Fórmula de Leibniz

1.0.0

Gerado por Doxygen 1.13.2

Capítulo 1

Índice dos Arquivos

1.1 Lista de Arquivos

Esta é a lista de todos os arquivos e suas respectivas descrições:

Capítulo 2

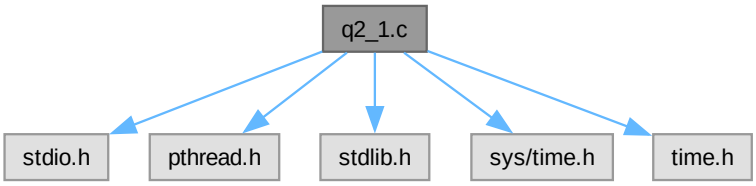
Arquivos

2.1 Referência do Arquivo q2_1.c

Calcula uma aproximação de Pi usando a fórmula de Leibniz com múltiplas threads.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
```

Gráfico de dependência de inclusões para q2_1.c:



Definições e Macros

- #define _POSIX_C_SOURCE 199309L
- #define SIZE 2000000000

  O número total de termos a serem calculados na série de Leibniz.
- #define NUM_THREADS 2

  O número de threads a serem usadas para paralelizar o cálculo.
- #define PARTIAL_NUM_TERMS ((SIZE) / (NUM_THREADS))

  O número de termos que cada thread irá processar.

Funções

- double calcular_tempo ()

  Calcula o tempo atual de alta precisão.
- long double partialFormula (int start_term)

  Calcula uma soma parcial da série de Leibniz.
- void ∗ partialProcessing (void ∗args)

  Função executada por cada thread.
- int main ()

  Ponto de entrada principal do programa.

Variáveis

- long double result = 0

  Variável global para armazenar o resultado final da aproximação de Pi.
- pthread_mutex_t mutex

  Mutex para sincronizar o acesso à variável global `result`.

2.1.1 Descrição detalhada

Calcula uma aproximação de Pi usando a fórmula de Leibniz com múltiplas threads.

Este programa divide o cálculo da série de Leibniz entre um número definido de threads para acelerar a computação. Cada thread calcula uma porção da série, e os resultados parciais são somados de forma segura usando um mutex para produzir o resultado final. O tempo de execução de cada thread e o tempo total de execução são medidos e exibidos.

Definição no arquivo q2_1.c.

2.1.2 Definições e macros

2.1.2.1 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 199309L
```

Definição na linha 11 do arquivo q2_1.c.

2.1.2.2 NUM_THREADS

```
#define NUM_THREADS 2
```

O número de threads a serem usadas para paralelizar o cálculo.

Definição na linha 28 do arquivo q2_1.c.

2.1.2.3 PARTIAL_NUM_TERMS

```
#define PARTIAL_NUM_TERMS ((SIZE) / (NUM_THREADS))
```

O número de termos que cada thread irá processar.

É calculado como o número total de termos (SIZE) dividido pelo número de threads (NUM_THREADS). Note que o resto da divisão não é distribuído uniformemente neste cálculo.

Definição na linha 37 do arquivo q2_1.c.

2.1.2.4 SIZE

```
#define SIZE 2000000000
```

O número total de termos a serem calculados na série de Leibniz.

Definição na linha 22 do arquivo q2_1.c.

2.1.3 Funções

2.1.3.1 calcular_tempo()

```
double calcular_tempo ()
```

Calcula o tempo atual de alta precisão.

Calcula o tempo atual do sistema com alta precisão.

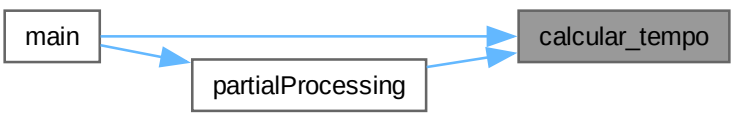
Retorna

O tempo atual em segundos, como um valor double. Utiliza CLOCK_MONOTONIC para garantir que o tempo seja sempre crescente e não afetado por mudanças no relógio do sistema.

O tempo atual em segundos, como um valor double. Utiliza CLOCK_MONOTONIC para medições de tempo que não são afetadas por mudanças no relógio do sistema.

Definição na linha 61 do arquivo q2_1.c.

Esse é o diagrama das funções que utilizam essa função:



2.1.3.2 main()

```
int main ()
```

Ponto de entrada principal do programa.

Inicializa o mutex, cria e gerencia as threads, distribui o trabalho entre elas, aguarda a conclusão de todas as threads, e então calcula e exibe o resultado final da aproximação de Pi e o tempo total de execução. Por fim, destrói o mutex.

Retorna

EXIT_SUCCESS em caso de sucesso.

A função `main` inicializa o mutex, cria e gerencia as threads, mede o tempo total de execução, e exibe o resultado final.

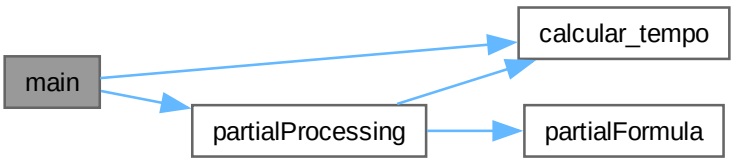
- Inicializa o mutex.
- Inicia a contagem do tempo total.
- Cria `NUM_THREADS` threads, passando a cada uma o seu termo inicial para o cálculo. A última thread é designada para calcular quaisquer termos remanescentes da divisão inteira.
- Aguarda a conclusão de todas as threads com `pthread_join`.
- Para a contagem do tempo total.
- Destrói o mutex.
- Imprime o valor aproximado de Pi e o tempo total de execução.

Retorna

EXIT_SUCCESS em caso de sucesso.

Definição na linha 139 do arquivo q2_1.c.

Este é o diagrama das funções utilizadas por essa função:



2.1.3.3 partialFormula()

```
long double partialFormula (
            int start_term)
```

Calcula uma soma parcial da série de Leibniz.

A fórmula é: (-1)^k / (2k + 1) Esta função calcula um número fixo de termos (`PARTIAL_NUM_TERMS`) a partir de um índice inicial.

Parâmetros

start_term	O índice 'k' inicial para o somatório.

Retorna

A soma parcial calculada como um `long double`.

A fórmula de Leibniz para Pi é: /4 = 1 - 1/3 + 1/5 - 1/7 + ... Esta função calcula um segmento desta série, começando em `start_term` e continuando por `PARTIAL_NUM_TERMS` iterações.

Parâmetros

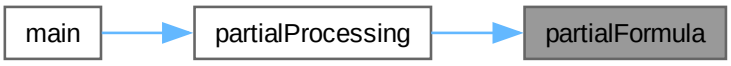
start_term	O índice inicial do termo na série a partir do qual o cálculo deve começar.

Retorna

A soma parcial dos termos calculados como um `long double`.

Definição na linha 79 do arquivo q2_1.c.

Esse é o diagrama das funções que utilizam essa função:



2.1.3.4 partialProcessing()

```
void * partialProcessing (
            void * args)
```

Função executada por cada thread.

A função de trabalho executada por cada thread.

Esta função gerencia o trabalho de uma única thread. Ela recebe o termo inicial, calcula a soma parcial chamando `partialFormula`, mede seu próprio tempo de execução, e adiciona seu resultado parcial à variável global `result` de forma segura.

Parâmetros

args	Um ponteiro para um inteiro alocado dinamicamente, que representa o termo inicial para o cálculo desta thread. A memória para `args` é liberada dentro desta função.

Retorna

NULL.

Esta função recebe o termo inicial para seu cálculo, chama `partialFormula` para obter a soma parcial, mede o tempo de execução dessa tarefa, e então adiciona seu resultado à variável global `result` de forma segura, usando um mutex.

Parâmetros

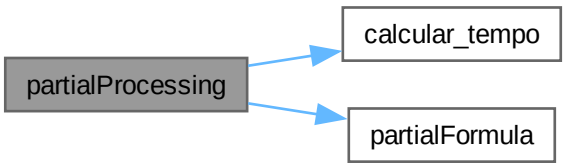
args	Um ponteiro para um inteiro alocado dinamicamente que contém o termo inicial para o cálculo desta thread. A memória para `args` é liberada dentro da função.

Retorna

NULL.

Definição na linha 109 do arquivo q2_1.c.

Este é o diagrama das funções utilizadas por essa função:



Esse é o diagrama das funções que utilizam essa função:

2.1.4 Variáveis

2.1.4.1 mutex

```
pthread_mutex_t mutex
```

Mutex para sincronizar o acesso à variável global `result`.

Mutex para garantir o acesso exclusivo à variável global `result` pelas threads.

Definição na linha 52 do arquivo q2_1.c.

2.1.4.2 result

```
long double result = 0
```

Variável global para armazenar o resultado final da aproximação de Pi.

Variável global para armazenar a soma final de todas as aproximações parciais de Pi. O acesso a esta variável é protegido por um mutex para evitar condições de corrida.

Esta variável é acessada por todas as threads e protegida por um mutex para evitar condições de corrida.

Definição na linha 46 do arquivo q2_1.c.

2.2 q2_1.c

Ir para a documentação desse arquivo.

```
00001
00010
00011 #define _POSIX_C_SOURCE 199309L
00012 #include <stdio.h>
00013 #include <pthread.h>
00014 #include <stdlib.h>
00015 #include <sys/time.h>
00016 #include <time.h>
00017
00022 #define SIZE 2000000000
00023
00028 #define NUM_THREADS 2
00029
00037 #define PARTIAL_NUM_TERMS ((SIZE) / (NUM_THREADS))
00038
00046 long double result = 0;
00047
00052 pthread_mutex_t mutex;
00053
00061 double calcular_tempo()
00062 {
00063     struct timespec time;
00064     clock_gettime(CLOCK_MONOTONIC, &time);
00065     return (double)time.tv_sec + (double)time.tv_nsec / 1e9;
00066 }
00067
00079 long double partialFormula(int start_term)
00080 {
00081
00082     const int num_terms = start_term + PARTIAL_NUM_TERMS;
00083
00084     long double pi_approximation = 0;
00085     double signal = 1.0;
00086
00087     for (int k = start_term; k < num_terms; k++)
00088     {
00089         pi_approximation += signal / (2 * k + 1);
00090         signal *= -1.0;
00091     }
00092
00093     return pi_approximation;
00094 }
00095
00109 void *partialProcessing(void *args)
00110 {
00111     pthread_t tid = pthread_self();
00112     int first_therm = *((int *)args);
00113     free(args);
00114
00115     double initial_time = calcular_tempo();             // comeca a contar o tempo de inicio
00116     long double sum = partialFormula((int)first_therm); // faz o calculo dos valores referentes a essa
    thread
00117     double end_time = calcular_tempo();                 // comeca a contar o tempo de fim
00118     double final_time = end_time - initial_time;        // calcula p tempo final
00119
00120     pthread_mutex_lock(&mutex);
00121     result += 4 * sum;
00122     pthread_mutex_unlock(&mutex);
00123
```

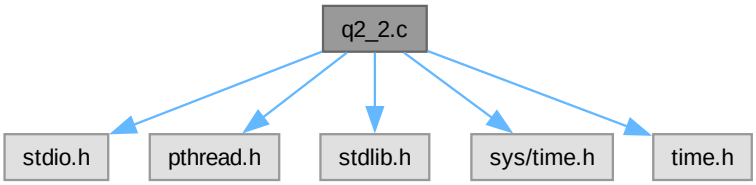
```
00124     printf("TID: %lu : %.2fs\n", (unsigned long)tid, final_time); // mostrar TID e tempo empregado na
      thread
00125
00126     return NULL;
00127 }
00128
00139 int main()
00140 {
00141
00142     // criar um mutex
00143     pthread_mutex_init(&mutex, NULL);
00144     // criar as threads
00145     pthread_t thread[NUM_THREADS];
00146     unsigned long args[NUM_THREADS];
00147
00148     // dividir o tamanho total pelo número de threads
00149     long long remainder = SIZE % NUM_THREADS;
00150
00151     // começamos a calcular o tempo de inicio do procesamento
00152     printf("Começando a calcular o valor de pi da série de Leibniz, com %d threads\n", NUM_THREADS);
00153     double total_start_time = calcular_tempo();
00154
00155     for (int i = 0; i < NUM_THREADS; ++i)
00156     {
00157         int *init = malloc(sizeof(int));
00158         *init = i * PARTIAL_NUM_TERMS;
00159
00160         int terms_to_compute = PARTIAL_NUM_TERMS;
00161
00162         // A última thread pega os termos restantes
00163         if (i == NUM_THREADS - 1)
00164         {
00165             terms_to_compute += SIZE % NUM_THREADS;
00166         }
00167
00168         pthread_create(&thread[i], NULL, partialProcessing, (void *)init);
00169     }
00170
00171     for (int i = 0; i < NUM_THREADS; ++i)
00172     {
00173         pthread_join(thread[i], NULL);
00174     }
00175
00176     double total_time_end = calcular_tempo();
00177     double total_final_time = total_time_end - total_start_time;
00178
00179     /* Liberar o mutex */
00180     pthread_mutex_destroy(&mutex);
00181
00182     printf("\nValor aproximado de pi: %.15Lf\n", result);
00183     printf("Tempo total de execução: %.2fs\n", total_final_time);
00184
00185     return EXIT_SUCCESS;
00186 }
```

2.3 Referência do Arquivo q2_2.c

Calcula uma aproximação de Pi usando a fórmula de Leibniz com paralelismo de threads.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
```

Gráfico de dependência de inclusões para q2_2.c:



Definições e Macros

- #define _POSIX_C_SOURCE 199309L
- #define SIZE 2000000000

  O número total de termos a serem calculados na série de Leibniz.
- #define NUM_THREADS 2

  O número de threads a serem usadas para paralelizar o cálculo.
- #define PARTIAL_NUM_TERMS ((SIZE) / (NUM_THREADS))

  O número de termos que cada thread irá processar. Calculado como SIZE / NUM_THREADS. O resto da divisão é tratado na thread principal.

Funções

- double calcular_tempo ()
- long double partialFormula (int start_term)
- void * partialProcessing (void *args)
- int main ()

Variáveis

- long double result = 0
- pthread_mutex_t mutex

2.3.1 Descrição detalhada

Calcula uma aproximação de Pi usando a fórmula de Leibniz com paralelismo de threads.

Este programa divide o cálculo da série de Leibniz em várias threads para acelerar o processamento. Cada thread calcula uma parte da série, e os resultados parciais são somados de forma segura usando um mutex para produzir a aproximação final de Pi. O tempo de execução de cada thread e o tempo total são medidos e exibidos.

Definição no arquivo q2_2.c.

2.3.2 Definições e macros

2.3.2.1 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 199309L
```

Definição na linha 12 do arquivo q2_2.c.

2.3.2.2 NUM_THREADS

```
#define NUM_THREADS 2
```

O número de threads a serem usadas para paralelizar o cálculo.

Definição na linha 29 do arquivo q2_2.c.

2.3.2.3 PARTIAL_NUM_TERMS

```
#define PARTIAL_NUM_TERMS ((SIZE) / (NUM_THREADS))
```

O número de termos que cada thread irá processar. Calculado como SIZE / NUM_THREADS. O resto da divisão é tratado na thread principal.

Definição na linha 36 do arquivo q2_2.c.

2.3.2.4 SIZE

```
#define SIZE 2000000000
```

O número total de termos a serem calculados na série de Leibniz.

Definição na linha 23 do arquivo q2_2.c.

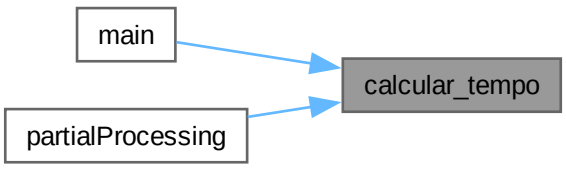
2.3.3 Funções

2.3.3.1 calcular_tempo()

```
double calcular_tempo ()
```

Definição na linha 57 do arquivo q2_2.c.

Esse é o diagrama das funções que utilizam essa função:

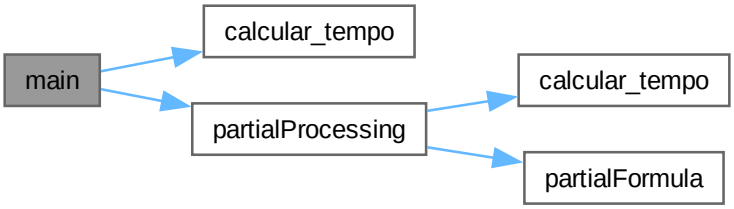


2.3.3.2 main()

```
int main ()
```

Definição na linha 141 do arquivo q2_2.c.

Este é o diagrama das funções utilizadas por essa função:



2.3.3.3 partialFormula()

```
long double partialFormula (
            int start_term)
```

Definição na linha 75 do arquivo q2_2.c.

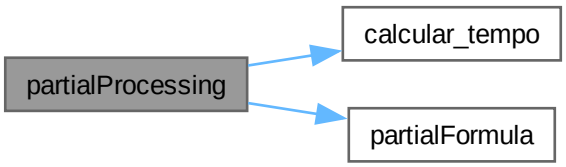
Esse é o diagrama das funções que utilizam essa função:

2.3.3.4 partialProcessing()

```
void * partialProcessing (
            void * args)
```

Definição na linha 104 do arquivo q2_2.c.

Este é o diagrama das funções utilizadas por essa função:



2.3.4 Variáveis

2.3.4.1 mutex

```
pthread_mutex_t mutex
```

Definição na linha 49 do arquivo q2_2.c.

2.3.4.2 result

```
long double result = 0
```

Definição na linha 43 do arquivo q2_2.c.

2.4 q2_2.c

Ir para a documentação desse arquivo.

```
00001
00011
00012 #define _POSIX_C_SOURCE 199309L
00013 #include <stdio.h>
00014 #include <pthread.h>
00015 #include <stdlib.h>
00016 #include <sys/time.h>
00017 #include <time.h>
00018
00023 #define SIZE 2000000000
00024
00029 #define NUM_THREADS 2
00030
00036 #define PARTIAL_NUM_TERMS ((SIZE) / (NUM_THREADS))
00037
00043 long double result = 0;
00044
00049 pthread_mutex_t mutex;
00050
00057 double calcular_tempo()
00058 {
00059     struct timespec time;
00060     clock_gettime(CLOCK_MONOTONIC, &time);
00061     return (double)time.tv_sec + (double)time.tv_nsec / 1e9;
00062 }
00063
00075 long double partialFormula(int start_term)
00076 {
```

```
00077
00078     const int num_terms = start_term + PARTIAL_NUM_TERMS;
00079
00080     long double pi_approximation = 0;
00081     double signal = 1.0;
00082
00083     for (int k = start_term; k < num_terms; k++)
00084     {
00085         pi_approximation += signal / (2 * k + 1);
00086         signal *= -1.0;
00087     }
00088
00089     return pi_approximation;
00090 }
00091
00104 void *partialProcessing(void *args)
00105 {
00106     pthread_t tid = pthread_self();
00107     int first_therm = *((int *)args);
00108     free(args);
00109
00110     double initial_time = calcular_tempo();             // comeca a contar o tempo de inicio
00111     long double sum = partialFormula((int)first_therm); // faz o calculo dos valores referentes a essa
    thread
00112     double end_time = calcular_tempo();                 // comeca a contar o tempo de fim
00113     double final_time = end_time - initial_time;        // calcula p tempo final
00114
00115     pthread_mutex_lock(&mutex);
00116     result += 4 * sum;
00117     pthread_mutex_unlock(&mutex);
00118
00119     printf("TID: %lu : %.2fs\n", (unsigned long)tid, final_time); // mostrar TID e tempo empregado na
    thread
00120
00121     return NULL;
00122 }
00123
00141 int main()
00142 {
00143
00144     // criar um mutex
00145     pthread_mutex_init(&mutex, NULL);
00146     // criar as threads
00147     pthread_t thread[NUM_THREADS];
00148     unsigned long args[NUM_THREADS];
00149
00150     // dividir o tamanho total pelo número de threads
00151     long long remainder = SIZE % NUM_THREADS;
00152
00153     // começamos a calcular o tempo de inicio do procesamento
00154     printf("Começando a calcular o valor de pi da série de Leibniz, com %d threads\n", NUM_THREADS);
00155     double total_start_time = calcular_tempo();
00156
00157     for (int i = 0; i < NUM_THREADS; ++i)
00158     {
00159         int *init = malloc(sizeof(int));
00160         *init = i * PARTIAL_NUM_TERMS;
00161
00162         int terms_to_compute = PARTIAL_NUM_TERMS;
00163
00164         // A última thread pega os termos restantes
00165         if (i == NUM_THREADS - 1)
00166         {
00167             terms_to_compute += SIZE % NUM_THREADS;
00168         }
00169
00170         pthread_create(&thread[i], NULL, partialProcessing, (void *)init);
00171     }
00172
00173     for (int i = 0; i < NUM_THREADS; ++i)
00174     {
00175         pthread_join(thread[i], NULL);
00176     }
00177
00178     double total_time_end = calcular_tempo();
00179     double total_final_time = total_time_end - total_start_time;
00180
00181     /* Liberar o mutex */
00182     pthread_mutex_destroy(&mutex);
00183
00184     printf("\nValor aproximado de pi: %.15Lf\n", result);
00185     printf("Tempo total de execução: %.2fs\n", total_final_time);
00186
00187     return EXIT_SUCCESS;
00188 }
```

Índice Remissivo