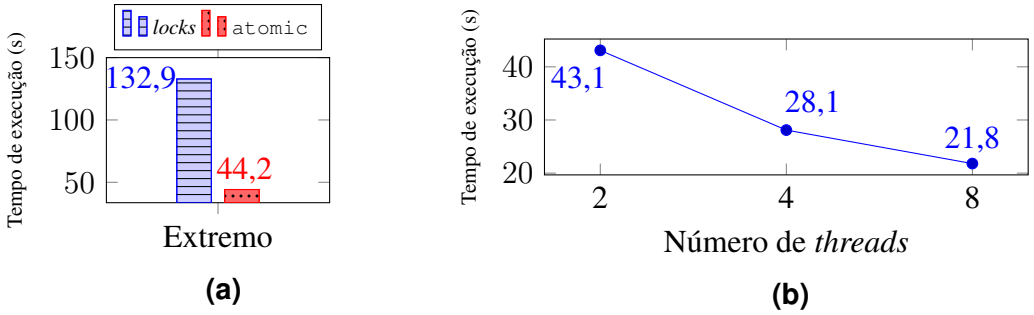

Figura 4. Tempo de execução médio dos programas paralelos, (a) usando 2 *threads* e (b) variando o número de *threads*, para *grids* binários partindo do ponto mais extremo

A implementação com a sincronização com *atomic* foi executada variando o número de *threads*, e os resultados são mostrados na Figura 4b. O pico do *speedup* é obtido quando 8 *threads* são utilizadas, apresentando *speedup* de $3,17$ em relação ao programa sequencial para *grids* binários partindo do ponto mais extremo.

4.3. Paralelismo na expansão a partir de dois pontos

A expansão a partir de dois pontos possui uma característica que permite explorar outra forma de paralelismo. É intuitivo pensar em uma *thread* ser responsável pelo cálculo das células alcançadas a partir da fonte e outra *thread* pelas células alcançadas a partir do terminal. Mas ainda é possível aplicar a mesma abordagem das versões anteriores, paralelizando o tratamento das células de um mesmo nível da expansão, tanto na expansão a partir da fonte quanto a partir do terminal, Em resumo, com 2 *threads*, cada

uma responsável por um dos dois pontos de expansão. Ao aumentar para 4 *threads*, são distribuídas duas *threads* para cada ponto, e assim por diante. Ao contrário das outras versões, aqui é possível explorar o paralelismo também na fase de *backtracking*, usando duas *threads* para realizar essa tarefa para cada um dos pontos de origem.

Foram implementadas versões paralelas de dois pontos para os dois tipos de *grids* e sem falso compartilhamento. Para a versão com *grids* binários, a sincronização *atomic* é usada. A Figura 5 apresenta os tempos de execução dessas versões, variando o número de *threads*. De forma geral, a versão com *grid* inteiro obteve melhores resultados, com *speedup* máximo de $2,64$ com 8 *threads*, em comparação à versão sequencial equivalente. A versão binária apresentou *speedup* máximo de $2,14$ com 16 *threads*, em relação à versão sequencial equivalente.

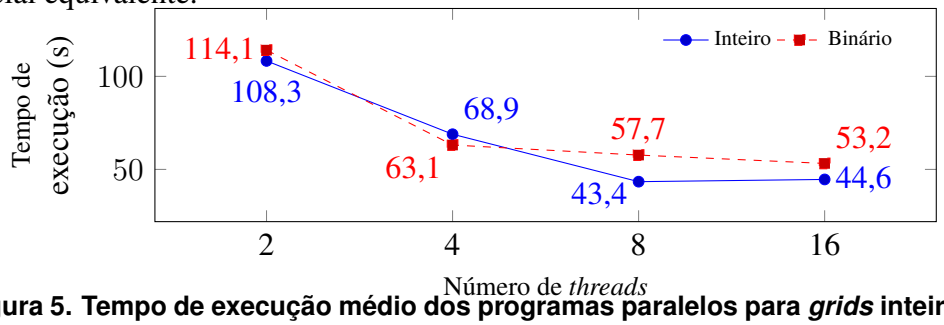

Figura 5. Tempo de execução médio dos programas paralelos para *grids* inteiros e binários, com expansão partindo de dois pontos

5. Resultados e Análise

Considerando as diversas versões implementadas, os melhores resultados foram obtidos nos programas paralelos que não possuem sincronização na visitação das células, e sem falso compartilhamento entre *threads*. A Figura 6 apresenta o tempo médio de execução para as versões sequenciais com *grids* inteiros e binários partindo do ponto mais extremo e dos dois pontos e os tempos obtidos pelas versões paralelas equivalentes. As Figuras 7a e 7b mostram os *speedups* das melhores versões paralelas em relação à versão sequencial equivalente, para *grids* inteiros e binários, respectivamente, variando-se o número de *threads* usadas.

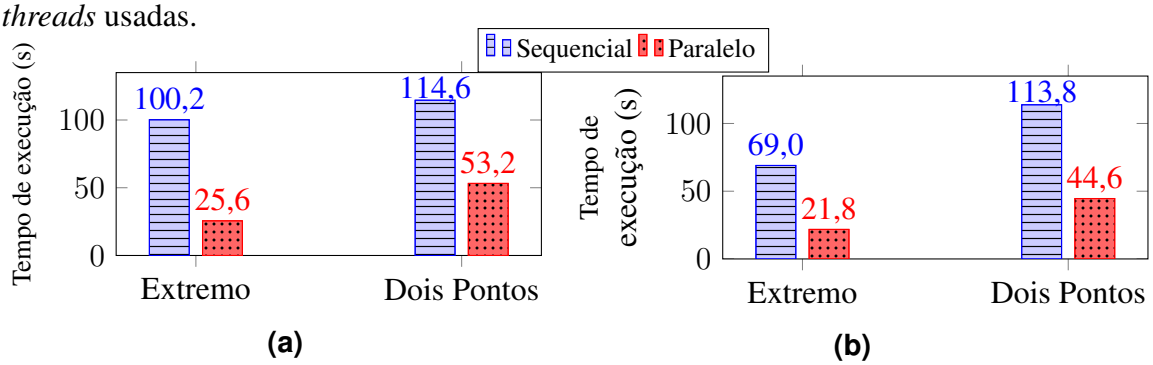

Figura 6. Tempo de execução médio dos programas sequenciais e paralelos para *grids* (a) inteiros e (b) binários, partindo do ponto mais ao extremo (8 *threads* no paralelo) e dos dois pontos (16 *threads* no paralelo)

O *speedup* cresce com o número de *threads* utilizadas em todas as versões paralelas, mais acentuado nas versões de *grids* inteiros. O *grid* binário, mesmo que utilizando menos memória, exige sincronização no acesso à célula, afetando negativamente sua performance, que fica evidente quando comparado com o *speedup* alcançado pelo *grid* inteiro. As versões que partem dos dois pontos não apresentaram a melhora esperada pela sua natureza paralela. Seus resultados foram superiores a todos os sequenciais, mas não foram melhores que as versões paralelas que partem do ponto mais extremo, devido o seu elevado nível de controle.

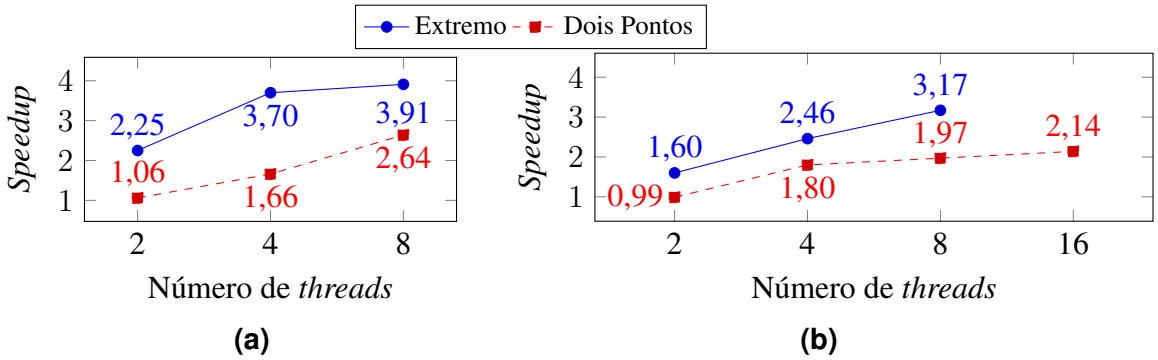

Figura 7. *Speedup* dos programas paralelos em relação ao sequencial correspondente para (a) *grids* inteiros e (b) *grids* binários

6. Conclusão

O algoritmo de Lee apresenta alto consumo de tempo e de memória e, a partir de otimizações propostas na literatura, essas exigências podem ser contornadas. Neste trabalho foi realizada uma implementação construtiva, onde, a cada passo, uma otimização foi agregada à implementação final. A sincronização e o efeito de falso compartilhamento nas caches foram alguns dos desafios tratados. Cada implementação paralela desenvolvida tendeu reduzir o tempo de execução necessário, mostrando-se como uma boa alternativa. Em sua versão final, obteve-se *speedup* de $2,25$ com 2 *threads*, $3,70$ com 4 *threads* e $3,91$ com 8 *threads* em relação ao sequencial.

Uma possível melhoria é alterar a indexação nos *grids* binários, onde cada elemento da matriz codifica um *subgrid* e não apenas uma linha do *grid*. Expandir a paralelização proposta aumentando esses parâmetros é uma alternativa de extensão deste trabalho. Otimizações em relação ao caminho, tal como aumentar a distância em relação aos componentes do chip ou diminuir o número de curvas da trilha – fatores que influenciam na funcionalidade da placa –, também são critérios que podem ser investigados.

Referências

Akers, S. B. (1967). A modification of Lee's path connection algorithm. *IEEE Transactions on Electronic Computers*, EC-16(1):97–98.

Chen, H.-Y. and Chang, Y.-W. (2009). Global and detailed routing. In *Electronic Design Automation*, pages 687–749. Elsevier.

Lee, C. Y. (1961). An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365.

Olukotun, O. A. and Mudge, T. N. (1987). A preliminary investigation into parallel routing on a hypercube computer. In *24th ACM/IEEE Design Automation*.

OpenMP Architecture Review Board (2018). OpenMP API version 5.0.

Sait, S. and Youssef, H. (1999). *VLSI Physical Design Automation: Theory and Practice*. Lecture Notes Series. World Scientific.

Seaton, C., Goodman, D., and Luján, M. (2012). Applying dataflow and transactions to lee routing. In *Workshop on Programmability Issues for Heterogeneous Multicores*.

Yen, I., Dubash, R., and Bastani, F. (1993). Strategies for mapping Lee's maze routing algorithm into parallel architectures. In *International Parallel Processing Symposium*.