Universidade do minho
Mestrado Integrado em Engenharia Informática

- Paradigmas de Computação Paralela – Trabalho Prático 1

Tema: Simulated Annealing

***

Docentes:

João Luís Ferreira Sobral

Desenvolvedores:

Daniel Cerveira Furtado Malhadas

A72293

Alexandre Romain Lucas Ventosa da Silva

A72502

Índice

1.Introdução

O presente documento serve de relatório ao primeiro trabalho prático da unidade curricular de paradigmas de computação paralela. Com este projeto teve-se a intenção de investigar a eficiência e qualidade de soluções paralelas do algorítmo "simulated annealing" sendo que essa paralelização foi alcançada com recurso a diretivas OpenMP aplicadas à linguagem C.

Começaremos por explicar uma possível implementação sequencial e uma paralela, seguindo-se depois um estudo aprofundado dos resultados obidos com o intuito de comparar as duas implementações.

Note-se que o algorítmo referido tem um caracter geral, podendo ser aplicado em diversos problemas de grande complexidade. Para este projeto, decidiu-se aplicar o algorítmo no contexto do "Traveling Salesman Problem".

2.Versão Sequencial

2.1.Utilização

O algorítmo "simulated annealing" tem o propósito de minimizar uma determinada função objectivo. No caso deste projeto, como referido na introdução, essa função objetivo será o caminho a percorrer passando por um conjunto de cidades. Temos então o propósito de saber qual o caminho que, passando por todas as cidades (e voltando à inicial) é o mais curto.

Para a utilização da versão sequencial é necessário que se forneça dois argumentos como input:

- **Conjunto de Cidades:** O conjunto de cidades referido deverá estar indicado num ficheiro de texto de input com a extensão ".in" onde cada linha irá caracterizar por completo uma cidade fornecendo, separados por espaços, os seguintes três parâmetros:
  - **[ID] [Cordenada x] [Cordenada y]**
- **Número de Iterações até Convergir:** Este número indica a quantidade de iterações que devem ocorrer sem que sejam encontrados novos caminhos relevantes até o programa terminar e interpretar a solução atual como a melhor que conseguiu encontrar. Fácilmente se entende então que, maior número de iterações irá levar a melhores soluções, mas por consequência leva também a um maior tempo de execução.

2.2.Explicação do Algorítmo

Segue-se o pseudo-código da implementação sequencial realizada:

```
S <- Solução inicial
T <- T inicial
Tentativa <- 0
For Tentativa to Maximo_Iterações :
        S' <- Nova solução aleatoria
        If(S' é aceite como um caminho relevante de acordo com T):
                If(S' < S):
                        S <- S'
                EndIf
                Tentativa <-0
        Else Tentativa <- Tentativa+1
        EndIf
        T <- T*Delta_T
EndFor
```

Ao observar este pseudo-código notamos que, apartir de uma solução inicial aleatória o sistema avança tentando a cada passo (iteração) reduzir o valor da distância a percorrer (neste caso em que nos referimos ao TSP).

Fácilmente se observa que seria talvez mais fácil para depois parallelizar se a condição do ciclo for (`For Tentativa to Maximo_Iterações`) fosse antes uma condição com fim definido (esta não o é pois voltamos a por `Tentativa` a zero ao encontrar um caminho relevante), no entanto isso, a nosso ver, tornaria o algorítmo menos "confiável", visto que as soluções obtidas não podiam ser confiadas a não ser que o utilizador soubesse já desde raiz um número certo de iterações do ciclo necessárias para ter uma "boa" solução, no entanto se o utilizador já tem essa informação, talvez então não tenha verdadeiro interesse em correr o programa sequencial. Por esse motivo optamos por esta interpretação do algorítmo baseada no pseudo código descrito no livro **"Parallel Programming in C with MPI and OpenMP"** por **Michael J. Quinn**. Que consideramos ser uma implementação mais "relevante" do que outras que fomos encontrando em outras fontes visto que esta, embora não nos dê garantia que a solução seja ótima dá-nos uma garantia que será próxima da ótima.

3.Versão Parallela

3.1.Utilização

Para a utilização da versão parallela é necessário que se forneça três argumentos como input:

- **Conjunto de Cidades:** Ver descrição no capítulo da implementação sequencial.
- **Número de Iterações até Convergir:** Ver descrição no capítulo da implementação sequencial.

- **Resultado Obtido com a Versão Sequencial:** Deverá ser fornecida a distância mais curta (output da versão sequencial). Visto que com este projeto temos o propósito de perceber quão mais rápido conseguimos em paralelo chegar à solução alcançada sequencialmente, por este motivo, depois de alcançado este valor, não faz sentido continuar e o programa

2.2.Explicação do Algorítmo

Segue-se o pseudo-código da implementação parallela realizada:

```
N <- Numero de threads

S[N] <- Fornecer solução aleatória (e diferente) para cada thread

T[N] <- Cada thread terá a sua temperatura privada (igual inicialmente)

Tentativa <- 0; Sequencial_alcançado <- 0

For Tentativa to Maximo_Iterações :

        Todas as threads em paralelo :

                n <- número da thread; passo <- 0; tentativa_T[n] <- Tentativa

                For (passo to npassos && tentativa_T[n] < Maximo_Iterações &&
                sequencial_alcancado = 0) :

                        S' <- Nova solução aleatoria

                        If(S' < valor_sequencial):

                                Sequential_alcancado++ (alcançamos o valor sequencial)

                        EndIf

                        If(S' é aceite como um caminho relevante de acordo com T[n]):

                                If(S' < S[n]):

                                        S[n] <- S'

                                EndIf

                                Tentativa[n] <-0

                        Else Tentativa[n]++

                        EndIf

                        T[n] <- T[n]*Delta_T

                EndFor

        Fim de bloco paralelo

        S <- encontrar melhor solução até agora entre todas as threads

        S[N] <- Fornecer uma modificação aleatória dessa solução a cada thread
(modificando tentativa[N], T[N], etc de acordo com os valores da thread que encontrou o melhor
caminho)

        EndFor
```

Consegue-se perceber que, a implementação sequencial está na base desta versão paralela tendo-se que as mudanças necessárias de fazer no código sequencial em sí foram mínimas (que é exatamente o objetivo das diretivas OpenMp). No entanto algumas coisas mudaram para apoiar a nossa estratégia paralela. Esta estratégia baseou-se principalmente no seguinte princípio: Se as soluções intermédias que se for encontrando forem melhores então iremos alcançar melhores soluções mais rápido, ou seja, mais rapidamente alcançaremos a solução obtida com o algorítmo sequencial. Para garantir que as soluções intermédias são melhores decidiu-se que, cada thread iria, a partir de uma solução inicial diferente, evoluir o sistema independentemente encontrando as suas soluções independentes. Ao fim de um certo número de iterações (descrito no pseudo-código como `npassos`) voltamos a ter uma só thread que irá determinar o melhor caminho encontrado entre todas as threads. O processo repete-se então, fornecendo-se a cada thread uma variação aleatória desse melhor caminho para que cada thread possa de novo evoluir independentemente.

Intuitivamente concluímos que, se temos várias threads ao mesmo tempo, partindo de soluções iníciais distintas, temos maior probabilidade de chegar a melhores soluções mais rápido do que com a versão sequencial. E isto será sempre verdade para o número de iterações, quanto mais threads, menos iterações iremos necessitar. A problemática desta solução para o tempo de execução estará somente no overhead do fork&join de um grande número de threads. E é para isso que temos a variável `npassos` que irá indicar o número de iterações até acontecer de novo um join. Não queremos que este número seja excessivamente grande, caso contrário, embora tenhamos menos overhead de fork&join, perdemos a vantagem da estratégia descrita anteriormente, mas se for demasiado pequeno então o bottleneck do fork&join será absurdo. Concluímos então que para cada input, `npassos` deve ser modificado de acordo, pois para pequenos inputs, não faz sentido um grando número, pois a convergência alcança-se rapidamente. Outra coisa a ter em conta é o número de threads. Para um elevado número de threads necessitaremos de menos iterações para convergir (em teoria), no entanto também teremos maior overhead fork&join se reduzirmos `npassos`. Por esta razão, para os valores apresentados no próximo capítulo `npassos` foi estudado e modificado de acordo com o input a receber de forma a ser obtida a melhor performance possível.

4.Análise de Resultados

Depois de implementadas as duas versões (sequencial e paralela) teve-se a intenção de comparar os seus tempos de execução. Para tal foram gerados, com recurso a um programa escrito em C, quarenta e um ficheiros de teste aleatórios, sendo que estes se integram em duas categrorias distintas:

- **Pequenos Inputs** – dez ficheiros, cada um com mais cem cidades que o anterior, sendo que o primeiro tem exatamente cem cidades e o décimo tem mil.
- **Grandes Inputs** – trinta e um ficheiros, cada um com mais mil cidades que o anterior, sendo que o primeiro tem exatamente dois mil e o último trinta e dois mil.

Como se pode notar foi realizado um grande número de testes com o intuito de que as nossas afirmações quanto à perfomance das duas implementações fosse o mais fiável possível. Sendo que, para a realização dos testes criou-se um script .sh que funciona da seguinte forma:

Para cada um dos quarenta e um ficheiros de teste, executa a versão sequencial cem vezes fazendo-se no fim uma média do tempo (Ttotal/cem). De seguida é dado o valor que a versão sequencial devolveu como argumento para correr a versão paralela. A versão paralela é então executada também cem vezes por cada core físico presente na máquina sendo que a cada uma destas cem medições se aumenta o número de threads a utilizar no programa com recurso à variável de ambiente **OMP_NUM_THREADS**.

Todos os testes realizados e resultados apresentados neste capítulo foram realizados com recurso a uma máquina com as seguintes caraterísticas:

```
Architecture: x86_64, CPU op-mode(s): 32-bit, 64-bit, Byte Order: Little Endian, CPU(s): 40,
Thread(s) per core: 2, Vendor ID: GenuineIntel, CPU family: 6, Model: 62, Stepping: 4, CPU MHz:
1200.000, L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 25600K
```

Os resultados mencionados foram depois estudados, permitindo concluir algumas coisas, por vezes não tão óbvias, sobre o nosso algorítmo paralelizado. Temos por exemplo o seguinte gráfico que nos indica a escalabilidade (a nível de threads por core físico) do nosso programa.

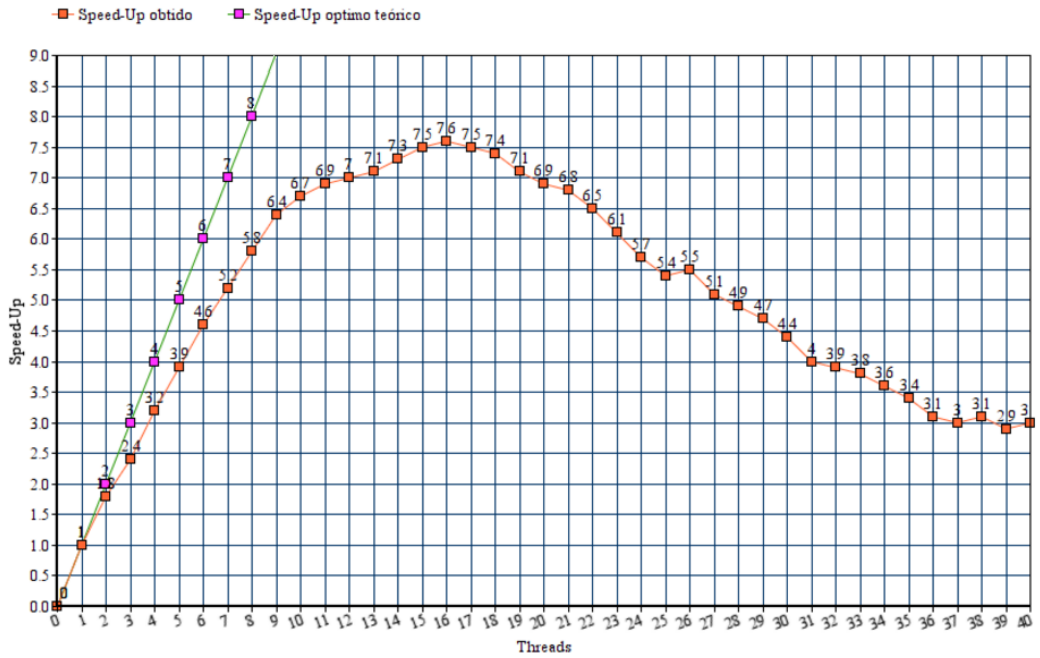

Figura 1 - Gráfico de Escalabilidade

Os valores apresentados no gráfico foram obtidos da seguinte forma:

Fez-se a média de todas as medições da versão sequencial (cem medições por cada um dos quarenta e um ficheiros), de seguida dividiu-se o valor obtido pela média de todas as medições da versão paralela para o número de threads pretendido (cem medições por cada um dos quarenta e um ficheiro com o número de threads pretendidas).

Com este gráfico conclui-se então que o nosso programa é relativamente bem escalável até às dezasseis threads sendo que a partir daí, embora hajam ganhos relativamente à versão sequencial os ganhos são cada vez menores. Atribuímos esta "perda" de speed-up a algo que já referimos no

capítulo anterior ao explicar o algorítmo paralelo. Referimos que para um grande número de threads, poderia acontecer que o número de iterações que cada thread realiza poderia não ser grande o suficiente para justificar o overhead causado pelo fork&join das mesmas. Também mencionamos que isto poderia então ser evitado ao mudar esse número de iterações. No entanto para certos inputs, nomeadamente os **pequenos inputs**, tal acaba por não ser possível, pois com dezassete ou mais threads basta um número ínfimo de iterações para alcançar o resultado pretendido, no entanto o overhead será enorme comparado com o tempo dispendido nessas infimas iterações o que fará com que demore mais até, por vezes, do que a versão sequencial. Se tentarmos aumentar o número de iterações até haver novo join das threads não iremos obter melhorias na mesma para os **pequenos inputs**, pois estaremos a fazer muito mais iterações do que as necessárias para converger com um input tão pequeno. Acabam por ser então estes **pequenos inputs** que têm o maior impacto na descida da curva apresentada anteriormente pelas razões descritas.

Temos ainda mais duas possíveis justificações que, embora achemos não ter tanto impacto devem ser referidas. Ambas são relativas à cache.

O nosso algorítmo paralelo, como referido no capítulo anterior, utiliza arrays onde cada índice é relativo à thread de número homónimo. Isto é vantajoso porque desta forma evitam-se race conditions desnecessárias, já que cada thread acede ao seu índice específico e bem definido. No entanto surge aqui um problema que advém do próprio funcionamento natural da cache. Cada core físico tem a sua cache privada e quando acede ao índice do array específico da sua thread é copiada grande parte do array para a sua cache de modo a poder no futuro aproveitar **o princípio da localidade e da temporalidade** (em vez de se copiar só o valor no índice acedido, acredita-se que os índices perto desse serão também acedidos num futuro não muito distante, copia-se então o máximo que se conseguir de índices do array para uma linha inteira da cache.) isto é um grande desperdício, pois sabemos que cada thread irá aceder somente ao seu próprio índice e que não necessita de nenhum dos outros, sendo que grande parte deste tipo de arrays irá estar a ser constantemente copiado para a cache de cada core sem necessidade alguma. Fácilmente percebemos então que, para um número maior de threads, o overhead causado por este problema será cada vez maior. Uma possível solução seria fazer "padding" deste tipo de arrays, de forma a que seja copiado para a cache de cada core apenas aquilo que realmente lhe interessa.

A outra justificação veio depois da utilização da ferramenta "callgrind" em conjunção com "kcachegrind" que nos permitiu perceber a percentagem de clock cycles dispendidos em cada função. Depois de várias repetições concluiu-se que o resultado caía sempre em algo como isto:

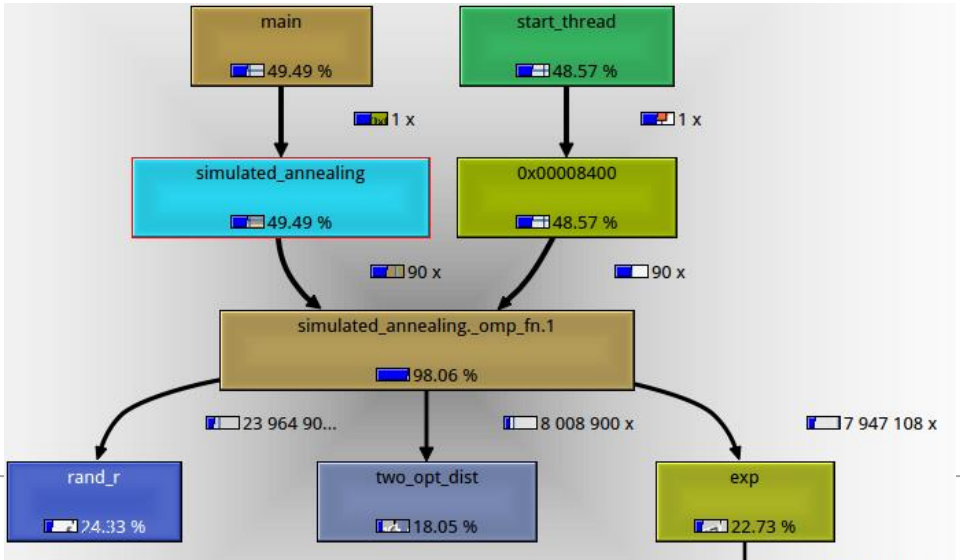

Figura 2 - Grafo representativo da percentagem de clock cycles por cada chamada de função

Aqui nota-se que o maior ênfase das atividades multi-threaded será sobre a função **simulated_annealing**. Observa-se que, em abiente multi-threaded, nessa função mencionada são chamadas várias vezes as funções **rand_r**, **two_opt_dist** e **exp**. Como rand_r e exp são funções já prédefinidas de bibliotecas importadas não se pode fazer muito quanto a elas, no entanto, a função two_opt_dist encontra-se definida no código e como tal poderá ser vantajoso se o seu conteúdo for copiado para dentro da função, ao invés de haver a necessidade de chamar a função. Isto pois o endereço de chamada da mesma pode vir a ocupar espaço importante para outras coisas na cache de cada core sem que haja verdadeira necessidade para isso.

Note-se também que, embora se note uma descida acentuada para um maior número de threads, quando o input é suficientemente grande temos grandes ganhos para mais threads (tal é difícil de observar por este gráfico simplesmente porque as perdas dos pequenos inputs são demasiado grandes). Mas podemos observar este exemplo, para um ficheiro de input com vinte e uma mil cidades (todos os tempos se encontram em segundos):

```
Tempo médio sequencial: 7.6400 segundos

(T = Número de threads, t = tempo médio de execução)

|T:2,  t:7.6008| |T:3,  t:7.9756| |T:4,  t:7.6924| |T:5,  t:7.5018|

|T:6,  t:7.3751| |T:7,  t:5.8862| |T:8,  t:5.8838| |T:9,  t:5.3033|

|T:10, t:5.9863| |T:11, t:5.0937| |T:12, t:4.8803| |T:13, t:4.9281|

|T:14, t:4.6155| |T:15, t:4.2414| |T:16, t:3.9903| |T:17, t:4.0984|

|T:18, t:3.6930| |T:19, t:3.8947| |T:20, t:3.7702| |T:21, t:3.9930|

|T:22, t:3.9566| |T:23, t:3.6659| |T:24, t:3.9731| |T:25, t:4.0550|

|T:26, t:3.3992| |T:27, t:3.4744| |T:28, t:3.4810| |T:29, t:3.7036|

|T:30, t:3.2847| |T:31, t:3.4226| |T:32, t:3.5828| |T:33, t:3.4021|

|T:34, t:3.2385| |T:35, t:3.1788| |T:36, t:3.3804| |T:37, t:3.1155|

|T:38, t:3.3022| |T:39, t:3.4152| |T:40, t:3.8433|
```

Neste exemplo concluímos que para inputs muito grandes poucas threads apenas causam overhead e não ajudam a converger mais rápido, apenas se verificam ganhos significativos a partir das sete threads sendo que apartir das dazassete/dezoito threads esse ganho "estagna", mas não deixa de ser significativo comparativamente com o sequencial. Em suma, para pequenos inputs, o ideal será essencialmente usar menos do que dezassete threads, mas para grandes inputs a partir das 17 é que se alcança o ganho máximo.

5.Conclusões

Terminado o primeiro projeto, depois de feita uma minunciosa avaliação do problema, chegou-se a conclusões (apresentadas ao longo do corpo deste documento) que se considera ser satisfatórias. Pensa-se que este relatório é capaz de apresentar explicações detalhadas acerca da performance da implementação paralela comparada com a sequencial do algorítmo simulated annealing assim como do próprio algoritmo em si. Com isto pretende-se afirmar que as conclusões chegadas e o plano de acção elaborado terá sido, a nosso ver, um sucesso tendo conseguido realizar tudo o que foi proposto de forma simples e eficaz com uma estratégia bem pensada e com lógica onde qualquer pormenor foi bem pensado e nunca simplesmente "deixado ao acaso".

Embora o speed-up alcançado não tenha sido o ideal teórico, sabemos as razões pelas quais isso acontece e entendêmo-las como algo que vai para além do nosso controlo, pois, como referido no capítulo anterior, certas configurações de threads serão sempre incompatíveis com certas configurações de input. Cabe a nós entender a melhor forma de equilibrar esses dois parâmetros, mas no final de contas estaremos sempre à mercê das limitações das diretivas OpenMP por serem somente extensões a uma linguagem inerentemente sequencial, algo que, a nosso ver, nunca irá superar uma linguagem unicamente construída com paralelismo puro em mente.