Heatplate - Gauss Seidel

Humberto Vaz e João Dias

***Abstract*—O presente documento apresenta o desenvolvimento do segundo projeto de *Algoritmos Paralelos* referente ao Mestrado Integrado de Engenharia Informática *(MIEI)* correspondente ao ano lectivo *2017/2018*, analisando o problema e as estratégias utilizadas para a sua resolução.**

Neste documento relatamos o trabalho realizado pelos autores com o intuito de atingir resultados relativos à performance de diferentes implementações para a resolução do problema *"Heatplate"* que retrata computacionalmente a difusão do calor numa placa metálica. É de referir que os resultados descritos são avaliados em dois aspetos (i) tempo de execução e (ii) número de passos e/ou iterações usadas. São então avaliadas 4 soluções para o problema em mãos (i) implementação do método de *Jacobi*, (ii) implementação Gauss Seidel e (iii) implementação Gauss Seidel Relaxada serão ainda criados traduções destes algoritmos com o intuito de medir os mesmos num ambiente de paralelismo recorrendo a (iv) memória partilhada, (v) memória distribuída e num (vi) ambiente híbrido. Sendo que todos estes *scripts* foram desenvolvidos em *C*.

Keywords — Heatplate, Difusão do Calor, *Gauss-Seidel*, *Red-Black*, Paralelização

1. Introdução

O problema "Heatplate" é um problema que pretende descobrir a convergência de uma "*mesh*" cujos valores estão associados a temperatura. Este domínio é decomposto por uma "*mesh*" retangular de duas dimensões. Trata-se de um processo iterativo, em que valor de cada nó da *mesh* é calculado/computado a cada passo até que o seu domínio convirja para uma tolerância ou valor estabelecido à priori. Isto é, assim que o método convirja para zero ou para uma tolerância aceitável a solução diz-se que aproxima a distribuição de temperatura e obtemos a solução do método.

A equação de *Poisson* trata-se de uma equação para resolver sistemas de equações parciais com bastante utilidade nos ramos da engenharia e física.

$$f(x,y) = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} \tag{1}$$

A equação que vai ser resolvida e estudada ao longo desta jornada trata-se de um caso particular da equação de *Poisson* descrita.

No entanto iremos usar diferenças finitas, isto é, valores aproximadas correspondentes a valores de temperaturas.

Sendo assim, podemos dizer que estaremos a resolver a equação de *Poisson* com valores finitos que correspondem a temperaturas utilizando dois métodos: o método de Jacobi, e o método *Gauss-Seidel*.

Ambos os métodos são métodos iterativos para resolução de sistemas de equações lineares. São bastante semelhantes, no entanto o método de *Gauss-Seidel* apresenta uma convergência mais rápida em relação ao método de *Jacobi*.

Visto que temos valores numéricos (escalares) na nossa *mesh*, não irá ser necessário calcular as supostas derivadas enunciadas na referida equação.

2. Considerações iniciais

2.1. Caracterização do Hardware

Para este projeto recorremos a duas máquinas, ambos nodos do Cluster SeARCH **781** e **662**. Essas máquinas são devidamente caracterizadas na tabela abaixo:

	662	781
Fabricante	Intel	Intel
Microarquitectura CPU	Ivy Bridge	Broadwell
# Cores	24 fis; 48 log	32 fis; 64 log
Freq. Relogio	2.4 Ghz	2.1 Ghz
L1	d:32K; i:32K	d:32K; i:32K
L2	256K	256K
L3	30720K	40960K
RAM	64G	256G

Estas escolhas são justificadas com o intuito de ter uma base de testes mais abrangente usando então duas micro-arquiteturas diferentes.

2.2. Dataset escolhido

Antes de analisarmos as soluções implementadas, é importante perceber quais os seus parâmetros de *input*. Observemos os seguintes requisitos: - Necessidade de um tamanho `N` ("altura/largura" da *mesh*) para a placa metálica. - Escolha de diferentes `N`'s para que o *Dataset* ocupem os vários níveis de memória.

Dessa forma iremos apresentar a inicialização da *mesh* sob a forma de código C:

```
void init(){
    int i, j;
    for (j = 0; j < N; j++) u[N-1][j] = 100.0f;
    for (j = 0; j < N; j++) u[0][j] = 0.0f;
    for (i = 0; i < N; i++) u[i][N-1] = 100.0f;
    for (i = 0; i < N; i++) u[i][0] = 100.0f;

    for (j = 0; j < N; j++) w[N-1][j] = 100.0f;
    for (j = 0; j < N; j++) w[0][j] = 0.0f;
    for (i = 0; i < N; i++) w[i][N-1] = 100.0f;
    for (i = 0; i < N; i++) w[i][0] = 100.0f;

    for (i = 1; i < (N-1); i++)
        for (j = 1; j < (N-1); j++)
            u[i][j] = w[i][j] = 50.0f;
```

```
}
```

Como é possível observar pelo código acima, utilizamos dois valores para as ”fronteiras” da *mesh*, o 100 e o `0`. Neste caso particular, o valor `100` corresponde ao ”calor” e o `0` corresponde ao ”frio”. Deste modo temos apenas uma única fonte de frio no ”teto” da placa metálica:

```
for (j = 0; j < N; j++) u[0][j] = 0.0f;
```

E três fontes de calor nas ”paredes” e ”chão” da ”placa metálica”:

```
for (j = 0; j < N; j++) u[N-1][j] = 100.0f;
for (i = 0; i < N; i++) u[i][N-1] = 100.0f;
for (i = 0; i < N; i++) u[i][0] = 100.0f;
```

Repare-se que inicializamos duas matrizes, `u` e `w` de forma a podermos executar os métodos *Jacobi* e *Gauss-Seidel*.
Mais à frente iremos explicar de forma mais concreta a utilização destas duas matrizes representativas da *mesh*.

De forma a tirarmos partido da localidade espacial e acedermos de forma mais rápida a dados necessitamos de que estes se encontrem próximos do CPU. Sendo assim e como é impossível manter esta quantidade de dados tão grande em registo, estes deverão estar preenchidos de forma a preencher o segundo nível de memória mais rápida da máquina, os vários níveis de Cache.

Dado que ambas as máquinas têm dimensões da cache L1 de `32K` para dados e `32K` para instruções, `256K` para a cache L2 e apenas diferem na cache L3, `30720K` (nó 662) e `40960K` (nó 781) iremos ter 4 *Datasets* diferentes:

1 - Dataset para a cache L1
2 - Dataset para a cache L2
3 - Dataset para a cache L3
4 - Dataset para a RAM

Tendo estes requisitos em mente chegamos a uma fórmula para calcular a dimensão do problema de interesse:

$$n \leq \sqrt{\frac{TamanhodaCache}{\#Matrix * sizeof(Double)}}$$

Uma das precauções que tivemos foi ainda a seleção de um **N** que ocupasse exatamente numa linha de cache, prevenindo assim custos desnecessários no carregamento de ”lixo”. Para tal, tendo em conta que nas máquinas utilizadas, uma linha de cache tem `64B` (8 `Doubles`) chegamos outra formula restringente:

$$LinhadeCache = \frac{N \times N}{8} \tag{2}$$

Como a única diferença na memória nas duas máquinas se trata da L3, decidiu-se utilizar o mesmo valor N para L3, o valor da cache L3 mais pequena, neste caso é a do nó 662. Sendo que para este nó, este nível de cache enche completamente e no 781 não acontece o mesmo no entanto é a única forma que nos permite ter um ponto de comparação entre os dois nós nós em todos os níveis de memória.

Sendo assim, esses tamanhos encontram-se calculados na seguinte tabela:

Memória	N
L1	44
L2	124
L3	4384
RAM	5000

3. Caso de Estudo

O caso de estudo deste trabalho é a difusão de calor também conhecido por **”Heatplate”**, pretende-se simular o efeito deste em um ambiente uniforme, determinando qual o seu estado de equilíbrio. Tal como referido antes, para podermos simular o seu efeito é necessário criar uma matriz que represente o ambiente em que se pretende trabalhar, em cada posição da matriz foi inserido o respetivo valor de temperatura (para simplificar o processo de difusão consideramos que o valor 0 representa o valor mais “frio” e 100 o valor mais “quente” representativo das fontes de calor).

4. Implementações

De forma a podermos comparar a performance e comportamento da difusão do calor, implementaram-se vários algoritmos com base nos métodos de *Jacobi* e *Gauss-Seidel*. Dado que este problema normalmente caracteriza-se por *meshs* com dimensões bastante grandes, faz sentido proceder à paralelização dos seus algoritmos no paradigma de memória partilhada (OpenMPI) e no paradigma de memória distribuída (OpenMP).
Para além destas diferentes abordagens, existem duas implementações por blocos *(tilling)* explicadas mais à frente.

4.1. Método Jacobi

O método de *Jacobi* é um algoritmo para resolver sistemas de equações lineares. Trata-se de uma versão simplificada do algoritmo de valores próprios de *Jacobi*.

No nosso caso de estudo, fazemos uso da equação de *Laplace* (como caso particular da equação de *Poisson*) que nos providencia o cálculo do *Steady-state* de temperatura, ou seja, o estado final.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{3}$$

A equação anterior, trata-se da equação de Laplace referida.

Dela podemos derivar a equação que representa a temperatura de um dado nó da *mesh* a cada iteração/passo:

$$T_{ij} = \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4} \tag{4}$$

Com o auxílio da equação anterior, conseguimos calcular a temperatura dos nós interiores da *mesh* que foi predefinida como quadrada.

A cada iteração iremos calcular a temperatura do seguinte número de nós:

$$N \times N - 4N \tag{5}$$

O cálculo anterior serve para mostrar que apenas estamos a atualizar os valores do interior da *mesh* deixando de fora as fronteiras *(4N)*.

Com o intuito de facilitar ao leitor a compreensão do algoritmo iremos mostrar uma figura (IV-A) que consegue ilustrar melhor o cálculo da temperatura para cada ponto da *heatplate*.

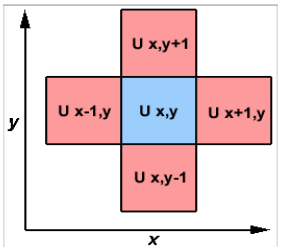


Fig. 1. Cálculo da temperatura para o nó x,y da ”mesh” da iteração seguinte

Na análise deste algoritmo depreendemos dois tipos de dependências de dados.

- dependências para com os vizinhos (Read after write (RAW) e Write after write (WAW)), que nos impossibilitam fazer a computação do valor na própria matriz;
- *bottleneck* de cada iteração, visto que iniciamos a nova iteração apenas quando a sua anterior termina;

Como método de resolução em parte estas dependências recorre-se a métodos iterativos, implementações essas abordadas de seguida.

A versão sequencial do algoritmo de Jacobi, começa com uma matriz inicial (inicializada no próprio programa) que é usada para calcular o próximo estado do sistema, o seu resultado guardado numa matriz auxiliar, `u`.

O processo de cálculo do próximo estado, implica calcular todas as posições do próximo estado do sistema a partir das posições da matriz atual, este processo é repetido um número de vezes, calculando os vários estados de evolução da placa.

No processo de difusão cada posição da matriz vai influenciar as posições vizinhas, como podemos ver na imagem IV-A, a posição U(x, y) da matriz `u` é a média aritmética do valor correspondente na matriz `w`, com as posições vizinhas identificadas a vermelho.

***Blocos* - sequencial**

Neste algoritmo foi ainda introduzido no contexto de blocos algo que na execução sequencial vai produzir um óbvio **overload** a nível computacional devido à necessidade de um processamento maior pela distribuição da *mesh/grid* por blocos. Assim sendo a **mesh** será dividida em partes, no caso da imagem abaixo em 4.

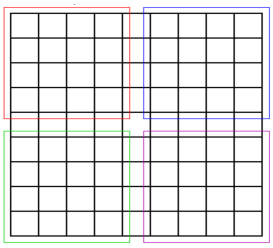


Fig. 2. Divisão de uma ”mesh” em 4 partes

Porém esperávamos utilizar esses resultados como um ponto de referência para as que demais se seguem. Tendo em conta que nas versões paralelos uma maior e melhor distribuição das grids pelos *cores* terão uma impacto óbvio nos tempos de processamento.

Pelos tamanhos considerados para teste acabamos por utilizar um parâmetro estático de 64 elementos por bloco, ou seja, *8*8*, apenas para manter a generalidade da aplicação, pois por resultados também testados para tamanhos de matrizes superiores sabemos que o lado de bloco de **32** apresenta melhores resultados, possivelmente motivados por um decréscimo do **overload** associado à troca de blocos, ou porque blocos de 64 elementos não exploram na sua capacidade total a localidade espacial associada a este tipo de problema.

4.2. Método Gauss-Seidel

O método de Gauss-Seidel é um método iterativo para resolução de sistemas de equações lineares. O seu nome é uma homenagem aos matemáticos alemães *Carl Friedrich Gauss* e *Philipp Ludwig von Seidel*.

Tal como tem vindo a ser referiod, este método é bastante parecido com o método de *Jacobi*. No entanto utiliza os valores mais atuais da temperatura de cada nó, ou seja, para alguns pontos da grid serão utilizados valores calculados já nessa iteração, o que faz com que convirja mais rapidamente para o *Steady-state*.

Com isso em mente foram criados 2 algoritmos:

- Considerando o acesso em apenas um ciclo à mesh;
- Estratégia conhecida por *Red-Black*;

Primeira abordagem

A primeira abordagem ao problema consiste na utilização de um ciclo que por sua vez tende em utilizar os valores que já foram calculados, neste caso seguindo a ordem de passagem pelo ciclo os valores tratados serão os correspondentes a

$$T_{i,j-1} \; e \; T_{i-1,j} \tag{6}$$

Aplica o principio do *Gauss-Seidel* havendo apenas a deficiência de não possuirmos os valores atuais dos nodos calculados posteriormente, não havendo aqui uma distinção

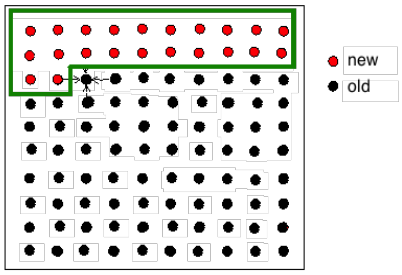
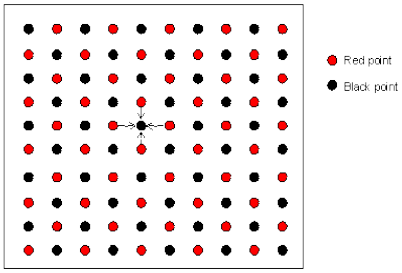


Fig. 3. Utilização dos valores mais recentes

entre os tipos de nodos para além se estes se encontram ou não numa região de fronteira, sendo todos calculados posteriormente da mesma forma.

Na imagem apresentada conseguimos facilmente distinguir a utilização dos valores mais recentes no calculo dos valores da iteração atual dos valores antigos.

Segunda abordagem



3

Fig. 4. Método ”Red - Black”

A estratégia *Red-Black* consiste no cálculo temperatura de cada nó, sendo que a temperatura de metade dos nós é calculada tendo em consideração a ultima iteração e a outra metade, conforme a natureza do solução, pode já utilizar os valores atualizados dessa mesma iteração, possibilitando assim uma pseudo-quebra na dependência de dados entre iterações.

Este grupo de nós é apelidado de ”Vermelho” (Red), sendo que o restante grupo é o grupo ”Preto” (Black).

Através deste método conseguimos o atualizar o seguinte número de nós da *grid* a cada iteração:

$$\frac{N^2}{2} \tag{7}$$

Dentro destas implementações tentamos aplicar também a mesma ideologia dos **blocos** supracitadas para o *Método Jacobi*, com pretensões de obter ganhos nas versões paralelas.

4.3. Gauss-Seidel - Relaxed

Melhorias adicionais ainda podem ser feitas nas versões sequenciais da primeira implementação do algoritmo **Gauss-Seidel**. Este algoritmo não depende somente do cálculo da média dos pontos da grid vizinhos, mas calcula essa média considerando ainda um determinado fator constante, vamos chama-lo de ω. Este fator influenciará a atualização da seguinte maneira:

```
updateval = (( u[i][j+1] + u[i+1][j]
+w[i-1][j] + w[i][j-1] ) *0.25) - w[i][j];
w[i][j] += updateval * w;
```

Foram criadas duas variantes deste algoritmo que apenas variam pela taxa/factor de relaxamento de relaxamento, uma pesquisa breve permite descobrir que esse valor se deve encontrar entre 1 ¡ω ¡ 2, para isso seguimos duas possibilidades:

- valor estático, fixado em 1.5;
- valor variável pelo tamanho da matriz

Sendo o valor variável calculado pela seguinte formula:

$$1 + \sin(\frac{\pi}{N+1}));$$

(8)
Valor retirado do livro de James W. Demmel;

5. OpenMP - Memória Partilhada

Para estes tipo de algoritmos em memória partilhada foram obtidos através aplicação de **diretivas do OpenMP** sobre os ciclos aquando da computação do valor da temperatura iteração atual.

5.1. Implementações gerais

As alterações a este algoritmo cingem-se à adição de

```
#pragma omp parallel for reduction(max : diff)
shared(w, u) private(i,j, aux)
```

Sendo que as dependências de dados por iteração são tratadas nas versões sequenciais podemos utilizar esta simples operação sem o risco de computar erradamente estes valores.

5.2. Implementações recorrendo a Blocos

Apenas nos casos em que a opção com blocos era apresentada é que se teve de ter um cuidado superior devido ao próprio tamanho dos blocos, fatores condicionantes do cálculo.

```
#pragma omp parallel for reduction(max : diff)
shared(w, u) private(i,ii,j,jj, aux)
```

Seguindo o mesmo principio aplicado para os métodos que não se baseavam em blocos, supostamente as prevenções extra realizadas permitir-nos-iam obter resultados satisfatórios.

Infelizmente, por motivos que não se revelam óbvios a obtenção desses resultados relativos à implementação de blocos quadrados num ambiente de memória partilhada provou-se um obstáculo superior ao previsto, por vezes obtendo um número variável de iterações ou produzindo um resultado final igual a:

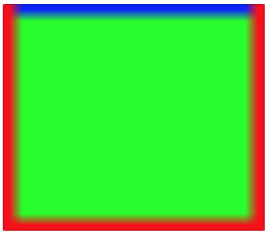


Fig. 5. Blocos - erro na computação

Pelos motivos supracitados achamos por bem não apresentar os tempos e número de iterações associados a esta implementação para não induzir o leitor em erro.

6. MPI - Memória Distribuída

Para o paradigma de memória distribuída foram desenvolvidos três algoritmos, todos partilham alguns pontos em comum, tais como:

- implementação em *Farming*;
- Variações do algoritmo *Gauss-Seidel* com implementação *Red-Black*;
- Processamento do valor de `diff` realizado sempre pelo processo **Master** aquando da transferência da resposta dos **Slaves/Workers** para a matriz **W**;

Assim sendo chegamos às seguintes implementações:

- Computação sobre blocos estático;
- Computação sobre blocos estáticos, comunicação através de *Broadcast*;
- Computação sobre linhas, comunicação dinâmica;

Uma consideração que temos ao criar esta implementação está intrinsecamente ligada a forma como as matrizes são alocadas em memoria. A utilização de uma estrutura estatica (grif) em formato de matriz, que o tornam basicamente uma lista de apontadores para vários arrays (linhas), impede um avanço mais *straightforward* que se resumia ao envio de blocos simples da matriz ou mesmo dela própria.

Sendo assim vamos ,antes de qualquer comunicação, converter as matrizes para um vectores, sendo estas ultimas estruturas a que serão enviadas entre as comunicações. Para evitar realizar esta reconversão para matriz nos slaves/workers criamos um novo algoritmo de computação que atua não sobre matrizes mas sim sobre vectores as operações necessárias, seguindo basicamente o mesma ideologia da versão sequencial. Apresentamos então um compute ”especial”:

```
void compute(double* linhas,
double* linha,int linesByBlock){
  int i,j;
  //#pragma omp parallel for private(j)
  for (i=1;i<linesByBlock-1;i++){
    for (i = 1; i < linesByBlock-1; i++)
    for (j = (i %2 == 0? 2 : 1); j < (N-1); j+=2){
linhas[j+((i-1)*M)]=
0.25*(linhas[((i-1)*M)+j]+linhas[((i+1)*M)+j]
/*calcular a colum */
+ linhas[(i*M)+j-1] + linhas[(i*M)+j+1]
/*calulate line*/  );
linha[j+((i-1)*M)] = linhas[j+((i-1)*M)];
    }
    //#pragma omp parallel for private(j)
     for (i = 1; i < linesByBlock-1; i++){
     linha[((i-1)*M)]=100;//heat font
        for (j = (i %2 == 0? 1 : 2); j < (N-1); j+
      linha[j+((i-1)*M)]=
      0.25*(linhas[((i-1)*M)+j]+linhas[((i+1)*M)+j
      /*calcular a colum */
    + linhas[(i*M)+j-1] + linhas[(i*M)+j+1]
    /*calulate line*/ );
    }
  linha[j+((i-1)*M)]=100;//heat font
}
}
}
```

6.1. Computação sobre blocos estático

Esta implementação consiste em enviar através da primitiva `MPI_SEND` o bloco da matriz respetivo (em formato *array*) a cada um dos *slaves/workers*.

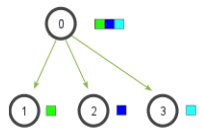


Fig. 6. Envio de blocos da matriz

Após computado os resultados são enviados ao *master*.

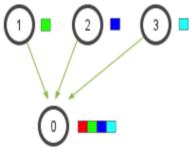


Fig. 7. Recepção de blocos da matriz

De notar que o *master* ao receber os resultados não tem nenhuma ordem especifica para tal, recebendo de qualquer fonte (`MPI_ANY_SOURCE`) que por sua vez os introduz na matriz `w` ,usando para isso a tag (`status.MPI_TAG`) onde está contida a linha a qual se inicia o bloco. Após receber todos os blocos está preparada para iniciar a próxima iteração, caso esta exista.

Sendo que a necessidade de haver ou não uma próxima iteração é realizada no **Master** aquando da copia dos valores das mensagens para a matriz principal **W**, neste transporte de dados é calculado e guardado o valor de *diff* e verificado se este ainda satisfaz a condição

```
(diff > TOL)
```

se assim for procede à realização de mais um iteração.

6.2. *Computação sobre blocos estáticos, comunicação através de Broadcast*

A implementação desta funcionalidade resume-se ao envio de toda a matriz (em formato de *array*) para todos os *slaves/workers* disponíveis.

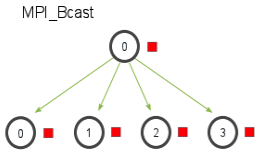


Fig. 8. Broadcast

Posteriormente, tendo em consideração o seu *rank*/identificador de processo, cada *slave/worker* vai computar o seu bloco da matriz respetivo. Sendo o resultado final enviado para o *master* que replica o processo da implementação anterior.

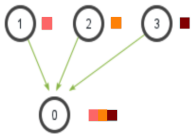


Fig. 9. Receção de blocos da matriz após Broadcast

O processamento e descoberta se a *mesh* já atingiu um *steady state* é realizada da mesma forma que tinha supracitada para o caso *Computação sobre blocos estático*.

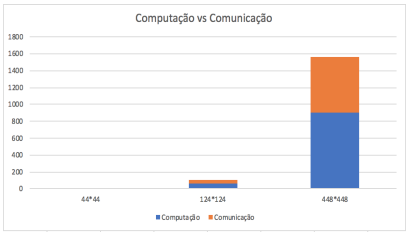


Fig. 10. Comunicação entre Master e *slaves*, comparação de tempos - 1 Nó: 4 Processos : 1 Master + 3 *slaves*

6.3. *Computação sobre linhas, comunicação dinâmica*

Uma implementação dinâmica apresenta problemas no sentido em que cada *slave* não sabe à priori quanto trabalho lhe será atribuído. A nossa resolução para este problema consiste em enviar uma mensagem inicial a comunicar se o processo deve ou não continuar a computação.

Esta implementação é referente ao envio de linhas para computação. Para cada linha computada esta vai necessitar das suas fronteiras, provocando um aumento no tamanho enviado que será três vezes tamanho da linha por processo.

Inicialmente existe um escalonamento estático para um número máximo de *slaves/workers*. Após um *slave* concluir a computação este envia o seu resultado ao *master*, que

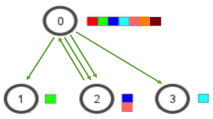


Fig. 11. Comunicação entre Master e *slaves*, computação por linha

adiciona essa linha à matriz auxiliar `u`.

Caso a haja, atribui uma nova tarefa ao *slave* que ficou livre, fazendo isto até todas as linhas da matriz exceto as fontes de calor, $Y = 0$ e $Y =$ *lado da matriz - 1*, estarem concluídas.

Nesta fase o *master* enviará para todos os *slaves* disponíveis a tag referente ao cessar trabalho e iniciará a próxima iteração. Em suma, esta implementação segue os princípios de um *fifo*.

Apenas neste exercício foi feita a medição dos tempos de comunicação sendo que é também esta implementação a mais rica neste factores.

A vermelho podemos ver os tempos de comunicação e a azul os de comunicação a vermelho. A implementação para 2 nos não difere muito das medidas apresentadas.

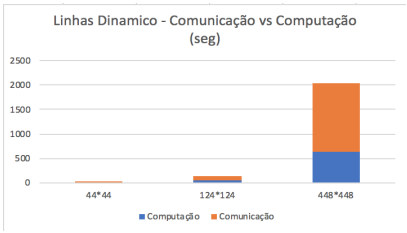


Fig. 12. Comunicação entre Master e *slaves*, comparação de tempos - 1 Nó: 4 Processos : 1 Master + 3 *slaves*

Uma análise rápida permite distinguir que esta implementação tem um grande *bottleneck* associado à sua comunicação. Nada de estranhar visto que a computação é feita linha a linha, porém os valores apresentados demonstram que esta não seria uma boa implementação para o paradigma de memoria distribuída, apresentando a comunicação como um custo na performance muitas vezes superior a 3 vezes o da computação.

Nota: A implementação destas medições foram feitas de modo *naive* eliminando apenas a computação do programa e correndo-o. Sendo o seu resultado posteriormente subtraído tempo ao total.

7. Versão híbrida - MPI OpenMP

A versão híbrida foi obtida através aplicação de diretivas do OpenMP sobre a computação de blocos estático. Analisando o algoritmo este representa aquele que mais tem a ganhar

com a aplicação do paralelismo.

Verificamos que não é possível aplicar diretamente o paralelismo seguindo as mesmas diretivas utilizadas para o Gauss-Seidel com a implementação em Red-Black seguindo o mesma ideologia que tinha sido aplicada para este mesmo algoritmo no contexto de *memória partilhada*.

Todo o restante processo é praticamente idêntico à versão **MPI** que lhe deu origem atacando apenas o problema nas fases de possível paralelização.

Assim sendo vamos paralelizar este código em 2 fases distintas:

- Envio dos blocos para os respetivos *slaves*;
- Computação do bloco de linhas;

Olharmos para o processo de cálculo de cada *slave* verificamos que é possível calcular qualquer posição do estado seguinte em paralelo, pois não há dependências de dados, sendo os valores para os **red** calculados inicialmente e sendo estes apenas dependentes dos valores **black** da iteração anterior e guardados tanto no vector de resposta coo na próprio vector enviado pelo **Master**, procedemos numa segunda fase ao calculo dos novos valores dos pontos **black** da *grid* são calculados a partir da matriz atual e apenas dependentes dos prontos previamente atualizados **red** quando calculados são guardados, cada um na sua respetiva posição no *vector* auxiliar de resposta.

Para concluir então esta versão bastou-nos adicionar a seguinte anotação ao ciclo exterior que envia os blocos `pragma omp parallel for private(i, aux, j)` bem como a anotação `pragma omp parallel for private(j)` ao ciclo exterior da função que executa a computação (compute).

8. Análise de Resultados

8.1. Consistência das soluções

O que acontece nestes algoritmos é a paralelização que tem por base quebrar em parte parte das dependências encontradas no algoritmo sequencial de Jacobi, algo que nos posteriores algoritmos sequenciais criados vai ser desconstruido tendo em consideração a natureza iterativa dessas soluções. Tendo em consideração que para a aplicação destes nas máquinas escolhidas, mais exatamente na **662** e o **781**, o resultado do número de iterações não vai nem se pode alterar, apenas podemos fazer uma comparação direta dos tempos de execução. Estes tempos de execução serão então analizados tanto para os algoritmos em sequencial como para os algoritmos que recorrem a paralelismo em memória partilhada. Como parâmetro de teste vamos variar o número de cores utilizados por máquinas, avaliando se para arquiteturas diferentes os tempos de execução se encontram na mesma ordem ou , como será de supor, uma arquitetura mais avançada produz por sua vez um melhor resultado.

A nossa escolha do número de processos vai então variar conforme as máquinas usadas. Como utilizamos o nó **662** e o **781** temos para o primeiro caso 24 cores físicos e 48 lógicos (usando *hyperthreading*) e no **781** 32 físicos e 64 lógicos, iremos limitar as nossas escolhas pelo de menor capacidade, introduzindo variação pela escolha de número de processos:

- 2
- 4
- 8
- 16
- 32
- 48

Passamos então para a confirmação de que os algoritmos produzem resultados esperados, posteriormente numa segunda fase passaremos para a comparação das duas máquinas utilizadas.

Aquando da produção das imagens finais no *Cluster SeARCH* existiram algumas dificuldades obrigaram-nos a produzir estas imagens num computador pessoal, algo que não afeta o compreensão e o propósito desta ação.

Através da utilização do seguinte código:

```
    void printi(char* name){
double value;
FILE *f = fopen(strcat(name,".ppm"), "wb");
fprintf(f, "P6\n%i %i 255\n", N, N);
for (int y=0; y<N; y++) {
    for (int x=0; x<N; x++) {
     value=w[y][x];

     if(value<50){
        fputc(0, f);
        // 0 .. 255
        fputc(value*(255/50), f);
        // 0 .. 255
         fputc(255-(value*(255/50)), f);
         // 0 .. 255
     }
     else{
     fputc((value-50)*(255/50), f);
     // 0 .. 255
        fputc(255-((value-50)*(255/50)), f);
        // 0 .. 255
         fputc(0, f);
         // 0 .. 255
     }
    }
}
fclose(f);
}
```

Podemos atestar a veracidade do resultado através de um formato visual:

Todos os algoritmos passam por este teste apresentando também esta imagem, confirmando que comportam de forma legitima.

Em segundo lugar passamos à comparação dos tempos de execução nas diferentes máquinas.

Para tal, a abordagem ao problema é relativamente direta. Escolhemos um valor do leque dos resultados obtidos pelo algoritmo **Gauss-Seidel** utilizando a implementação **Red-Black**, a partir daí faz-se uma comparação com o tempo necessário por ambas as maquinas para atingir esse resultado. O mesmo

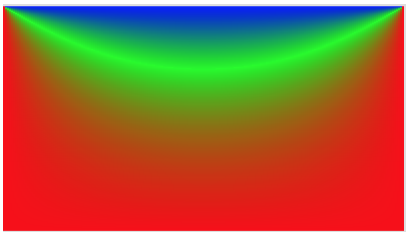


Fig. 13. Estado final da *grid* após atingir o *Steady-state* da temperatura

pode aplicar-se aos outros algoritmos, escolhemos o **Gauss-Seidel- Red-Black** tendo em consideração a sua importância e devido á fácil paralelização do algoritmo, apresentando a melhor predisposição para beneficiar com uma paralelização devido à sua independência de dados por iteração e à seu vasto trabalho computacional que de uma forma sequencial se traduz em tempos de execução **altos** mas que no contexto paralelo pode ser reduzido e traduzido em ganhos visíveis de *performance*.

Temos ainda o cuidado de fazemos uma comparação com vários processos através do speedup:para simular a utilização de:

- apenas cores físicos;
- cores físicos e lógicos;

Temos a noção que vai ocorrer um *overhead* intrínseco à alocação das *threads*,especialmente no caso das 48 impedindo assim uma justiça total neste problema, porém estamos convictos que na utilização das matrizes maiores dimensões este valor seja desprezável introduzindo assim um maior equilíbrio na abordagem deste problema.

De notar que para esta medição como para todas as restantes ao longo deste relatório foi utilizado o modelo **K-best** de **5**,ou seja, das 5 medições foram sempre retirados o melhor valor, sendo que quando os resultados apresentavam uma variância superior a **10%** estes eram ignorados e novas medições eram retiradas.

Crê-se que estes casos pontuais em que os resultados apresentavam valores não esperados poderiam ser motivados por:

- factores externos (temperatura da maquina aquando das medições, etc.)
- dificuldade em reservar todos os cores, o que diminui a capacidade de controlo de carga no **CPU**, escalonamento, etc.)

O próximo gráfico relata os resultados obtidos do nó **662** vs **781** na medição destes tempos em picosegundos

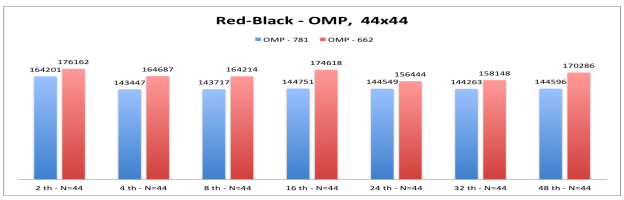


Fig. 14. DataSet 44x44 - 662 vs 781

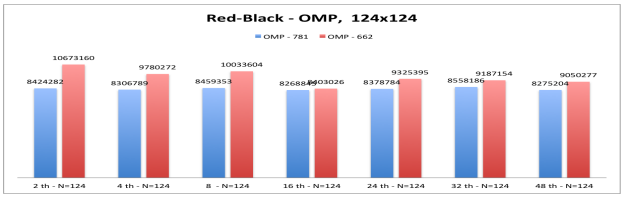


Fig. 15. DataSet 124x124 - 662 vs 781

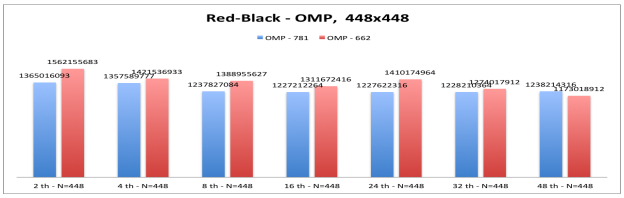
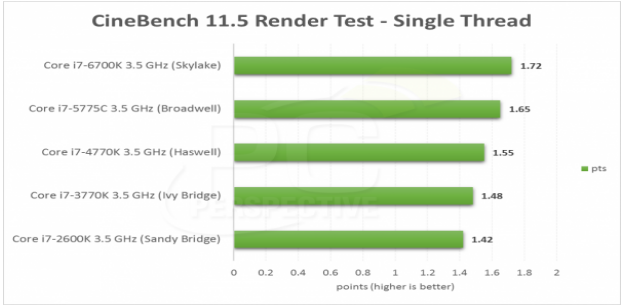


Fig. 16. DataSet 448x448 - 662 vs 781

Quanto ás máquinas utilizadas, nós **662** e **781**, é possível observar um desvio de **0.3** Ghz entre eles na frequência de relógio, sendo que os nós do **662** ficam a frente neste aspeto, o que poderia se traduzir em melhores resultados a nível de *performance*, algo que não se verifica, sendo que o nó **781** com um frequência de **2.1** GHz, consegue, na sua generalidade, possuir melhores resultados. Suspeita-se que o motivo para estes resultados está diretamente relacionado com a **Microarquitectura** de cada **CPU**, sendo que o **662** e **781**, divergem nesse aspeto, sendo um uma máquina **Ivy Bridge** e a outra **Broadwell** respetivamente.

KNOW YOUR CODENAMES			
CODENAME AND YEAR	PROCESS	PROMINENT CONSUMER CPU BRANDING	TICK/TOCK
Westmere (2010)	32nm	Core i3/i5/i7	Tick (new process)
Sandy Bridge (2011)	32nm	Second-generation Core i3/i5/i7	Tock (new architecture)
Ivy Bridge (2012)	22nm	Third-generation Core i3/i5/i7	Tick
Haswell (2013)	22nm	Fourth-generation Core i3/i5/i7	Tock
Broadwell (2014/2015)	14nm	Fifth-generation Core i3/i5/i7, Core M	Tick
Skylake (2015?)	14nm	TBA	Tock

Como podemos observar pela tabela acima, a **Ivy Bridge** saiu em **2012** e corresponde a terceira geração da arquitetura Core da Intel enquanto a **Broadwell** saiu em **2014/2015** e corresponde a quinta geração Core. Recorrendo a outros **benchmarks** conseguimos verificar que esta diferença de gerações vai ,de forma transversal, ter consequências na **performance**, sendo que as gerações mais recentes são caracterizadas por melhores performances, algo que analisado de forma análoga tem o seu sentido.



Concluímos então que os resultados obtidos dentro de toda esta perspetiva fazem sentido, tento ainda consideração que a máquina **662** entra em **hyperthreading** a partir das **24** *threads* enquanto o **781** apenas o faz passando a barreira das **32**, justifica ainda mais este estudo.

Passamos agora ao estudo do comportamento dos diversos algoritmos, por motivos de facilidade em alocação de recursos cingimos agora o nosso estudo à maquina **662**.

8.2. Interpretação de resultados

Em primeiro lugar torna-se importante revelar que tivemos de diminuir a nossa base de testes, o que queremos dizer com isto é que para as matrizes de maiores dimensões (acesso a **L3** e **RAM**) não conseguimos medir tempo devido à restrição de tempo associadas a cada entrada no *Cluster SeARCH*, assim sendo é de elevada importância referir que tivemos de encurtar o tamanho da nossa *grid* no caso do acesso a **L3** para **448** e não **4384** como tinha sido previamente planeado, sendo que para o caso da **RAM** já não se torna possível testar.

- Análise da Complexidade

De forma a antevermos a convergência dos algoritmos chegamos à conclusão que estes apresentam uma complexidade na ordem de:

Implementação	Complexidade
Sequential Jacobi	O(N)×$O(N)$
Parallel Jacobi	O(N) ×$O(N/p))$
Parallel RedBlack PoissonGS	O(N) ×$O(N/p))$
Parallel Poisson GS SOR	O($\sqrt{N}$ × $O(N/p)$

p - processos N - tamanho do lado da grid

- **Comparação de Algoritmos Sequenciais**

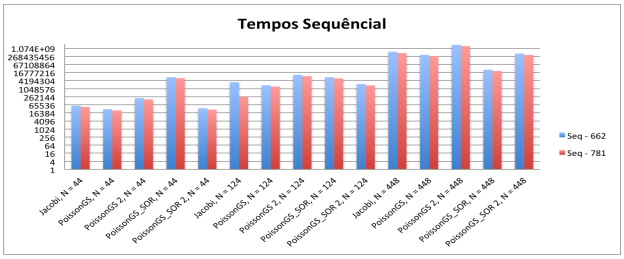


Fig. 17. Comparação entre os vários tempos conseguidos nos nós 781 e 662 (em pico-segundos)

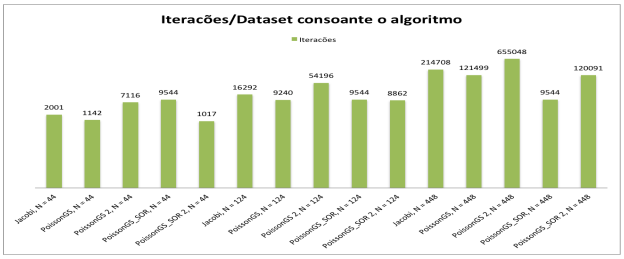


Fig. 18. Comparação entre o número de iterações e as várias implementações consoante o tamanho do *Dataset*

A seguinte tabela diz respeito ás implementações sequenciais:

Implementação	N	Iterações
Jacobi	44	2001
PoissonGS (Redblack)	44	1142
Poisson GS 2	44	7116
Poisson GS SOR	44	9544
Poisson GS SOR2	44	1017
Jacobi	124	16292
PoissonGS (Redblack)	124	9240
Poisson GS 2	124	54196
Poisson GS SOR	124	9544
Poisson GS SOR2	124	8862
Jacobi	448	214708
PoissonGS (Redblack)	448	121499
Poisson GS 2	448	655048
Poisson GS SOR	448	9544
Poisson GS SOR2	448	120091

A partir desses gráficos e tabela, verificamos que, entre todas as máquinas, que de forma geral os métodos baseados em de Gauss-Seidel com Successive Over-Relaxation são os mais rápidos para alcançar a convergência em termos de tempo de execução. Isso esta diretamente associado com o facto de possuirem um número menor de iterações totais que os outros métodos (o que é verdade, e pode ser confirmado com a consulta da tabela das iterações totais que cada método levou para alcançar a convergência apresentada). O próximo método mais rápido acaba sendo Gauss-Seidel, seguido por Jacobi. Esta ordem não é surpresa (e permanece tanto para as versões sequenciais quanto para as paralelas) porque Gauss-Seidel não é nada além de uma versão otimizada do método Jacobi, enquanto o método Gauss-Seidel com Successive Over-Relaxation fornece otimização ainda maior, como foi explicado na secção anterior deste relatório.

- **Comparação das Implementações Gauss-Siedel**

Dentro dos modelos apresentados para **Gauss-Siedel** a primeira implementação vai apresentar uma melhor performance a nível de tempos de execução e a nível de número de iterações,convergindo mais rapidamente para o *Steady-State*. Esta diferença no número de iterações está associada ao próprio algoritmo, pois enquanto a segunda implementação do **Gauss-Siedel** (Red-Black), apenas utiliza valores mais recentes (calculados nessa mesma iteração) para **metade** dos valores da grid (Black ou Red) o primeiro, apesar de parcialmente (2 vizinhos), já utiliza esses valores mais recentes tendo um impacto direto na convergência para um *Steady-State*.De notar ainda que apenas para o *Dataset* de maiores dimensões o tempo de execução total e número de iterações da primeira implementação **SOR** é inferior ao da primeira implementação, podendo afirmar que para *Datasets* de tamanho inferior esse algoritmo tem um bom desempenho.

- **Comparação Successive Over-Relaxation estático vs dinâmico**

Uma analise as tabelas globais apresentadas anteriormente, destrói alguns dogmas que tínhamos criado. Especulamos que o valor dinâmico seria traduzido de forma transversal em

melhores resultados, algo que não se prova verdadeiro. Contrariamente ao que seria de esperar, a versão estática apresenta melhores resultados para *Datasets* de maiores dimensões.

(N=448) ω= 1.00699680851;

1.00699680851 <1.5;

Isto acontece pois a formula utilizada para para calcular a taxa de relaxamento para valores N consideravelmente superiores vai tender para 1, isto vai implicar um maior número de iterações para convergir, consequentemente um maior tempo de execução.

Tudo isto nos leva a concluir que esta versão dinâmica *(SOR2)* tende a apresentar realmente melhores resultados apenas para matrizes de tamanhos inferiores, pois à medida que o **N** aumenta, a taxa de relaxamento tende a diminuir, havendo consequentemente uma convergência mais lenta.

- **Análise de speedups e tempos associados á implementação em Memória Partilhada**

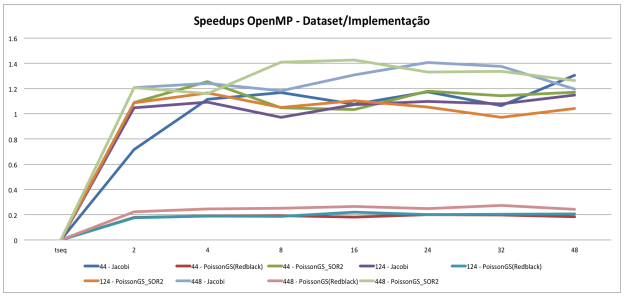


Fig. 19. Speedups OpenMP

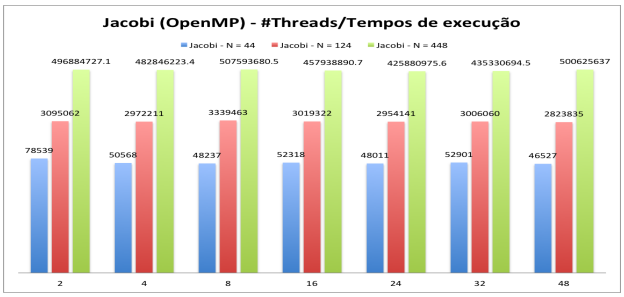


Fig. 20. Tempos da implementação Jacobi

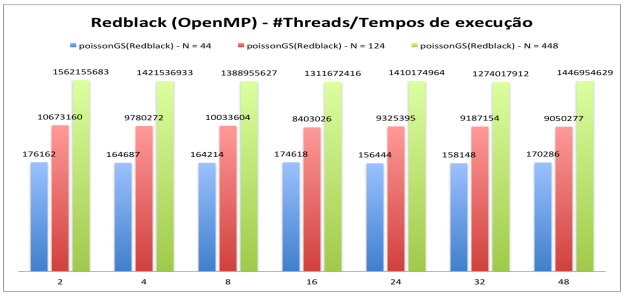


Fig. 21. Tempos da implementação Red-Black

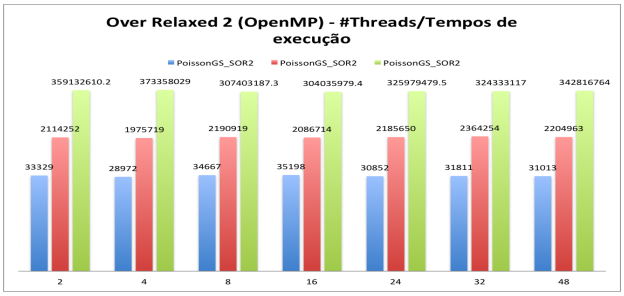


Fig. 22. Tempos da implementação Over-Relaxed

A partir destes gráficos, vemos que, entre todas as máquinas, o método de Gauss-Seidel com Successive Over-Relaxation (**SOR**) continua o mais rápido para alcançar a convergência em termos de tempo de execução. Uma implicação pratica disto pode ser traduzido por um número menor de iterações totais do que os outros métodos (o que através é do gráfico anterior pode ser comprovado). O próximo método mais rápido por ser a implementação de Gauss-Seidel, seguido pelo Jacobi. Esta ordem não é surpreendente (algo que já vi tinha sido revisto nas versões sequenciais permaneceu para as paralelas) porque Gauss-Seidel não é nada além de uma versão otimizada do método Jacobi, enquanto o método Gauss-Seidel com Sucessive Over-Relaxation fornece otimização ainda mais, como foi explicado nas secções anteriores do relatório.

Inicialmente previámos que o algoritmo PoissonGS Redblack escalasse com o aumento de *threads* no entanto tal não acontece. Após uma análise mais detalhada, presumimos que o *bottleneck* deste algoritmo esteja na alocação de threads após a troca de ”Red” para ”Black” e vice-versa que não consegue fazer face ao tempo ganho ao paralelizar a computação da difusão do calor. Quantos aos speedups, todos apresentam um comportamento normal, isto é, globalmente escalam quando aumentamos o número de *threads* acompanhado pelo aumento do *Dataset*, apesar destes mesmos speedups se encontrarem bastante longe do tecto ideal, ou seja, um speedup perfeito.

Em seguida, procuramos o número de iterações até a convergência de cada método com cada conjunto de dados.

A seguinte tabela diz respeito ás implementações em partilhada:

Implementação	N	Iterações
Jacobi	44	2001
PoissonGS (Redblack)	44	5906
PoissonGS SOR2	44	1017
Jacobi	124	16292
PoissonGS (Redblack)	124	45157
PoissonGS SOR2	124	8862
Jacobi	448	214708
PoissonGS (Redblack)	448	546035
PoissonGS SOR2	448	120091

Consegue-se perceber que o algoritmo que converge mais rapidamente trata-se do `PoissonGS SOR2`, tal como

referido acima, esta diferença em relação ao `PoissonGS (Redblack)`, trata-se da natureza do algoritmo introduzida pela variação do "hiper-parâmetro", representado por ω, que consiste numa taxa de relaxamento consoante o tamanho da matriz. Para valores muito grandes, ω, vai tender para 1, ou seja, vai aumentar o número de iterações. Quanto ao método de Jacobi, o seu número de iterações proporcionalmente com o tamanho da *grid* no entanto é o método que converge mais lentamente, ou seja, apresenta um maior número de iterações em todos os *Datasets* devido à natureza da sua implementação que utiliza sempre os valores da *mesh* da iteração anterior.

Análise de speedups e tempos associados ás implementações em Memória Distriubuída e em ambiente Híbrido

A seguinte tabela diz respeito ás implementações em memória distribuída e híbrida:

Implementação	N	Iterações
Blocos Estatáticos	44	6012
Broadcast Total	44	6012
Híbrido	44	6012
Linhas Dinâmico	44	6223
Blocos Estatáticos	124	45304
Broadcast Total	124	45304
Híbrido	124	45304
Linhas Dinâmico	124	48621
Blocos Estatáticos	448	545302
Broadcast Total	448	545302
Híbrido	448	545302
Linhas Dinâmico	448	599524

Os valores obtidos no que diz respeito ao número de iterações comparando com os algoritmos processados em memória partilhada, apresentam resultados inferiores, ou seja,o número de iterações aumentou em todos os algoritmos de forma global, e em parte também por isso o tempo de execução. Uma das motivações para tal ocorrer é a necessidade de enviar as fronteiras de cada bloco, estes valores vão permanecer estáticos o que ao longo do algoritmo se pode traduzir numa diminuição na taxa de convergência. Quanto à implementação em linhas dinâmica, converge mais lentamente do que as restantes pelos mesmos motivos supracitados, uma vez que a computação de uma linha necessita o envio de três linhas, ou, se for de mais fácil compreensão para o leitor, de um bloco de tamanho três, o que de forma óbvia se reflete num maior número de iterações.

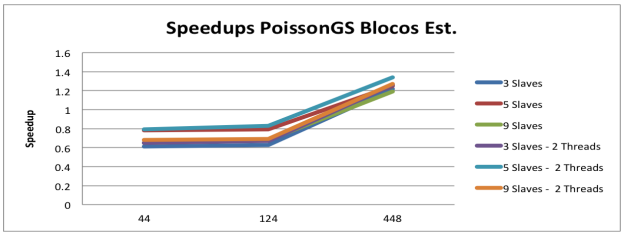


Fig. 23. Speedups da implementação com blocos estáticos

O gráfico da figura 23 mostra uma análise do comportamento da implementação por blocos em relação à implementação sequencial. A figura pretende mostrar o rácio entre o tempo de execução da implementação sequencial e o tempo de execução da implementação por blocos estáticos.
Consegue-se perceber que apenas começa a compensar, isto é, a obter speedup *(rácio maior do que 1)* quando usamos um *Dataset* de `448x448` a razão deste speedup "tardio" terá a ver com o *bottleneck* da comunicação entre processos, pois estamos a enviar uma grande quantidade de dados da *mesh* numa implementação em *farming*. Também consegue-se perceber que no que o uso de threads neste algoritmo híbrido apenas apresenta *speedups* maiores (cerca de 1,4) quando fazemos uso de 5 slaves e não 9 ou 3. Este facto dever-se-á ao compromisso "número de carga/*slaves* ser ótimo com um número médio como 5. Se adotássemos 9 slaves teríamos de aumentar a carga do bloco ou do *Dataset* em questão. Se ao contrário utilizássemos 3 slaves teríamos de reduzir à carga do bloco ou ao tamanho do *Dataset*.

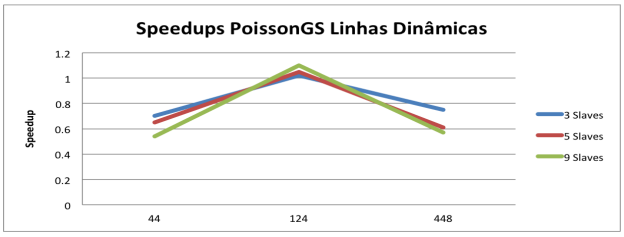


Fig. 24. Speedups da implementação com linhas dinâmicas

O gráfico da figura 24 representa o comportamento da implementação de linhas dinâmica, que consiste no envio de linhas mediante a disponibilidade dos *slaves*. Trata-se também de um rácio entre o tempo de execução da implementação sequencial e o tempo de execução da implementação atual.
Conclui-se pois que apenas para *Datasets* de tamanho "médio" como o de 124x124 é que começamos a obter melhorias (temporais) em relação à execução sequencial. Dever-se-à ao facto de enviar linha por linha para *Datasets* pequenos como o de 44x44 traduzir-se numa comunicação desnecessária, isto é, o tempo perdido na comunicação facilmente se traduziria em tempo de computação efetivamente necessária e aproveitada por apenas um processo.
Também dá para perceber que este algoritmo não será eficiente para *Datasets* "muito grandes" como o de 448x448 (onde também não temos speedups) provavelmente pelo facto do compromisso "tamanho/número de comunicações" não ser o ótimo. Isto é, ou reduzimos o tamanho do dataset ou reduzimos às comunicações fazendo uma implementação por exemplo em *heartbeat*.

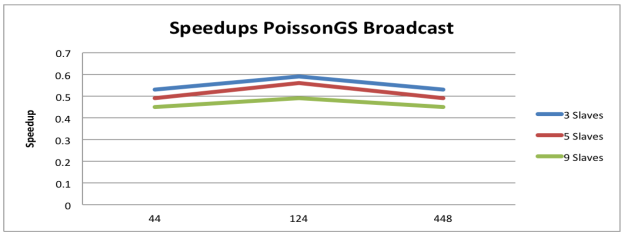


Fig. 25. Speedups da implementação em Broadcast

Em último, a figura 25 apresenta o comportamento tempo de execução da implementação fazendo uso do *Broadcast* do OpenMPI. Consiste no envio da matriz inteira sob a forma de vetor para todos os *slaves* sendo que estes apenas executas o seu trabalho.
Pela figura acima consegue se perceber que para os *Datasets* escolhidos não conseguimos obter um "rácio positivo", isto é, `>0`, provavelmente devido ao facto do bottleneck da comunicação não fazer compensar o uso deste mecanismo (message passing) proporcionado pelo OpenMP.
Ou seja, a carga enviada é demasiado grande. É de presumir que se o tamanho do dataset fosse menor iriamos obter *speedups*, no entanto como não é, este não compensa.

9. Conclusão e Trabalho Futuro

Olhando para todas as conclusões do capítulo anterior, acreditamos que Gauss-Seidel com *Overrelaxation* é o melhor método tanto na versão sequencial quanto na versão paralela. Infelizmente a produção do código de blocos provou-se infrutífera, não conseguindo produzir soluções validas e sólidas no panorama geral tornou o trabalho despendido nesses algoritmos um pouco ingrato, pois apenas pudemos supor como os blocos se comportariam em ambiente paralelo, e sem resultados que possam comprovar ou desaprovar as teorias desenvolvidas. Não tendo grande utilidade se não forem provadas.
Em suma, para esses casos acreditamos que poderíamos ter melhores resultados num ambiente de memoria partilhada, tendo possivelmente melhores *speedups*.

Nas versões implementadas em MPI (memória distribuída), a implementação que demonstra melhores resultados neste caso será a algoritmo que usa a distribuição por Blocos Estáticos, que foi posteriormente utilizada como base para a implementação em **ambiente híbrido**, demonstrando-se a que possui o melhor compromisso entre *comunicação/computação*.

Existe a consciência que uma abordagem em **Heartbeat** e não em **Farming** produziria provavelmente melhores resultados, diminuindo drasticamente o tempo de comunicação que é um grande **bottleneck** para as implementações criadas como solução para o problema em questão. Através do **Heartbeat** realizaríamos apenas o envio dos blocos estáticos através do vector auxiliar inicialmente computando dentro do próprio **slave** até este chegar a um **Steady-State** e depois enviando essa resposta para o **master**. Algo que infelizmente não implementamos, mas que acreditamos que produziria resultados de interesse.

Estamos também cientes que existem outros métodos que não foram testados aqui, como o **Gradiente Conjugado** entre outros, que poderiam até proporcionar resultados superiores aos obtidos. Todavia, para aqueles que foram realmente implementados e testados, estamos crentes das nossas afirmações.

10. Referências

- `http://adl.stanford.edu/cme342/Lecture_Notes_files/lecture10-14.pdf`
- `Applied Numeric Linear Algebra James W. Demmel`