# Estratégias de Otimização e Paralelização do Algoritmo de Lee para o Problema de Roteamento

**William Felipe C. Tavares, Nahri Moreano**

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 consumption. This paper evaluates optimizations described in the literature that reduce the time and memory consumption of the algorithm. The paper also proposes parallelization techniques for the algorithm, evaluating different strategies for thread synchronization, persistence and data sharing. Through a detailed experimental evaluation, we reached the final results with speedups 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. O artigo também propõe técnicas para a paralelização do algoritmo de Lee, avaliando diferentes estratégias para a sincronização entre as threads, a persistência das mesmas e compartilhamento de dados entre elas. Através de uma avaliação experimental detalhada, obteve-se como resultado final speedups 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, a automação da produção se tornou mais complexa e exaustiva. Na produção de um *chip* está presente a fase de projeto físico, constituído por procedimentos que possuem o objetivo de distribuir os componentes em uma placa considerando algumas limitações, tais como área, restrições de tempo e consumo de energia. A partir disso, então, originaram-se diversos problemas designados de projeto *VLSI* (*Very Large Scale Integration*) [Sherwani 1993]. Particionamento lógico, posicionamento, roteamento e compactação compõem a sequência de procedimentos necessários para o projeto físico. Nesse sentido, cada procedimento pode ser tratado de forma independente, respeitando a sequência estipulada. O problema de roteamento é o foco deste artigo.

O roteamento é responsável por definir rotas que conectam os componentes do *chip*. Esse procedimento recebe como entrada uma placa com as localizações dos componentes definidas pela fase de posicionamento [Chen and Chang 2009]. Espaços não ocupados, conhecidos como região de roteamento, estão disponíveis para rotear as trilhas, isto é, as rotas que conectam os componentes. Define-se como roteamento a tarefa de estabelecer trilhas conectando os componentes de uma placa. Tendo em vista características realísticas, as trilhas não podem se interceptar. Também é tarefa do roteamento minimizar o comprimento das trilhas, que afeta diretamente o desempenho e o custo de

produção do *chip*. Dependendo das restrições de fabricação, podem ser incluídas mais especificações no problema, como placas que possuem mais de uma camada, roteamentos de várias trilhas em uma mesma placa ou uma trilha com múltiplas conexões. Como em *VLSI* um *chip* contém milhões de transistores, são necessárias muitas trilhas com incontáveis possibilidades de roteamento, resultando em um problema computacionalmente custoso [Wu 2011]. Para este trabalho foram escolhidas restrições de placa com uma única camada e roteamento de uma única trilha conectando dois pontos.

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 um consumo alto de tempo e de memória, o que o torna quase inviável no âmbito de *VLSI*. Assim, o algoritmo de Lee 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.

O presente 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, avaliando diferentes estratégias para a sincronização, persistência e compartilhamento de dados entre as *threads*. A Seção 5 analisa os resultados obtidos com as implementações paralelas. A Seção 6 conclui o artigo e descreve trabalhos futuros.

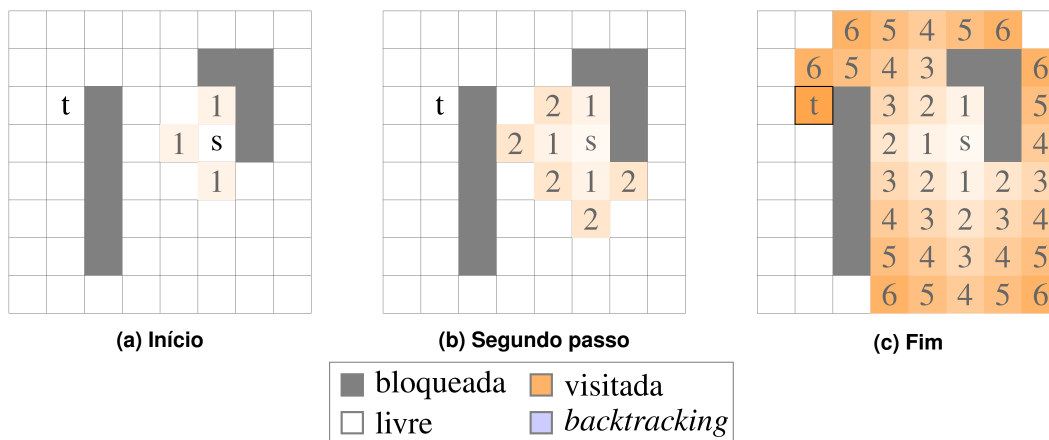
## 2. Algoritmo de Lee e Otimizações

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, norte, direita, sul e esquerda.

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.

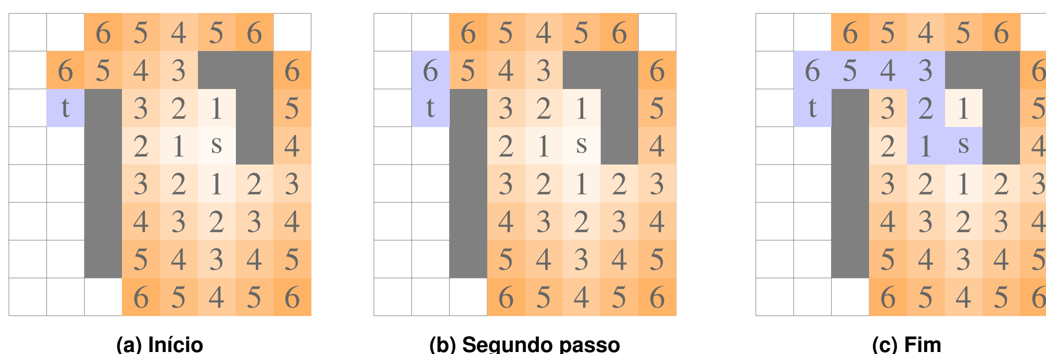
O algoritmo de Lee possui duas fases: expansão e *backtracking*. A primeira fase inicia na fonte e realiza uma onda de expansão em busca do terminal. Nessa busca, os vizinhos de cada célula são tratados e suas distâncias são calculadas com uma unidade a mais da distância da célula. Esses vizinhos são tratados apenas quando estão livres e não tenham sido visitados ainda. A Figura 1 ilustra a fase de expansão do algoritmo. Ao iniciar na fonte com distância $0$, seus vizinhos então terão distância $1$ (Figura 1a). Em seguida, os vizinhos dessas células terão distância $2$ (Figura 1b), e assim por diante, criando uma onda de expansão até o terminal ser encontrado (Figura 1c). Caso não exista um caminho que conecte ambos os pontos, a fase de expansão chegará a um momento em que não haverá células para serem tratadas, sem que o terminal tenha sido alcançado.

Na fase de *backtracking*, o caminho mais curto entre a fonte e o terminal é determinado. A Figura 2 ilustra essa fase. A partir do terminal, a segunda fase é iniciada (Figura 2a), consultando os seus vizinhos e avaliando dentre eles, qual possui a distância que sinaliza o menor caminho para a fonte (Figura 2b). Efetuando essa operação repetidamente, encontra-se o caminho de volta para a fonte (Figura 2c).




**(a) Início** **(b) Segundo passo** **(c) Fim**

**Figura 1. Fase de expansão do algoritmo de Lee: células em cinza representam obstáculos, células em branco representam a região de roteamento, e células em laranja representam a onda de expansão**



**(a) Início** **(b) Segundo passo** **(c) Fim**

**Figura 2. Fase de *backtracking* do algoritmo de Lee: células em azul claro representam o menor caminho de $s$ a $t$**

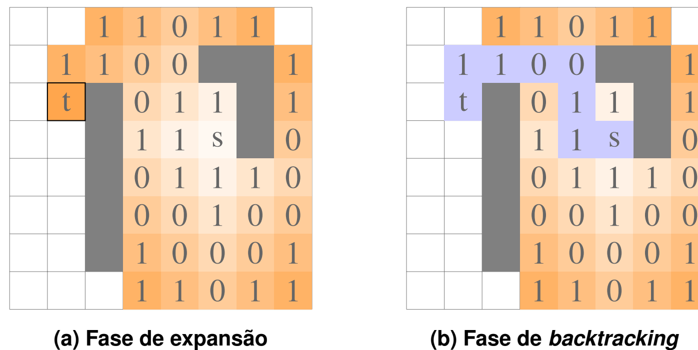
No Algoritmo 1 são descritos os passos da técnica. A *flag* $encontrado$ é utilizada para sinalizar quando o terminal é alcançado. A fila $Q$ armazena as células a serem tratadas, organizadas em ordem FIFO, e é inicializada apenas com a célula fonte. A fila $P$ armazena as células do caminho encontrado de $s$ a $t$, se ele existir. A onda de expansão inicia no laço *enquanto*. Para cada célula na fila é verificado se ela é o terminal, o que finalizará a fase de expansão. Caso contrário, seus vizinhos são visitados, calculando-se suas distâncias e inserindo-os na fila $Q$. Assim que a expansão se encerra, é verificado se o terminal foi encontrado. Se foi, a fase de *backtracking* é iniciada. Caso contrário, o algoritmo retorna um caminho vazio. Na fase de *backtracking*, o terminal é inserido no caminho $P$ e, dentre os vizinhos do terminal, avalia-se qual possui a distância que sinaliza a origem do caminho. Essa célula é então inserida em $P$, seus vizinhos são avaliados, e assim por diante, até alcançar a fonte. Dessa maneira, o algoritmo retorna o caminho $P$

encontrado. Esse algoritmo garante encontrar o menor caminho entre $s$ e $t$, se ele existir, e possui complexidade de tempo e espaço $O(m \times n)$, onde $m$ e $n$ são as dimensões do *grid*.

**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
```

Algumas otimizações foram propostas para o algoritmo em questão. Ackers 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 a distância das células, utilizar uma rotulação com apenas um bit $0, 1, 1, 0, 0, 1, 1, 0, \ldots$ (Figura 3a) e durante o *backtracking* será possível descobrir o caminho de volta (Figura 3b) [Akers 1967]. Assim, cada célula pode ser representada com apenas dois bits, um representando o rótulo e outro o estado (bloqueado ou liberado). Essa otimização diminui significativamente o consumo de memória da implementação do algoritmo.

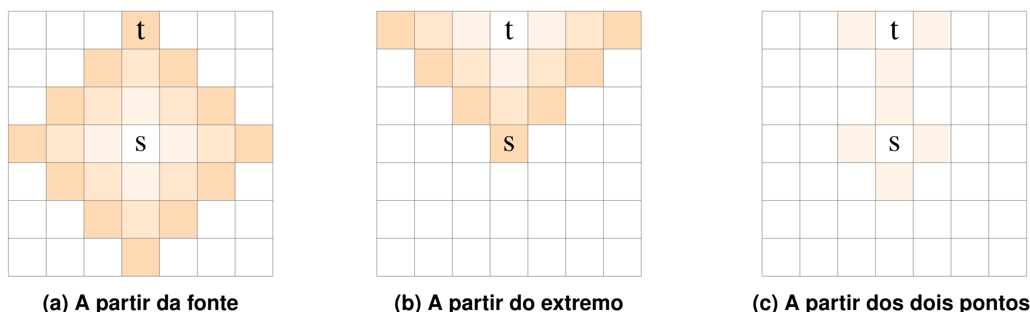



**(a) Fase de expansão** **(b) Fase de *backtracking***

**Figura 3. Fases do algoritmo de Lee com otimização de Acker**

Sait propôs algumas otimizações para diminuir o número de células tratadas [Sait and Youssef 1999]. Dentre elas há a expansão a partir do ponto, fonte ou terminal, mais próximo à borda do *grid* e a expansão a partir dos dois pontos. Isso só é possível pois o caminho não exige uma orientação, logo os pontos $s$ e $t$ podem ser trocados em relação a início e fim de expansão, ou ainda, a expansão pode ser iniciada a

partir de $s$ e $t$, até que um ponto intermediário seja alcançado a partir das duas origens. A Figura 4 mostra de forma visual as células visitadas pela onda de expansão para as diferentes formas de expansão. O número de células tratadas pode diminuir a cada uma dessas otimizações, porém isso pode não acontecer quando incluídos obstáculos no *grid*.




**(a) A partir da fonte** **(b) A partir do extremo** **(c) A partir dos dois pontos**

**Figura 4. Diferentes formas de iniciar a expansão, propostas por Sait**

## 3. Resultados Preliminares e Análise

O Algoritmo 1 sem otimizações foi implementado utilizando a linguagem de programação C++. Os testes foram conduzidos em um computador com processador Intel $i7$ de $3.6$GHz (com quatro núe com $32$GB de memória RAM e cada experimento foi executado $5$ vezes. Foram gerados aleatoriamente nove *grids* de dimensão $70.000 \times 70.000$ a $90.000 \times 90.000$. Estas dimensões foram definidas baseadas na limitação da memória RAM. Além do algoritmo original, em que a expansão é iniciada na **fonte**, foram implementadas as expansões iniciadas no ponto mais ao **extremo** (fonte ou terminal) e nos **dois pontos** (fonte e terminal). Para cada uma dessas variações, foram desenvolvidos programas em que o *grid* é representado como uma matriz de inteiros ou uma matriz de bits, respectivamente, *grids* **inteiros** e **binários**.

Algumas considerações sobre a implementação são necessárias. Os *grids* foram representados por matrizes de elementos inteiros alocadas dinamicamente – tanto para os *grids* inteiros quanto para os *grids* binários. O primeiro tipo foi utilizado para os *grids* com valores inteiros nas células, representando a distância da célula em relação à fonte; células bloqueadas foram representadas por $-1$ e células livres por $+2.147.483.647$, o maior número inteiro de $64$ bits. Os *grids* binários são manipulados através de operações de manipulação de bits, sendo possível representar $32$ células do *grid* em um elemento da matriz, que contém $64$ bits. Cada célula então possui dois bits, o bit mais significativo representa a rotulação, isto é, a ordem na onda de expansão (sequência $0, 1, 1, 0$), enquanto o bit menos significativo representa o estado ($1$ para células bloqueadas e $0$ para células livres). Nesse caso, a distância da célula em relação à fonte não é calculada nem armazenada.

Para a otimização em que a expansão é iniciada a partir da célula mais ao extremo, foi necessário utilizar a distância euclidiana da fonte e do terminal em relação ao centro do *grid* para selecionar o extremo. Para implementar a expansão que inicia dos dois pontos, é necessário que cada célula possua uma informação a mais, indicando se ela foi visitada pela expansão partindo de $s$ ou de $t$. Para o *grid* inteiro optou-se por representar essa informação usando a paridade do valor associado à célula: números pares indicam a expansão partindo da fonte e números ímpares partindo do terminal. Nessa implementação

utiliza-se duas filas, cada uma responsável por manter as células alcançadas a partir de um dos pontos, alterando minimamente o Algoritmo 1. No *grid* binário com expansão de dois pontos, foi necessário acrescentar mais um bit na representação de cada célula, totalizando 3 bits para cada célula e representando 21 células em um elemento da matriz. Com oito valores diferentes, foi possível definir um valor para células bloqueadas, um para células livres, três para rotular caminhos originados da fonte e três do terminal.

A Figura 5a apresenta o tempo médio de execução para cada programa implementado, para todos os *grids* gerados. O *grid* inteiro, por utilizar uma maior quantidade de memória, exige muitos acessos aos diferentes níveis de memória 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 5b. 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.

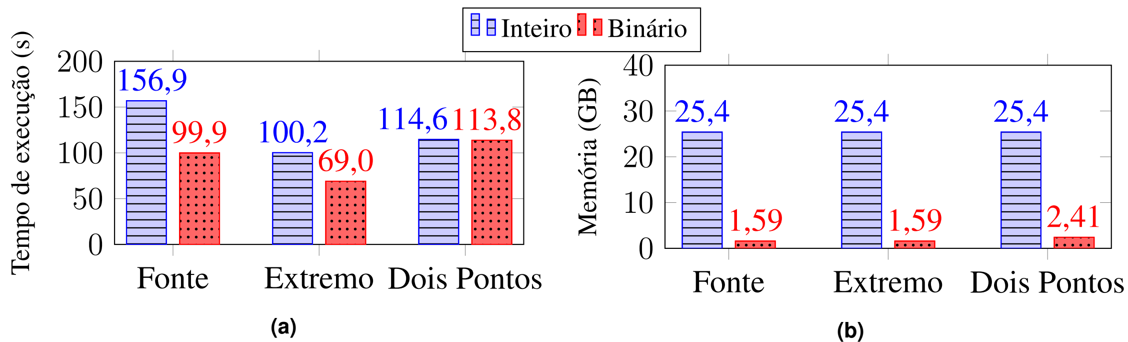



**Figura 5. (a) Tempo de execução médio e (b) quantidade média de memória utilizada para cada forma de expansão, para *grids* inteiros e binários**

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, pois um número menor de células são visitadas (como visto na Tabela 1). Entretanto, 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* | $70.000 \times 70.000$ | $80.000 \times 80.000$ | $90.000 \times 90.000$ | 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**

## 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,

Kurç et al. 1991], 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, no roteamento de uma única trilha no *grid*.

Para a implementação das versões paralelas foi utilizado modelo de programação OpenMP [OpenMP Architecture Review Board 2018]. As execuções foram realizadas no mesmo computador usado para as implementações sequenciais (descrito na Seção 3), que possui quatro núcleos. Foram utilizados os mesmos grids gerados para os experimentos anteriores.

### 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 ao mesmo tempo 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). O Algoritmo 2 apresenta essa modificação da fase de expansão.

**Algoritmo 2:** Expansão: tratamento em paralelo das células do 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 ← ∅
```

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 *grid* precisa ser definido como uma estrutura de dados compartilhada entre as *threads*, portanto é necessária uma sincronização na visita a um vizinho, para evitar que uma célula seja visitada ao mesmo tempo por duas ou mais *threads* e tenha seu valor sobrescrito. Essa sincronização foi realizada inicialmente usando a construção `critical` do OpenMP e a visita é tratada como uma única seção crítica, o que deu origem ao primeiro programa paralelo desenvolvido. Entretanto, essa solução realiza sincronizações

desnecessárias, dado que células diferentes podem ser visitadas em paralelo, sem necessidade de sincronização.

Para contornar a situação e restringir a seção crítica para uma célula em específico, foram utilizadas variáveis *lock* do OpenMP. Idealmente, cada célula deveria possuir um *lock*, o que exigiria $n \times m$ variáveis desse tipo, o que seria inviável. Porém, como não são todas as células que são visitadas em um nível de expansão, é viável reduzir o número de *locks* para $n + m$, uma quantidade razoável de células visitadas a cada nível de expansão. Esse tipo de sincronização caracterizou o segundo programa paralelo desenvolvido.

Todavia, como os vizinhos visitados em paralelo estão em um mesmo nível da expansão, a sobrescrita de uma célula, em paralelo por duas ou mais *threads*, com o valor de distância da célula, não causa inconsistência, pois todas as *threads* escrevem o mesmo valor na célula visitada. Considerando isso, foi implementado um terceiro programa em que não há sincronização nenhuma no acesso às células do *grid* e permite-se que mais de uma *thread* visite uma mesma célula e sobrescreva o seu valor.

Essas três estratégias de sincronização entre as *threads* 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 6 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, restringindo muito o paralelismo. O programa com *lock*, em que a sincronização é realizada apenas para visitas a uma mesma célula, ainda possui *overhead*. O programa sem sincronização possui o melhor desempenho, com *speedup* de $1,67$ em relação ao programa sequencial equivalente, por as duas *threads* tratarem conjuntos diferentes de células em paralelo.

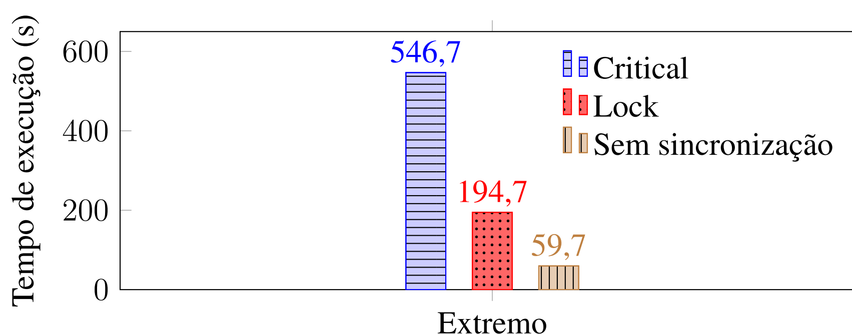



**Figura 6. Tempo de execução médio dos programas paralelos com sincronização `critical` e *lock* e sem sincronização, usando $2$ *threads*, para *grids* inteiros partindo do ponto mais extremo**

Dada a forma como a paralelização foi implementada, é necessário que as *threads* sejam criadas no início e destruídas ao final de cada nível da expansão. É possível, então, que haja um *overhead* excessivo para a criação e destruição dessas *threads*. Para que essa criação de *threads* seja realizada apenas uma vez, optou-se por criá-las antes do laço *enquanto* da fase de expansão, incluindo a construção `#pragma omp parallel` antes dele no Algoritmo 2. Essa nova versão exigiu uma sincronização de barreira entre as *threads* ao fim de cada iteração do laço *enquanto*, pois as *threads* devem trabalhar no mesmo nível da expansão.

As duas estratégias de gerenciamento de *threads*, criação/destruição e persistência, foram avaliadas. A Figura 7 mostra o tempo de execução dessas soluções. A primeira solução, onde há criação e destruição das *thread* a cada nível de expansão, obteve um desempenho melhor que a solução onde as *threads* são reutilizadas, provavelmente devido à necessidade de sincronização. Desta forma, o gerenciamento de criação e destruição de *threads* foi mantido e a proposta de reutilização foi descartada.

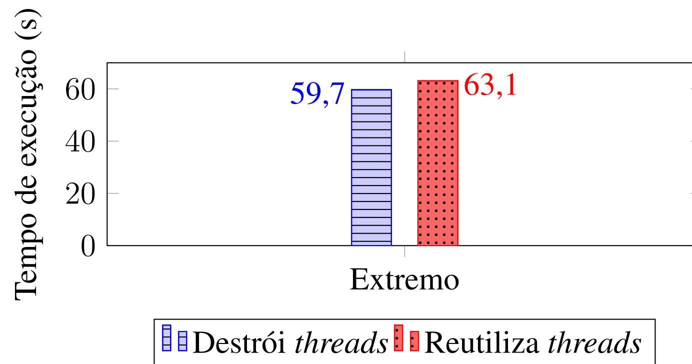



**Figura 7. Tempo de execução médio dos programas paralelos com criação/destruição e reutilização de *threads*, usando 2 *threads*, para *grids* inteiros partindo do ponto mais extremo**

As estruturas utilizadas para a fila auxiliar $Q_{aux}$ geram um falso compartilhamento [Hennessy and Patterson 2011] 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. Essas estruturas de dados foram rearranjadas para evitar que o falso compartilhamento aconteça. A Figura 8 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 $1,34$.

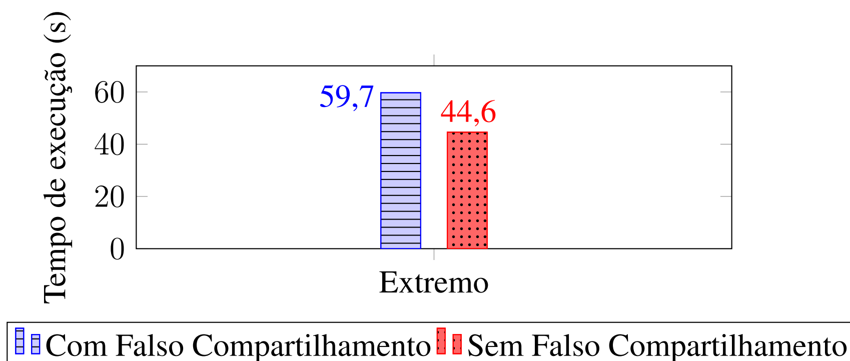



**Figura 8. Tempo de execução médio dos programas paralelos com e sem falso compartilhamento, usando 2 *threads*, para *grids* inteiros partindo do ponto mais extremo**

A Figura 9 mostra o tempo de execução da versão 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.

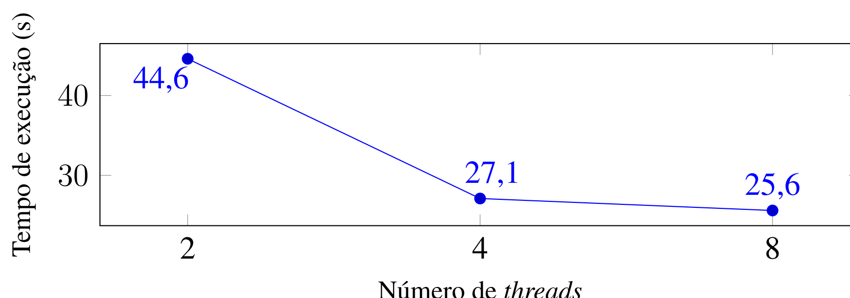



**Figura 9. Tempo de execução médio do programa paralelo sem falso compartilhamento para *grids* inteiros partindo do ponto mais extremo**

### 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 10a 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.

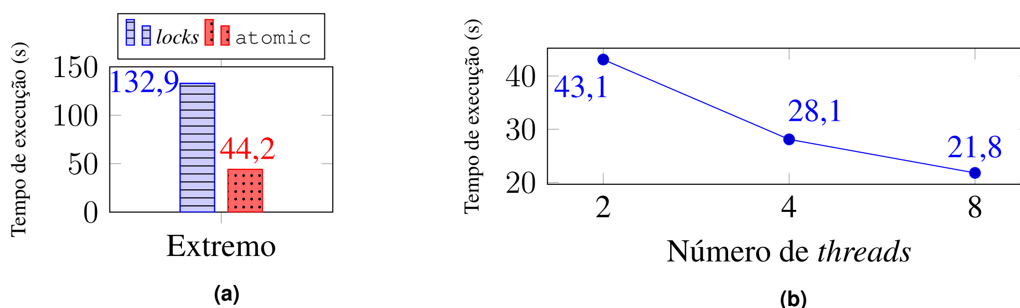



**Figura 10. Tempo de execução médio dos programas paralelos, (a) usando $2$ *threads* e (b) variando o número de *threads*, para *grids* binários partindo do ponto mais extremo**

A implementação com a sincronização com *atomic* foi executada variando o número de *threads*, e os resultados são mostrados na Figura 10b. 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. E ainda é possível aplicar, de forma conjunta, 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 na a partir do terminal. Em resumo, no seu nível de paralelismo mais básico, essa versão possui $2$ *threads*, cada uma responsável por um dos dois pontos de expansão . Ao aumentar para $4$ *threads*, são distribuídas duas *threads* para cada ponto de expansão, e assim por diante. Ao contrário das outras versões paralelas, aqui é possível explorar 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 *grids* inteiros e binários, também reorganizando as estruturas para não haver falso compartilhamento. Para a versão com *grids* binários, a sincronização *atomic* é usada. A Figura 11 apresenta os tempos de execução dessas versões, variando o número de *threads*. De forma geral, a versão com *grids* inteiros 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.

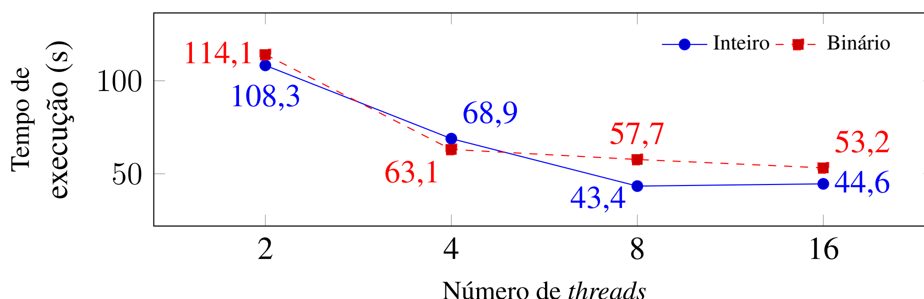



**Figura 11. Tempo de execução médio dos programas paralelos para *grids* inteiros e binários, com expansão partindo de dois pontos**

## 5. Resultados e Análise

Considerando as diversas versões implementadas, os melhores resultados foram obtidos nos programas paralelos sem sincronização na visitação das células, com criação e destruição de *threads* e sem falso compartilhamento entre as *threads*. A Figura 12 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 13a e 13b 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.

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

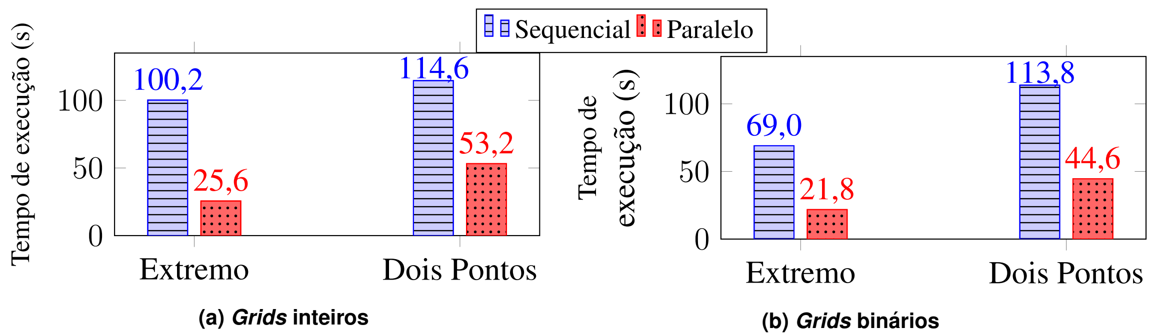



**Figura 12. Tempo de execução médio dos programas sequenciais e paralelos para *grids* (a) inteiros e (b) binários, partindo do ponto mais ao extremo ($8$ *threads* no paralelo) e dos dois pontos ($16$ *threads* no paralelo)**

menos memória, exige sincronização no acesso à célula, afetando negativamente sua performance, o 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 ao seu elevado *overhead* de controle.

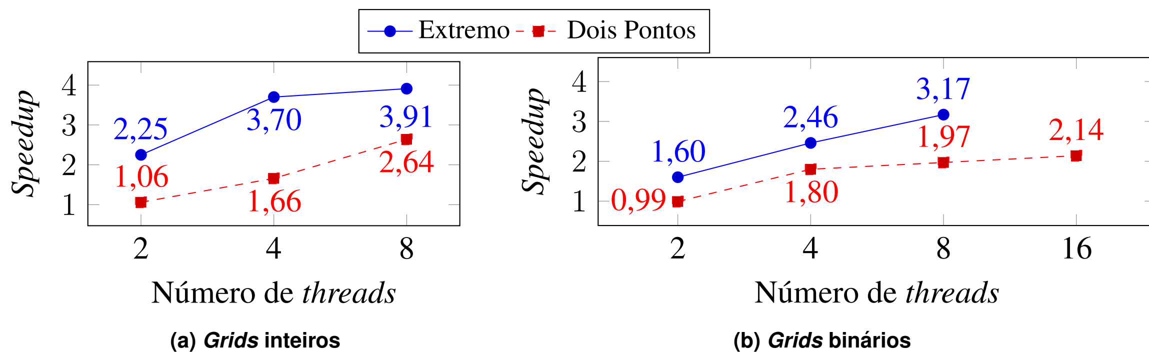



**Figura 13. *Speedup* dos programas paralelos em relação ao sequencial correspondente para (a) *grids* inteiros e (b) *grids* binários**

## 5.1. Testes de esforço

Os melhores resultados foram obtidos pelas versões paralelas sem sincronização no tratamento das células, sem persistência de *threads* e sem falso compartilhamento entre elas, para os *grids* inteiros e binários, com propagação a partir do ponto mais extremo. Essas versões foram submetidas a testes de esforço, utilizando três *grids* de dimensão $100.000 \times 100.000$ a $120.000 \times 120.000$. Essas instâncias, por se tratarem de *grids* de dimensões grandes, exigiram um alto tempo de paginação. Desejou-se, então, descobrir se o paralelismo ainda é uma alternativa considerável para instâncias dessa magnitude.

A versão com *grids* inteiros apresentou tempo de execução médio de $5178,3$ segundos para o programa sequencial e $3481$ segundos para o programa paralelo com $8$ *threads*. A versão com *grids* binários obteve melhor desempenho, executando em média em $200$ segundos e $64$ segundos, com os programas sequencial e paralelo, respectivamente. É um contraste grande entre as versões de matrizes diferentes, influenciada pela quantidade de acessos à memória que exige-se no *grid* de inteiros.

Na Figura 14 encontram-se os *speedups* obtidos nos testes, variando a quantidade de *threads*. A paralelização para os *grids* inteiros, apesar de diminuir o tempo de execução, não apresentou um crescimento grande do *speedup* linear, com o aumento do número de *threads*. Por outro lado, o *speedup* foi crescente para os *grids* binários.

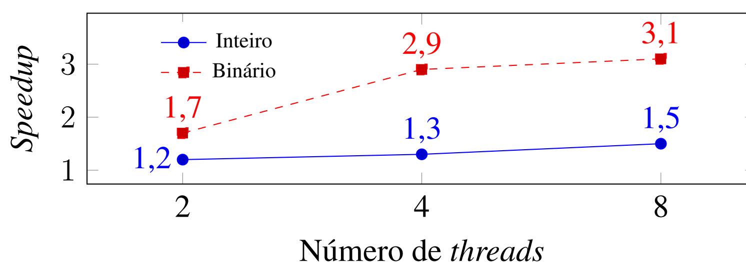



**Figura 14. *Speedup* dos melhores programas paralelos em relação aos sequenciais, para *grids* inteiros e binários, para *grids* do teste de esforço**

## 6. Conclusão

O algoritmo de Lee apresenta alto consumo de tempo e de memória e, a partir de otimizações, essas exigências podem ser contornadas. Neste trabalho foi realizada uma implementação construtiva, onde, a cada passo, diferentes estratégias foram avaliadas e uma otimização foi agregada à implementação final. A sincronização entre as *threads*, a persistência ou não das mesmas e o efeito do falso compartilhamento nas caches foram alguns dos desafios tratados. Cada implementação paralela desenvolvida tendeu a 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, para *grids* de dimensão $70.000 \times 70.000$ a $90.000 \times 90.000$.

A memória exigida permanece como o grande gargalo do algoritmo e limitação dos testes no computador utilizado. As versões em que os *grids* são modelados de forma binária, mesmo que com menor consumo de memória, possuem alto grau de controle, o que afetou negativamente o desempenho.

Uma possível melhoria é modificar a indexação nos *grids* binários, de forma que cada elemento da matriz codifique um *subgrid* e não apenas uma linha do *grid*, otimizando o acesso aos vizinhos. Outra ideia interessante é explorar outras abordagens de paralelização, como por exemplo o modelo mestre/trabalhador, onde cada trabalhador é responsável por um conjunto de células.

Neste trabalho trabalhou-se com *grids* com uma única camada e conectando dois pontos com uma única trilha. Expandir a paralelização proposta aumentando esses parâmetros é uma possibilidade 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*, pages 687–749. Elsevier.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition.

Kurç, T. M., Aykanat, C., and Erçal, F. (1991). *Parallelization of Lee's routing algorithm on a hypercube multicomputer*, pages 244–253. Springer Berlin Heidelberg, Berlin, Heidelberg.

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 *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 814–820.

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 *Proceedings of the Workshop on Programmability Issues for Heterogeneous Multicores*, page 54.

Sherwani, N. A. (1993). *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, Norwell, MA, USA.

Wu, T.-H. (2011). *A parallel integer programming approach to global routing*. PhD thesis, University of Wisconsin–Madison.

Yen, I., Dubash, R., and Bastani, F. (1993). Strategies for mapping Lee's maze routing algorithm into parallel architectures. In *Proceedings of the International Parallel Processing Symposium*, pages 672–679.