Quadratura Adaptativa Paralela

Descrição dos testes realizados, problemas identificados e não resolvidos.

*por **Fernando Lima** (**2020877**)*

Neste trabalho duas variantes do algoritmo da quadratura adaptativa, utilizando OpenMP e uma com Pthreads. O programa utiliza X threads para computar a aproximação, pelo método de trapézios, da área abaixo da curva formada por uma função f. O número de trapézios a ser calculado para realizar a aproximação depende de uma tolerância. Em resumo, deve-se implementar um método seguindo as instruções:

1. Para determinado intervalo [a,b], o algoritmo calcula a área do trapézio com extremidades (a, b, f(a), f(b)), e a área dos dois subtrapézios formados por (a, (b-a)/2, f(a), f((b-a)/2)) e por ((b-a)/2, b, f((b-a)/2), f(b)).
2. Se a diferença entre a área do trapézio maior e a soma dos dois subtrapézios fica abaixo de uma tolerância, o cálculo da área entre a e b está completo.
3. Caso contrário, repet-se o mesmo procedimento para os trapézios menores. A tolerância na diferença entre um trapézio e os dois subtrapézios "seguintes" deve ser uma porcentagem da soma de áreas dos dois subtrapézios.

Os programas de teste devem:

- Aceitar facilmente a mudança para outras funções e intervalos, parametrizando o cálculo da integral pelos extremos e por um ponteiro de função.
- Executar cada variante para diferentes números de threads (1, 2 e 4), com algumas medidas preliminares dos tempos obtidos para cada combinação.
- Experimentar com diferentes tolerâncias e com diferentes funções;

A função escolhida foi o módulo da função *sinc*, dado por:

$$\int_l^r \left| \frac{sin(x)}{x} \right| dx$$

Primeira Variante

A primeira implementação faz cada thread calcula um subintervalo pelo qual será responsável e calcula o resultado para esse subintervalo inteiro. Quando todas as threads terminarem, a thread principal deve mostrar o resultado final. Essa variante foi implementada com pthreads e com OpenMP.

A quadratura adaptaiva pode ser implementada por uma função sequencia, esse bloco pode ser chamado e divido por mais threads, basta passar os intervalos, a função e aproximação como argumentos de entrada e o total de saída, conforme o critério de parada especificado na terceira instrução.

```
double* adaptavive_quadrature(adaptavive_quadrature_args* args)
{
```

```
    double* total = (double*) malloc(sizeof(double));
    double m = (args->r + args->l)/2;
    int approx = _acceptable_approx(args, total);

    if(/** Acceptable Aprox **/ approx)
    {
        //result
        return total;
    }
    else {

        //left
        adaptavive_quadrature_args largs = {args->l, m, args->func, args->approx};
        double* lresult = adaptavive_quadrature(&largs);

        //right
        adaptavive_quadrature_args rargs = {m, args->r, args->func, args->approx};
        double* rresult = adaptavive_quadrature(&rargs);

        *total = *lresult + *rresult;
        return total;
    }
}
```

Com esse bloco pode-se implementar rotinas usando Pthreads e OpenMP. A função Pthread é descrita abaixo

```
void* pthread_adaptavive_quadrature(void* arg, int num_intervals)
{
    adaptavive_quadrature_args* args = (adaptavive_quadrature_args*) arg;
    double* total = (double*) malloc(sizeof(double));
    *total = 0;

    adaptavive_quadrature_intervals intervals = _get_intervals(args,
num_intervals);

    pthread_t* thds = (pthread_t*) malloc(intervals.divisions *
sizeof(pthread_t));
    for (int i = 0; i < intervals.divisions; i++)
    {
        pthread_create(&(thds[i]), NULL, _call, &(intervals.args[i]));
    }
    void* res;
    for (int i = 0; i < intervals.divisions; i++)
    {
        pthread_join(thds[i], &res);
        *total += *(double*)res;
    }
    return total;
}
```

Enquanto a mesma rotina pode ser implementada, de forma análoga, utilizando blocos de *pragmas* com OpenMP.

```
void* omp_adaptavive_quadrature(adaptavive_quadrature_args* args, int
num_intervals)
{
    double* total = (double*) malloc(sizeof(double));
    *total = 0;

    adaptavive_quadrature_intervals intervals = _get_intervals(args,
num_intervals);
    double res;
    #pragma omp parallel
    {
        #pragma omp for private(res)
        for (int i = 0; i < intervals.divisions; i++)
        {
            res = *(adaptavive_quadrature(&(intervals.args[i])));
            #pragma omp critical (Total)
            {
                *total += res;
            }
        }
    }
    return total;
}
```

Ambas recebem o intervalo inicial, função e aproximação como argumentos e mais o número de intervalos que a função será quebrada e paralelizada. Nessa variante o número de intevalo é igual ao número de threads, cada thread iniciada, no design de *fork-and-join* irá calcular um intervalo e após todas as threads terminarem (*join*) o total é retornado.

Segunda Variante

Nessa segunda, a thread principal inicialmente cria uma lista de tarefas, contendo os extremos dos intervalos, com NUMINICIAL tarefas. Cada thread executa uma tarefa, e se ela gerar subtarefas, coloca uma delas na fila global e processa a outra, até que não encontre mais tarefas na fila. A thread principal espera as demais terminarem e mostra o resultado final. Essa variante foi implementada apenas com OpenMP.

A implementação pode ser divida em duas partes. Um método *worker*, na qual é executado pelas threads abertas pelo OpenMP e loop, até a a fila de tarefas estar vazia. E a outra parte é composta pelo método *administrator* na qual gerencia o *Pool* de threads e a lsita de tarefas.

```
void omp_adaptavive_quadrature_admin(
    double* total,
    adaptavive_quadrature_args* initial,
    Queue_v* queue,
    int num_intervals
)
```

```
{
    adaptavive_quadrature_intervals intervals = _get_intervals(initial,
num_intervals);
    for (int i = 0; i < intervals.divisions; i++)
    {
        enqueue(queue, &(intervals.args[i]));
    }

    int empty = 0;
    #pragma omp parallel
    {
        #pragma omp for firstprivate(empty)
        for (int i = 0; i < omp_get_num_threads(); i++)
        {
            while(!empty)
            {
                adaptavive_quadrature_args* initial;
                #pragma omp critical
                {
                    initial = !empty ? (adaptavive_quadrature_args*)
dequeue(queue) : NULL;
                }
                if(initial != NULL)
                    omp_adaptavive_quadrature_worker(total, initial, queue);
                empty = isEmpty(queue);
            }
        }
    }

}

void omp_adaptavive_quadrature_worker(
    double* total,
    adaptavive_quadrature_args* initial,
    Queue_v* queue
)
{
    adaptavive_quadrature_args* args = initial;
    double tarea = 0;
    double m;
    int approx;

    m = (args->r + args->l)/2;
    approx = _acceptable_approx(args, &tarea);


    if(!approx)
    {
        adaptavive_quadrature_args* largs =
create_adaptative_quadrature_args(args->l, m, args->func, args->approx);
        adaptavive_quadrature_args* rargs = create_adaptative_quadrature_args(m,
args->r, args->func, args->approx);
        #pragma omp critical
        {
```

```
            enqueue(queue, largs);
            enqueue(queue, rargs);
        }
    } else {
        *total += tarea;
    }

}
```

Note que nos métodos *administrator* e *worker* é determinada uma região crítica quando uma tarefa é inserida ou retirada da fila. Isso é feito para assegurar o sincronismo na fila compartilhada.

Resultados

Conforme estabelecido na introdução desse relatório, as trêws variantes (Pthread, OpenMP v1, OpenMP v2) foram avaliadas a partir do tempo de execução total de todas as threads abertas, com a função *abs_sinc* descrita na introdução, com três aproximações (0.1, 0.001, 0.0000001) com 1,2 e 4 threads abertas.

Foi utilizado o método de contagem de microsegundos da biblioteca *benchmark.h* criada para testar o desempenho de uma rotina.

```
import <benchmark.h>
...
int main()
{
  ...
  long int utime = (long int) ustopwatch(fork_join_routine, &args);
  printf("Elapsed: %ld microseconds\n\n", utime);
  ...
}
```

A função mede o tempo através do método *gettimeofday* :

```
double stopwatch(void (*routine)(void*), void* args)
{
    struct timeval current_time;

    gettimeofday(&current_time, NULL);
    double tic = (double)current_time.tv_sec + current_time.tv_usec / 1000000.0;

    routine(args);

    gettimeofday(&current_time, NULL);
    double toc = (double)current_time.tv_sec + current_time.tv_usec / 1000000.0;

    return (toc - tic);
}
...
double ustopwatch(void (*routine)(void*), void* args)
```

```
{
    double seconds = stopwatch(routine, args);
    return seconds * 1000000;
}
```

(desempenho, area)	1 thread			2 threads			4 threads		
	0.1	0.001	0.0000001	0.1	0.001	0.0000001	0.1	0.001	0.0000001
Pthread (us|u.a)	(341, 4.452)	(304, 4.755)	**(1204, 6.330)**	(499, 4.452)	(342, 4.755)	**(808, 6.330)**	(777, 6.623)	(787, 6.303)	(1517, 6,330)
OpenMP v1 (us|u.a)	(4545, 4.452)	(2069, 4.755)	**(6470, 6.330)**	(5942, 4.452)	(4467, 4.755)	**(4257, 6.330)**	(4591, 6.623)	(4884, 6.303)	**(7935, 6.330)**
OpenMP v2 (us|u.a)	(45, 4.452)	(50, 4.755)	**(1088, 6.330)**	(188, 4.452)	(200, 4.755)	**(1324, 6.330)**	(344, 4.452)	(321, 4.755)	**(1417, 6.330)**
WolframAlpha*	**Total: 6.330**								

**Obtido em: https://www.wolframalpha.com/input/?i2d=true&i=Integrate%5Babs%5C%2840%29sinc%5C%2840%29x%5C%2841%29%5C%2841%29%2C%7Bx%2C-25%2C25%7D%5D*

A tabela acima mostra os resultados obtidos pelo cálculo da área na função *abs_sinc* partindo dos intervalos (l,r) = (-25, 25), conforme mostrado o link acima. O WolframAlpha serviu como fonte de verdade para comparar os resultados obtidos com a variação de threads e aproximações. Dos resultados mostrados para tabela é seguro dizer que o valor converge para a fonte de verdade a partir da aproximação de *0.0000001* em todas as combinções de threads. Os valores foram medidos na casa dos microsegundos, por isso pequenas variações não são tão relevantes para se observar um desempenho significativo ao adicionar mais threads. Notoriamente, a implementação com Pthreads se mostrou a mais veloz entre as combinações e o Pool de threads na versão OpenMP v2 otimizou a versão v1 reduzindo o tempo os patamares da versão Pthreads (até com o desempenho levemente maior), portanto ouve uma melhora. Vale ressaltar que nesse patamar de tempo, é necessário considerar o tempo do processo abrir threads e a proximidade das memórias alocadas aos processadores, o que, nesse patamar, pode adicionar componentes de tempo as implementações.