UNIVERSIDADE FEDERAL DE SÃO CARLOS

CAMPUS SOROCABA

COMPUTAÇÃO PARALELA

STRING MATCHING: PTHREAD/OPENMP

Erico Alexandre Nielsen Matthiesen | 400556

Gabriel Stankevix Soares | 511340

Maurício Spinardi | 408174

Setembro de 2015

SUMÁRIO

1. INTRODUÇÃO AO EXPERIMENTO

Uma linha ou encadeamento de execução (thread) é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrencialmente. Permite, por exemplo, que o usuário de um programa utilize uma funcionalidade do ambiente enquanto outras linhas de execução realizam diferentes operações em paralelo. Sabendo disso, podemos afirmar que é muito útil para evitar comportamentos bloqueantes e incrementar o desempenho de aplicações com as características adequadas. Por outro lado, identificar quais características são adequadas para a divisão de tarefas pode ser uma atividade extremamente custosa no processo de desenvolvimento.

Para ilustrar um cenário real, com um problema que justifique o esforço da paralelização, este experimento irá, de forma sucinta, reintroduzir as APIs de programação paralela PThread e OpenMP, que serão aplicadas em conjunto, de forma a incrementar exponencialmente o número de linhas de execução concorrentes e estressar ao máximo a(s) máquina(s) de teste disponível(is). Esse último objetivo visa evidenciar o incremento de performance que um hardware pode trazer, quando bem explorado, ao mais inocente dos algoritmos.

2. PTHREAD/OPENMP

Considere a definição de linhas de execução onde a assumimos por uma forma de um processo dividir a si mesmo em duas ou mais tarefas, que podem ser executadas concorrencialmente. Tendo isso em mente, para um desenvolvedor de software, threads de memória compartilhada são como programas independentes, subordinados à um coordenador, com quem divide informações. Na prática, por sua vez, não constituirão programas distintos em si, apenas ramos diferentes de uma mesma aplicação.

O que então diferencia PThread e OpenMP? No primeiro caso, o desenvolvedor precisa explicitamente definir o comportamento de cada linha de execução, da criação ao encerramento, acesso/escrita concorrente e prioridade de execução. A API dá total liberdade, sendo possível desenvolver código de mais baixo nível, ainda que o desenvolvedor fique mais vulnerável à eventuais erros de projeto. No último, estamos falando de um conjunto de diretivas de memória compartilhada que, em C e C++, são trechos de código e bandeiras específicas convertidas diretamente em instruções de hardware. Tais bandeiras são conhecidas como *pragmas,* normalmente implementadas para dar suporte a comportamentos que não são naturais e, por isso mesmo, compiladores que não as reconhecem são livres para ignorá-las (o que permite que códigos cuidadosamente escritos possam ser executados independentemente do suporte à API).

Devido às características de ambas, talvez o leitor já tenha se perguntado se é possível mesclar seu uso no desenvolvimento de uma única aplicação, e se isso traria benefícios práticos no produto resultante. A resposta para ambas as questões pode ser positiva, considerando qual o problema a ser resolvido.

3. STRING MATCHING

Tome nota que o problema sugerido aqui para ilustrar um cenário de desenvolvimento é, essencialmente, uma busca. Algoritmos dessa classe estão entre os mais significativos e estudados da computação, juntamente com indexação, ordenação e classificação, por exemplo. No entanto, como o protagonismo desse experimento deve ser das APIs de programação paralela, não apresentaremos a versão mais robusta do pareamento de padrões. Nos contentaremos em demonstrar como paralelizar a versão mais inocente da solução para esse problema.

3.1. SUBSTRING SEARCH

O problema da busca por uma substring consiste em encontrar determinado padrão em uma entrada maior. Por exemplo, encontrar “Alice”[1] em:

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, `and what is the use of a book,' thought Alice `without pictures or conversation?'

Sendo assim, generalize que, dado uma string `pat` e uma string `txt`, encontraremos todas as ocorrências de `pat` em `txt`. Logo, `pat` será o padrão a ser reconhecido e `txt` será a entrada.

3.2. APLICAÇÕES

É espantoso ver o quanto um problema relativamente simples pode se estender para diferentes aplicações, como é o caso da busca por padrões de strings. Estamos falando de cenários que vão desde aqueles comuns ao usuário menos avançado, como os mecanismos de busca em editores de texto e páginas web, aos que abrangem detecção de spam e hackeamento de senhas.

[1] Note que existem duas ocorrências desse padrão na entrada proposta.

Essa realidade torna mais fácil compreender porque são problemas recorrentes no estudo de computação, e porque é importante saber como explorar todos os recursos de hardware para aumentar a performance das soluções existentes.

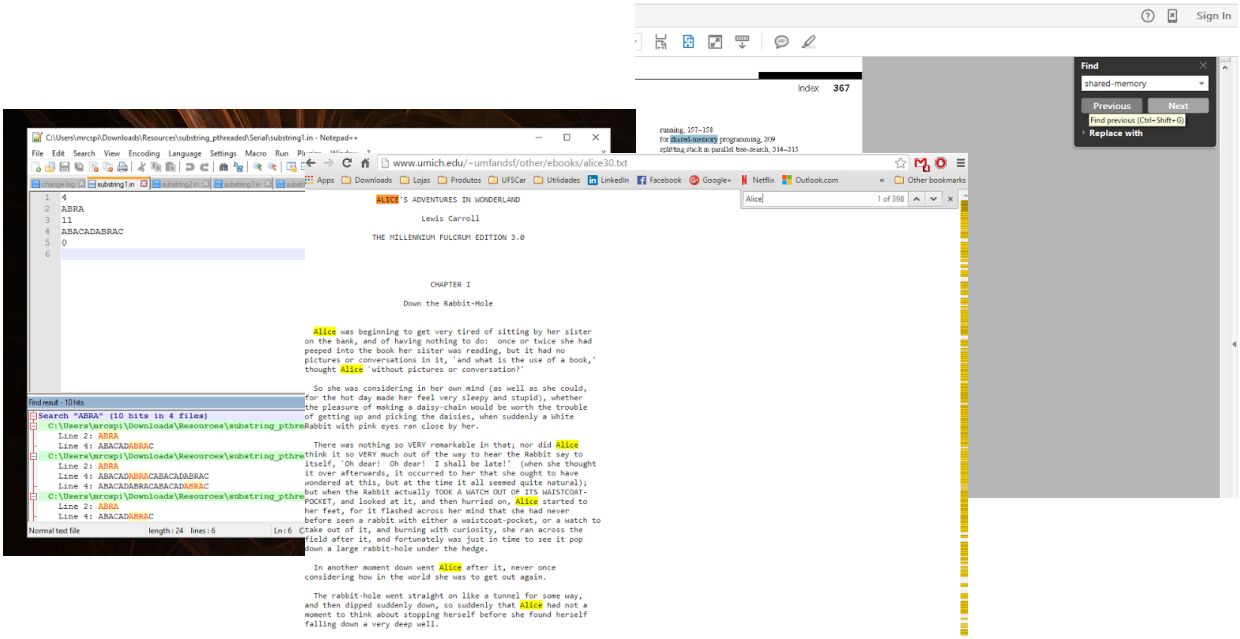


FIGURA 1 - MOSAICO COM APLICAÇÕES DA BUSCA POR PADRÕES

3.3. SOLUÇÃO INOCENTE

Existem três etapas essenciais para a solução[2] do problema proposto. A primeira e mais óbvia é a entrada de dados: é preciso descobrir qual é o padrão a ser buscado e onde. A segunda trata tanto de viabilizar a busca em mais de uma entrada como de encontrar mais de uma ocorrência em cada uma. E a terceira, trata da forma como encontrar cada ocorrência (ou a busca propriamente dita).

3.3.1 ENTRADA/SAÍDA

Essencialmente, a primeira e segunda etapas para a solução do problema constituem laços como o que podemos ver abaixo, que permite ao usuário informar onde buscar um padrão:

[2] Código-fonte disponível em arquivo de extensão .C.

```
scanf("%d", &M);      // size of pattern
      pat = malloc(sizeof(char)*M);
      scanf("%s", pat);     // pattern

      while (N != 0) {
            scanf("%d", &N);      // size of text
            if (N == 0)
                  break;

            txt = malloc(sizeof(char)*N);
            scanf("%s", txt);     // text
...
            if (!find(M, N, pat, txt))
                  printf("Not found!\n");
...
...
      }
```

3.3.2 BUSCA

Já a busca propriamente dita constitui uma função que recebe argumentos como o padrão e a entrada, para varrer um em função do outro:

```
int find(int M, int N, char *pat, char* txt) {
      int end = 0;
      int i, j;

      for( i = 0; i <= N-M; i++) {
            for (j = 0; j < M; j++) {
                  if (txt[i+j] != pat[j])
                        break;
                  if( j+1 == M) {
                        printf("Found at %d\n", i);
                        end = 1;
                        break;
                  }
            }
      }
      return end;
}
```

4. DESENVOLVENDO UMA SOLUÇÃO PARALELA

Detectar os aspectos adequados para a correta divisão de tarefas é uma atividade custosa no processo de propor uma “nova” solução para um problema conhecido. Por exemplo, no caso do pareamento de padrões, existem pelo menos três aspectos a se considerar:

- bloqueio de e/s;
- entradas múltiplas;
- busca sequencial.

Como se isso não fosse o bastante, é também preciso considerar o que explorar em termos de recursos de hardware antes de iniciar o desenvolvimento. Note que uma máquina servidora de alto desempenho irá se beneficiar mais quando a fragmentação da aplicação é maior (ou seja, quando há mais linhas de execução concorrentes), pois possui uma arquitetura que permite tirar proveito disso, enquanto uma máquina doméstica tende a apresentar melhor desempenho quando a fragmentação é controlada.

Para esse experimento, considere simular ambos os casos, no tocante à recursos de hardware. Assim será possível, ao menos, visualizar o gargalo que os componentes físicos trazem à “nova” solução. Aliás, o objetivo desse experimento será abordar também os três casos iniciais, de forma a cobrir todas as possibilidades e evidenciar os aspectos positivos e os riscos do desenvolvimento de soluções paralelas.

4.1. CONSIDERAÇÕES

Considere os trechos de código a partir daquele apresentado em 3.3.1. Esse é, por sinal, é o trecho que caracteriza não só comportamento bloqueante como entradas múltiplas, na versão serial. Tendo isso em mente, a primeira parte da proposta de paralelização aborda justamente uma reforma nesse segmento.

Assuma que é possível tratar em paralelo cada entrada do tipo `txt`. Essa possibilidade elimina comportamento bloqueante e, de quebra, permite um ganho de performance em máquinas com mais de um núcleo. Para conseguir esse resultado, nesse trecho temos código de mais baixo nível, desenvolvido com PThreads:

```
int thread_count = 0;
struct arg_struct *args;
long thread = -1; // long para sistemas 64 bits
pthread_t* thread_handles; // variável de threads
thread_handles = malloc(sizeof(pthread_t));

…

while (N != 0) {
     ...

     scanf("%d", &N); // tamanho da entrada
     if (N == 0)
          break;

     thread_count += 1;
     thread_handles = realloc(thread_handles,
thread_count*sizeof(pthread_t));
     ID += 1;

     txt = malloc(sizeof(char)*N);
     scanf("%s", txt);    // text

     // preenche a estrutura de argumentos para uma nova
thread
     args = malloc(sizeof(struct arg_struct));
     (*args).OMP = strtol(argv[1], NULL, 10);
     (*args).ID = ID;
     (*args).M = M;  (*args).N = N;
     (*args).pat = pat;
     (*args).txt = malloc(sizeof(char)*N);
strcpy((*args).txt, txt);
     ...

     // cria uma nova thread
     thread++;
     pthread_create(&thread_handles[thread], NULL, &find,
(void *) args);

     free(txt);
}

for ( thread = 0; thread < thread_count; thread++ )
     pthread_join(thread_handles[thread], NULL);
```

Como é possível observar no trecho acima, para evitar o comportamento bloqueante, um mesmo aspecto ganha uma interpretação positiva e outra negativa: o número de threads, nessa proposta, cresce dinamicamente, de acordo com o número de entradas do usuário.

Uma vez dependente do usuário, não é possível garantir que ocorra o balanceamento de carga entre as ramificações que podem ser criadas em uma execução do código proposto. Isso nos leva a considerar a possibilidade de balancear a carga de cada thread em si, trabalhando mais do que o comportamento bloqueante e as múltiplas entradas: paralelizando a busca propriamente dita. Observe a função:

```
void *find(void *arguments) {
      struct arg_struct *args = arguments;

      int end = 0;
      int i, j;

      /*
       * paraleliza a busca do algoritmo inocente, computando um
       * ganho que pode facilmente ultrapassar 100% do tempo
       * de esforço original.
       */
      # pragma omp parallel for private(j) num_threads(args->OMP)
      for( i = 0; i <= args->N - args->M; i++) {
            for (j = 0; j < args->M; j++) {
                  if (args->txt[i+j] != args->pat[j])
                        break;
                  if( j+1 == args->M) {
                        printf("Found at %d in text '%d'\n", i,
args->ID);
                        end = 1;
                        break;
                  }
            }
      }
      if (!end)
            printf("Not found in text '%d'!\n", args->ID);

      free(args);
      return EXIT_SUCCESS;
}
```

Como é possível observar no segmento acima, dessa vez são as ferramentas disponíveis por OpenMP que permitem distribuir a carga de cada thread. A diretiva aplicada

permite que máquinas com suporte à API possam ramificar ainda mais a aplicação, sem condenar a execução em outros casos. No entanto, a combinação entre o uso de PThreads e OpenMP, nesse caso, pode fazer com que o número de threads cresça muito rapidamente, e muito mais do que uma determinada máquina possa tirar proveito[3]. Para os propósitos desse experimento de paralelização, isso permite que avaliemos os possíveis cenários discutidos anteriormente, com um conjunto apropriado simulando entradas do tipo "pior caso".

4.2. ANÁLISES

Para verificar as diferenças de desempenho entre as versões paralelas e serial, foram realizados diversos testes, incluindo entradas que simulam 2, 4 e 8 linhas de execução (quando consideramos apenas[4] o crescimento dinâmico dado pela porção de código que faz uso de Pthreads). Todos esses testes[5] foram realizados mais de uma vez, a fim de garantir a acurácia dos resultados e, no caso do códigos paralelo, esse processo se repete para ramificações diferentes.

Vale lembrar que o cenário utilizado para essa verificação foi o de uma máquina configurada em modo de alta performance, com as seguintes características:

Sistema operacional: Ubuntu 14.04.3 LTS
Tipo de processador: QuadCore Intel Core i5-3570K, 3700 MHz (37 x 100)
Memória do Sistema: 8074 MB (DDR3-1600 DDR3 SDRAM)
Adaptador gráfico: NVIDIA GeForce GTX 970 (4 GB)
Armazenamento: Corsair Force 3 SSD (SATA-III)

Considere o resultado dos testes com parte do conjunto de entradas[6], selecionado para análise:

substring2.in

SERIAL	PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
17,45	15,50	11,03	8,37	7,33	7,20

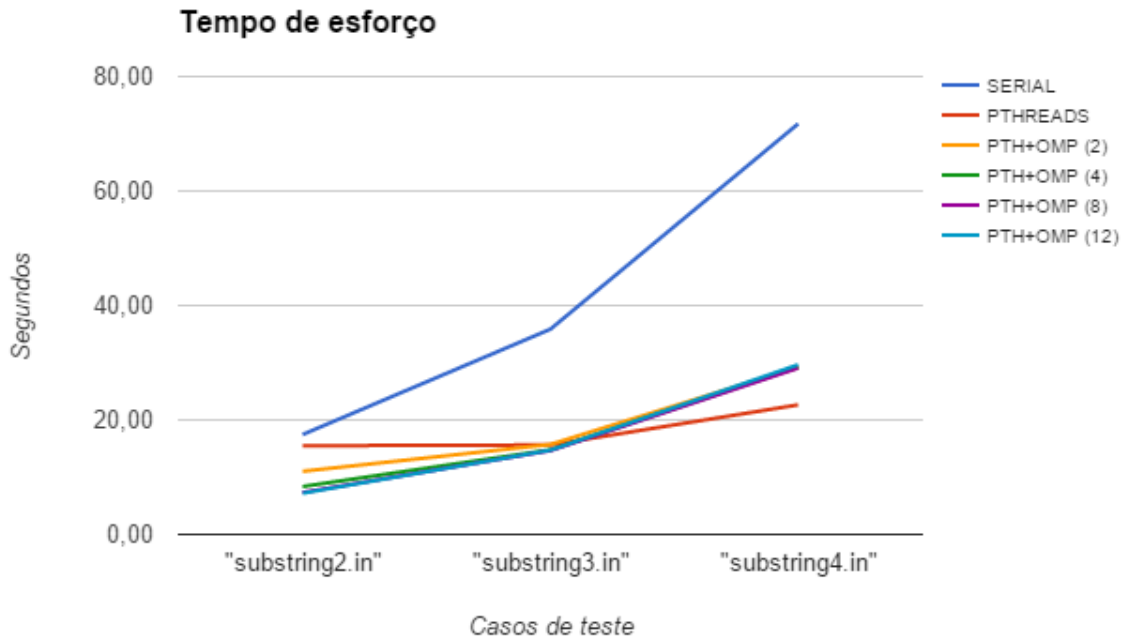
[3] Considere, por exemplo, quatro entradas e quatro ramificações em cada: isso já constituirá 16 threads executando em paralelo.
[4] Ao todo, podemos chegar a 96 linhas de execução: PTH + OMP (12) para *substring4.in* (8*12).
[5] Todos os valores são referentes à média dos resultados, dados em segundos.
[6] Cada caso simula o crescimento dinâmico de uma forma diferente, numa escala 2-4-8 threads, respectivamente.

substring3.in

SERIAL	PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
35,86	15,64	15,72	14,79	14,66	14,72

substring4.in

SERIAL	PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
71,75	22,64	29,20	29,23	29,07	29,65



Para a realização dos cálculos de *speedup*, utilizaremos a seguinte formula:

- *S = T serial / T paralelo*

Assuma que “T serial” é o tempo de execução do programa serial e “T paralelo” é o tempo de execução do programa paralelo.

substring2.in

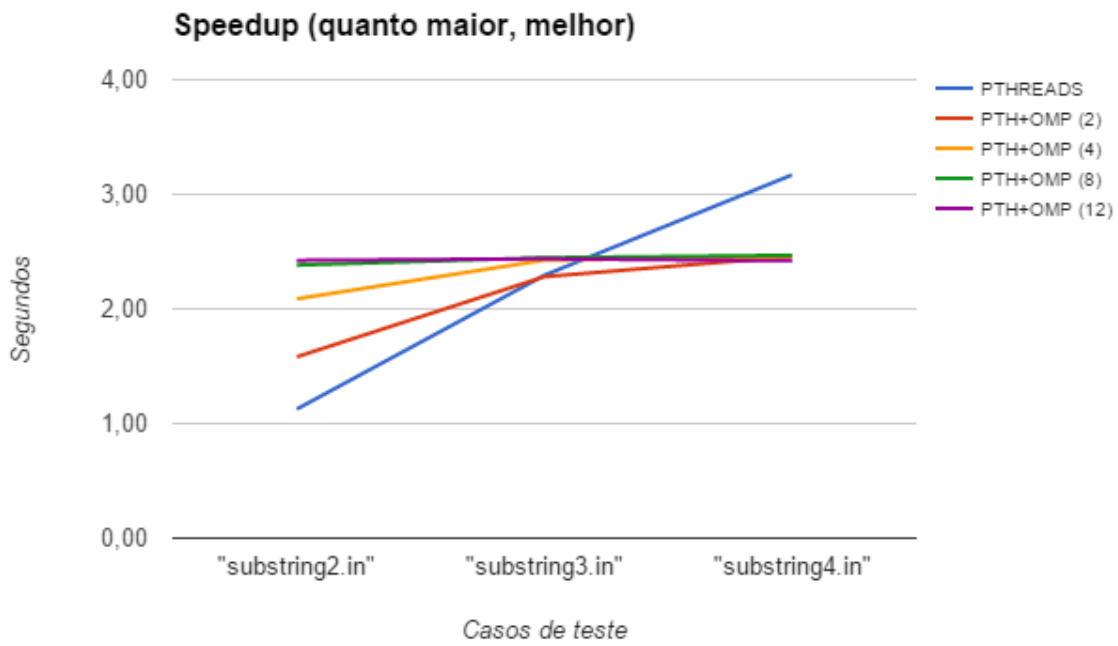
PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
1,13	1,58	2,09	2,38	2,42

substring3.in

PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
2,29	2,28	2,43	2,45	2,44

substring4.in

PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
3,17	2,46	2,45	2,47	2,42



Agora, munidos do *speedup* dos testes acima, calcularemos a eficiência do programa paralelo utilizando a seguinte fórmula:

- *E = S / p = (T seria / T paralelo) / p = T serial / p*T paralelo*

Assuma que “p” é o numero de linhas de execução, “T serial” é o tempo de execução do programa serial e “T paralelo” é o tempo de execução do programa paralelo.

substring2.in

PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)

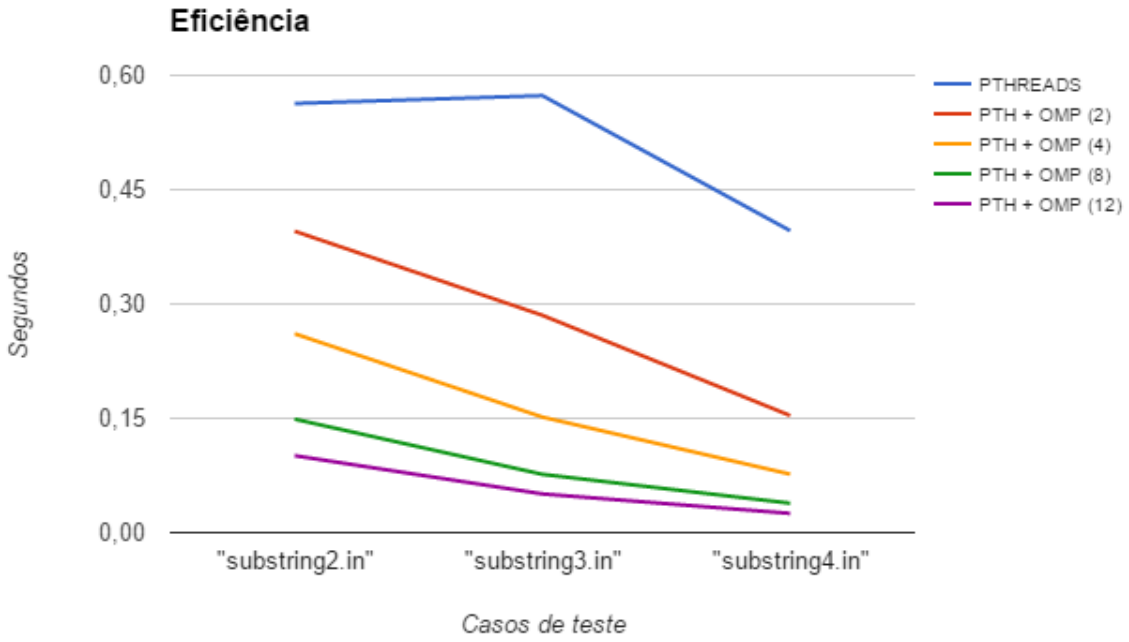
0,56	0,40	0,26	0,15	0,10

substring3.in

PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
0,57	0,29	0,15	0,08	0,05

substring4.in

PTH	PTH + OMP (2)	PTH + OMP (4)	PTH + OMP (8)	PTH + OMP (12)
0,40	0,15	0,08	0,04	0,03



5. CONCLUSÃO

A realização deste experimento proporcionou uma nova visão sobre a possibilidade de mesclar ferramentas de programação paralela para obter resultados mais robustos (ou até mesmo para expandir os limites impostos por apenas uma). A maturidade, bagagem do aprendizado obtido em atividades anteriores, também permitiu programar em um nível mais baixo, explicitamente controlando o comportamento da aplicação através de PThreads, e obter resultados promissores. Em adição, vale lembrar que também foi possível estressar o hardware de teste de forma a estabelecer um *breaking point* entre hardware e software: o momento em que extrair todo o potencial de uma máquina deixa de representar um ganho de performance.