Partes 1 e 2 do projeto
Serial vs OpenMP

Trabalho realizado por: Grupo 15
Francisco Melo nº86998 - Martim Pereira nº87075 - Tiago Jacinto nº87127

Abordagem para a paralelização

Para explicar a abordagem tomada para a paralelização do programa, vamos explicar o funcionamento do programa em *serial mode*.
O programa começa por abrir e ler o ficheiro, tirando deste os parâmetros necessários à resolução do problema. De seguida, criamos as matrizes: L, B, R^T, L_sum e R_sum. As duas últimas matrizes são utilizadas para guardar a cada iteração os valores a atualizar nas matrizes L e R respetivamente. A matriz R^T é criada ao invés da R, para que possamos ter os valores que precisamos de R na cache aquando a sua atualização ou na multiplicação com a matriz L. A criação da matriz R^T em vez de R vai provocar mais *misses* na cache, durante a criação, mas é preferível desta forma, visto que se criássemos a matriz R encontraríamos estes *misses*, mas num for onde o programa passa grande parte do tempo. Deste modo, conseguimos otimizar o programa. Depois da criação das matrizes preenchemo-las com valores aleatórios e fatorizámo-las. Obtemos, finalmente, o resultado final.
Para a paralelização utilizamos o método abordado nas aulas: Inspeção e análise do programa *serial*. Dito isto optamos por fazer as seguintes alterações. Optámos por não paralelizar duas partes: a leitura do ficheiro e a criação das matrizes. A leitura do ficheiro é I/O Bound, o que significa que a velocidade com que executa depende do ficheiro que está a ler, não havendo vantagens em paralelizar esta parte. Já a criação de matrizes foi porque por analise percebemos que isso tornaria o programa mais lento, algo que não queríamos. Como o programa passa muito pouco tempo nessas operações optámos por deixar *serial*. Todo o resto analisamos e optamos por paralelizar sendo que em algumas secções tivemos de por certas variáveis como privadas para que cada *thread* tivesse a sua e não houvesse nenhum tipo de sobreposição.

Decomposição utilizada

No que diz respeito à decomposição do programa, utilizamos um vetor de estruturas que guarda as entradas não-nulas da matriz A, ao invés de guardar a matriz em si. Como cada entrada não-nula é responsável por atualizar uma linha de L e uma coluna de R(linha de R^T) esta abordagem permite que cada *thread* trate de um conjunto de entradas não-nulas e que seja possível correr o programa em paralelo. Há, no entanto, um problema. Quando há duas entradas não-nulas com linhas ou colunas iguais que querem atualizar ao mesmo tempo. Para resolver este problema utilizamos o comando *atomic* do omp que, como abordado mais à frente, impede duas threads de alterar a mesma posição de memória ao mesmo tempo. Assim, uma das *threads* vai fazer as alterações enquanto a outra espera e só depois é que esta ultima faz a suas alterações. Dito isto, todas as operações de atualização e multiplicação entre matrizes serão realizadas em paralelo, com exceção da situação acima referida. As matrizes L_sum e R_sum que usamos no nosso programa têm guardados os valores para atualizar em de L e R no final de cada iteração. Estes valores

também são protegidos com *atomic*, pela mesma razão acima referida. Decidimos utilizar *atomic* e não *critical*, porque queremos apenas que a atualização de valores nas mesmas entradas ao mesmo tempo não se realizem em paralelo, mas que todas as outras sim. A utilização do comando *critical* não permite isto. Também não protegemos os casos de acesso à mesmo posição de memoria quando o valor que lá se guarda é fixo. Neste caso a ordem em que é escrito é irrelevante e atrasaria o programa.

Problemas de sincronização

Relativamente à sincronização das *threads*, utilizamos *barrier* já implementada nos *parallel for* que impede uma *thread* de continuar para outro ciclo antes de todas a *threads* terem chegado à *barrier*. No que diz respeito à sincronização das posições de memória, utilizamos o comando *atomic*, que garante que a escrita e a leitura numa posição de memória são feitas por apenas uma *thread* de cada vez.

Load Balancing

A decomposição usada no programa foi schedule *static*, visto que as threads criadas têm todas uma quantidade de trabalho semelhante, pelo que o tempo de espera pelo fim das outras é mínimo.

Resultados e comentários

Os seguintes resultados foram obtidos nos computadores do laboratório. Não tivemos tempo de correr os programas com os ficheiros de maior dimensão, devido a apenas termos terminado em cima do tempo, uma má gestão da nossa parte. No entanto, com os resultados obtidos é possivel analisar o esperado.
A tabela mostra os tempos de cada ficheiro para os vários modos de funcionamento.

Ensaios realizados no PC 5 do lab 13 às 21:40 de dia 3/4.

inst		0	1	2	30	200	400	500	600	1000	50000	100k	1M
serial		0min0.011	0min0.094	0min0.102	0min0.982	1min4.245	1min30.078	2min30.879	4min13.790	0min45.780	-	-	-
omp	1 thread	0min0.012	0min0.193	0min0.147	0min1.069	1min2.947	1min18.137	2min28.219	3min58.861	0min46.704	-	-	-
	2 thread	0min0.012	0min0.225	0min0.191	0min0.660	0min32.341	0min40.786	1min34.208	2min29.272	0min27.815	-	-	-
	4 thread	0min0.317	0min0.324	0min0.252	0min0.457	0min16.487	0min20.227	0min40.007	1min2.746	0min12.069	-	-	-
	8 thread	0min0.126	0min2.460	0min1.254	0min0.959	0min18.105	0min21.442	0min48.738	1min11.730	0min13.284	-	-	-

Por analise dos resultados da tabela obtemos um *speedup* perto dos 3.8 para os ficheiros grandes e de 0.3 para os ficheiros mais pequenos. Isto é algo esperado, visto que a versão paralela, para ficheiros mais pequenos tem *overheads* e, portanto, não compensa usar este modo de funcionamento nestes ficheiros. Verifica-se também que a versão *omp* com 1 thread é mais lenta que a versão *serial*. Tal é esperado e acontece porque há custos de sincronização e o *kernel* fica mais sobrecarregado. Há, no entanto, alguns casos em que isto não se verifica. Isto acontece porque o programa foi, para além de paralelizado, melhorado e esta melhoria não foi realizada no programa *serial* testado por lapso do grupo, no entanto irá na próxima versão juntamente com o mpi. Para mais de 4 *threads* é esperado que o programa seja mais lento que com 4 *threads*, pois o computador só tem 4 *cores* e ter mais *threads* que *cores* só adiciona tempo de sincronização e *overheads* desnecessários.