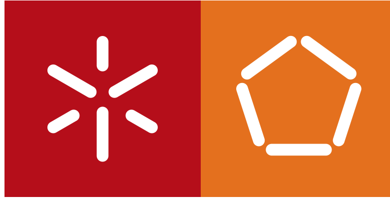
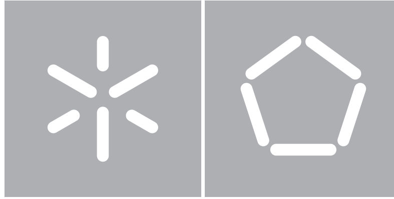


Universidade do Minho
Escola de Engenharia

Diogo da Cunha Rodrigues

Simulação em ambientes paralelos da dispersão de fotões num guia de luz

Janeiro de 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo da Cunha Rodrigues

Simulação em ambientes paralelos da dispersão de fotões num guia de luz

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor João Sobral
Professor Eduardo Pereira

Janeiro de 2014

Agradecimentos

Nem sempre é possível pegar num determinado acontecimento e removê-lo totalmente do contexto onde está inserido, como que erradicando todas as ligações pré-estabelecidas, esquecendo o passado, e tudo o que teve por base a que esse acontecimento tivesse lugar.

Este trabalho, além do valor isolado que representa, assume-se também como o final de um ciclo, de um percurso duradouro, de uma aprendizagem multi-nível, e do que para muitos é, a melhor fase da vida de um ser humano.

Nesse sentido, quero deixar um agradecimento especial à minha Família, aos meus Amigos, por todo o apoio e motivação que me deram ao longo desta etapa. Quero agradecer também aos colegas de trabalho pela opinião crítica e objetiva, em especial ao Rafael Silva que sem o seu conhecimento técnico e espírito de ajuda não seria possível ultrapassar certos desafios.

Por fim, agradecer aos meus orientadores, o Prof. João Sobral e o Prof. Eduardo Pereira, pela disponibilidade e empenho que dedicaram, mas acima de tudo, pelo carácter técnico e motivador que sempre demonstraram, ajudando-me no apuro de um sentido prático de engenharia, com uma visão objetiva, analítica, crítica e construtiva.

Resumo

Nesta dissertação pretende-se desenvolver um programa que simule a propagação de fotões num guia de luz. De forma a simular a realidade com precisão será necessário calcular a trajetória dum elevado número de fotões ($\pm$ 10^{10}), fazendo com que o cálculo seja extremamente demorado se for executado sequencialmente. De forma a torná-lo exequível num período de tempo mais curto e, eventualmente com maior precisão (mais fotões), serão exploradas técnicas de processamento paralelo.

Este trabalho insere-se num grupo de problemas da área de ótica. A solução será alcançada através de ajustes nas propriedades óticas do guia de luz, para obter uma dispersão homogénea de luz na saída.

A simulação de fotões, em geometrias complexas, é implementada através de métodos de Monte Carlo para retratar os vários elementos aleatórios. Foi feita uma análise a vários geradores de números aleatórios para implementar o que se destaca na qualidade dos números, paralelização e período da sequência. A simulação foi paralelizada num ambiente de memória partilhada, distribuída e híbrido.

Do trabalho produzido concluiu-se que independentemente do ambiente de paralelização, os resultados de desempenho e eficiência foram muito positivos. Os resultados da simulação mostram que o problema físico ficou resolvido, a homogeneidade na saída apresenta uma uniformidade de 79% e $0,018$ de Black-MURA, valores que são concordantes com os obtidos num protótipo real.

A paralelização obteve um ganho máximo, de 40, foi atingido com o modelo de memória distribuída utilizando 48 processos. Devido à natureza do problema, a independência dos fotões, a eficiência para memória partilhada e distribuída foi de 90% e 84%, respetivamente.

O modelo híbrido, devido à computação extra para a gestão dos fios de execução, não apresentou resultados positivos.

Abstract

This masters thesis will be focused on the development of a computer simulation program to solve a particular problem of Monte Carlo, the photon migration on a light guide. To simulate light, the problem has to be broken down to the simulation of each photon (approximately 10^{10}*)*. This task, if done sequentially, consumes a lot of computational resources. Therefore, the problem will be approached, and implemented, in a parallel fashion way, using parallel paradigms to make the execution time shorter and eventually obtain a higher level of precision (more photons). The final result will be achieved through tweaking different properties of the light guide in order to achieve an homogenous light distribution at the output.

The photon migration simulation is part of a wide array of problems which are solved through Monte Carlo processes. This group shares a common property, the randomness of certain events. Therefore, it was done an analysis to different Random Number Generators to discover which would have the best quality, the longest period and more flexible in terms of paralelism. The simulation was paralellized in three environments, shared and distributed memory and a hybrid combination of both.

This simulation adapted well to the different parallel environments with positive results of speedup and efficiency. The light distribution at the exit was achieved with 79% of uniformity and a Black-MURA factor of $0,018$.

The maximum speedup, 40, was achieved with the distributed memory model with 48 processes. Due to the nature of the problem, photons are independent, the efficiency of the shared and distributed memory was 90% and 84%, respectively. The hybrid model did not presented positive results due to the extra computation to create, manage and destroy threads.

Índice

I Introdução 1

1.1 O problema do transporte de luz 1

1.2 Método de MC aplicado ao transporte de fotões 2

1.3 Objetivos 3

II Estado da arte 4

2.1 Métodos de Monte-Carlo 4

2.2 Geração de números aleatórios 4

2.3 Técnicas de geração de números pseudo-aleatórios 5

2.4 Geração de números em paralelo 9

2.5 Paradigmas de paralelização 10

2.5.1 Memória partilhada 10

2.5.2 Memória distribuída 12

2.5.3 Híbrido 13

2.6 Plataformas de simulação do transporte de fotões em meios túrbidos 13

III Desenvolvimento da solução 15

3.1 Sistema físico a modelar 15

3.1.1 Descrição do software 17

3.2 Implementação base 19

3.2.1 Transporte para C e validação 19

3.2.2 Gerador de números aleatórios 19

3.2.3 Melhoramento da qualidade da simulação 20

3.2.4 Otimização de parâmetros 21

3.2.5 Propriedades óticas implementadas 23

3.2.6 Critérios de qualidade 25

3.3 Abordagem de paralelização 26

3.3.1 Memória partilhada 26

3.3.2 Memória distribuída 30

3.3.3 Híbrido 32

IV Resultados e discussão **35**
4.1 Validação de resultados 35
4.2 Otimização do desempenho da versão base 36
4.3 Memória partilhada 40
4.3.1 Paralelismo ao nível dos LEDs 41
4.3.2 Paralelismo ao nível dos Fotões 42
4.3.3 Paralelismo ao nível dos LEDs & Fotões 42
4.3.4 Conclusões 43
4.4 Memória distribuída 45
4.4.1 Balanceamento dinâmico e granularidade 45
4.4.2 Híbrido 46
4.4.3 Conclusões 47

V Conclusão e trabalho futuro **48**

Índice de figuras

2.1 *Middle square method.* 7
2.2 Exemplar de um modelo da arquitetura de memória partilhada. . . 10
2.3 Hierarquia de memória de um sistema NUMA com 2 sockets independentes para dois processadores Quadcore. 11
2.4 Exemplar de um modelo da arquitetura de memória distribuída. . . 12
3.1 Ecrã iluminado por 64 LEDs (32 no topo e 32 na parte inferior) . . 15
3.2 Modelo 3D (em corte) do guia de luz inserido no invólucro. 16
3.3 Vista frontal do modelo real (os focos luminosos laterais são os LEDs). 16
3.4 Fluxograma da execução - a azul parte sequencial - laranja parte que foi paralelizada. 17
3.5 Distribuição das rugosidades nas faces de topo e inferior (LEDs a vermelho) 21
3.6 Saída do guia de luz (face de topo). 22
3.7 Reflexão Lambertiana de uma superfície. 23
3.8 Casos em que acontece a reflexão Lambertiana no guia de luz. . . . 24
3.9 Reflexão Fresnel de uma superfície. 24
3.10 DA - Área do *display* considerada para medir a qualidade. 25
3.11 Fluxograma da execução em memória distribuída 32
3.12 Fluxograma da execução híbrida - a laranja a paralelização em memória partilhada. 34
4.1 Perfil de execução da versão com funções trigonométricas. 36
4.2 Perfil de execução da versão sem funções trigonométricas. 38
4.3 Percentagem do código que foi alvo de paralelização dependendo do nº de fotões (notar que a escala dos XX é logarítmica). 39
4.4 Comparação entre as versões sequenciais (notar que a escala dos XX é logarítmica). 40
4.5 Ganho da versão LED comparativamente à versão serial. 41
4.6 Ganho da versão Fotões comparativamente à versão LEDs. 42
4.7 Ganho de todas as versões em memória partilhada. 43
4.8 Eficiência de todas as versões em memória partilhada. 44
4.9 Tempo de execução com a variação da granularidade (notar que a escala dos XX é logarítmica). 45
4.10 Tempo de comunicação extra do processo Mestre devido à granularidade(notar que a escala dos XX é logarítmica). 46

4.11 Ganho das várias combinações do modelo híbrido. 47

Índice de tabelas

2.1 Os tempos de execução das diferentes versões do MCML demonstram que o GCMCML é o mais rápido. A GPU usada foi uma 9800 GT (112 CUDA CORES) e o CPU foi o Intel Core2Duo E7300 (2 fios de execução em *Hyper-Threading*) @ 2.67 GHz. 14

3.1 Tabela com os parâmetros de entrada otimizados. Conjunto de valores $\{V1\,V2\,V3\}$. 22

4.1 Resultados de saída da simulação com os parâmetros definido em 3.2.4 35

4.2 Tabela de ganhos das versões em memória partilhada. 43

4.3 Tabela da eficiência das versões em memória partilhada. 44

4.4 Tabela de ganhos das versões em memória partilhada e híbrida com 12 processos/máquina. 47

Lista de símbolos e abreviaturas

CUDA Compute Unified Device Architecture

DCMT Dynamic Creation of Mersenne Twisters

GCMCML GPU Cluster Monte Carlo Multi Layered

LCD Liquid-crystal display

LCG Linear Congruential Generator

LED Light-emitting Diode

MC Monte Carlo

MCML Monte Carlo Multi Layered

MPI Message Passing Interface

NUMA Non-uniform Memory Access

PRNG Pseudo-Random Number Generator

Capítulo I

Introdução

Esta dissertação é focada em desenvolver um programa que permita, em tempo útil, simular o processo de propagação de fotões num guia de luz. O guia de luz é um componente crítico dos *Liquid-Crystal Displays* (LCD) fabricados para o uso na industria automóvel (sistemas de navegação, auto-rádios, etc.) onde o objetivo é otimizar a distribuição de luz que sai na face frontal. Esta ferramenta computacional permitirá a otimização do dispositivo físico de forma mais eficiente do que o estudo experimental empírico.

Na simulação das trajetórias dos fotões, os métodos de Monte Carlo (MC) são usados para representar o comportamento dos fotões ao incidirem num material. Isto é, podem ser refletidos, refratados ou até absorvidos e é este cálculo para milhões de fotões a razão para as simulações serem computacionalmente pesadas.

Assim, é criada a necessidade de explorar o paralelismo neste tipo de problemas. As trajetórias são independentes e existem em grande número, tornando atrativa a exploração de paralelismo para reduzir o tempo de execução.

1.1 O problema do transporte de luz

Recorrer a métodos analíticos para descrever o movimento de fotões, através de equações diferenciais, é uma tarefa árdua e extensa. Em geometrias complexas, o cálculo necessário para reproduzir resultados cientificamente aceitáveis é extenso e difícil, sendo apenas exequível para geometrias ideais, pouco importantes na prática[1].

Assim, nasce a computação gráfica que através da simulação do transporte de luz cria imagens virtuais de um mundo real. Ao fornecermos a descrição deste mundo através da geometria dos objetos e das suas propriedades óticas, a(s) fonte(s) de luz e do ponto de vista donde a imagem será gerada, os algoritmos de transporte de luz simulam as propriedades físicas de forma a reproduzir imagens próximas do real.

O problema é simular cada fotão, assim, o disparo de milhões de fotões torna-se numa tarefa computacionalmente pesada. No entanto, de acordo com as propriedades físicas dos fotões estes são independentes, ou seja, o comportamento de um não influencia outro. Isto facilita a exploração de paralelismo, fazendo com que seja possível simular n fotões em paralelo.

No mundo da simulação de fotões existem dois tipos de simulações, *sequencial* e

*non-sequencial ray-tracer*s, sendo o primeiro um tipo de simulação onde já se sabe *a priori* o percurso que cada fotão irá percorrer, no segundo o percurso de cada fotão é imprevisível devido às refrações e múltiplas reflexões[2]. O problema aqui a simular é o último, um modelo *non-sequencial ray-tracer*.

1.2 Método de MC aplicado ao transporte de fotões

Atualmente várias áreas da ciência recorrem a métodos MC para estudar problemas complexos. Alguns exemplos são, a propagação de luz numa atmosfera e em tecidos biológicos, diagnósticos cancerígenos e tratamentos usando radiação em meios túrbidos, otimização de lâmpadas fluorescentes, etc. Nestes sistemas complexos, a simulação através de processos de MC é considerada como a regra de ouro porque os resultado analíticos só são possíveis em geometrias ideais[3].

A modelação do transporte de fotões com o método de MC é uma abordagem flexível, porém, computacionalmente intensiva. As regras de transporte, para cada superfície, são expressas na forma de distribuições de probabilidades que determinam o movimento dos fotões através do ângulo de reflexão aquando o embate numa superfície.

Na sua essência a modelação de MC é estatística, como tal, para atingir a exatidão cientifica é necessário reduzir o erro. A qualidade do resultado estatístico (número de casas decimais com precisão) depende da quantidade de amostras simuladas (neste caso fotões). Assim, para obter um cálculo 10 vezes mais exato é necessário simular 100 vezes mais fotões, fórmula 1.

$$Exatidão\,de\,cálculo \propto \frac{1}{\sqrt{Número\,de\,fotões}} \tag{1}$$

Ainda, estas simulações permitem manter o registo de várias propriedades físicas com diferentes resoluções temporais e espaciais, exigindo apenas mais tempo de computação e memória.

Esta flexibilidade torna a modelação de MC uma ferramenta poderosa, no entanto, computacionalmente pesada. Ainda assim é considerada a norma não só no transporte de fotões como noutras áreas da ciência.

1.3 Objetivos

A parceria existente ente os departamentos de física e informática e a Bosch Car Multimedia fez com que esta tese se centrasse em :

1. Construir um código para simular a trajetória percorrida pelos fotões num guia de luz garantindo que os resultados são cientificamente corretos. O código deverá também possibilitar a alteração de vários parâmetros da simulação e a fácil visualização dos resultados da simulação.

2. Estudar técnicas de processamento paralelo visando a obtenção de resultados em tempo útil, identificando também quais as plataformas/técnicas de processamento paralelo mais adequadas.

3. Pesquisar técnicas de geração de números aleatórios em ambientes paralelos e existência de bibliotecas públicas de geradores, incluindo a avaliação da qualidade dos números gerados.

Capítulo II

Estado da arte

Neste capítulo serão apresentadas as várias técnicas de geração de números, a abordagem paralela nos diferentes modelos e ferramentas existentes de simulação.

2.1 Métodos de Monte-Carlo

Os métodos MC são usados para simular sistemas matemáticos e físicos, sendo também especialmente úteis quando se trata de sistemas com elevados graus de liberdade. Na matemática recorre-se a MC para avaliar integrais multi-dimensionais com condições de fronteira complexas, também é uma ferramenta utilizada nas áreas de negócio para calcular o risco de certas decisões. Os métodos de MC são melhores que a intuição humana quanto à previsão de falhas, excesso de custos e atrasos em projetos[4].

A robustez de um método de MC depende da qualidade dos número gerados. Tais critérios são, o período da sequência de números, o desempenho e a precisão dos números a gerar (*float* ou *double* por exemplo). O desempenho é uma característica que, em problemas onde são necessários vários números num curto espaço de tempo, afetará diretamente o tempo de execução da simulação.

2.2 Geração de números aleatórios

A geração de números desenvolveu-se, tal como outras, com o desenrolar de uma guerra. Tanto na área da criptografia como na física a qualidade dos geradores era um fator crítico para produzir os resultados requeridos pelas diferentes fações. Assim, foi desenvolvida uma nova abordagem a este tipo de problemas, introduzida por Stanislaw Ulam e John von Newmann que desenvolveram o método de MC.

Este método permitiu a descoberta da bomba atómica, através do "Manhattan Project", que não teria sido possível se não fossem desenvolvidos geradores de elevada qualidade (para a época). Inicialmente eram usadas listas de "verdadeiros" números aleatórios o que em termos de performance era extremamente lento. John von Newmann desenvolveu uma forma de calcular números pseudo-aleatórios (*Pseudo-Random Number Generator* - PRNG). Este algoritmo gerava uma sequência de números que se aproximava das propriedades dos números verdadeiramente aleatórios mas com uma performance superior.

Antes de adiantar mais acerca da geração de números aleatórios é de ter em atenção o comentário de von Newmann, "quem considerar que métodos aritméticos são capazes de produzir números aleatórios está, claramente, em estado de pecado". O que von Newmann quer dizer é que por mais longo que o período seja e com as melhores propriedades estatísticas, estes métodos (aritméticos) serão sempre uma aproximação do real onde o resultado final poderá ser previsto (determinístico).

Várias técnicas foram abordadas na geração de números, e de facto, parece existir uma linha que divide os geradores de números verdadeiramente aleatórios dos geradores de números pseudo-aleatórios. Num hemisfério geradores por *hardware*, no outro por *software*, isto é :

- Os geradores em *hardware* surgiram devido à exigência, na área da criptografia, de produzir sementes[1] verdadeiramente aleatórias. Aqui, é usada como fonte de aleatoriedade eventos físicos, por exemplo, fenómenos quânticos, ruído num sensor ou até fenómenos fotoelétricos que teoricamente são considerados completamente aleatórios.

- Geradores em *software*, ao usarem algoritmos determinísticos, produzirão valores que não são verdadeiramente aleatórios, no entanto, como não recorrem a elementos físicos têm um desempenho superior.

Por vezes é exigido que uma simulação seja completamente aleatória. Assim, estes dois tipos de geradores complementam-se, isto é, ao usar-se geradores verdadeiramente aleatórios para gerar a semente inicial e pseudo-aleatórios para gerar a sequência de números para a simulação é garantido que o ponto de partida da simulação foi verdadeiramente aleatório. Por outro lado, pode ser útil utilizar a mesma semente visto que reproduzirá os mesmo resultados, ajudando na deteção de erros.

2.3 Técnicas de geração de números pseudo-aleatórios

Na geração de números pseudo-aleatórios existem muitas formas de abordar os mais variados problemas. Sabendo em concreto as necessidades do problema, restringe-se o leque de procura visto serem conhecidas as propriedades de cada um.

O critério de decisão acerca de um gerador se é ou não bom para um determinado problema pode variar. Porém existem vários critérios transversais que definem a qualidade de um gerador :

[1] Uma semente, é o valor que o gerador usa para iniciar a sequência. Sementes iguais, equivalem a sequências iguais.

- Gerar uma distribuição uniforme em $[0, 1]$.
- Livre de correlações sequenciais.
- Período suficientemente grande.

Para classificar um gerador quanto à qualidade da geração de números existem diferentes testes que avaliam estas características.

Durante vários anos a bateria de testes criada por Marsaglia, *DIEHARD Battery of tests of Randomness* foi considerada como o padrão a seguir[5]. Com o avançar da tecnologia novos padrões surgiram e os testes mudaram. Atualmente, sugerido por L'Ecuyer, a bateria de testes da framework TestU01, que contém 3 testes diferentes já pré-definidos, SmallCrush (consiste em 10 testes), Crush(96 testes) e BigCrush (106 testes) é o padrão de referência. A descrição específica destes testes está descrita no Manual de Utilização do TestU01[6].

Estes testes verificam se um gerador contém falhas de estatísticas (distribuição uniforme, correlação na sequência) na produção de números. Porém, um gerador que passe todos os testes não é garantido que não irá produzir erros em simulações MC. Aliás o que diferencia um bom PNRG dum mau é que o mau falhará em testes mais simples, enquanto que o bom falhará somente em testes muito complexos[7].

Os geradores primitivos utilizavam técnicas de *bit-shifting*. Ao deslocar um bit para a esquerda conseguem gerar uma sequência *quasi-random*[2] muito rapidamente. O período deste geradores é de 2^w (onde w é o tamanho da palavra, *word*[3], do CPU). No entanto, técnicas de *shifting* são atualmente usadas na geração de números em complemento a técnicas avançadas, visto serem a forma mais básica e rápida de manipulação de bits.

O primeiro gerador criado por Jon von Newmann utiliza a técnica de *middle-square*. Este gerador usa apenas uma operação aritmética, a multiplicação, para gerar o novo número. Por exemplo, usando a semente, faz-se o quadrado desta e do resultado, retiram-se os 4 dígitos do meio, figura 2.1. Rapidamente concluiu-se que este algoritmo tinha várias falhas, uma delas por exemplo, se um dos resultados fosse 2340000738, o algoritmo gerava eternamente 0. No entanto, para o problema de Jon von Newmann este gerador era adequado.

Um gerador de números aleatórios genérico e recente é o *Linear Congruential Generator* (LCG) que produz uma distribuição uniforme em $[0, 1]$ e é suficientemente bom para ser o gerador, por omissão, de várias linguagens de programação.

[2]Uma sequência aleatória, porém, não aleatória o suficiente para ser considerada pseudo-aleatória

[3] *Word* é o número fixo de bits que o intruction set/hardware processa como sendo uma unidade

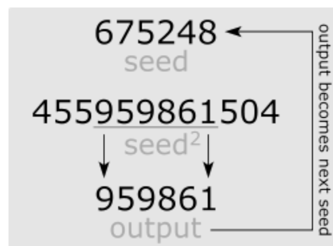


Figura 2.1: *Middle square method.*

Como já foi sujeito a um vasto período de investigação, conhecem-se os parâmetros que determinam certos tipos de comportamentos estatísticos. O algoritmo está descrito na fórmula 2 onde m é o módulo, a o multiplicador, c a constante e X_n a semente.

$$X_{n+1} = (a \cdot X_n + c) \times mod\, m \tag{2}$$

Para o LCG, o valor de m determina o período máximo da sequência. No entanto, na maioria dos casos o período acaba por ser mais curto. Knuth ao estudar o LCG concluiu que, se c não for 0 o LCG terá um período completo para todas as sementes se e só se [8]:

- c e m forem coprimos
- $a - 1$ for divisível por todos os fatores primos de m
- $a - 1$ for múltiplo de 4 se m for múltiplo de 4

Estes três requisitos produzem resultados bons para a maioria das aplicações. Porém, este algoritmo é extremamente sensível aos parâmetros c, m e a. Da investigação realizada sobre o LCG, resultou uma lista de vários parâmetros que produzem diferentes resultados. Por norma o valor de m é uma potência de 2, visto que assim o módulo resulta numa operação computacional de truncar bits. Apesar de ser um gerador genérico, o facto de produzir uma série de correlações entre valores faz com que não seja adequado para simulações de MC.

Para conseguir preencher os requisitos de uma simulação de MC, é necessário algoritmos com elevada qualidade nos números gerados.

Uma alternativa é o algoritmo *Mersenne twister* (fórmula 3) que oferece um período mais vasto ($2^{19937} - 1 \approx 4 \times 10^{6001}$) e uma uniformidade distribuída em várias dimensões[9]. Este algoritmo, é baseado numa variante do *Lagged Fibonacci Generator* (que por sua vez é uma variante do LCG) e consiste em “torcer” (*twist)*

a matriz A de $b \times b$ bits que está numa forma simples, permitindo assim uma implementação eficiente em termos de operações de deslocamento de bits. Para este algoritmo também foi desenvolvida uma tabela com parâmetros, conseguindo-se assim reproduzir o período máximo sendo a qualidade do números gerado bastante boa. A operação | concatena os bits mais à esquerda de x_{n-N} com os mais à direita de x_{n-N+1}.

$$x_n = (x_{n-N} \,|\, x_{n-N+1})\, A \oplus x_{n-N+M} \tag{3}$$

Outro algoritmo proposto é o *XORShift*[10], fórmula 4, que resulta da operação binária XOR entre uma palavra (*word*) e uma versão de si própria deslocada.

$$x_n = x_{n-1}(I \oplus L^a)\,(I \oplus R^b)\,(I \oplus L^c) \;=:\; x_{n-1}M \tag{4}$$

Representando a *word* de tamanho w como um vetor de bits $x = (x_1, x_2, ..., x_w) \in \{0,1\}^w$, um *shift* à esquerda pode ser representado como uma multiplicação matricial $(x_1, x_2, ..., x_w)L = (x_2, ..., x_w, 0)$ resultando uma matriz esparsa :

$$L = \left\{ \begin{matrix} 0 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 1 & 0 \end{matrix} \right\} \tag{5}$$

O *XORShift* combinando uma palavra de 160 bits com um gerador de Weyl define o gerador *XORWOW* que foi implementado nas bibliotecas de *Compute Unified Device Architecture* (CUDA)*[11].* A álgebra de Weyl é álgebra associativa que não é comutativa, pertence a uma classe importante da álgebra, chamadas de anéis de operadores diferenciais de variedades algébricas.

Um aspeto interessante na geração de números é o facto de esta não se restringir ao uso de um gerador para produzir uma sequência. Uma forma de melhorar a qualidade dos número gerados, ou de esconder as fraquezas de um gerador, consiste em concatenar a geração de números com outros geradores diferentes. Assim, é possível reproduzir, além de uma qualidade mais elevada, períodos mais longos do que os geradores em que são baseados[12].

2.4 Geração de números em paralelo

Com o aumento das necessidades computacionais, a exploração de novas formas de gerar números foi também alargada a novas arquiteturas. Visto que a geração de números aleatórios de forma centralizada, em CPU ou GPU, não é a resposta para as simulações mais exigentes, foram exploradas várias técnicas de paralelismo. Assim, ao invés da geração centralizada, optou-se por atribuir a cada fio de execução (*thread*) uma instância diferente de um PRNG. No entanto, esta abordagem, se for mal configurada, pode gerar muitos acessos à memória, tornando-se esta a principal limitação na geração de números e não a velocidade das unidades aritméticas[13]. Como consequência, os geradores de números em paralelo necessitam de cumprir determinados requisitos específicos, além dos requisitos gerais no ponto 2.3 [14]:

- Tem de ser replicável (10^3 por vezes 10^4).
- De forma a minimizar os acessos à memoria, o estado de cada gerador tem de ser pequeno (em tamanho de bits).
- Garantir que não existe correlação na geração de números nas sub-sequências.

De forma a realizar a geração em paralelo, podem ser aplicadas várias técnicas :

- Dividir uma sequência de números com um período suficientemente grande em n partes.
- Utilizar geradores com períodos muito grandes de forma a que se forem iniciados com sementes diferentes a probabilidade de se sobreporem seja mínima.
- Implementar geradores independentes da mesma classe, com parâmetros diferentes.

Em CPU já existem também vários geradores preparados para a geração em paralelo. O *Dynamic Creation of Mersenne Twisters*[15] (DCMT) consegue reproduzir várias sequências, um por fio de execução, garantindo que a probabilidade de se correlacionarem é muito baixa. Como é baseado no *Mersenne Twister*, a qualidade dos números aleatórios gerados é elevada.

2.5 Paradigmas de paralelização

Atualmente existem vários modelos de programação paralela e várias formas de implementação. O consenso dentro de um modelo de programação é importante pois permite que o software seja transportado entre diferentes arquiteturas.

Há um antigo ditado Chinês que diz “Um artesão, primeiro afia as suas ferramentas antes de começar a sua obra”. Isto é uma verdade no desenvolvimento de software, particularmente na área de aplicações paralelas em que é necessário escolher a ferramenta correta para cada tipo de problema.

A programação em paralelo apresenta várias dificuldades - condições de corrida são as mais comuns e mais difíceis de resolver num programa com vários fios de execução. Uma desvantagem da paralelização é a criação de um acréscimo de computação (*overhead*) devido à sincronização dos fios de execução, outro, o balanceamento de carga que pode fazer com que certos fios de execução obtenham mais trabalho, acabando por ter um impacto severo na performance.

É necessário um estudo profundo, com conhecimento não só técnico como também do tipo de problema a paralelizar de forma a conseguir desenhar o melhor modelo de paralelização. Para este problema, serão usadas técnicas de paralelização em memória partilhada, memória distribuída e um modelo híbrido que tenta agrupar os benefícios de cada uma das abordagens.

2.5.1 Memória partilhada

O modelo de memória partilhada permite que um bloco de memória seja acedido simultaneamente por vários fios de execução com o intuito de providenciar comunicação e/ou evitar cópias redundantes.

Hardware Um exemplo de memória partilhada é mostrada na figura 2.2, onde a RAM é partilhada por ambos os núcleos de processamento.

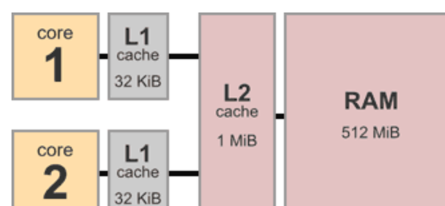


Figura 2.2: Exemplar de um modelo da arquitetura de memória partilhada.

A hierarquia de memória das máquinas e a sua capacidade/rapidez faz com que um programa mal estruturado seja suscetível de falhas na *cache*. Isto é, uma falha

na leitura ou escrita dum bloco resulta num acesso à memória principal que tem uma latência muito superior ($\sim 65ns$- RAM vs $4 \sim 10$ ciclos de *clock* para *cache* L1-L2[16].). Assim, uma boa organização e reutilização de dados é crucial para o funcionamento rápido de programas de simulação.

Nesta tese as máquinas usadas são máquinas NUMA, figura 2.3, estas máquinas têm dois *sockets* para processadores onde um processador pode aceder à memória do outro processador. No entanto, é mais rápido um processador aceder à memória privada do que partilhada, o tempo de acesso depende assim da localização da memória relativa ao processador.

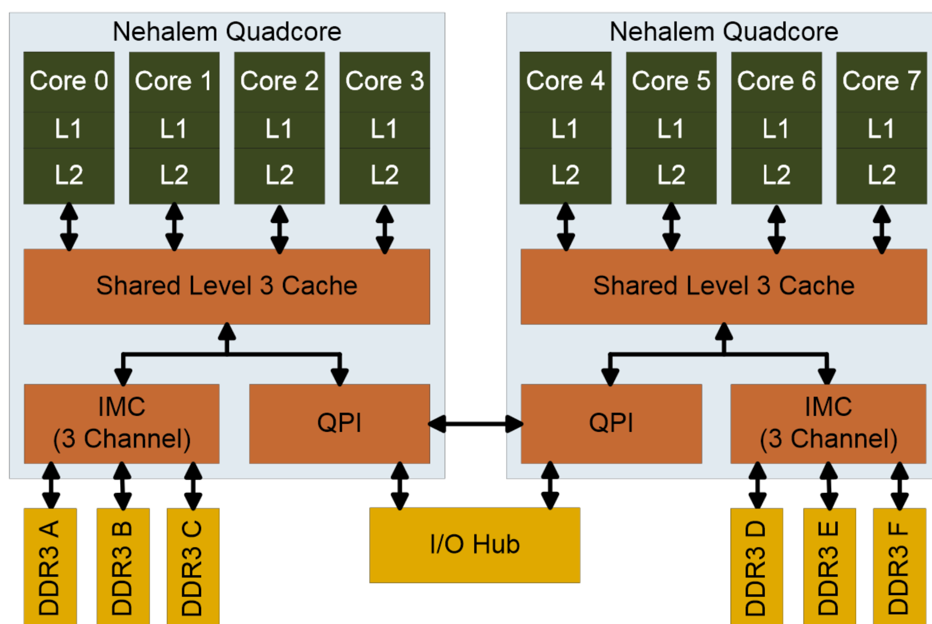


Figura 2.3: Hierarquia de memória de um sistema NUMA com 2 sockets independentes para dois processadores Quadcore.

Software O modelo de memória partilhada é implementado através do OpenMP. Este, apresenta um conjunto de diretivas, rotinas e variáveis de ambiente que permitem ao utilizador especificar o paralelismo em Fortran e C/C++.

O paralelismo pode ser implementado ao nível dos ciclos *for*, dividindo a carga total pelos vários fios de execução disponíveis. Após o trabalho ser calculado existem diretivas para recolher o trabalho feito pelos fios de execução no fio de execução principal.

2.5.2 Memória distribuída

Hardware Um sistema de memória distribuída é composto por várias máquinas, cada máquina contém os seus processadores e memória, figura 2.4. A ligação entre máquinas é feita através de um sistema de comunicação, neste caso *Ethernet*. Assim, é possível a comunicação, partilha de dados e mensagens de comunicação.

Software Para implementar este modelo de paralelização é atribuída ao programador a decisão sobre como acomodar, distribuir e recolher os dados. O acréscimo de comunicação devido à distribuição dos dados por n máquinas é natural de um sistema de memória distribuída. No entanto, é possível "esconder" este acréscimo de comunicação quando o ganho de efetuar n tarefas em paralelo compensa os custos de comunicação.

O *Message Passing Interface* (MPI) é um *standard* de comunicação através da troca de mensagens que é portátil e usado em várias plataformas. Este ambiente fornece ferramentas para C e Fortran, comunicação ponto-a-ponto, operações coletivas, definição de tipos de estruturas, entre outras.

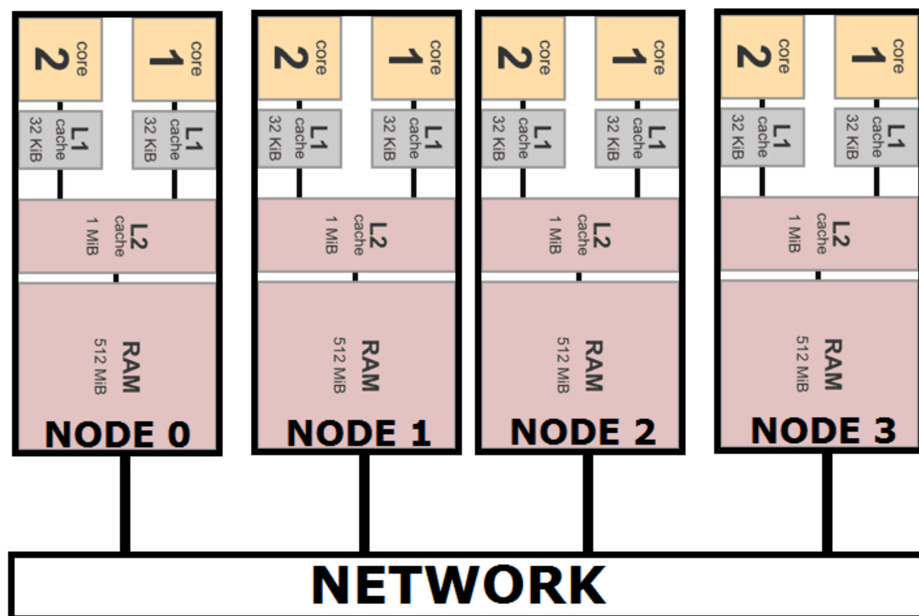


Figura 2.4: Exemplar de um modelo da arquitetura de memória distribuída.

2.5.3 Híbrido

Após efetuar a paralelização em memória partilhada e memória distribuída, é possível juntar as duas técnicas como forma de tentar extrair o máximo de desempenho. Este modelo híbrido pretende explorar ambas as vantagens dos dois modelos : poupança de memória e facilidade de programação do modelo de memória partilhada e a escalabilidade do modelo de memória distribuída.

Cada vez mais o desenvolvimento caminha para máquinas com mais núcleos por processador, o agrupamento desta máquinas em *cluster* permite a exploração deste modelo de paralelismo, paralelizando ao nível das máquinas com MPI e dos núcleos de uma máquina com OpenMP.

2.6 Plataformas de simulação do transporte de fotões em meios túrbidos

Várias plataformas foram desenvolvidas ao longo dos anos, no início dos anos 90 o *Monte Carlo Multi-Layered* (MCML), desenvolvido inicialmente em Pascal por Marleen Keijzer et al era a plataforma mais avançada. Em 1992 foi desenvolvida uma versão em C, por Lihong Wang et al que acabou por ser colocada na Internet de forma a ser desenvolvida pela comunidade de investigadores. O MCML suporta uma simulação *steady-state* de MC em meios com várias camadas. Neste, é possível alterar as propriedades de cada camada, isto é, absorção, difusão, refração etc.,

Mais tarde, em 2008, usando o MCML como base desenvolveu-se o CUDAMCML que resultou duma parceria entre informáticos e físicos. Este, resolve os mesmos problemas do MCML no entanto, aproveita a existência de um GPU para resolver o transporte de fotões em paralelo. No entanto, existem algumas limitações do CUDAMCML [17]:

- Número máximo de fotões é $2^{32} - 1$.
- A primeira camada não pode ser de vidro.
- No máximo podem existir 100 camadas diferentes.
- Apenas suporta geração de números a 32 bits.
- Suporta apenas 1 GPU.

Em 2009, foi desenvolvido o GPU CLUSTER MCML (GCMCML) que suportando agora vários GPU's em rede obtém performances superiores (tabela 2.1) mas sofre das mesmas limitações[18].

Plataforma	Nº de fotões	Tempo de exec. (s)
MCML (CPU)	10^7	396
GCMCML (1 Nó)	10^7	4,4
GCMCML (4 Nós)	10^7	1,1

Tabela 2.1: Os tempos de execução das diferentes versões do MCML demonstram que o GCMCML é o mais rápido. A GPU usada foi uma 9800 GT (112 CUDA CORES) e o CPU foi o Intel Core2Duo E7300 (2 fios de execução em *Hyper-Threading*) @ 2.67 GHz.

O Zemax, fundado em 1988, que recentemente fundiu-se com Radiant criando o Radiant Zemax, é um programa de design ótico utilizado para desenhar e analisar sistemas como lentes de câmaras e sistemas de iluminação através de *ray-tracing*. A versão 64 bits está otimizada para funcionar até 64 processos em paralelo[19].

Capítulo III

Desenvolvimento da solução

O capítulo 3 visa focar o desenvolvimento desde a fase inicial às propriedades físicas implementadas como também a paralelização.

3.1 Sistema físico a modelar

Para esta tese de mestrado, o problema a resolver foi a homogeneização da saída de luz num ecrã retro-iluminado, figura 3.1, com 20 cm de comprimento 10 cm de altura e 4 cm de espessura. Este dispositivo é usado na consola central de automóveis fornecendo diversas informações. É necessário otimizar a saída de luz de forma a tornar a visualização fácil mesmo em situações noturnas, e também como forma de diminuição do consumo energético.

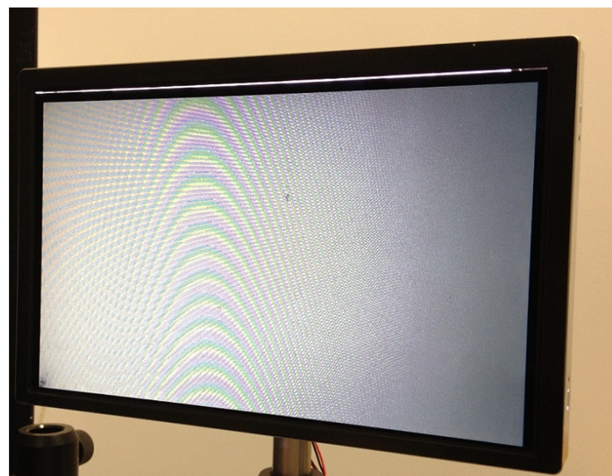


Figura 3.1: Ecrã iluminado por 64 LEDs (32 no topo e 32 na parte inferior) .

O ecrã é constituído por um guia de luz e várias camadas de plástico fino que influenciam a trajetória dos fotões à saída do guia de luz (*backlight*). Estas camadas alteram a direção dos fotões, fazendo com que a luz saia numa determinada direção em maior quantidade que outras. O guia de luz é a *interface* entre os *Light-Emitting Diodes* (LEDs) e o utilizador, sendo o alvo do software de simulação desenvolvido nesta dissertação. A resolução do guia de luz foi medida no modelo real e contém 1752 por 1036 pixeis.

O guia de luz possui várias características para modelar o fluxo de luz :

- Nas zonas próximas aos LEDs existem saliências que fazem com que a luz de cada LED seja espalhada mais no plano horizontal do que no plano vertical.

- A face de topo contém rugosidades microscópicas que alteram a trajetória dos fotões, facilitando a transmissão de fotões para exterior quando encontra uma rugosidade.

- O guia de luz está embutido numa moldura revestida de branco, de forma a minimizar desperdícios energéticos, fazendo com que todos os fotões que saiam pelas faces laterais e inferior, sejam refletidos novamente para dentro do guia, figura 3.2 e 3.3.

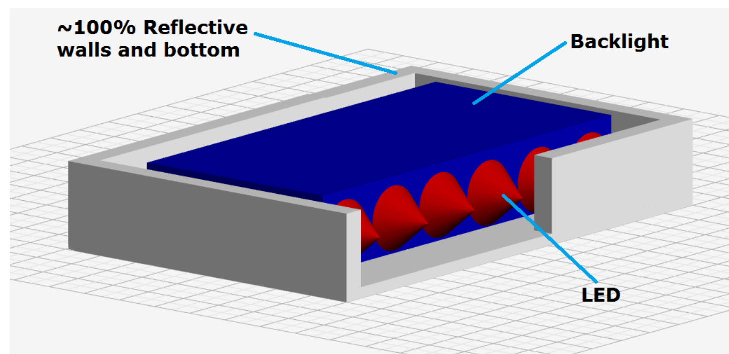


Figura 3.2: Modelo 3D (em corte) do guia de luz inserido no invólucro. .

Figura 3.3: Vista frontal do modelo real (os focos luminosos laterais são os LEDs).

3.1.1 Descrição do software

O software pretende simular o transporte de fotões no guia descrito e apresentar os resultados de forma resumida no final da simulação.

A execução sequencial da simulação, figura 3.4, inicia por ler do ficheiro de entrada as dimensões do guia, o número de LEDs e outros parâmetros de entrada que são necessários para o pré-processamento. O pré-processamento prepara as estruturas de dados de entrada e de saída para o cálculo tais como a posição de cada LED e o mapa de saída. O gerador de números é iniciado com a semente e de seguida é iniciado o ciclo de cálculo para os fotões onde é calculada a trajetória de cada fotão sendo registado no mapa de saída a posição final. No final são calculados os dados estatísticos da simulação e apresentados na consola de saída, algoritmo 1.

De forma a apresentar os vários elementos do software serão explicadas de seguida as etapas cruciais do desenvolvimento.

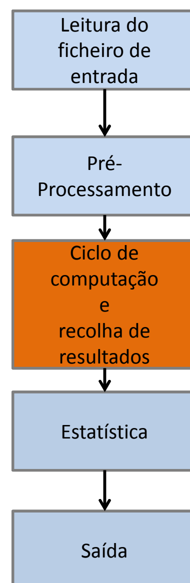


Figura 3.4: Fluxograma da execução - a azul parte sequencial - laranja parte que foi paralelizada.

Algoritmo 1 Pseudo-código do primeira versão sequencial com o ciclo a iterar nos LEDs

```
int main(void){
  //Estruturas e alocação de memória
  //Inicialização do mapa de saída
  //Inicialização gerador de números
  for (i = 0; i < Total_LEDs; i++) {
    calc_LED(COut, LED[i]);
  }
  //Cálculo de estatística
  //Apresentação resultados
}
```

3.2 Implementação base

Este trabalho baseou-se num estudo prévio feito pelo Professor Eduardo Pereira que desenvolveu, em Fortran, um esboço do programa que simula o comportamento dos fotões. Este estudo foi crucial pois forneceu uma base cientificamente correta para o desenvolvimento em C deste software. Optou-se pela linguagem C, para facilitar a implementação do software. De forma cronológica, este trabalho foi construído da seguinte forma :

1. Transporte para C da versão em Fortran e validação dos resultados.
2. Decisão sobre o gerador de números aleatórios e paralelização em OpenMP.
3. Melhoramento da qualidade da simulação e versão inicial MPI.
4. Otimização de ambas as versões (OpenMP e MPI) e de parâmetros.
5. Refinamento das versões e recolha de testes(Sequencial - OpenMP - MPI).

Esta estrutura foi o seguimento natural do desenvolvimento, representando apenas a ordem cronológica dos eventos. No entanto, algumas decisões críticas tiveram de ser tomadas no desenvolvimento do projeto que serão explicadas de seguida.

3.2.1 Transporte para C e validação

O código já existente em Fortran permitiu o transporte de forma modular. Cada função foi implementada de forma otimizada, e testada. Assim, foi possível validar cada etapa do desenvolvimento de forma sistemática.

Após o software estar concluído foi testado com o mesmo gerador de números aleatórios e semente, para garantir que os resultados finais coincidiam.

3.2.2 Gerador de números aleatórios

O gerador de números aleatórios é um componente crítico da simulação, porém, como será explicado no capítulo 4, a execução não está limitada pela geração de números aleatórios. Assim, a decisão sobre qual o gerador a usar deixa de ser centrada no desempenho e mais na sua adequabilidade ao problema. Visto o propósito desta tese ser a paralelização, um gerador já preparado para produzir sequências de números independentes em processos/máquinas distintos(as) seria a ferramenta perfeita.

O gerador usado foi uma variação do Mersenne Twister, o DCMT, implementado por Makoto Matrsumoto e Takuji Nichimura. Resumidamente, este Mersenne Twister permite que o utilizador insira o tamanho da palavra (w) e do expoente do período (p) desejado ($2^p - 1$).

Reproduzir uma sequência de números aleatórios de período $2^p - 1$ com w bits é computacionalmente pesado para valores de p grandes, reduzir esta tarefa garantindo que nunca se repetem os números, foi uma componente importante.

Do estudo feito à simulação concluiu-se que 93% dos fotões que entram no guia de luz, em média, ao fim de 5 iterações saem do guia. Após a análise das funções óticas, conlcui-se que no pior dos casos um fotão necessita de 7 números aleatórios por iteração, assim, é possível concluir que para 10^{50} fotões:

$$2^p - 1 \geq 7 \times 5 \times 10^{50} \Leftrightarrow p \geq log_2\left(7 \times 5 \times 10^{50} - 1\right) \Leftrightarrow p \geq 171.2256 \tag{6}$$

Infelizmente o DCMT não permite especificar o valor de p, fornece uma lista de expoentes a usar que variam de 521 até 44497. Para esta simulação é claro que o limite inferior (521) é mais do que suficiente para satisfazer as necessidades mais rígidas dos utilizadores (10^{50} fotões). Mesmo para um p pequeno, 521, o calculo de inicialização da sequência pode demorar algum tempo, aumentando o p, o tempo de procura aumenta respetivamente, sendo que para um p grande é esperado que a complexidade seja de $O(p^3)$. A palavra (w) usada foi 32, visto que atualmente o DCMT apenas permite a utilização de 31 ou 32.

A vantagem principal do DCMT é permitir a criação de várias sub-sequências independentes. Matematicamente falando, o polinómio característico da recursão são coprimos entre si. Para garantir que as sub-sequências são independentes, quando é especificado p e w, acrescenta-se o *id* que especifica o número do fio de execução ou processo, a cada *id* estará associada uma sub-sequência. Assim, apenas é necessário uma semente para vários fios de execução visto que está garantido que as sub-sequências geradas são independentes.

3.2.3 Melhoramento da qualidade da simulação

Tal como foi mencionado, este trabalho é resultante de duas frentes de investigação. Do ponto de vista físico, a redução do tempo total de execução através de exploração de paralelismo, é uma forma de obter resultados mais rapidamente, mas também uma forma de obter mais e melhor resultados na mesma janela temporal. Com o

software validado, a simulação pode ser usada no acerto das melhores características do guia de luz.

Nas primeiras versões, por exemplo, era permitido que de acordo com uma probabilidade, os fotões saíssem do guia de luz pelas faces laterais. No entanto, após uma análise ao modelo real isto não se verifica. Outra característica que inicialmente não foi implementada, foram as rugosidades existentes nas faces de topo e base. Estas rugosidades, figura 3.5, representam uma característica muito importante para a simulação, alterando drasticamente a distribuição de saída dos fotões. Estas rugosidades são modelas por três parâmetros, um mínimo e um máximo de probabilidade (p_min e p_max) de ocorrência, variando linearmente com a distância ao centro do guia, e ainda uma probabilidade de, um fotão ao encontrar uma rugosidade a conseguir atravessar (p_out). Ainda assim, numa análise ao modelo real, concluiu-se que esta aproximação não é a mais realista, visto que as rugosidades não variam linearmente, mas sim de acordo com uma distribuição ainda não conhecida. O estudo do tipo de distribuição que as rugosidades apresentam no modelo real será feito futuramente.

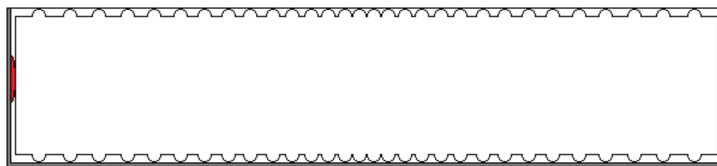


Figura 3.5: Distribuição das rugosidades nas faces de topo e inferior (LEDs a vermelho)

3.2.4 Otimização de parâmetros

Um dos pontos cruciais da simulação são os parâmetros de entrada. É possível alterar várias características do guia de luz, que consequentemente refletem-se no comportamento dos fotões no tempo de execução. Assim, foi necessário definir um conjunto de dados que reunissem as características adequadas para atingir a solução do problema (homogeneidade na face de saída, figura 3.6) e que não fossem fisicamente impossível de reproduzir[4].

Os parâmetros usados estão na tabela 3.1.

[4] Por exemplo, ajustar uma probabilidade a 0,00001 ou 0,999999.

Parâmetros fixos	Dimensões x, y e z (cm)	200 100 4
	Dimensões grelha de saída (pixeis)	1752 1036
	Nº de LEDs por face (4 faces)	0 32 0 32
Parâmetros ajustáveis	Rugosidades p_min e p_pmax	0.15 0.8
	Rugosidades p_out	0.15
	Nº de fotões	10^{10}

Tabela 3.1: Tabela com os parâmetros de entrada otimizados. Conjunto de valores $\{V1\,V2\,V3\}$.

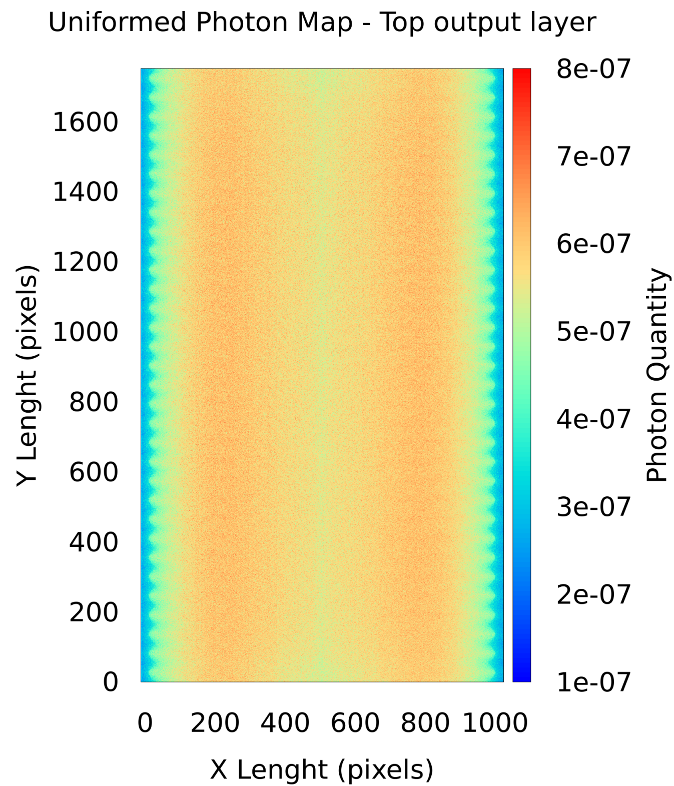


Figura 3.6: Saída do guia de luz (face de topo).

3.2.5 Propriedades óticas implementadas

De forma a modelar este sistema, é necessário implementar um número de propriedades físicas. No entanto, não é possível, num período de uma tese, implementar toda a complexidade do sistema.

Neste modelo foram implementadas duas propriedades óticas, a difusão *Lambertiana* e a reflexão especular através dos coeficientes de *Fresnel*.

Difusão Lambertiana Uma superfície *Lambertiana* emite e reflete a mesma luminância em todas as direções (isotrópico[5]). Por exemplo, um folha de papel branco igualmente iluminada é aproximadamente Lambertiana. No entanto, não tem intensidade isotrópica.

A figura 3.7 ilustra o princípio Lambertiano, a quantidade de energia refletida numa direção em particular é proporcional ao cosseno do ângulo refletido. O cosseno é medido relativamente à normal da superfície.

Neste modelo a difusão Lambertiana é usada para a transmissão e reflexão. Existem três situações em que é usado, figura 3.8 :

- Quando um fotão é disparado por um LED (Lambert_LED),
- Quando embate numa rugosidade e é refletido para dentro do guia (Lambert_Side),
- Quando sai do guia por uma das laterais e volta a entrar (Lambert_ReEntrance).

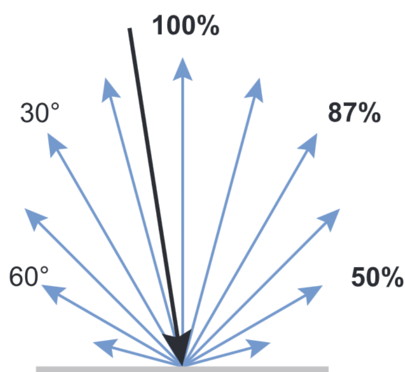


Figura 3.7: Reflexão Lambertiana de uma superfície.

[5]Qualquer meio em que a luz atua igualmente em todas as direções.

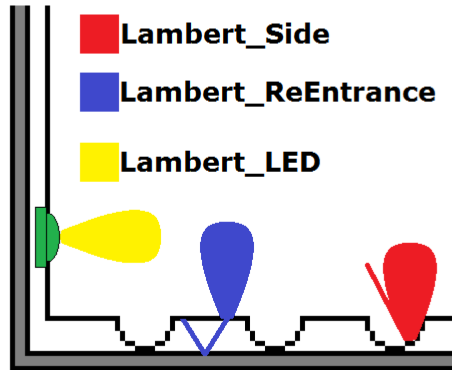


Figura 3.8: Casos em que acontece a reflexão Lambertiana no guia de luz.

Reflexão especular de Fresnel A reflexão de Fresnel ocorre na presença de dois meios diferentes por exemplo, quando um fotão colide com uma das paredes laterais ou nas de topo e inferior (quando não encontra uma rugosidade), figura 3.9.

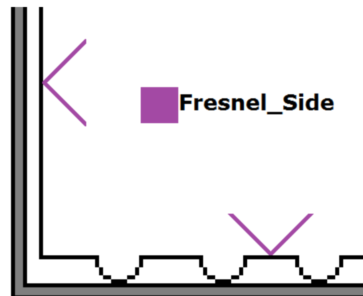


Figura 3.9: Reflexão Fresnel de uma superfície.

Uma substância transparente transmite quase toda a luz, mas reflete uma quantidade muito pequena em cada uma das duas superfícies. Esta reflexão acontece quando há uma alteração no índice de refração. Numa incidência normal (ângulo de incidência = 0º), a lei de Fresnel quantifica o efeito de acordo com a equação 7, onde r_λ é a reflexão perdida, n_1 o índice refrativo do meio 1 e n_2 o índice refrativo do meio 2[20].

$$r_\lambda = \frac{(n_2 - n_1)^2}{(n_2 + n_1)^2} \tag{7}$$

3.2.6 Critérios de qualidade

De forma a determinar se o resultado final é aceitável, foi necessário implementar duas medidas de classificação quantitativa, a uniformidade e o *Black MURA*. Para efetuar a classificação corretamente, é apenas tido em consideração a área visível ao utilizador, isto é, despreza-se uma ligeira margem nas bordas do ecrã , figura 3.10, correspondente a 10 pixeis.

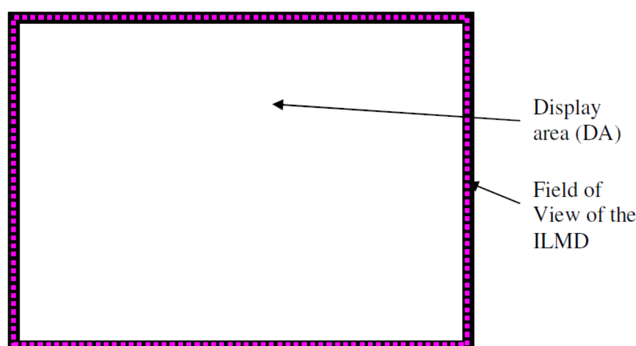


Figura 3.10: DA - Área do *display* considerada para medir a qualidade.

Uniformidade A uniformidade, fórmula 8, é calculada usando um método de varrimento que simula o método convencional de medição de luminância apenas num ponto (neste caso um retângulo designado por "*Box*") que é deslocado em iterações de um pixel através da área do display DA.

$$u = \frac{min(Y_{Box}(i,j))}{max(Y_{Box}(k,l))}\ (i,j),(k,l) \in (DA) \tag{8}$$

Black MURA Para a deteção das áreas *black MURA*[6] é calculado a magnitude do gradiente da luminância, que é a primeira derivada da luminância relativa a cada pixel. Resumidamente, para cada pixel, são calculadas as componentes x e y do gradiente (Y_{ax} e Y_{ay}) e calculado o módulo do gradiente Y_a, fórmulas 9, 10 e 11 (para uma janela de 10 pixeis).

$$Y_{ax}(i,j) = \frac{1}{110}\sum_{l=-10}^{l=10} l \times Y(i, j+l) \tag{9}$$

$$Y_{ay}(i,j) = \frac{1}{110}\sum_{l=-10}^{l=10} l \times Y(i+l, j) \tag{10}$$

[6] *Mura* vem do Japonês que significa desigualdade, não uniforme, irregularidade.

$$Y_a(i,j) = \sqrt{Y_{ax}^2(i,j) + Y_{ax}^2(i,j)} \quad (11)$$

O *MURA* é mais crítico em situações noturnas devido à ausência de reflexões, assim, calcula-se o valor relativo à luminância média, equação 12.

$$Y_{a,rel}(i,j) = \frac{Y_a(i,j)}{\bar{L}w} \quad (12)$$

Todos os pixeis da imagem são testados para calcular o valor máximo do gradiente da luminância. Aqui, é necessário ter em conta a "*Box*" mencionada em 8.

$$Y_{a,rel,max} = max\left(\frac{Y_{a,Box}(i,j)}{\bar{L}w}\right) \quad (13)$$

O valor $Y_{a,rel,max}$ é o valor final do MURA que tem de ser classificado.

O cálculo de ambas as medidas está especificado no manual *Uniformity measurement standard for Displays*[21].

3.3 Abordagem de paralelização

A execução da simulação contém blocos que podem ser paralelizáveis e outros que são exclusivamente sequenciais. A natureza do transporte de fotões mostra que cada trajetória é independente, assim, é possível calcular as trajetórias de vários fotões em paralelo.

3.3.1 Memória partilhada

Para paralelizar o ciclo de computação em memória partilhada, cada fio de execução terá associado um número de fotões para calcular. Cada fio de execução trabalhará no seu bloco de memória privada, acedendo apenas ao bloco partilhado inicialmente, para copiar as constantes auxiliares ao cálculo. Estando garantido que as variáveis e estruturas de dados são mantidas em dados locais ao fio de execução, o código torna-se *"amigo da cache"*[7]. Ao estruturar a simulação desta forma garante-se que os acessos à memória partilhada são reduzidos.

Existem dois mapas de saída, COut e SOut. COut representa o mapa local do fio de execução, enquanto que SOut é utilizado para agrupar a informação de todos os COuts locais.

[7] Código que é pequeno o suficiente para ser armazenado nos níveis de *cache* mais próximos do processador (ex. L1 ou L2)

Após todos os fotões serem calculados, todos os fios de execução efetuam uma redução. A redução é feita de forma mutuamente exclusiva, obrigando a que num dado instante, apenas um fio de execução adicione a SOut os seus resultados.

Paralelização ao nível dos LEDs A paralelização com o ciclo de computação a iterar nos LEDs foi a primeira abordagem, algoritmo 2. Após a entrada na região paralela (*#pragma omp parallel*) as variáveis e estruturas locais são criadas, o gerador é iniciado com o identificador do fio de execução.

O número total de fotões a calcular é dividido pelo número total de LEDs. Assim, cada fio de execução calcula todas as trajetórias ao LED correspondente e acumula o resultado em COut. No final, esses valores são adicionados à estrutura partilhada SOut de forma exclusiva.

Algoritmo 2 Pseudo-código do ciclo computacional *for* da versão de memória partilhada com a paralelização nos LEDs.

```
int main(void){
  //Leitura do ficheiro de entrada
  //Estruturas e alocação de memória globais
  //Inicialização do mapa de saída SOut
  #pragma omp parallel {
    //Variáveis privadas
    //Estruturas e alocação de memória privadas
    //Inicialização do mapa de saída COut
    //Iniciação do gerador de números aleatórios (semente,id)
    #pragma omp for
    for (i = 0; i < Total_LEDs; i++) { //Iterador sobre os LEDs
      calc_LED(COut, LED[i] ); //Calculo de todos os fotões para o
LED
    }
    //Redução dos resultados
    #pragma omp critical
    reduceHits(SOut,COut);
  }
  //Cálculo de estatística
  //Apresentação resultados
}
```

Paralelização ao nível dos Fotões Alterando o ciclo de computação para iterar sobre o número total de fotões alarga-se a possibilidade de executar tarefas em paralelo, visto existirem sempre mais fotões (10^9a 10^{12}) do que LEDs (64). A estrutura mantém-se semelhante, criando apenas computação adicional para associar cada fotão a um LED (*whereIsPhoton()*), algoritmo 3.

As alterações necessárias para esta versão foi a mudança no ciclo *for* para o número total de fotões e o cálculo de associar um fotão i a um dos LEDs.

Algoritmo 3 Pseudo-código da parte paralelizada da versão de memória partilhada com a paralelização nos fotões.

```
int main(void){
  //Leitura do ficheiro de entrada
  //Estruturas e alocação de memória globais
  //Inicialização do mapa de saída SOut
  #pragma omp parallel {
    //Variáveis privadas
    //Estruturas e alocação de memória privadas
    //Inicialização do mapa de saída COut
    //Iniciação do gerador de números aleatórios (semente,id)
    //Calcular os limites de cada LED
    #pragma omp for
    for (i = 1; i <= Total_Fotões; i++) { //Iterador nos fotões
      whereIsPhoton(LED,limites,i); //Localiza o fotão i ao LED
para mapeamento
      calc_Fotão(COut, LED); //Calcula o fotão i injetado pelo LED
    }
    //Redução dos resultados
    #pragma omp critical
    reduceHits(SOut,COut);
  }
  //Cálculo de estatística
  //Apresentação resultados
}
```

Paralelização ao nível dos LEDs & Fotões Como forma de tornar o código mais legível, foi abordada a paralelização através do encadeamento de ciclos. Um ciclo exterior para iterar nos LEDs e um interior para iterar nos fotões de cada LED, algoritmo 4. A paralelização foi alcançada através da diretiva *collapse*, que transforma os dois ciclos num só ciclo.

Algoritmo 4 Pseudo-código da parte paralelizada da versão de memória partilhada com ciclos encadeados

```
int main(void){
  //Leitura do ficheiro de entrada
  //Estruturas e alocação de memória globais
  //Inicialização do mapa de saída SOut
  #pragma omp parallel {
    //Variáveis privadas
    //Estruturas e alocação de memória privadas
    //Inicialização do mapa de saída COut
    //Iniciação do gerador de números aleatórios (semente,id)
    //Calcular os limites de cada LED
    #pragma omp for collapse(2)
    for(j = 0; j < Total_LEDs ; j++) { //Iterador nos LEDs
      for (i = 1; i < Fotões_p_LED; i++) { //Iterador nos Fotões
relativo ao LED[j]
        calc_Fotão(COut, LED);
      }
      LED = LED[j+1]; //Atualiza a estrutura LED para o proximo LED
    }
    //Redução dos resultados
    #pragma omp critical
    reduceHits(SOut,COut);
  }
  //Cálculo de estatística
  //Apresentação resultados
}
```

3.3.2 Memória distribuída

A paralelização em memória distribuída pode ser implementada de várias formas, com um balanceamento dinâmico ou estático.

O balanceamento estático divide inicialmente a carga de trabalho total por todas as máquina disponíveis, ideal para quando as máquinas executam a tarefa no mesmo espaço de tempo. No entanto, se existirem máquinas diferentes ou alguma aleatoriedade, o tempo de execução total é limitado pelas máquinas que demoram mais tempo.

O balanceamento dinâmico requer uma implementação mais complexa. No entanto, é mais robusto quanto à heterogeneidade das máquinas e ao balanceamento de carga. Um processo (mestre) responsabiliza-se apenas pela distribuição de carga em tempo real e os "escravos" por executarem a tarefa computacional, figura 3.11 e algoritmo 5.

Por cada tarefa que é calculada é necessária comunicação. No entanto, devido tratar-se de fotões, a mensagem transporta apenas os limites de cada tarefa, dois *longs*. O tamanho da tarefa está associada a um rácio entre o número de processos escravos e o número de fotões a calcular (Granularidade), fórmula 14. Quando um processo escravo terminar de calcular os fotões que lhe foram atribuídos, comunica com o processo mestre que, se tiver mais fotões para calcular envia uma mensagem com os próximos dados, caso contrário envia uma mensagem de paragem (sinalFim = 1 - Paragem).

$$Tamanho\, da\, tarefa = \frac{N^{o}\, total\, de\, fotões}{(N^{o}\, de\, processos - 1) * Granularidade} \tag{14}$$

O compromisso existente entre o tamanho da tarefa e a quantidade de comunicação é critico para o desempenho em memória distribuída. Se as tarefas forem de grandes dimensões a comunicação é mínima mas poderá surgir um problema de balanceamento de carga. Pelo contrário, se as tarefas forem pequenas é garantido um bom balanceamento com comunicação adicional por tarefa. Assim, o mesmo trabalho pode ser dividido em muitas tarefas pequenas ou em poucas tarefas grandes.

Para perceber o impacto que o tamanho da tarefa tem no desempenho, foram testados vários valores para a granularidade, entre $2^0 \leq Granularidade \leq 2^{11}$, em potências de 2.

A abordagem implementada foi o balanceamento dinâmico com base na versão de memória partilhada com a paralelização nos fotões.

Algoritmo 5 Pseudo-código do código do balanceamento dinâmico para os processos Mestre e Escravos

```
int main(void){
  //Inicialização MPI
  //Estruturas e alocação de memória globais
  //Inicialização do mapa de saída SOut
  if (processoMestre) { //Se for o processo Mestre (0)
    //Estruturas e alocação de memória - mestre
    //Ajuste da granularidade
    while (houverFotões) { //Enquanto houver fotões para calcular
      MPI_Recv(processoEnviar); //Recebe o id do processo a enviar
      prepMPI(MPI_Struct); //Prepara a estrutura de envio
      MPI_Send(MPI_Struct,processoEnviar,sinalFim); //Envia com o
sinalFim a (0) ou (1)
    }
  }
  else { //Para os processos escravos
    //Estruturas e alocação de memória - escravos
    //Calculo limites LEDs
    //Inicialização gerador de números (semente , id processo)
    while(sinalFim == 0) { // Se o sinal de paragem estiver a 0
      MPI_Send(nºProcesso); //Envia o id para receber a tarefa
      MPI_Recv(MPI_Struct); //Recebe a tarefa
      for ( i = MPI_Struct.limits[0] ; i < MPI_Struct.limits[1] ;
i++) {
        whereIsPhoton(LED,limites,i);
        calc_Fotão(COut, LED);
      }
    }
  }
  MPI_Reduce(SOut,COut); //Reduz todos os resultados de todos os
processo escravos para o SOut do processo mestre
  if (processoMestre) {
    //Cálculo de estatística
    //Apresentação resultados
  }
}
```

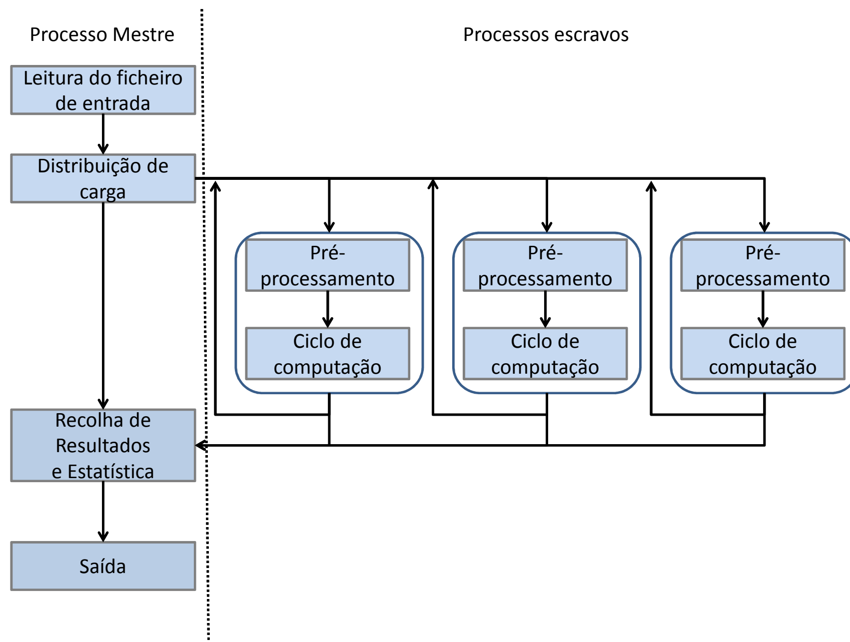


Figura 3.11: Fluxograma da execução em memória distribuída

3.3.3 Híbrido

O modelo híbrido pretende alcançar a escalabilidade da memória distribuída e a eficiência da memória partilhada. O modelo assemelha-se ao de memória distribuída, acrescentando, no ciclo de cálculo das trajetórias, a paralelização OpenMP do modelo de memória partilhada, figura 3.12.

Este modelo necessita de estruturas auxiliares adicionais para a nova região paralela. Assim, é criada uma nova estrutura local (ao fio de execução) COutOMP que guarda os resultados do cálculo em memória partilhada. No final do cálculo de tarefa é necessário agrupar os resultados do todos os fios de execução na estrutura COut. É a estrutura COut que é reduzida para o processo mestre, que guardará os resultados finais em SOut.

Para garantir que, de tarefa em tarefa, os fios de execução não reiniciam o gerador com a mesma semente, é aplicado o *Middle square method* à semente inicial após o cálculo de cada tarefa.

Algoritmo 6 Pseudo-código do código do modelo de Híbrido com o ciclo *for* paralelizado.

```
int main(void){
  //Inicialização MPI
  //Estruturas e alocação de memória globais
  //Inicialização do mapa de saída SOut
  if (processoMestre) { //Se for o processo Mestre (0)
    //Estruturas e alocação de memória - mestre
    //Ajuste da granularidade
    while (houverFotões) { //Enquanto houver fotões para calcular
      MPI_Recv(processoEnviar); //Recebe o id do processo a enviar
      prepMPI(MPI_Struct); //Prepara a estrutura de envio
      MPI_Send(MPI_Struct,processoEnviar,sinalFim); //Envia com o
sinalFim a (0) ou (1)
    }
  }
  else { //Para os processos escravos
    //Estruturas e alocação de memória - escravos
    //Calculo limites LEDs
    while(sinalFim == 0) { // Se o sinal de paragem estiver a 0
      MPI_Send(nºProcesso); //Envia o id para receber a tarefa
      MPI_Recv(MPI_Struct); //Recebe a tarefa
      #pragma omp parallel { // Parte de memória partilhada
        //Inicialização de variáveis locais
        //Inicialização gerador de números (semente , id do fio de
execução)
        #pragma omp for
          for ( i = MPI_Struct.limits[0] ; i < MPI_Struct.limits[1]
; i++) {
          whereIsPhoton(LED,limites,i);
          calc_Fotão(COutOMP, LED);
        }
        //Redução dos resultados
        #pragma omp critical
        reduceHits(COut,COutOMP); //Reduz os resultados dos fios de
execução para COut.
      }
      semente = semente * semente; //Reduz a probabilidade de
repetição de números.
    }
  }
  MPI_Reduce(SOut,COut); //Reduz todos os resultados de todos os
processo escravos para o SOut do processo mestre
  if (processoMestre) {
    //Cálculo de estatística
    //Apresentação resultados
  }
}
```

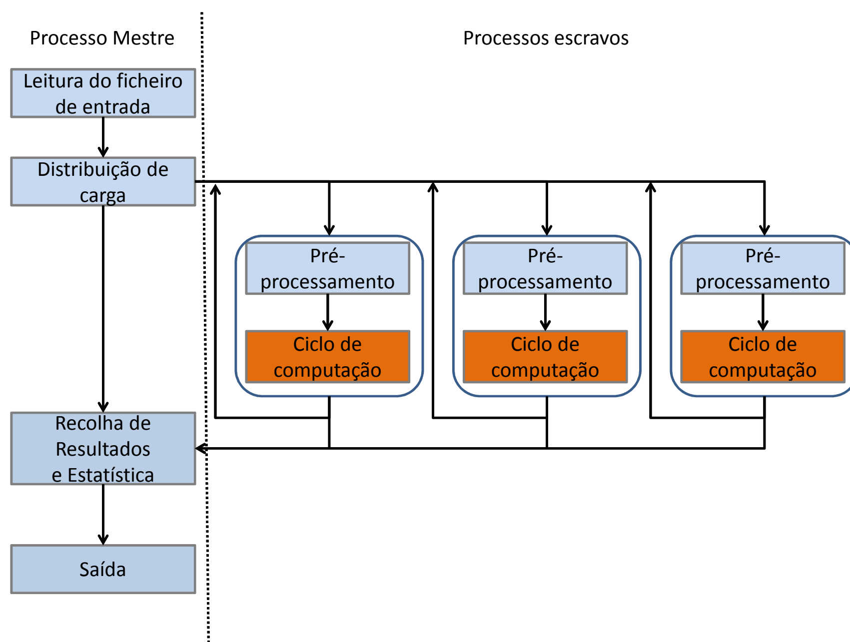


Figura 3.12: Fluxograma da execução híbrida - a laranja a paralelização em memória partilhada.

Capítulo IV

Resultados e discussão

Neste capítulo serão apresentados os resultados e conclusões de cada etapa da otimização e paralelização. Para o desenvolvimento de software em que existe a necessidade de recolher medidas exatas e com o mínimo de ruído, é necessário recorrer a máquinas que executem apenas os processos essenciais do sistema. Assim, foi usado como plataforma de trabalho o *cluster* SEARCH da Universidade do Minho, onde estão instaladas vários tipos de máquinas e arquiteturas. Contudo, de forma a manter coerência nos testes e resultados, apenas foram usados os nós de computação 601 que contêm:

- Dois *Intel Xeon X5650* com 12 núcleos por nó
- 12 GB Memória RAM
- 2.66GHz
- Compilador Intel *icc* versão 11.0 (com flags -O3)

4.1 Validação de resultados

Esta simulação ao pretender homogeneizar a saída dos fotões deverá conseguir produzir resultados que cumpram padrões de qualidade, nomeadamente a uniformidade e o *black MURA*. Assim, foram feitos vários testes e analisados os resultados de forma a garantir que a simulação permanece dentro dos padrões utilizados por um fabricante de LCD's, tabela 4.1.

Fotões simulados	10^{10}
Uniformidade	79%
Black Mura	$0,018$

Tabela 4.1: Resultados de saída da simulação com os parâmetros definido em 3.2.4
.

4.2 Otimização do desempenho da versão base

Após a conclusão da versão inicial (com a paralelização nos LEDs), foi feita uma análise ao perfil de execução do software para visualizar o tempo consumido por cada função. Da figura 4.1, é percetível que, mais de 30% do tempo total de execução é consumido em cálculos trigonométricos (*sin*, *cos*, *acos* e *asin*).

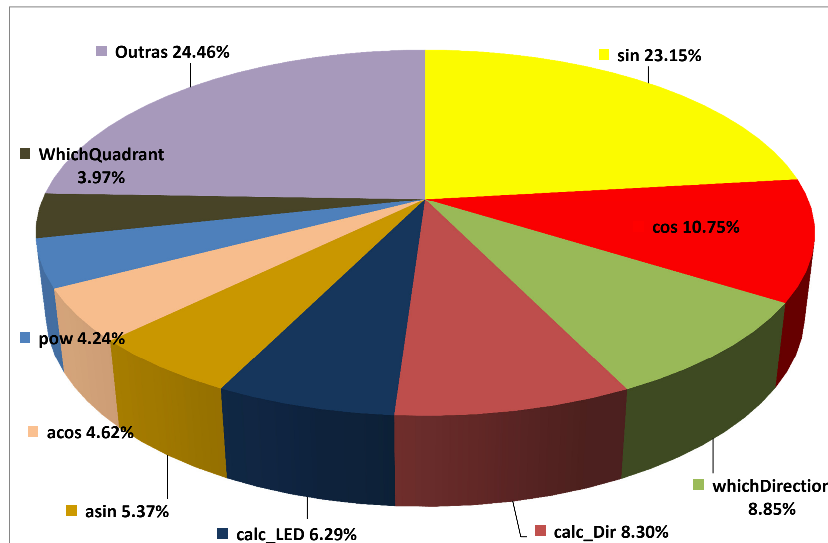


Figura 4.1: Perfil de execução da versão com funções trigonométricas.

O código é composto por 5 funções que representam o movimento físico dos fotões. Cada uma contém várias funções trigonométricas, por exemplo, a função Lambert_LED_RE_Entrance contém 7, algoritmo 7. Cada chamada destas funções necessita de, aproximadamente, 100 ciclos de relógio para ser executada[22].

Assim, foi feita uma reestruturação do código para remover as funções trigonométricas. Para o caso de Lambert_LED_Re_Entrance, esta otimização resultou numa redução de 4 vezes o número de chamadas trigonométricas, algoritmo 8. Com esta otimização, é de esperar um tempo de execução mais curto visto serem calculadas menos vezes as funções que consomem mais tempo.

Algoritmo 7 Parte da função Lambert_LED_Re_Entrance com funções trigonométricas.

```
void lambert_LED_Re_Entrance (...){
  theta_p = asin(sgendrand_mt(mt));
  phi_p = TWOPI * sgendrand_mt(mt);
  theta_f = theta_p;
  thetaL_f = asin(n_out_in * sin(theta_f));
  theta_fs = theta_f + thetaL_f;
  theta_fd = theta_f - thetaL_f;
  theta_p = thetaL_f;
  x_ref = sin(theta_p) * cos(phi_p);
  y_ref = sin(theta_p) * sin(phi_p);
  z_ref = cos(theta_p);
}
```

Algoritmo 8 Parte da função Lambert_LED_Re_Entrance sem funções trigonométricas.

```
void lambert_LED_Re_Entrance (...){
  sin_theta_p = sgendrand_mt(mt);
  phi_p = TWOPI * sgendrand_mt(mt);
  sin_phi_p = sin(phi_p);
  cos_phi_p = cos(phi_p);
  sin_theta_f = sin_theta_p;
  cos_theta_f = sqrt(1.0 - pow(sin_theta_f, 2));
  sin_thetaL_f = n_out_in * sin_theta_f;
  cos_thetaL_f = sqrt(1.0 - pow(sin_thetaL_f, 2));
  sin_theta_p = sin_thetaL_f;
  cos_theta_p = cos_thetaL_f;
  x_ref = sin_theta_p * cos_phi_p;
  y_ref = sin_theta_p * sin_phi_p;
  z_ref = cos_theta_p;
}
```

Para comparação foi medido o perfil de execução da versão sem funções trigonométricas figura 4.2, onde é visível que esta otimização reduziu o consumo da função *sin* em 20%, sendo agora a função *pow* responsável por 25% do tempo total de execução. A função *pow* leva cerca de 40 ciclos de relógio para ser executada.

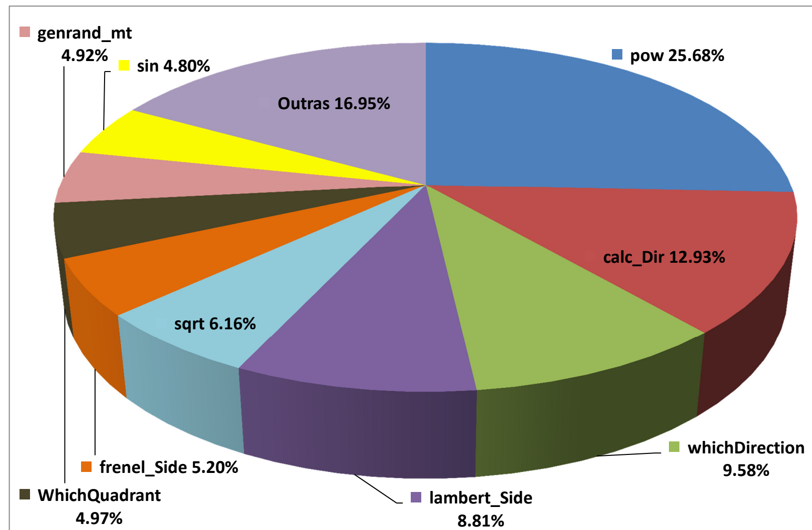


Figura 4.2: Perfil de execução da versão sem funções trigonométricas.

Após esta otimização foi feito um estudo relativo à percentagem do código que é possível paralelizar. Com a ajuda de funções de relógio [8], foi medido que para 10^{10} fotões $99,9\%$ do tempo de execução é consumido no ciclo computacional, figura 4.3.

[8] *OpenMP timers - omp_get_wtime();*

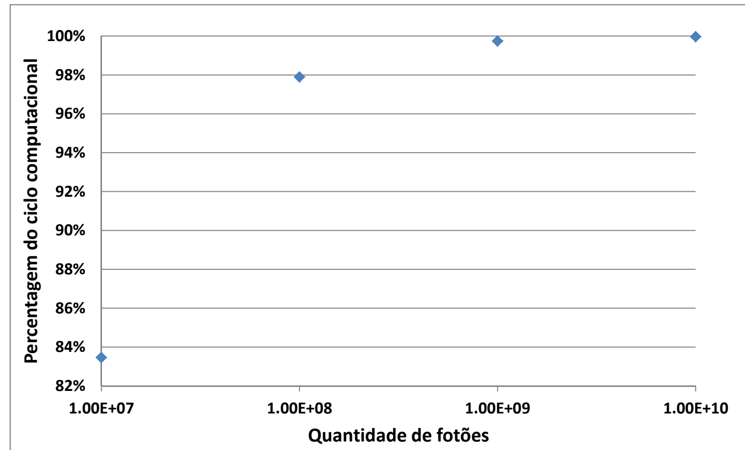


Figura 4.3: Percentagem do código que foi alvo de paralelização dependendo do nº de fotões (notar que a escala dos XX é logarítmica).

Estas otimizações resultaram num tempo de execução mais curto. Comparando as várias versões sequenciais, LED (com funções trigonométricas), LED (sem funções trigonométricas), Fotões (sem funções trigonométricas) e *Collapse* (aninhamento de ciclos sem funções trigonométricas) figura 4.4, é visível que, para as versões LED houve um ganho de $1,6$, para 10^{10} fotões. Notar também que a versão Fotões é ligeiramente mais lenta, 2%, que a versão LED (sem funções trigonométricas), isto deve-se ao cálculo adicional necessário para associar o fotão i a cada LED (*whereIsPhoton()*).

A versão com melhor desempenho é a *Collapse*, porém, destaca-se apenas por $0,5\%$ comparativamente à versão LEDs. A forma automática de aninhar os ciclos obtém um desempenho superior para a versão sequencial.

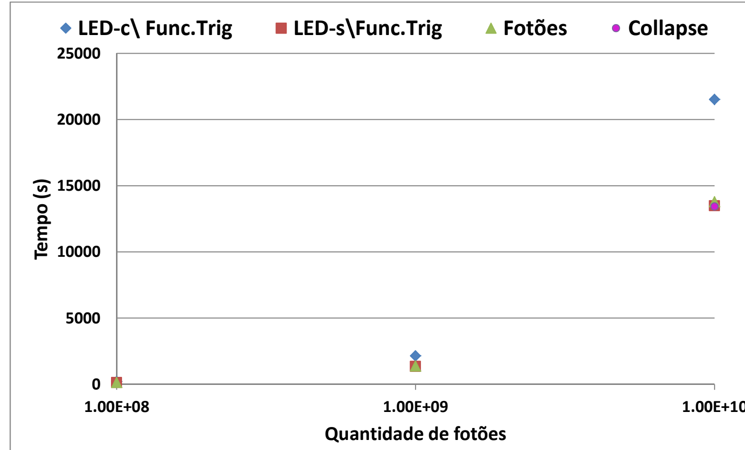


Figura 4.4: Comparação entre as versões sequenciais (notar que a escala dos XX é logarítmica).

4.3 Memória partilhada

Nesta secção serão apresentados os resultados das versões paralelizadas em memória partilhada. Para obter medidas do desempenho de execução é medido o ganho , fórmula 15,

$$Ganho = \frac{Tsequencial}{Tparalelo}(s) \tag{15}$$

e comparado à lei de *Amdahl* (ganho teórico). As medidas apresentadas de ganho são relativas à melhor versão sequencial, *Collapse*.

No entanto, a lei de Amdahl apresentada é a versão revista para vários núcleos, onde é especificado que o ganho máximo por núcleo em *Hyper-Threading* é, no máximo, 30% [23]. Assim, para n =número de fios de execução, $12 < n \leq 24$ e $Amdahl's_n = \frac{1}{(1-P)+(\frac{P}{n})}$, onde P corresponde à parte paralelizável do código ($0,999$ para 10^{10} fotões), tem-se que:

$$Amdahl's\,HT = Amdahl's_{12} + (Amdahl's_n - Amdahl's_{12}) * 1.3 \tag{16}$$

Nos vários gráficos que serão apresentados, é visível um fenómeno comum. Quando é feita a transição para a região com *Hyper-Threading*, 12 a 24 fios de execução, há uma quebra no desempenho. Este fenómeno deve-se às máquinas terem 12 núcleos físicos, por exemplo, quando apenas 2 núcleos estão em *Hyper-Threading*

(14 fios de execução) causa um mau balanceamento de carga.

4.3.1 Paralelismo ao nível dos LEDs

A primeira abordagem à paralelização foi implementada com o ciclo *for* a iterar nos LEDs, onde a cada LED são atribuídos fotões relativos à divisão $\frac{n^{\text{o}}\text{total de fotões}}{n^{\text{o}}\text{ total de LEDs}}$.

Ao dividir o nº total de LEDs pelos fios de execução, como o numerador é muito pequeno (64 LEDs), certos fios de execução terão de calcular 2, 3 ou 4 LEDs. Isto causará um mau balanceamento de carga, como é visível na figura 4.5. Aqui, é notável a perda de desempenho quando são utilizados 14, 16, 18 e 20 fios de execução.

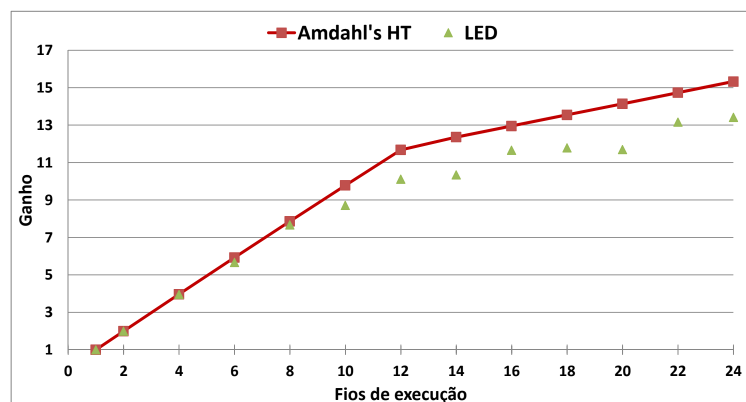


Figura 4.5: Ganho da versão LED comparativamente à versão serial.

Conclui-se que apesar deste mau balanceamento o algoritmo exibe o comportamento esperado, isto é, na região sem *Hyper-Threading* (até 12 fios de execução), o ganho segue aproximadamente a curva teórica revelando características de um escalamento linear. Esta versão apresenta um ganho máximo, para 24 fios de execução, de $13,4$.

Como forma de resolver o mau balanceamento de carga, avançou-se para a versão com a paralelização ao nível dos fotões.

4.3.2 Paralelismo ao nível dos Fotões

O processo de alterar o ciclo *for* para iterar nos fotões, resultou num acréscimo de computação devido à natureza do problema. Agora, cada fio de execução calculará os limites (número de fotões atribuídos) de cada LED, inferior e superior. Este cálculo adicional e o processo identificar a que LED cada fotão (função *whereIsPhoton()*) está associado, resulta num acréscimo de computação de 2% no tempo total de execução, passa de 13.491 segundos para 13.792 (com 24 fios de execução).

Esta solução nunca irá criar um balanceamento ideal, existirá sempre um ligeiro desajuste entre os fios de execução. No entanto, esta divisão reflete-se em mais ou menos um fotão que não representa impacto no desempenho.

Assim, é visível na figura 4.6 que o escalamento com e sem *Hyper-Threading* melhorou significativamente, estando agora mais próximo da linha teórica até 12 fios de execução e melhorando progressivamente após a utilização do *Hyper-Threading*. O ganho com 24 fios de execução é de $13,8$.

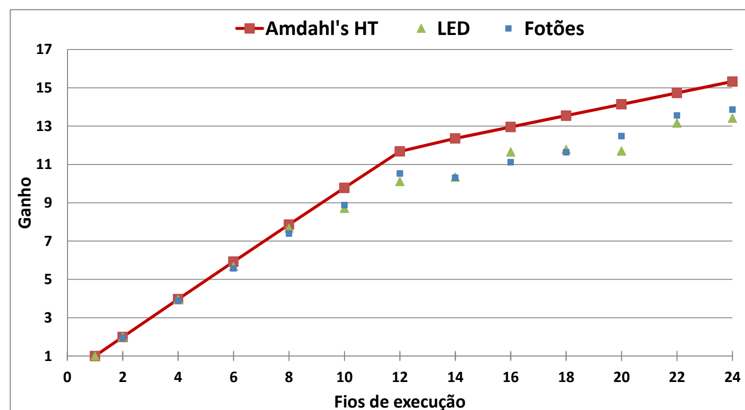


Figura 4.6: Ganho da versão Fotões comparativamente à versão LEDs.

4.3.3 Paralelismo ao nível dos LEDs & Fotões

Como forma de tornar o código mais legível, foi abordada a paralelização através de ciclos aninhados.

Um ciclo exterior para iterar nos LEDs e um interior para iterar nos fotões de cada LED. A paralelização foi alcançada através da diretiva *collapse*, que, em

compilação, transforma os dois ciclos num só ciclo.

A versão sequencial com o *Collapse* resultou numa melhoria de 2%, porém, não houve melhorias de desempenho em memória partilhada. Houve uma melhoria em relação à versão LEDs, no entanto, é mais lento que a versão Fotões, figura 4.7. O *Collapse* também aparenta ser "sensível" à distribuição de carga aquando é usado 14, 16, 18 e 20 fios de execução. O ganho com 24 fios de execução é de 13.

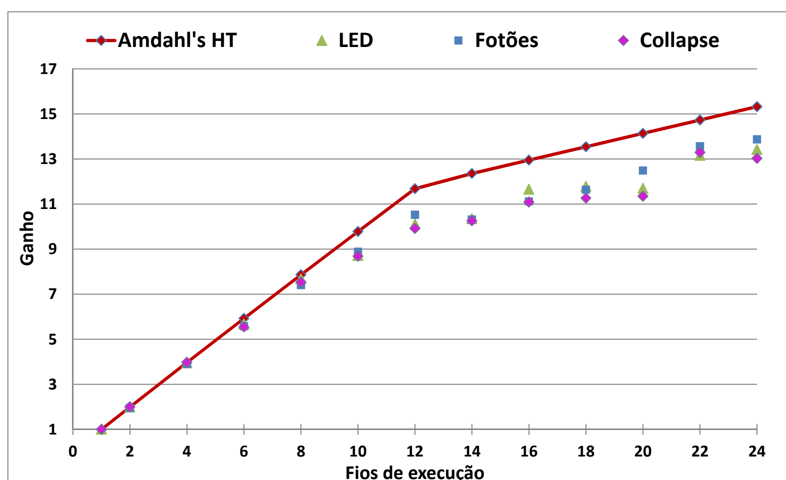


Figura 4.7: Ganho de todas as versões em memória partilhada.

4.3.4 Conclusões

Das várias abordagens apresentadas, a que reflete melhor desempenho é a versão Fotões, tabela. 4.2. Também aparenta ser a mais robusta quanto à divisão de carga com e sem *Hyper-Threading*.

Versão\Fios de execução	12	24
LED	10,1	13,4
Fotões	10,5	13,8
LED&Fotões	9,9	13

Tabela 4.2: Tabela de ganhos das versões em memória partilhada.

Relativamente à eficiência, as 3 versões foram comparadas à eficiência de *Amdahl*, fórmula 17. Na figura 4.8, é visível que tanto para 12 como 24 fios de execução a versão Fotões tem uma eficiência de 90%, tabela 4.3. Notar também a perda de eficiência devido ao balanceamento de carga para 14, 16, 18 e 20 fios de execução.

$$Eficiência\,de\,Amdahl = \frac{Amdahl's\,Hyper\,Threading}{\#\,fios\,de\,exec.} \qquad (17)$$

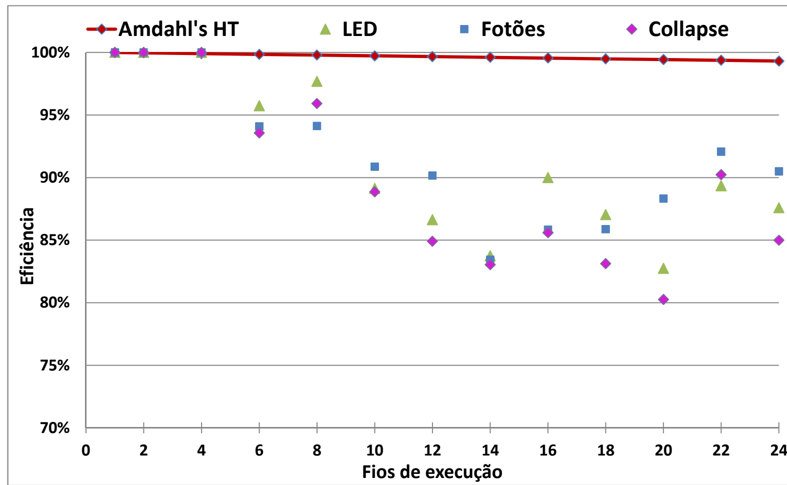


Figura 4.8: Eficiência de todas as versões em memória partilhada.

Versão\fios de execução	12	24
LED	87%	88%
Fotões	90%	90,5%
LED&Fotões	85%	85%

Tabela 4.3: Tabela da eficiência das versões em memória partilhada.

4.4 Memória distribuída

A paralelização em memória distribuída foi implementada a partir da versão Fotões. Para os testes seguintes foram usadas quatro máquinas 601 que totalizam 48 processos, 12 por máquina com possibilidade de *Hyper-Threading*.

4.4.1 Balanceamento dinâmico e granularidade

Para perceber o impacto que a granularidade tem no desempenho, foram executados vários testes (com 48 processos - 12 por máquina - 1 por núcleo), variando o valor da granularidade entre $2^0 \leq Granularidade \leq 2^{11}$, em potências de 2. A granularidade afeta apenas $5,6\% - (20segundos)$ do tempo médio de execução, quando comparando os limites testados, figura 4.9.

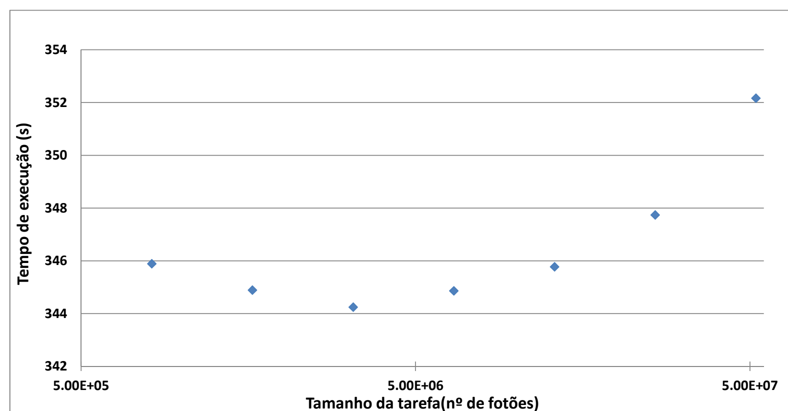


Figura 4.9: Tempo de execução com a variação da granularidade (notar que a escala dos XX é logarítmica).

Quanto mais tarefas necessitam de ser distribuídas maior será a comunicação. No entanto, visto tratar-se de mensagens muito curtas contendo apenas os limites de cálculo, dois *longs*, este excesso é quase insignificante. Assim, para o pior caso, 2^{11}, o tempo de comunicação total do processo mestre reflete-se em $0,35$ segundos do tempo total de execução, figura 4.10.

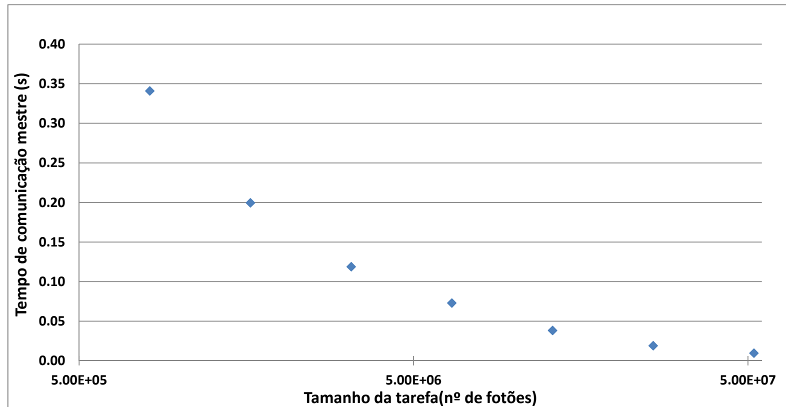


Figura 4.10: Tempo de comunicação extra do processo Mestre devido à granularidade(notar que a escala dos XX é logarítmica).

O desempenho para o pior caso apresenta um ganho de 38, para o melhor, 2^9, um ganho de 40 com 48 processos.

A eficiência mantêm-se acima dos 80% para todos os casos. Para 2^9, a eficiência é de 84%.

4.4.2 Híbrido

O modelo híbrido foi implementado a partir da versão de memória partilhada anteriormente apresentada com granularidade de 2^9. Para estes estes foram também usadas 4 máquinas com e sem Hyper-Threading para 2, 4, 6 e 12 processos.

A figura 4.11 demonstra que quanto maior o número de processos melhor é o desempenho. É visível também que o uso de Hyper-Threading prejudica o desempenho em cerca de 10%, para 12 processos . Assim, os melhores resultados são obtidos para 12 processos sem *Hyper-Threading* com um ganho de $33,7$.

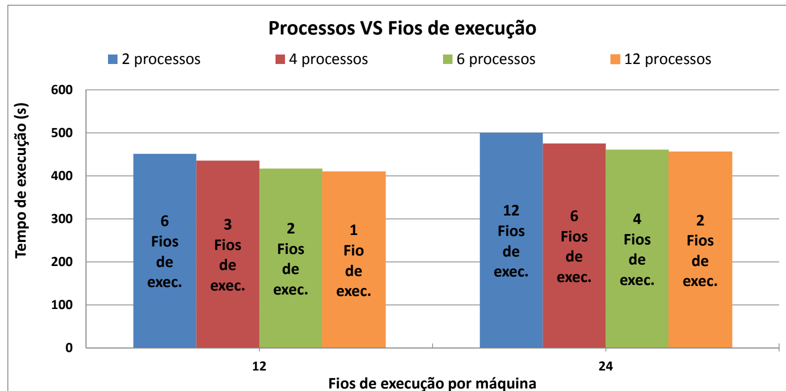


Figura 4.11: Ganho das várias combinações do modelo híbrido.

4.4.3 Conclusões

Os testes em memória distribuída apresentaram resultados positivos, exceto para a versão híbrida. Apesar da especulação teórica acerca da versão híbrida, na prática não obtém um desempenho superior. A versão de memória distribuída apresenta os melhores resultados pois recorre apenas a processos.

A utilização do OpenMP reduz o desempenho significativamente, tabela 4.4. Esta computação adicional é superior face ao ganho em tempo de execução resultante do paralelismo.

Versão	sem HT(12 processos)	com HT(24 processos)
Memória partilhada	40	-
Híbrida	34	30

Tabela 4.4: Tabela de ganhos das versões em memória partilhada e híbrida com 12 processos/máquina.

Quanto à eficiência, sem HT, a versão de memória distribuída tem resultados acima dos 80% enquanto que a versão híbrida, para o melhor caso, obtém 70%.

Capítulo V

Conclusão e trabalho futuro

No decorrer deste período de desenvolvimento a ferramenta informática ficou concluída, produz resultados fidedignos e está capaz de suportar vários ambientes paralelos.

Em termos de desempenho a abordagem de memória distribuída apresentou os resultados mais positivos, um ganho de 40 com 48 processos. Quanto à eficiência, o modelo de memória partilhada superou, o de memória distribuída, com um resultado acima dos 90% para 24 fios de execução.

A razão pela alta eficiência e desempenho das várias implementações resulta da natureza do problema a resolver. Os fotões ao serem independentes permitem uma paralelização ideal do problema, a cada processo/fio de execução é atribuído um conjunto de fotões, ficando por resolver a recolha dos resultados de cada processo/fio de execução.

Fisicamente, o problema da distribuição homogénea de luz ficou resolvido. Foi necessário ajustar vários parâmetros, dentro de uma gama verosímil, para concluir que a homogeneidade e o *black-MURA* encontram-se dentro dos padrões da indústria.

Trabalho futuro

Ferramenta de simulação

- Aplicar a ferramenta informática produzida à otimização do mundo real. Esta, pode ser usada como base de desenvolvimento e investigação para o *design* e otimização do produto.

- Investigar e implementar as propriedades reais das rugosidades. Nesta fase inicial foi feita uma aproximação que, apesar de reproduzir resultados credíveis, não representa o modelo real.

- Estudar o modelo real para recolher informação sobre fenómenos físicos que não foram implementados.

Análise e melhoria de desempenho

- Testar a ferramenta num ambiente heterogéneo. O modelo implementado em memória distribuída está pronto para resolver o balanceamento de carga com máquinas de diferente desempenho, porém, não foi testado.
- Explorar técnicas de paralelização alternativas para o modelo Híbrido. O resultado desta paralelização não foi positivo, no entanto, a premissa teórica é de alto desempenho.
- As otimizações feitas para remover as funções trigonométricas e alterar o ciclo de computação para iterar nos fotões, poderão ser benéficas para um ambiente de GPU's. Explorar o paralelismo nestas arquiteturas.
- De forma a aumentar a robustez e versatilidade da ferramenta, implementar os vários ambientes de memória partilhada, memória distribuída e GPUs numa ferramenta robusta para usufruir de todos os recursos disponíveis.

Referências

[1] Lihong V. Wang and Hsin-i Wu. *Biomedical Optics - Principles and Imaging.* Wiley-Interscience, 2007.

[2] M. Stevenson. Optical software: which program is right for me? *Institute of Physics and IOP Publishing Ltd.*, 2006.

[3] E. Alerstam, T. Svensson, and S. Andersson-Engels. Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration. *Journal of biomedical optics*, 13:060504, 2008.

[4] D.W. Hubbard. *The Failure of Risk Management: Why It's Broken and How to Fix It.* Wiley, 2009.

[5] G. Marsaglia. Diehard battery of tests of randomness. Florida State University, 1995.

[6] P. L'ecuyer and R.Simard. Testu01 - a software library in ansi c for empirical testing of random numbers generators. *ACM Trans. Math. Softw.*, 33:40, 2007.

[7] P. L'ecuyer and R.Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), August 2007.

[8] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] M. Makoto and N. Takuji. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[10] F. Panneton and P. L'ecuyer. On the xorshift random number generators. *ACM Trans. Model. Comput. Simul.*, 15(4):346–361, October 2005.

[11] Lawrence M. Murray N. Nandapalan, Richard P. Brent and Alistair P. Rendell. High-performance pseudo-random number generation on graphics processing units. *CoRR*, abs/1108.0486, 2011.

[12] P. D. Coddington. Analysis of random number generators using monte carlo simulatio. *Northeast Parallel Architecture Center. Paper 14*, 1994.

[13] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[14] M. Manssen, M. Weigel, and A.K. Hartmann. Random number generators for massively parallel simulations on gpu. *The European Physical Journal*, 210:53–71, 2012.

[15] M. Makoto and N. Takuji. Dynamic creation of pseudorandom number generators. In *Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, pages 56–69, June 1998.

[16] T. Rolf. Cache organization and memory management of the intel nehalem computer architecture. *University of Utah Computer Engineering*, 2009.

[17] T. Svensson E. Alerstam and S. Andersson-Engels. *CUDAMCML - User and implementation notes*. Department of Physics, Lund University, Sweden, April 2009.

[18] P. Li C. Jiang and Q. Luo. *GCMCML - User Manual for GCMCML*. Britton Chance Center for Biomedical Photonics, Wuhan National Laboratory for Optoelectronics, Huazhong, University of Science and Technology, Wuhan 430074, PR China, 2009.

[19] Radiant Zemax. Multi-cpu - http://www.radiantzemax.com/zemax/features/multi-cpu - visitado a 22 de janeiro 2014.

[20] Alma E. F. Taylor. *Illumination Fundamentals*. Rensselaer Polytechnic Institute, 2000.

[21] DFF OEM Work group displays. *Uniformity measurements standard for Displays*, 2010.

[22] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, July 2013.

[23] S. D. Case. How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, 6:11, 2011.