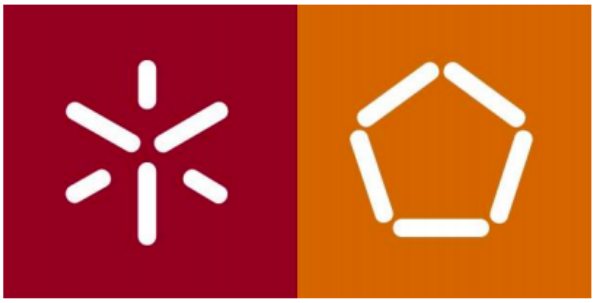

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Algoritmos Paralelos

Room Assignement Problem

Trabalho Prático 1

Eduardo Lourenço da Conceição (A83870)
Rui Nuno Borges Cruz Oliveira (A83610)

27/03/2021
Braga

Índice

1 Introdução

Para o primeiro trabalho prático de Algoritmos Paralelos, foi-nos proposta a implementação de uma solução para o problema de atribuição de quartos (*Room Assignement Problem*). A solução que implementaríamos tem como base a solução implementada pelo Professor Rui Ralha, com base nos Métodos de Monte Carlo, e foi escrita na linguagem de programação C. Neste relatório iremos expôr as decisões que tomamos para a implementação da solução, nomeadamente com a implementação de um algoritmo *greedy* para resolver o problema, como paralelizamos o código e, no final, iremos explorar os resultados dos vários testes que realizamos e as conclusões a que chegamos com os mesmos.

2 *Room Assignement Problem*

O problema de atribuição de quartos parte do princípio que temos *N* pessoas, sendo *N* par, que vamos distribuir em pares em *N/2* quartos. Estas pessoas têm um coeficiente de incompatibilidade umas com as outras, e o objetivo é encontrarmos uma combinação que minimize o custo total da solução, em termos do total de incompatibilidade.

Para representarmos o conjunto de pessoas e as suas respetivas incompatibilidades, utlizamos uma matrix $N \times N$, dita *D*. Esta matriz é simétrica, os coeficientes na diagonal são 0 e em *D(i,j)* (e, por extensão, *D(j,i)*), temos o coeficiente representativo do grau de incompatibilidade entre a pessoa *i* e a pessoa *j*.

Tendo portanto um enunciado do problema, partimos agora para explorar a complexidade do mesmo.

2.1 Complexidade

Para podermos procurar uma solução para este problema, devemos olhar primeiro para o espaço de soluções possíveis.

Assumamos que temos *N* pessoas. Partamos de uma qualquer pessoa i_0. Para se juntar a outra, tem de escolher uma entre *N-1* hipóteses. Quando a sua escolha for feita, temos o primeiro par (i_0,j_0) formado. Seguindo para o próximo, a partir de um qualquer i_1 poderemos escolher entre *N-3* hipóteses (uma vez que não podemos escolher i_0, j_0 ou i_1), e daqui formamos o segundo par. Se seguirmos esta lógica com os seguintes pares, temos que o espaço de soluções possíveis contém:

$$(N-1) \times (N-3) \times (N-5) \times ... \times 3 \times 1$$

soluções.

O aspeto mais importante que poderemos retirar daqui é que o crescimento extremamente rápido do número de soluções possíveis. Para uma matriz relativamente pequena 100x100 temos 2.72×10^{78} combinações possíveis. Como podemos ver que o problema é NP, não podemos tentar resolvê-lo com métodos de força bruta, e muito dificilmente utilizando um método que obtenha consistentemente uma solução ótima. Assim, viramo-nos para uma solução heurística.

2.2 Resolução

No âmbito do programa de Algoritmos Paralelos, foi utilizado um **Método de Monte Carlo** como base, que foi expandido utilizando *Simulated Annealing*. Por fim, elaboramos também um algoritmo *greedy* simples e determinístico contra o qual poderemos contrastar as soluções obtidas com os dois primeiros métodos.

2.2.1 Obtenção de uma Solução Inicial

Para obter uma aproximação inicial S_0, começamos por criar uma matriz inicialmente com os valores:

[[0,1] , [2,3] , [4,5] , ... , [N-2,N-1]]

De seguida, iteramos num ciclo de 0 a $\frac{N}{2}$ com a vairável i, gerando em cada iteração quatro índices, dois entre 0 e $\frac{N}{2}$ (x_0 e x_1) e os outros dois ou 0 ou 1 (y_0 e y_1), e fazemos a troca entre os elementos $S[i][0]$ e $S[x_0][y_0]$ e os elementos $S[i][1]$ e $S[x_1][y_1]$. Assim, no final deste algoritmo, teremos uma solução inicial aleatória que poderemos utilizar na procura de uma solução heurística.

2.2.2 Algoritmo

A primeira implementação feita segue o seguinte algoritmo:

Tendo uma matriz *D*, com *N* linhas, construímos uma matriz *Nx2* de emparelhamentos aleatórios. Esta servirá como a solução inicial, a partir da qual obteremos uma solução heurística. O custo da solução inicial é dito C_0

De seguida, vamos fazer uma troca aleatória entre dois quartos vizinhos, ficando:

$$(i_k, j_k), (i_{k+1}, j_{k+1}) \implies (i_{k+1}, j_k)(i_k, j_{k+1})$$

Ao fazer isto, calculamos a diferença entre o custo antes da troca e depois da troca, Δ:

$$\Delta = D(i_k, j_k) + D(i_{k+1}, j_{k+1}) - (D(i_{k+1}, j_k) + D(i_k, j_{k+1}))$$

Se o custo da nova solução for maior do que o da solução que tínhamos antes da troca, ou seja, se $\Delta \geq 0$, então rejeitamos esta solução. Caso contrário, aceitamos a nova solução e atualizamos o custo:

$$C_i := \Delta + C_{i-1}$$

O algoritmo para quando forem feitas *MAX_ITER* iterações consecutivas em que a nova solução é rejeitada.

Em teoria, este algoritmo permite-nos obter uma solução heurística com o custo minimamente otimizado e em tempo útil. A solução pode também, em teoria, ser mais próxima da ótima para valores maiores de *N* aumentando o *MAX_ITER*, algo que testamos e cujos resultados podemos ver mais à frente.

2.2.3 Algoritmo com *Simulated Annealing*

Simulated Annealing é uma técnica probabilística de otimização com o objetivo de simular a diminuição da energia, ou temperatura, do sistema ao longo da execução do algoritmo. Em geral, quanto maior for a temperatura T, maior é a permeabilidade do sistema a mudanças.

Em prática, o que vai ser diferente é a forma como vemos se uma potencial solução é aceitável ou não. Para além de verificarmos se Δ é menor que zero, também verificamos a seguinte condição:

$$e^{-\frac{\Delta}{T}} \geq r$$

sendo r um número real aleatório entre 0 e 1. Caso qualquer uma das duas condições seja verificada, aceitamos a solução. A temperatura é então reduzida.

Isto deverá resultar em execuções do algoritmo que produzem soluções com custos mais próximos dos das soluções ótimas, mas o número de passos necessários para completar o algoritmo variará muito mais, tendo tendência a produzir soluções com quatro ou cinco vezes o número de passos da primeira implementação, em algumas ocasiões.

2.2.4 Algoritmo *Greedy*

Para além dos dois algoritmos não determinísticos, implementamos também um algoritmo *greedy* determinístico, de modo a podermos comparar a solução que este pode produzir em relação aos métodos de Monte Carlo.

O funcionamento deste é bastante simples: tendo uma matriz D, $N \times N$, e um *array* auxiliar *aux* de tamanho *N*, inicialmente com todos os valores a 0, percorremos a matriz D linha a linha, e, para cada linha i, procuramos o menor coeficiente $D(i, j)$. Quando o encontramos, o que implica percorrer a linha toda, adicionamos o par (i, j) à matriz das soluções, e atualizamos o custo:

$$C_{total} := C_{total} + D(i, j)$$

Agora, atualizamos o *array aux*, com *aux[i] = 1* e *aux[j] = 1*. Este *array* auxiliar vai-nos permitir, em iterações posteriores, ignorar as linhas que já estão emparelhadas (neste caso, a linha j).

A utilização de *aux* permite-nos consultar as linhas já emparelhadas em O(1), e faz com que a complexidade deste algoritmo seja proporcional a $N/2$ ($O(\frac{N}{2})$), uma vez que só teremos de verificar metade das linhas da matriz. No entanto, temos de entender que cada passo neste caso é percorrer uma linha inteira da matriz, o que torna a complexidade em $O(\frac{N}{2} \times N)$. Desta forma podemos ver que a complexidade deste algoritmo é quadrática, o que é consideravelmente mais do que os outros dois, o que também poderemos confirmar mais tarde.

Não obstante, não é esperado que a solução produzida por este algoritmo seja melhor que a dos outros dois, especialmente quando aumentamos a amplitude dos valores da matriz D e o seu tamanho.

3 Implementação da Solução

Como dissemos anteriormente, a implementação do algoritmo foi feita em C. Nesta secção explicaremos como fizemos a implementação dos três algoritmos, bem como algumas otimizações que experimentamos.

As implementações são adaptações diretas do algoritmo que o professor implementou em MATLAB. Cada um dos algoritmos é implementado numa função com o mesmo nome que o algoritmo (*Rooms*, *RoomsSA* e *RoomsGreedy*), e têm como auxiliar uma função para gerar uma solução inicial, chamada *randperm*.

Em termos de tipagem e assinatura das funções, são as seguintes:

- *int** randperm(size_t N)*: Função que recebe como parâmetro um tamanho *N* e devolve uma matriz $\frac{N}{2} \times 2$ com pares aleatórios com os números de 0 a *N-1*, sob a forma de um duplo apontador para inteiros.
- *int** rooms(int** D, size_t N, int* cost, int* steps, int max_iter)*: Implementação do algoritmo básico que recebe uma matriz *D* e o seu tamanho *N*, juntamente com um apontador onde vai guardar o custo da solução e o número de passos e um inteiro representativo do número máximo de iterações que podemos fazer sem haver alterações na solução, e devolve a solução (matriz $\frac{N}{2} \times 2$, sendo ela também um duplo apontador para inteiro).
- *int** roomsSA(int** D, size_t N, int* cost, int* steps, int max_iter, double temp_multiplier)*: Semelhante ao caso anterior, mas aqui passamos também o multiplicador que altera sucessivamente o valor da temperatura *T* na versão com *Simulated Annealing*.
- *int** roomsGreedy(int** D, size_t N, int* cost, int* steps)*: Com o mesmo funcionamente que as outras duas implementações, mas não passamos o número máximo de iterações ou o multiplicador da temperatura, uma vez que este algoritmo não utiliza nenhum dos dois.

A implementação destas funções encontra-se no ficheiro "rooms.c".

3.1 Otimizações

Em termos de otimizações, um aspeto que nos chamou a atenção foi a utilização da função *exp()*, do módulo "math.h", uma das bibliotecas standartizadas de C, na implementação com *Simulated Annealing*. Esta função, que dado um *double x*, calcula e^x, consome algum tempo de execução, pelo que tentamos arranjar uma alternativa para a implementar, que fosse ligeiramente mais rápida, que poderia prescindir de alguma precisão numérica, desde que mantivesse o comportamento aproximado da função exponencial.

Em primeiro lugar, sabemos que:

$$e^x = \lim_{n \to +\infty} (1 + \frac{x}{n})^n$$

Mais ainda, como sabemos que, no contexto do problema, o valor de x é igual a $-\frac{\delta}{T}$, substituindo na igualdade anterior, ficamos com:

$$e^{-\frac{\Delta}{T}} = \lim_{n \to +\infty} (1 - \frac{\Delta}{T \times n})^n$$

A partir deste limite, podemos fazer uma aproximação rudimentar mas suficiente do resultado utilizando a seguinte função [3]:

```
static inline
double fast_inverse_exp_256(double x, double T)
{
    x = 1.0 - ( x  / ( T * 256.0) );
    x *= x; x *= x; x *= x; x *= x;
    x *= x; x *= x; x *= x; x *= x;

    return x;
}
```

No entanto, quando a utilizamos em testes, acabamos por reparar que os ganhos apenas são significativos em situações que temos valores de x (correspondente ao Δ) bastante elevados, muito maiores do que os valores que seriam relevantes para os testes. Mais ainda, a otimização só é relevante se o programa for compilado com as *flags* de compilação *-O3* ou *-O2* (para o GCC). Como tal, a função não é utilizada mas o seu código encontra-se também no ficheiro "rooms.c".

Outro problema que estavamos a encontrar surgia apenas quando utlizávamos tamanhos pequenos (para $N < 60$) para a matriz *D* na versão com *Simulated Annealing*. Quando o valor de T ficava suficientemente baixo, a certas alturas o programa entrava num ciclo infinito, por razões que nós próprios não conseguimos explicar, mas que suspeitamos estarem relacionadas com problemas de arredondamento na atualização do T. Como tal, decidimos fazer um *check* na atualização do T, tal que quando este é menor que um qualquer valor aleatoriamente pequeno (optámos por 10^{-8}), em vez de T ser atualizado com o multiplicador *temp_multiplier*, colocamos o seu valor imediatamente a 0, pelo que, depois, na verificação da exponencial, assumimos que a mesma é igual a 0 e que o sistema chegou ao seu estado de menor energia (ou seja, chega ao ponto em que o seu comportamento é igual ao do algoritmo que não usa *Simulated Annealing*).

Um fator interessante que se provou, ao longo do desenvolvimento do trabalho prático, bastante problemático, foi a geração de números aleatórios. A função **rand()**, standart no C, não é de todo compatível com paralelização. A sua implementação continha por trás uma forma de *locking* que prevenia que mais do que um processo tentasse gerar um número aleatoriamente, pelo que tivemos de procurar uma alternativa. A função **rand_r()** foi a solução para o nosso problema, pois prescindia do problema de *locking*. Difere também da função *rand()* na medida que não precisa de uma função auxiliar para inicializar a *seed*, pois a *seed* é-lhe passada sob a forma de um apontador para *unsigned int*.

3.2 Paralelização

Em termos de paralelização, temos de analisar quais são os aspetos que são mais propícios a paralelização nos algoritmos.

Olhando primeiro para o algoritmo original e a sua variante com *Simulated Annealing*, devido ao facto que cada iteração vai aceder a partes independentes mas inicialmente desconhecidas da matriz, uma vez que são aleatórias, não temos nenhum trabalho aqui propício a paralelização, porque iria ser necessário implementar algum mecanismo de *locking* para as várias *threads* ou processos saberem que quartos estão a ser alterados, o que iria indubitavelmente degradar a *performance* e invalidar qualquer esforço de paralelização.

Em contraste, o algoritmo *greedy* parece, à primeira vista, ser mais paralelizável. Uma vez que a matriz D é percorrida linha a linha, poderíamos atribuir algumas linhas a cada processo e processá-las independentemente. No entanto, na verdade, isto também não é possível, uma vez que, quando olhamos para uma linha i, sabemos que vamos esolher um elemento qualquer dessa linha e invalidar a pesquisa numa linha que pertença ao intervalo $[i + 1, N - 1]$. Assim, não sabemos também de princípio que linhas devemos ou não ver, pelo que também não existe uma forma boa de paralelizar este algoritmo.

Resta-nos o quê, então? Bem, os testes que nós fazemos a uma dada matriz apenas partilham entre eles variáveis que para todos os efeitos são *read-only*, como a matriz *input*, as suas dimensões, o número máximo de iterações e por aí fora. Mais ainda, os teste são completamente independentes uns dos outros, pelo que são embaraçosamente paralelizáveis. Como tal, são os candidatos ideiais a paralelização.

Para o fazer, simplesmente utilizamos um ciclo *for*, que é paralelizado utilizando a cláusula *#pragma omp parallel for* do *OpenMP*. Nós utilizamos, de modo a passar por referência os valores do custo e dos passos necessários para completação às funções, duas variáveis que são declaradas fora da *scope* do ciclo, pelo que estas são declaradas como privadas na cláusula.

Por fim, será interessante falarmos do escalonamento da sua execução. A implementação que utiliza *Simulated Annealing* apresenta uma característica interessante no facto que, em alguma execuções, realiza muitas mais iterações do que em outras, o que é normal, mas é importante notar que, se mapearmos as execuções um para um com *threads* do *OpenMP*, teremos *threads* que realizam muito mais trabalho do que outras, o que degrada a performance do programa, pelo que optamos por escolher um escalonamento **dinâmico** para esta versão do algoritmo. Para os outros dois algoritmos, como variam muito menos em termos de número de *steps* (em particular, o *greedy* não varia de todo para a mesma matriz *input*), e, portanto, cada *thread* executa aproximadamente o mesmo trabalho, utilizamos escalonamento **estático**.

4 Testes

Nesta secção explicaremos os testes que realizámos, o seu ambiente e as conclusões a que chegamos com os mesmos.

4.1 Condições de Teste

O programa foi testado no *SeARCH Cluster*[2] da Universidade do Minho. O nodo utilizado (662) contém dois *Intel Xeon Processors E5-2695 v2* de 12 *cores*[1], sempre requisitado com 1 nodo e 8 processos, na fila *mei*.

Para testar várias combinações, testamos com os seguintes valores:

- Número de Threads: 1,2,4,8,16,32;
- Tamanho da Matriz Input: 100,200,300,...,9900,10000;
- Multiplicador da Temperatura: 0.999 (*default*), 0.8, 0.5, 0.2;
- Número Máximo de Iterações sem mudanças: 50, 100 (*default*), 200, 500

Mais ainda, a matriz *input* é gerada aleatoriamente, sendo ela simétrica com coeficientes entre 1 e 10 e a diagonal igual a 0.

Os resultados apresentados são médias para o custo e para o número de passos, de modo a podermos normalizar os resultados, e o melhor tempo para o caso do tempo de execução, para apenas um dos *datasets*.

4.2 Resultados

Nesta secção avaliaremos os resultados que obtivemos para várias configurações e procuraremos uma explicação para os mesmos.

Para simplificar a interpretação e análise dos resultados, em todos os casos exceto na avaliação do tempo de execução utilizaremos a versão sequencial.

4.2.1 Custo da Solução

Olhando para o seguinte gráfico:

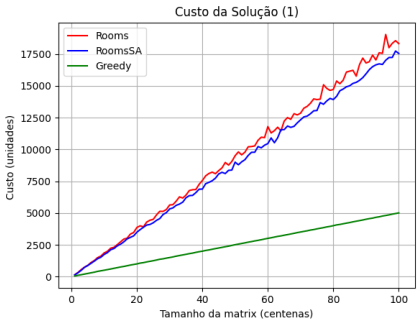

Figure 1: Custo da Solução

Como podemos obervar, a solução gerada pelo algoritmo com *Simulated Annealing* é em geral mais "barata" que a gerada pelo algoritmo normal, mas ambas são consideravelmente mais caras que a solução gerada pelo algoritmo *greedy*.

Para podermos ver como a alteração do *MAX_ITER* e do *TEMP_MULTIPLIER* afetam o custo, podemos olhar para os dois seguintes gráficos:

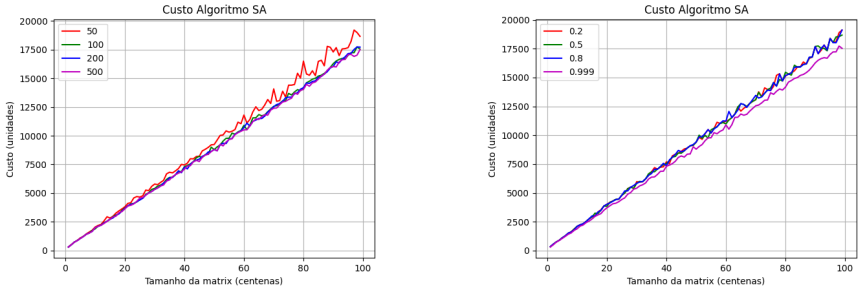

Figure 2: Comparação do Custo com vários valores de *MAX_ITER* (esquerda) e *TEMP_MULTIPLIER* (direita)

Como podemos ver, o custo da solução aumenta em geral para valores de *MAX_ITER* e *TEMP_MULTIPLIER* mais baixos, apesar de não ser uma diferença muito notória. Ambos fazem sentido. Quando diminuímos o *TEMP_MULTIPLIER* fazemos com que o sistema "perca energia" mais rapidamente, ou seja, demora menos tempo até o algoritmo se comportar de maneira mais semelhante ao algoritmo sem *Simulated Annealing*. Quanto ao *MAX_ITER*, ao permitirmos que hajam menos iterações consecutivas sem alterações vamos dar menos "tempo" para encontrar uma alteração, o que fará com que algumas alterações benéficas não sejam consideradas e, portanto, temos um custo mais elevado. O inverso acontece quando aumentamos o *MAX_ITER*.

4.2.2 Número de Iterações

Podemos comparar o número de iterações entre os três algoritmos no seguinte gráfico:

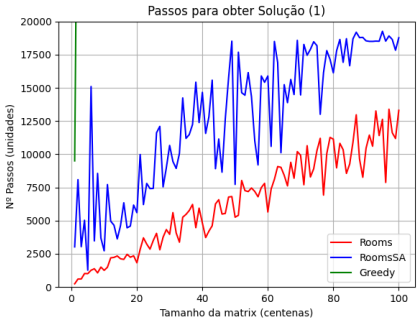

Figure 3: Número de Iterações

Como podemos ver, o número de iterações para algoritmo *greedy* cresce imensamente rápido. Se nós fizermos *zoom out* no gráfico, podemos até ver que o seu crescimento é plenamente quadrático, o que seria de esperar visto que, como dissemos anteriormente, este verifica $\frac{N}{2} \times N$ elementos da matriz D.

Olhando agora para os outros dois, podemos ver que o número médio de iterações varia imenso, o que é de esperar com Métodos de Monte Carlo, mas são mais ou menos lineares com o tamanho da matriz *input*. Com *Simulated Annealing*, no entanto, podemos ver que o número de iterações é sempre maior, o que é de esperar, visto que com *SA* temos a possibilidade de aceitar muitas mais configurações por aceitarmos configurações que aumentam o custo.

Avaliemos aqui também o efeito da variação do *MAX_ITER* e do *TEMP_MULTIPLIER*.

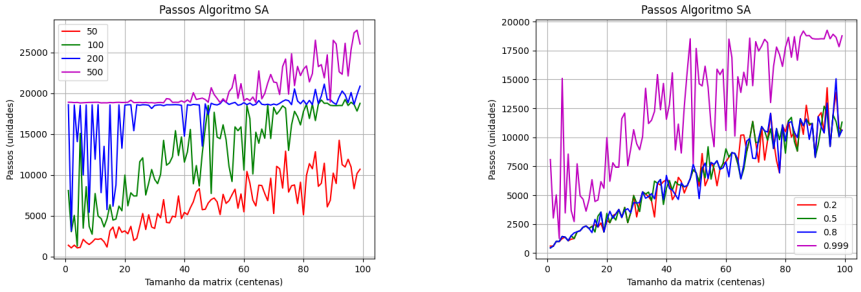

Figure 4: Comparação do Custo com vários valores de *MAX_ITER* (esquerda) e *TEMP_MULTIPLIER* (direita)

Para o *MAX_ITER*, como é óbvio, com o seu aumento, aumentam também o número de iterações, e o inverso acontece com a sua diminuição. Como se sucedia com o custo, quando diminuímos o *TEMP_MULTIPLIER*, também diminuímos o número de iterações, pois o comportamento aproxima-se do do algoritmo "rooms" que não usa *Simulated Annealing*. Como pudemos ver na figura 4, a diferença nos passos é muito mais notória do que no custo. Se prestarmos particular atenção à linha para $MAX\_ITER = 200$, podemos ver que é muito constante a partir de N = 3000. Acreditamos que isto se deve apenas a uma anomalia na medição e não é um aspeto relevante a ter em conta, uma vez que não se conforma de modo nenhum com o resto dos dados recolhidos.

4.2.3 Tempo de Execução

O aspeto mais relevante a analisar aqui é o *speedup* relativo à versão sequencial. Para este efeito, utilizamos os valores para N=5000, uma vez que conclusões que retiremos daqui poderão ser extendidas para os outros tamanhos.

Seguindo a **Lei de Amdahl**, uma vez que a fração de trabalho sequencial deste programa é nula, o *speedup* teórico que poderemos obter é diretamente proporcional ao número de *processos*. Os resultados que obtivemos foram os seguintes:

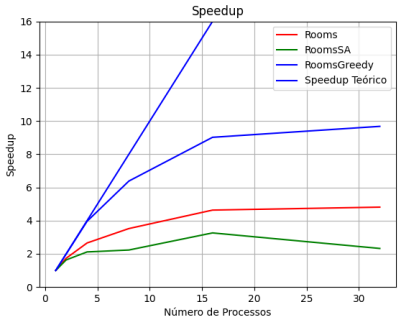

Figure 5: *Speedup* obtido

Como podemos ver, todos os *speedups* estão aquém do valor teórico. Para este facto podemos encontrar duas explicações:

- Em primeiro lugar, nós apenas pedimos oito processos, uma vez que o *cluster* esteve bastante carregado nos dias em que pudemos realizar os testes. Mais ainda, como pedimos apenas um *nodo*, a partir das 8 *threads* obrigamos a utilização de *hyper-threading*, o que justifica o *drop off* a partir das 16 *threads*;

- O *RoomsSA* e o *Rooms*, em particular, têm uma carga de trabalho que é um pouco errática, o que significa que a execução vai estar limitada pela maior execução em termos de *steps*. Mesmo tentando mitigar o problema com a utilização de escalonamento dinâmico, este tem na mesma um efeito notório. Isto acaba também por se refletir no *speedup* do *RoomsGreedy*, que, por ser o com a carga de trabalho mais igualmente distribuída, acaba por ser o com o melhor *speedup*.

No entanto, como podemos ver, conseguimos ter na mesma *speedups* significativos que justificam a paralelização do código da forma que optamos.

5 Conclusão

Em geral acreditamos ter alcançado o objetivo do trabalho. Conseguimos uma implementação dos vários algoritmos pedidos e pudemos fazer uma análise sucinta do seu comportamento quando certos parâmetros são alterados. Mais ainda, conseguimos paralelizar o algoritmo de uma forma que nós acreditamos ser satisfatória.

No entanto, existem alguns aspetos que poderiam ser melhorados. O algoritmo *greedy*, por exemplo, poderia ser implementado de uma forma menos simplista de modo a tomar partido do facto que a matriz é simétrica para diminuir o número de *steps*.

6 Referências

[1] *Intel Xeon Processor E5-2695 v2*. URL: https://ark.intel.com/content/www/us/en/ark/products/75281/intel-xeon-processor-e5-2695-v2-30m-cache-2-40-ghz.html.

[2] *SeARCH Cluster*. URL: http://search6.di.uminho.pt/wordpress/.

[3] *Using Faster Exponential Approximation*. URL: https://codingforspeed.com/using-faster-exponential-approximation/.