Introdução às POSIX Threads
Performance Relativa de Kernels em ambiente de Memória Partilhada

Filipe Oliveira
Departamento de Informática
Universidade do Minho
Email: a57816@alunos.uminho.pt

1. Introdução – Contextualização das POSIX Threads

As POSIX Threads são um standard em ambientes UNIX, relativamente ao desenvolvimento de aplicações paralelas em ambiente de memória distribuída em C/C++. Permitem-nos criar novos fios de execução para um mesmo processo, possibilitando um ambiente propício à computação paralela e distribuída sem o overhead associado à criação de processos, via **fork()**. Às threads em geral está também associado o termo **Light-Weight Processes** uma vez que o sistema, ao contrário dos processos, não atribui a estas um espaço de endereçamento de memória virtual.

As POSIX Threads são objectos opacos que têm de ser tratados mediante os métodos definidos na sua API. Esta inclui, entre outras, operações de criação, terminação, sincronização de threads, e ”schedulling”. É importante salientar que uma thread não mantém uma lista das threads pertencentes ao mesmo processo, não registando sequer a referência ao processo que a criou.

Threads do mesmo processo partilham:

- Variáveis globais;
- Descritores de ficheiros;
- ”signals” e ”signal handlers”;
- ”User id” e ”Group id”;

Cada Thread tem exclusivamente:

- Thread ID;
- Conjunto de registos e um Stack Pointer;
- Stack de variáveis globais;
- return address;

É importante compreender o anteriormente enumerado para associar por vezes a falta de speed-up a um conjunto de factores intrinsecamente associados a estas propriedades. Enquanto que em ambiente de memória distribuída, por exemplo, problemas como a invalidação de dados partilhados em memória e o acesso atómico a variáveis globais, não se colocam como uma grande barreira aos ganhos de performance, neste tipo de ambiente de memória partilhada esses factores podem ser cruciais para a decisão de manter ou iniciar outra abordagem a um qualquer problema computacional.

2. Contextualização das métricas de performance em estudo

Escolhido o ambiente de memória a utilizar para a implementação dos nossos kernels paralelos, resta-nos especificar quais as métricas em medição no caso de estudo. Ora, é importante compreender o comportamento do sistema mediante a criação de um elevado número de POSIX Threads. É requerido que calculemos o tempo médio necessário à criação e terminação de um fio de execução, e estudemos a sua variância mediante o aumento do número de threads.

Numa segunda parte do caso de estudo, serão definidos 3 kernels paralelos, cada um implementando o mesmo algoritmo com a ressalva para a forma de exclusão mútua de uma secção crítica do mesmo. Iremos recorrer portanto ao método de espera-activa (busy-wait), MUTEX, e Semáforos, e registar a influência de cada escolha no tempo total para o tempo da solução.

3. Caracterização do Hardware do ambiente de testes

Especificadas as métricas de performance em estudo, resta-nos especificar os ambientes de teste nos quais pretendemos realizar as benchmarks. Através da análise do hardware disponível no Search6 [1], uma das nossas plataformas de teste, foram seleccionados nós do tipo compute-431, sendo a disponibilidade global dos mesmos o principal factor. Iremos ainda incluir no caso de estudo o **student laptop** dado que pretendemos, para a primeira fase do caso de estudo recorrer à ferramenta **dtrace** como forma de monitorização da criação e terminação de ”light-weight processes”. Nas tabelas 1 e 2 encontram-se especificadas as principais características dos sistemas em teste.

1. Services and Advanced Research Computing with HTC/HPC clusters

TABLE 1. CARACTERÍSTICAS DE HARDWARE DO NÓ 431

Sistema	compute-431
# CPUs	2
CPU	Intel® Xeon® X5650
Arquitectura de Processador	Nehalem
# Cores por CPU	6
# Threads por CPU	12
Freq. Clock	2.66 GHz
Cache L1	192KB (32KB por Core)
Cache L2	1536KB (256KB por Core)
Cache L3	12288KB (partilhada)
Ext. Inst. Set	SSE4.2
#Memory Channels	3
Memória Ram Disponível	48GB
Peak Memory BW Fab. CPU	32 GB/s

TABLE 2. CARACTERÍSTICAS DE HARDWARE DO **STUDENT LAPTOP**

Sistema	student laptop
# CPUs	1
CPU	Intel® Core™ i7-3635QM
Arquitectura de Processador	Ivy Bridge
# Cores por CPU	4
# Threads por CPU	8
Freq. Clock	2.4 GHz
Cache L1	128KB (32KB por Core)
Cache L2	1024KB (256KB por Core)
Cache L3	6144KB (partilhada)
Ext. Inst. Set	SSE4.2 & AVX
#Memory Channels	2
Memória Ram Disponível	16GB
Peak Memory BW Fab. CPU	25.6 GB/s

4. Determinação do tempo médio necessário para criar e terminar um fio de execução

4.1. Nós compute-431

Por forma a calcular o tempo médio necessário para criar e terminar um fio de execução foi criado um kernel, que apenas realizava essas mesmas operações e registados os valores para os diferentes número de threads. A tabela 3 apresenta a relação entre média e desvio padrão de criação/terminação para um diferente número de POSIX Threads para os nós do tipo compute-431.

TABLE 3. RELAÇÃO ENTRE MÉDIA (EM μS) CRIAÇÃO/TERMINAÇÃO PARA UM DIFERENTE NÚMERO DE POSIX THREADS PARA OS NÓS COMPUTE-431

# POSIX Threads	Média de Criação/Term. (em μs)
1	59
2	41.5
4	25
8	24
16	23.063
32	25.656
64	23.219
128	22.609
256	25.109

4.2. Student Laptop

Em adição à forma de cálculo do tempo médio necessário para criar e terminar um fio de execução realizada no Search6, foi ainda criado um script na ferramenta **dtrace** que permitiu ter acesso aos tempos de criação de terminação de das POSIX Threads, sendo para tal registados os tempos das ocorrência de **proc:::lwp-start** e **proc:::lwp-exit** para cada thread, e registado o tempo de início do Processo através do registo de **dtrace:::BEGIN** e **dtrace:::END**. Desta forma conseguimos relacionar os tempos de criação das threads com o processo que as criou, tendo obtido diagramas como os apresentados nas figuras 1, 2 e 3, para 8, 16 e 32 POSIX Threads.

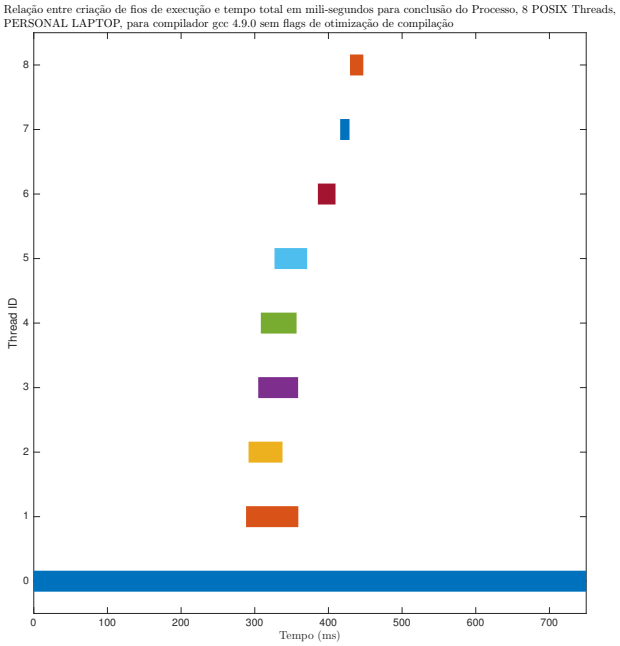

Figure 1. Relação entre criação de fios de execução e tempo total em mili-segundos para conclusão do Processo, 8 POSIX Threads, para o student laptop, para compilador gcc 4.9.0 sem flags de otimização. **Nota: Thread 0 refere-se ao processo.**

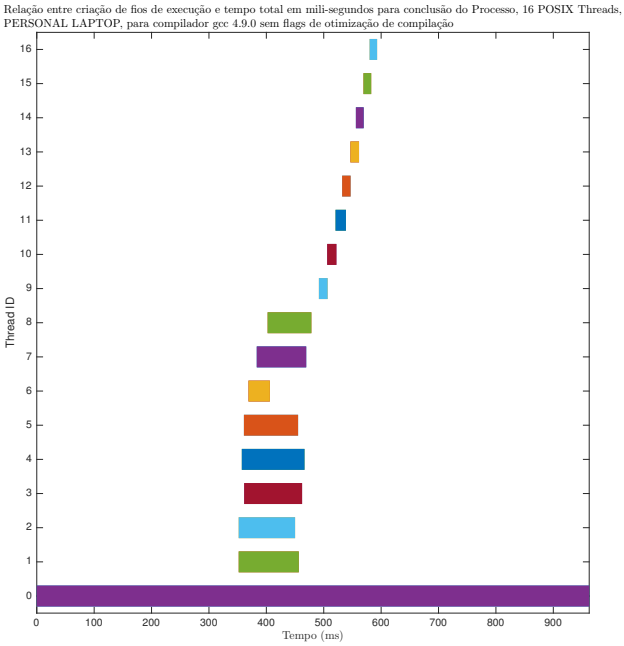

Figure 2. Relação entre criação de fios de execução e tempo total em mili-segundos para conclusão do Processo, 16 POSIX Threads, para o student laptop, para compilador gcc 4.9.0 sem flags de otimização. **Nota: Thread 0 refere-se ao processo.**

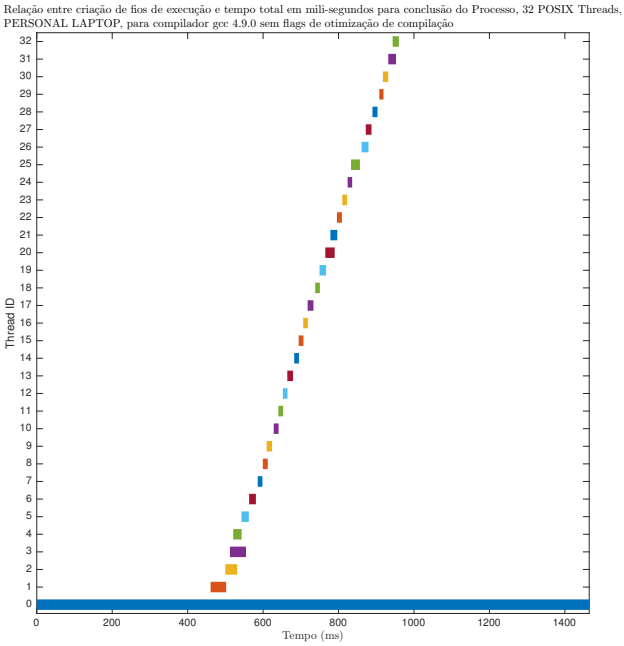

Figure 3. Relação entre criação de fios de execução e tempo total em mili-segundos para conclusão do Processo, 32 POSIX Threads, para o student laptop, para compilador gcc 4.9.0 sem flags de otimização. **Nota: Thread 0 refere-se ao processo.**

Denote que para todos os casos distintos, é na inicialização/terminação das primeiras POSIX Threads que despendemos a maior porção de tempo associado às threads – tal como pode ser confirmado na tabela 4. Podemos ainda inferir que o pior resultado registado em termos de tempo total para a solução se dá para o número de POSIX Threads igual a 16 (o dobro do número de threads físicas disponíveis no CPU), e que é para um número de POSIX Threads igual a 128 que obtemos tanto o menor desvio como o melhor valor em termos de média. Para a média esta afirmação é válida tanto para os nós compute-431 como para o student laptop.

TABLE 4. RELAÇÃO ENTRE MÉDIA E DESVIO PADRÃO DE CRIAÇÃO/TERMINAÇÃO PARA UM DIFERENTE NÚMERO DE POSIX THREADS PARA O STUDENT LAPTOP

# POSIX Threads	Média de Criação/Term. (em μs)	Desvio P.
1	23.3620	0
2	24.7585	2.3342
4	40.1410	10.7082
8	39.8469	19.8955
16	51.6829	41.1291
32	17.6375	7.8276
64	26.5381	11.5596
128	14.6927	7.8023
256	16.7902	9.2130

Podemos portanto concluir que um aumento no número de fios criados/terminados por um processo leva a uma diminuição do tempo médio necessário para a realização de tais operações, contribuindo ainda para um estabilização desse mesmo tempo em intervalos com uma desvio padrão menor.

5. Implementação da regra trapezoidal

Tal como referimos anteriormente, nesta fase iremos submeter a controlo de métricas de performance 3 kernels paralelos, cada um implementando o mesmo algoritmo - **regra trapezoidal** - com a ressalva para a forma de exclusão mútua de uma secção crítica do mesmo. Iremos recorrer portanto ao método de espera-activa (busy-wait), MUTEX, e Semáforos, e registar a influência de cada escolha no tempo total para o tempo da solução.

Ora, no nosso algoritmo existe apenas uma seção crítica, sendo esta a adição de uma valor calculado a cada iteração das POSIX Threads envolvidas na solução a uma variável global a todas as Threads.

Foi escolhido a não recorrência a flags de otimização uma vez que, para o kernel que implementa a exclusão mútua via espera-activa, otimizações automáticas como a execução fora de ordem de instruções podem levar a situações de dead-lock. Ora, assim e dado que necessitamos de comparar todos os kernels com iguais flags de compilação nenhum dos 3 kernels apresenta otimizações.

Atente no gráfico 4. Como se pode confirmar, a melhor solução para um elevado número de POSIX Threads é alcançada recorrendo a MUTEX's. Ora, podemos considerar que os MUTEX's como um tipo especial de semáforos

(semáforos binários), sendo portanto esperado que os valores alcançados pelo método que implementa exclusão mútua via semáforos apresentasse a mesma ordem de grandeza de valores dos apresentados pelo kernel que implementa a exclusão mútua via MUTEX's. Contudo, tal não se verifica, tendo o kernel implementado recorrendo a semáforos os mesmos fracos resultados que os apresentados pelo kernel que implementa espera-activa.

Ora, esta grande diferença poderá estar relacionada como o facto de num MUTEX's apenas a Thread que bloqueou a variável de condição a pode desbloquear. Talvez por essa premissa se consigam reduções no tempo de verificação da variável de condição e consequentemente melhores resultados.

Não obstante o anteriormente dito resta-nos ressalvar que este algoritmo é demasiado simplista e de fácil resolução da dependência de dados para podermos extrapolar conclusões para qualquer outro algoritmo. No que toca à resolução da necessidade a cada iteração da secção crítica, cada POSIX Thread poderia ter a variável approx local – **approx_local**, sendo apenas esse valor reduzido no final do trabalho de cada thread – retirando a necessidade de **n** (no nosso caso 1048576) operações atómicas, passando apenas a ser necessárias **n_threads** (no nosso caso máximo 256) operações atómicas, retirando quase por completo todo o overhead gerado pela operação de adição de valores à variável global (Podemos pensar nesta alteração como a implementação de um conceito map/reduce).

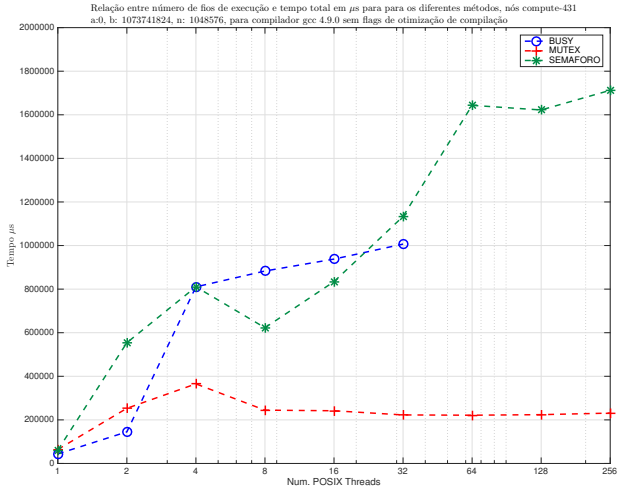

Figure 4. Relação entre criação de fios de execução e tempo total em μ segundos para conclusão do Processo, para diferentes números de POSIX Threads, para os diferentes métodos de exclusão mútua de secção crítica, para os nós compute-431, para compilador gcc 4.9.0 sem flags de otimização, para a=0, b=1073741824, e n=1048576.
Nota: Para um número de POSIX Threads superior a 32 não foi possível recolher os valores de tempo total em mili-segundos para conclusão do Processo nos nós 431

5.1. Implementação da regra trapezoidal recorrendo a OpenMP (PThreads vs OpenMP)

Tal como nas POSIX Threads, o OpenMP recorre a uma API para implementação de kernels paralelos em memória partilhada. A grande diferença assenta no "esforço" necessário para paralelizar kernels. Enquanto que via OpenMP através de um conjunto de directivas conseguimos obter código paralelo, recorrendo a PThreads necessitamos de explicitamente especificar o processo de criação, terminação, e comportamento de cada Thread.

Analisemos portanto a facilidade de paralelizar (sem otimizações quer via mudança do algoritmo quer via adição de flags de compilação) o nosso algoritmo:

```
#pragma omp parallel num_threads(thread_count)
{
  #pragma omp parallel for schedule(static,my_n)
    for (i = interval_a ;i<=interval_b-1;i+=h){
      x_i = interval_a + i*h;
      #pragma omp atomic
      approx  +=  f(x_i);
    }
}
```

Por forma a obtermos o mesmo "comportamento" adicionamos o scheduling static e de tamanho my_n (o mesmo que para o nosso algoritmo implementado via PThreads).

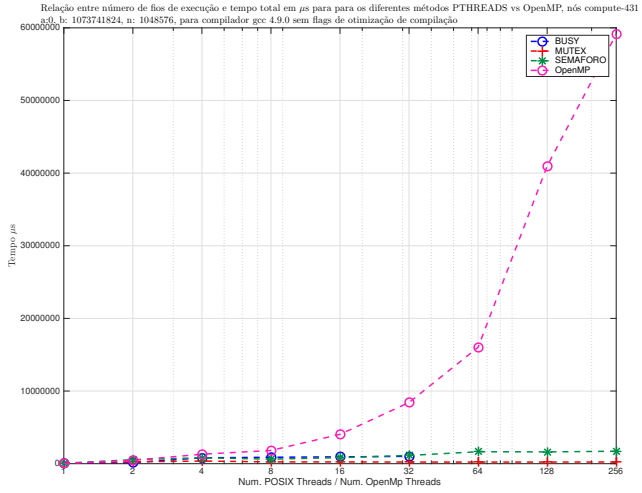

Figure 5. Relação entre criação de fios de execução e tempo total em μ segundos para conclusão do Processo, para diferentes números de POSIX Threads / OpenMp Threads, para os diferentes métodos de exclusão mútua de secção crítica, para os nós compute-431, para compilador gcc 4.9.0 sem flags de otimização, para a=0, b=1073741824, e n=1048576.
Nota: Para um número de POSIX Threads superior a 32 não foi possível recolher os valores de tempo total em mili-segundos para conclusão do Processo nos nós 431

As Pthreads são portanto mais "baixo nível" e permitem otimizações e especificação de comportamentos específicos por thread. Em OpenMP somos confrontados com a "facilidade" de paralelização a custo de por vezes performance

(ver figura 5) e interações de mais baixo nível entre Threads difíceis de computar.

Contudo, não podemos afirmar que o recurso a PThreads é sempre vantajoso – se necessitarmos de uma otimização máxima e parelelização superior à conseguida através de ciclos e directivas, então deverão ser escolhidas PThreads. Caso seja "suficiente" o grau de paralelismo alcançado com as diretivas OpenMP, dado o pouco esforço necessário à produção de código paralelo simples, então a opção deverá ser a de recorrência a OpenMP. Denote que a paralelização via OpenMP tem ainda a vantagem da manutenção de código sequencial, bastando para isso compilar sem a flag -fopenmp.

6. Conclusão

Relativamente aos kernels em estudo podemos concluir que o recurso a MUTEX's, **para o nosso caso específico**, se revelou mais vantajoso, mas, tal como referido anteriormente dada a simplicidade do algoritmo não se podem retirar conclusões acerca da generalidade de mecanismos de exclusão mútua. Neste caso também se verificou um ganho de performance relativamente à paralelização em ambiente de memória partilhada via OpenMP, por si só espectável – o overhead de criação das zonas paralelas, dado o pouco de recurso do algoritmo a ciclos (apenas 1) não se verificou compensatório.

O valor do algoritmo deste caso de estudo ultrapassa os resultados obtidos, prendendo-se uma vez mais com o desenvolvimento e prática de ferramentas (**dtrace**), e métodos de tratamento e análise de métricas de sistemas de computação de alta perfomance, assim como uma forma de maior ambientação no ambiente de clustering presente no Search6.