Relatório - PrefixSum com Pthreads

Nome: Paulo Mateus Luza Alves
GRR: GRR20203945

1 Conteúdo do .tar.gz

Dentro do arquivo compactado temos:

- Relatório.pdf: Este relatório;
- chrono.c: Arquivo feito pelo professor para contagem de tempo;
- prefixSumPth.c: Arquivo main com a implementação do programa;
- Makefile: makefile para compilação do programa;
- roda.sh: Script para rodar os esperimentos conforme solicitado pelo professor;
- prefixSum-Pthreads-v1-e-v2: Planilha fornecida pelo professor para construção dos gráficos.

2 Implementação

O processo de cálculo da soma de prefixos se baseou na utilização de 2 conceitos, sendo eles:

- Barreiras (Apenas uma);
- Pool de threads.

Para isso, foram utilizadas 2 funções para a paralelização do cálculo, denominadas ‘parallelPrefixSumPth’ e ‘prefixSum’. Ademais, alguns ajustes como nomenclatura de variáveis, comentários, dentre outros foram modificados, no arquivo do programa.

2.1 A função parallelPrefixSumPth

Nesta função encontramos o ‘core’ para o início da paralelização. Basicamente, ela gerencia a criação da barreira e da pool de threads utilizada no programa. um ponto de atenção é a variável ‘alreadyInit’, ela está presente para garantir que o espaço do programa que inicializa a barreira e a pool de threads rode apenas uma vez.

Por fim, após a construção das threads, ela executa a função ‘prefixSum’ para a thread 0 finalizando seu papel.

2.2 A função prefixSum

Aqui encontramos a real execução da soma de prefixos. Algumas ferramentas foram utilizadas para a otimização deste cálculo, sendo elas:

- Uma barreira;
- Unroll & jam.

A estruturação da função é dividida em 3 etapas:

- Preparação das informações (Executada uma vez);
- Etapa pré-barreira (Executada diversas vezes);
- Etapa pós barreira (Executada diversas vezes);

2.2.1 Preparação das informações

Esta etapa engloba a parte das linhas 54 a 66 no arquivo 'prefixSumPth.c'. Aqui realizamos o setup das variáveis que vão ser utilizadas nas outras duas etapas. Ou seja, calculamos o início e o fim da seção que a thread vai calcular e o tamanho do 'ulroll_size' para este pedaço, dentre outros.

2.2.2 Etapa pré-barreira

Esta etapa engloba a parte das linhas 67 a 79 no arquivo 'prefixSumPth.c'. Aqui temos a parte da função que calcula o preenchimento de um vetor global chamado 'partialSum', este vetor é responsável por conter a soma de todos os elementos da seção que a thread é responsável, e que posteriormente é usada para calcular a soma dos prefixos. Aqui encontramos a primeira utilização do unroll & jam, realizando uma pequena paralelização no cálculo dessa soma.

2.2.3 Etapa pós-barreira

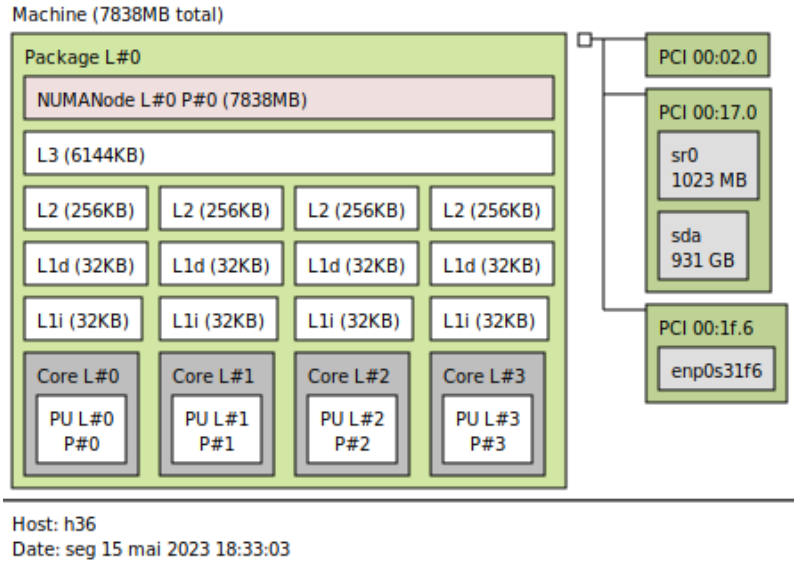
Esta etapa engloba a parte das linhas 81 a 99 no arquivo 'prefixSumPth.c'. Aqui temos a finalização da função, onde a thread utiliza o vetor 'partialSum' previamente calculado por todas as threads (pois está antes da barreira) para calcular o valor do seu primeiro prefixo, e a partir disso executa o cálculo do vetor de prefixos como conhecido.

2.2.3 Observações

Dentro da função temos um while (true), esta é uma caracterização para o pool de threads, pois com isso nós 'nunca matamos' a thread, ela apenas fica em standby na barreira, aguardando a próxima chamada da função 'parallelPrefixSumPth'.

3 O processador

Para executar os testes foi utilizado o computador h36 do laboratório do dinf. Foram utilizados os programas ‘lscpu’ e ‘lstopo’ para coletar os dados, os prints abaixo resumem a estrutura do processador.



CPU:i5-7500 CPU @ 3.40GHz
CPU(s): 4
Thread(s) per núcleo: 1
Núcleo(s) por soquete: 4
CPU MHz: 1112.082
CPU MHz máx.: 3800,00
CPU MHz mín.: 800,00
cache de L1d: 128 KiB
cache de L1i: 128 KiB
cache de L2: 1 MiB
cache de L3: 6 MiB

Opções: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmpe rf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe po pcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti tpr_shad ow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdseed adx smap c lflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp

4 Os Experimentos

Para a execução dos experimentos foi executado o seguinte processo:

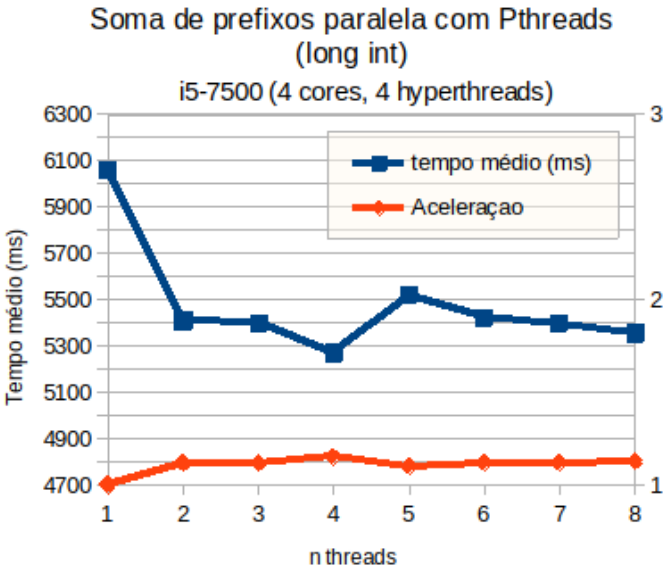
dentro da pasta do projeto, no terminal, o executável foi compilado com o comando ‘make’. Após a confirmação da compilação, foi executado o comando ‘./roda.sh 5000000’. O script ‘roda.sh’, fornecido pelo professor, executou o programa 10 vezes para [1…8] threads.

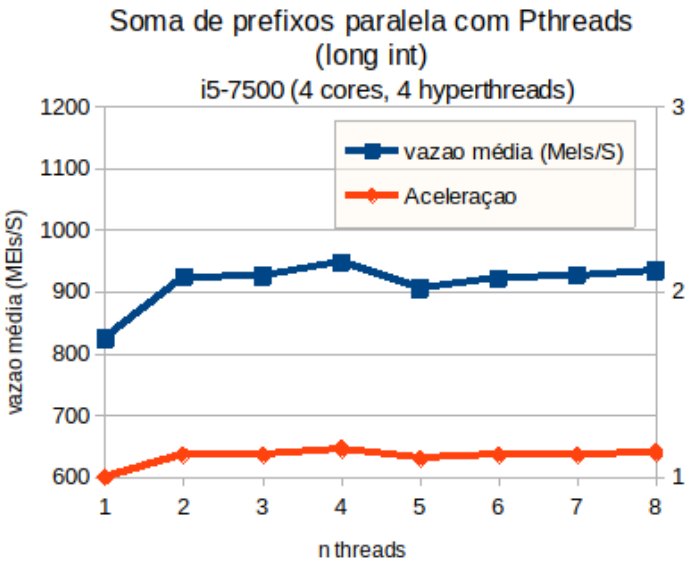
Por fim, o resultado do script foi copiado e adicionado na tabela fornecida pelo professor, onde foi possível verificar o processo de otimização por meio dos gráficos.

Para repetir a experiência, basta executar os mesmos passos na sua máquina pessoal.

5 Sumarização dos resultados

Ao executar os experimentos indicados na CPU apresentada foi possível obter os seguintes resultados:





	Tempo médio							
threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
tempo médio (ms)	**6059,46**	**5.410,00**	**5400**	**5272**	**5519**	**5421**	**5396**	**5356**
Aceleração	**1.00**	**1,12**	**1,12**	**1,15**	**1,10**	**1,12**	**1,12**	**1,13**

	Vazão Média							
threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
vazão média (Mels/S)	**825**	**924**	**926**	**948**	**906**	**922**	**927**	**934**
Aceleração	**1.00**	**1.12**	**1.12**	**1.15**	**1.10**	**1.12**	**1.12**	**1.13**