Relatório do Segundo Trabalho Prático
Computação de Alto Desempenho

Pedro Hollanda Boueke {phboueke@poli.ufrj.br}
Erik Fernandes Tronkos {erik.tronkos@poli.ufrj.br}
UFRJ

11/07/2016

Os dados obtidos e utilizados para a realização desse relatório serão entregues junto ao mesmo. Ademais, os códigos fonte completos também estarão disponíveis em https://github.com/pboueke/CAD-COC472, no diretório "/2". Os experimentos e iterações foram realizados por meio de scripts bash e python, que também serão disponibilizados, responsáveis pela execução ordenada dos programas e geração de tabelas a partir dos dados obtidos.

Exerccíco 1

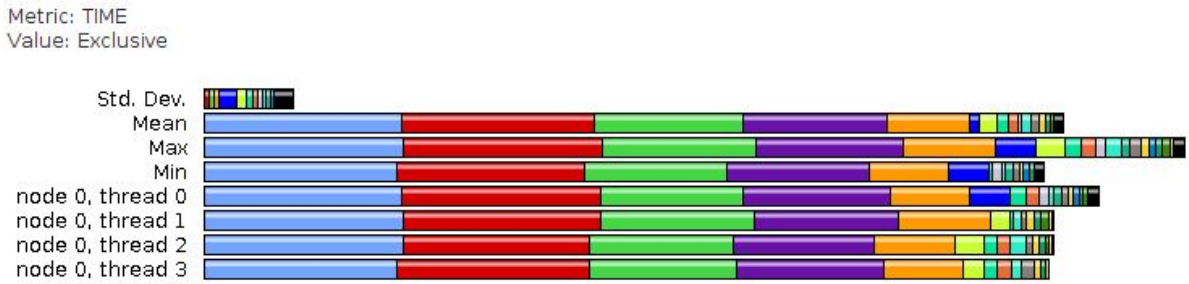

Figura 1. Tela inicial do paraprof, dados colhidos a partir da paralelização do código stream.

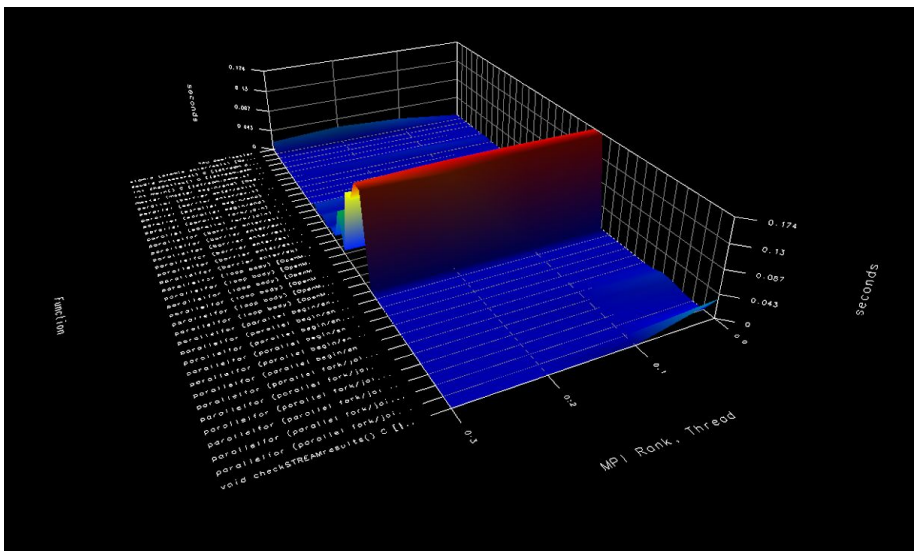

Figura 2. Vizualização 3d gerada pelo paraprof a partir dos dados colhidos da execução do código stream.

Figura 3. Vizualização 3d gerada pelo paraprof a partir dos dados colhidos da execução do código stream otimizado, com foco nas linhas de maior consumo de recursos.

Podemos observar, por meio da figura 3, que as linhas que mais consomem recursos computacionais são 334 e 344, que se tratam de laços que utilizam paralelização openmp. Também podemos observar que a carga, nas funções que demandam mais tempo de execução, está bem distribuída.

Exercício 2

Para esse exercício, o código wave otimizado foi modificado para o teste de desempenho com diversas clausulas e diretivas do OpenMP. Alguns dos testes não foram bem executados devido à *grande dificuldade* do uso do tau_cc.sh para sua compilação (foi tentada a compilação com diversos flags de compatibilidade e a utilização do compilador tau para c++, porém nada funcionou. As mensagens de erro mais frequentes foram "frontend_failed" e "rose_exception"). Os dados obtidos nas execuções estão disponíveis em 2/2/old_data e correspondem à compilação dos códigos em 2/2/wave/old.

Com base nos resultados obtidos, foi concluído que o melhor resultado deve ser uma combinação da cláusula de escalonamento "schedule(dynamic)" com a clausula "collapese(3)".

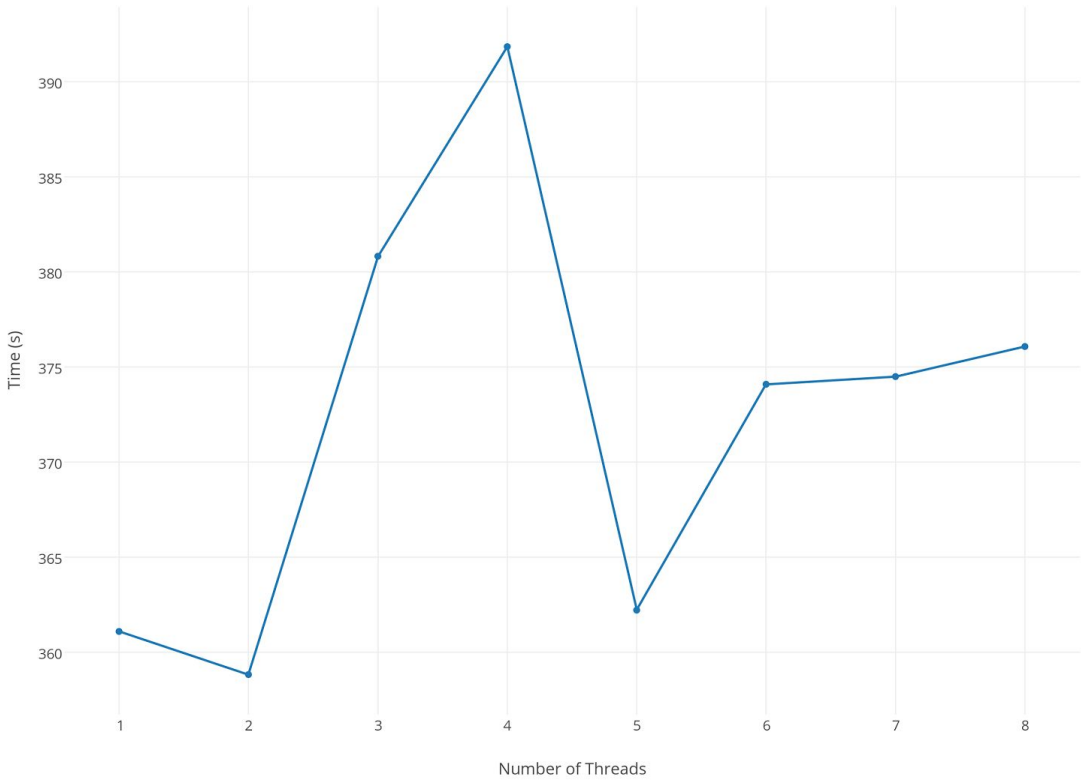

Figura 7. Tempos de execução do programa wave modificado em função do número de threads..

O gráfico apresenta uma grande inconsistência referente aos tempos de execução, nos tornando incapaz de tecer conclusões com relação à influência do número de threads para a execução do programa.

Conclui-se que, devido à incapacidade de compilar o programa com o tau, foram gerados resultados inconclusivos.

Exercício 3

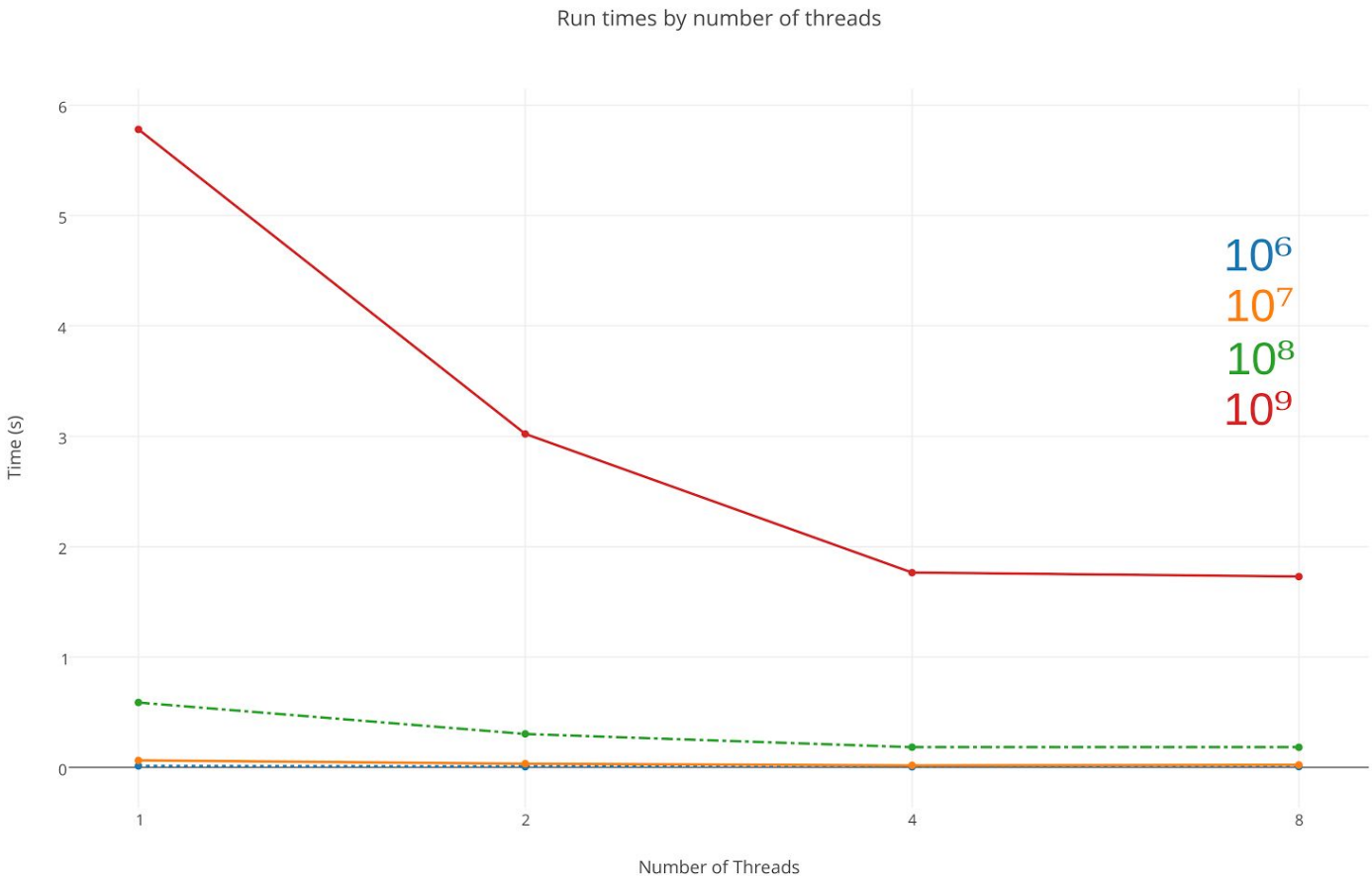

Figura 4. Tempos de execução (eixo vertical) da s simulações de Monte Carlo em função do número de threads (eixo horizontal), agrupadas pelo número total de pontos utilizados (linhas).

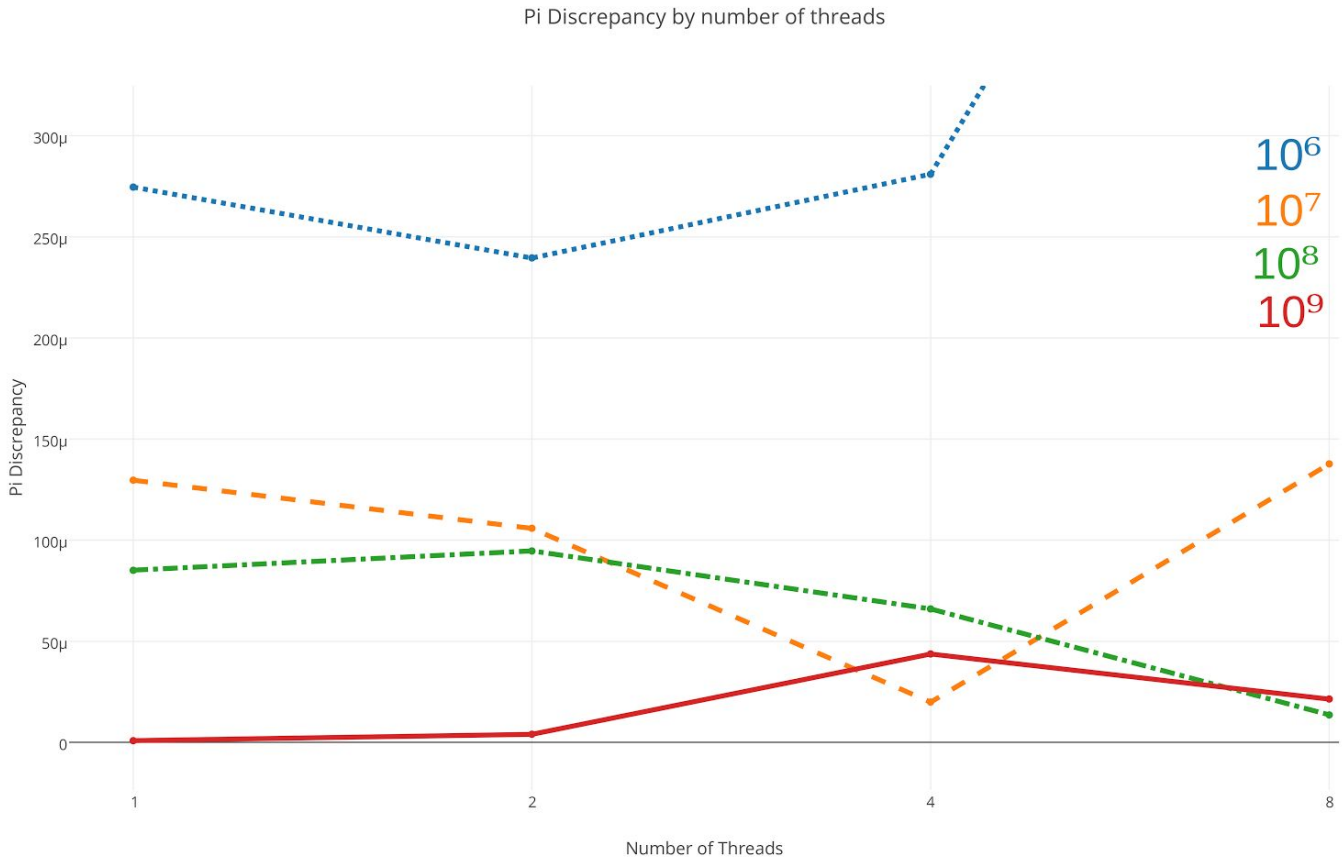

Figura 5. Discrepância (eixo vertical) encontrada no valor calculado de Pi (da forma (PiCalculado - PiConhecido) / PiConhecido) em função do número de threads (eixo horizontal) agrupado pelo número total de pontos utilizados (linhas).

Por meio dos dados apresentados nos gráficos acima (figuras 4 e 5), podemos notar que, escolhendo grandes números de pontos, existe a tendência de que tanto a discrepância do PI calculado pelo método de Monte Carlo quanto o tempo de execução diminuam com o aumento do número de threads, i.e., confirmando que há a melhora estatística em umm cenário de escalonamento fraco.

Exercício 4

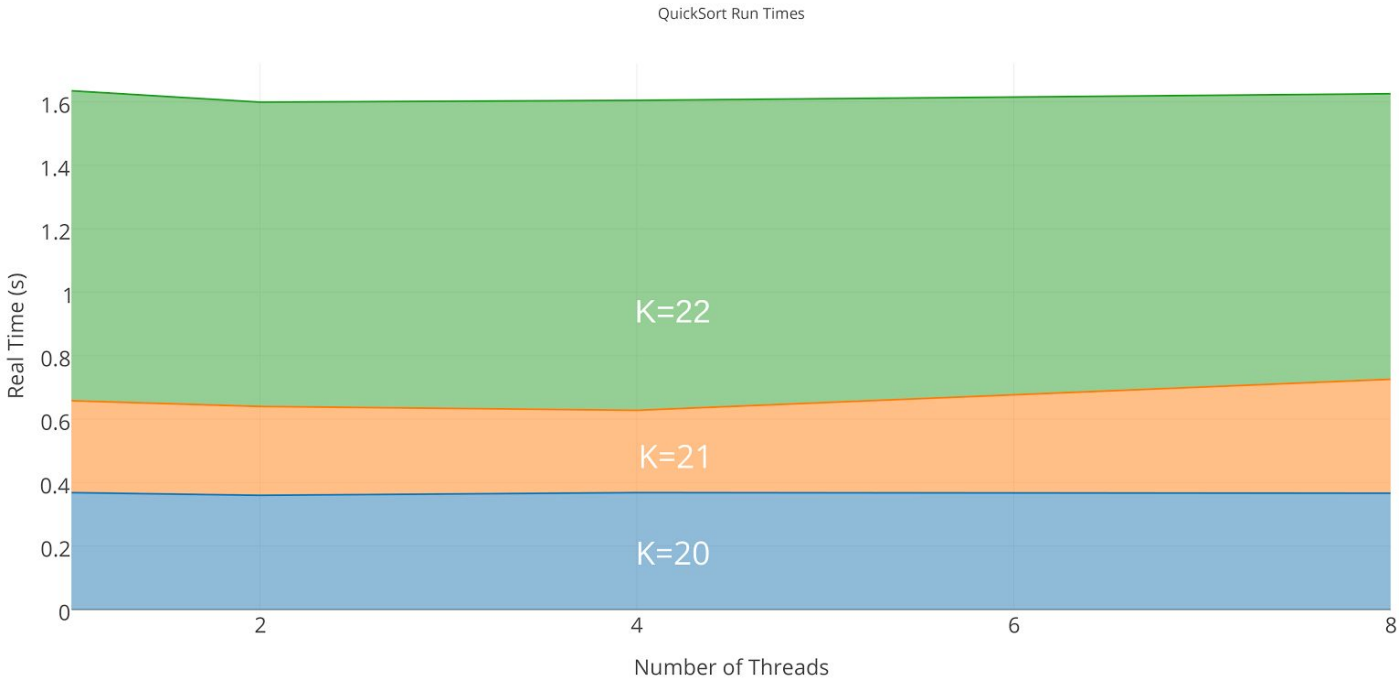

Figura 6. Tempos de execução (eixo vertical) em função do número de threads (eixo horizontal, com valores de 1 a 8) agrupados pelo valor K (áreas), tal que n (número de registros a serem ordenados) é calculado na forma 2 elevado a k-ésima potência. Dados foram obtidos com o código do Quick Sort paralelizado por meio do OpenMP.

O gráfico apresenta os dados referentes a 240 execuções do algoritmo QuickSort, distribuídas de forma que o tempo apresentado para cada ponto representa a média de 20 execuções para cada combinação de número de threads (1 - serial, 2, 4 e 8) e tamanho de K (20, 21 e 22).

Os dados obtidos indicam que, para o algoritmo Quick Sort, a paralelização indicada, por meio do uso de "OpenMp parallel sections" não provocou um efeito visível nos tempos de execução. O trecho abaixo indica a utilização empregada do OpenMP:

```
void quickSort(int arr[], int left, int right) {
   //Inicialização
   ….
   //Particionamento
   ….
   #pragma omp parallel sections
      #pragma omp section
         if (left < j) { quickSort(arr, left, j); }
      #pragma omp section
         if (i < right) { quickSort(arr, i, right); }
}
```

O código real se encontra em root/2/4/quicksort.cpp