UNIVERSIDADE FEDERAL DO PARANÁ

ARTHUR MUNHOZ AMARAL
GRR20177243

TRABALHO 1: UTILIZAÇÃO DE PROCESSOS E THREADS EM C

LINK: https://youtu.be/E0C7nA25mv4

CURITIBA
2020

SUMÁRIO

INTRODUÇÃO

A fim de evidenciar o real funcionamento dos processos e threads em c, foi proposta a implementação de dois programas que fizessem o cálculo do fatorial de um número. Os códigos completos podem ser vistos em: https://github.com/aar7hur/EmbeddedSystems.

O relatório tem como finalidade discutir e discorrer a respeito da diferença de tempo de execução do fatorial quando implementado com mais processos ou mais threads.

Componentes do sistema utilizados

Os componentes de hardware do computador utilizados são:

O comando lscpu retorna informações a respeito da CPU, podem ser vistas abaixo.

```
arthur@note-Arthur:~$ lscpu
Arquitetura:                x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes:            Little Endian
CPU(s):                     2
Lista de CPU(s) on-line:    0,1
Thread(s) per núcleo:       2
Núcleo(s) por soquete:      1
Soquete(s):                 1
Nó(s) de NUMA:              1
ID de fornecedor:           AuthenticAMD
Família da CPU:             21
Modelo:                     19
Nome do modelo:             AMD A4-5150M APU with Radeon(tm) HD Graphics
Step:                       1
CPU MHz:                    2694.862
CPU MHz máx.:               2700,0000
CPU MHz mín.:               1400,0000
BogoMIPS:                   5389.75
Virtualização:              AMD-V
cache de L1d:               16K
cache de L1i:               64K
cache de L2:                1024K
CPU(s) de nó0 NUMA:         0,1
```

Figura 1: Versão da CPU

O comando hwinfo --memory retorna informações a respeito da memória usada e pode ser visto na imagem abaixo:

```
arthur@note-Arthur:~$ hwinfo --memory
01: None 00.0: 10102 Main Memory
  [Created at memory.74]
  Unique ID: rdCR.CxwsZFjVASF
  Hardware Class: memory
  Model: "Main Memory"
  Memory Range: 0x00000000-0x1bfb50fff (rw)
  Memory Size: 7 GB
  Config Status: cfg=new, avail=yes, need=no, active=unknown
```

Figura 2: Quantidade de memória

O comando lsb_release -a retorna as informações sobre o sistema operacional utilizado

```
arthur@note-Arthur:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.5 LTS
Release:        18.04
Codename:       bionic
```

Figura 3: Versão do SO

Algoritmo utilizando *fork*

O principal conceito do fork é criar um processo filho a partir de um processo parental. Quando o fork é chamado, uma cópia do processo filho é gerada. O algoritmo tem como entrada dois parâmetros: o número do fatorial a ser calculado e o número de processos a ser utilizado. O trecho do código que faz a verificação das entradas do usuário pode ser visto abaixo:

```
boolean validUserInput(int argc, char *argv[])
{

   // if number of CLI inputs is different from 3
   if (argc != 3)
   {
       fprintf(stderr, "Arguments are missing to run the application\n\n");
       return False;
   }

   argvNumberProcess = (uint8_t)atoi(argv[2]);
   argvFat = atoi(argv[1]);

   // All CLI arguments must be grater than 0
   if ((argvFat <= 0) || (argvNumberProcess <= 0) || (argvNumberProcess > MAX_NUMBER_PROCESS))
   {
       fprintf(stderr, "Arguments must be greater than 0\n\n");
       return False;
   }
   else
   {
       return True;
   }

}
```

A validação consiste em verificar se o usuário entrou com dois parâmetros - número do fatorial e número de processos. Tais parâmetros devem ser necessariamente maiores do que zero.

Após a validação da entrada do usuário o script, é criado um array de structs. Cada posição desse array representa um processo. Esse array de structs é preenchido conforme demonstra a imagem abaixo.

```
childProcessStruct *organizeStructForEveryProcess(int numberProcess, int factorialNumber)
{
   int initEndNumber = 0, rest = 0, division = 0;

   // Allocate memory for all child structs
   childProcessStruct * childs = (childProcessStruct *)malloc(numberProcess *
sizeof(childProcessStruct));


   // If the rest of the division is 1 a process will calculate one more number
   if (factorialNumber % numberProcess)
   {
```

```
        rest = 1;
    }

    division = (int)factorialNumber/numberProcess;

    // If division is equal to the number to calculate factorial,
    // only one process will be used, so return without continue.
    if (division == factorialNumber)
    {
        return childs;
    }

    for (uint8_t counter = 0; counter < numberProcess; counter++)
    {
        initEndNumber = ((numberProcess - (counter+1))*division) +1;
        if (counter == 0)
        {
            initEndNumber = initEndNumber - rest;
        }

        childs[counter].startNumber = factorialNumber;
        childs[counter].endNumber = initEndNumber;
        factorialNumber = initEndNumber - 1;
    }

    return childs;
}
```

Cada struct terá um range de fatorial para calcular. Esse range é calculado através da seguinte lei de formação:

```
initEndNumber = ((numberProcess - (counter+1))*division) +1;
```

O número inicial é calculado com base no número de processos, e no quociente do número fatorial pela quantidade de processos usada.

Após ser feito o condicionamento de cálculo para cada processo, um array de file descriptor é criado no arquivo process.c. Este é usado para fazer a comunicação entre os processos filhos e o processo pai. Cada filho tem sua própria struct e sabe qual o range de fatorial que precisa calcular.

```
if (numberProcess > MIN_PROCESS)
    {
        // Creates variables to handle with creating proccess
        // use in file descriptor and aux variables
        pid_t process;
        int fileDescriptor[numberProcess][FILE_DESCRIPTOR_LEN];
        uint8_t counter = 0;

        while (counter < numberProcess)
        {
            if  (pipe(fileDescriptor[counter]))
            {
                perror("Pipe Error\n");
                _exit(EXIT_FAILURE);
            }
            process = fork();
```

```
        // If occurs some erros in creating fork
        if (process == ERROR)
        {
            printf("Error creating child process");
            _exit(EXIT_FAILURE);
        }

        // if the process is a child of the parent
        else if(process == CHILD_PROCESS)
        {
            // close read file descriptor because child w'ont use it
            close(fileDescriptor[counter][READ_FD]);

            // performs factorial and writes to file descriptor
            writeToPipe(fileDescriptor[counter], &data[counter]);
              // close write file descriptor because child w'ont use it
             close(fileDescriptor[counter][WRITE_FD]);
             exit(CHILD_FINISHED);
         }

         counter++;
      }
```

Por outro lado, processo pai fica lendo o túnel de comunicação entre o filho-pai e multiplicando uma variável resultado, conforme abaixo.

```
// If the process is parent
       if (process > 0)
      {
         counter = 0;

         while (counter < numberProcess)
         {
             long double parcialResult = 0;

             // close write file descriptor because child w'ont use it
             close(fileDescriptor[counter][WRITE_FD]);

             // waits until any child process finish
             wait(NULL);

             // read from file descriptor and saves into parcialResult variable
             read(fileDescriptor[counter][READ_FD], &parcialResult, sizeof(parcialResult));

             // final result is is multiplied and incremented at each loop
             result->result = parcialResult * result->result;

             // close read file descriptor because child w'ont use it
             close(fileDescriptor[counter][READ_FD]);

             counter++;
         }
      }
```

A main do código do processo pode ser vista abaixo:

```c
int main(int argc, char *argv[])
{
   // Gets initial clock to save how many time programm needs to execute
   struct timeval begin, end;
   gettimeofday(&begin, 0);


   // Checks if user entered with valid inputs
   if (validUserInput(argc, argv) == False)
   {
       exit(EXIT_FAILURE);
   }

   // Creates file descriptor to communicate between
   // process using PIPE
   int fileDescriptor[2], nbytes;

   // Allocate memory to create an array to separate
   // the numbers that the processes will calculate
   childProcessStruct *childs = organizeStructForEveryProcess(argvNumberProcess, argvFat);

   // Allocate memory to direct access struct
   pipeStruct *data = allocateMemory(sizeof(*data));
   data->numberProcess = argvNumberProcess;
   data->factorialNumber = argvFat;
   data->result = MIN_PROCESS;

   // Handles with all process
   handleProcess(childs, data, argvNumberProcess);

   printf("\nFactorial Result is: %Lf\n", data->result);
   free(data);
   free(childs);

   gettimeofday(&end, 0);
   long seconds = end.tv_sec - begin.tv_sec;
   long microseconds = end.tv_usec - begin.tv_usec;
   double elapsedTime = seconds + microseconds*1e-6;
   saveToCsvFile(elapsedTime, argvNumberProcess, argvFat);

}
```

Algoritmo utilizando *Thread*

O principal conceito da Thread é ter múltiplos núcleos rodando em paralelo e que podem utilizar memória compartilhada. Quando a thread é chamada ela guarda um ponteiro do programa que chamou. Além disso, ela pode ser direcionada para uma função . O algoritmo tem como entrada dois parâmetros: o número do fatorial a ser calculado e o número de processos a ser utilizado. O trecho do código que faz a verificação das entradas do usuário é o mesmo utilizado pelo fork e pode ser visto na página 5.

Após a validação da entrada do usuário o script, é criado um array de structs. Cada posição desse array representa uma thread. Esse array de structs é preenchido conforme demonstrado abaixo.

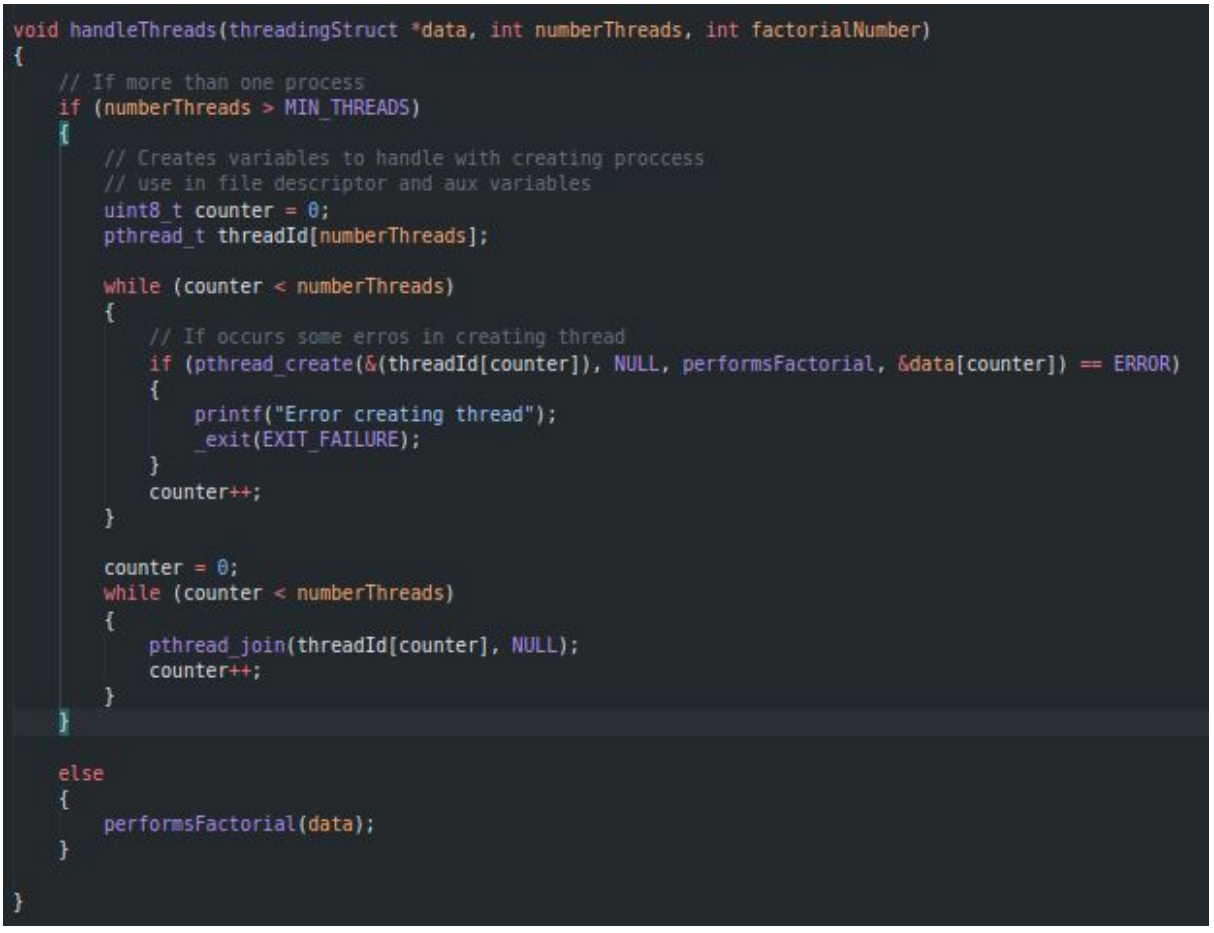
```
threadingStruct *organizeStructForEveryThread(int numberThreads, int factorialNumber)
{
   int initEndNumber = 0, rest = 0, division = 0;

   // Allocate memory for all child structs
   threadingStruct * thread = (threadingStruct *)malloc(numberThreads * sizeof(threadingStruct));

   // If the rest of the division is 1 a process will calculate one more number
   if (factorialNumber % numberThreads)
   {
       rest = 1;
   }

   division = (int)factorialNumber/numberThreads;
   /*
   if (division == factorialNumber)
   {
       printf("retornando thread")
       return thread;
   }
   */
   for (uint8_t counter = 0; counter < numberThreads; counter++)
   {
       // init number is calculated using the number of threads passed by the user,
       // the division between the number to be calculated, the factorial and the
       // number of processes.
       // Counter -->  is the variable used to make the looping interval. It is added to one
       // in the formula because it starts at 0.
       // Division --> is the result of dividing the number to be calculated and the number
       // of processes. It is used to know how many multiplications each process will do.
       // +1 -->  in the formula is used because the starting number is always 1 more than the
       // previous ending number. Example:
       //                       start number 120; end number 81; start number 80; end number 41
       // Rest is the remainder of the division of the number to be calculated with the number of
       // processes. If the rest is 1, one of the processes will multiply 1 more number
       //

       initEndNumber = ((numberThreads - (counter+1))*division) +1;
       if (counter == 0)
       {
           initEndNumber = initEndNumber - rest;
       }

        thread[counter].startNumber = factorialNumber;
        thread[counter].endNumber = initEndNumber;
        factorialNumber = initEndNumber - 1;
   }
   return thread;
```

Cada struct terá um range de fatorial para calcular. Esse range é calculado através da seguinte lei de formação:

```
initEndNumber = ((numberThread - (counter+1))*division) +1;
```

O número inicial é calculado com base no número de threads, e no resultado da divisão do número fatorial pela quantidade de processos usada.

Após ser feito o condicionamento de cálculo para cada thread, a parte que lidará com as threads está implementada no arquivo thrading.c. As threads são criadas de acordo com a quantidade solicitada pelo usuário. As threads escrevem na posição referente a ela no array de structs. O thread da main espera as outras threads finalizaram com a função pthread_create, conforme demonstrado abaixo.



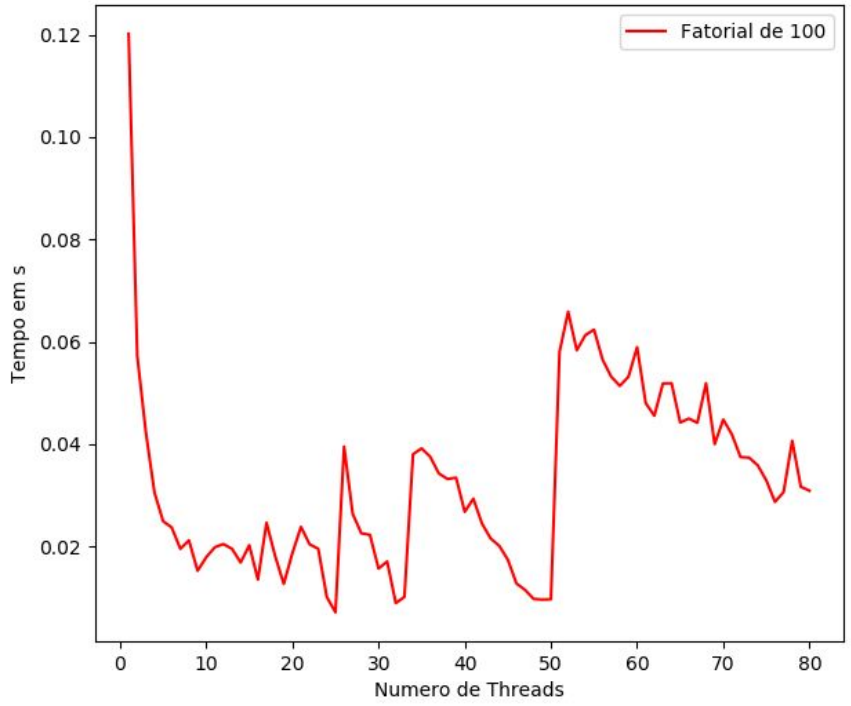
```c
void handleThreads(threadingStruct *data, int numberThreads, int factorialNumber)
{
    // If more than one process
    if (numberThreads > MIN_THREADS)
    {
        // Creates variables to handle with creating proccess
        // use in file descriptor and aux variables
        uint8_t counter = 0;
        pthread_t threadId[numberThreads];

        while (counter < numberThreads)
        {
            // If occurs some erros in creating thread
            if (pthread_create(&(threadId[counter]), NULL, performsFactorial, &data[counter]) == ERROR)
            {
                printf("Error creating thread");
                _exit(EXIT_FAILURE);
            }
            counter++;
        }

        counter = 0;
        while (counter < numberThreads)
        {
            pthread_join(threadId[counter], NULL);
            counter++;
        }
    }

    else
    {
        performsFactorial(data);
    }

}
```

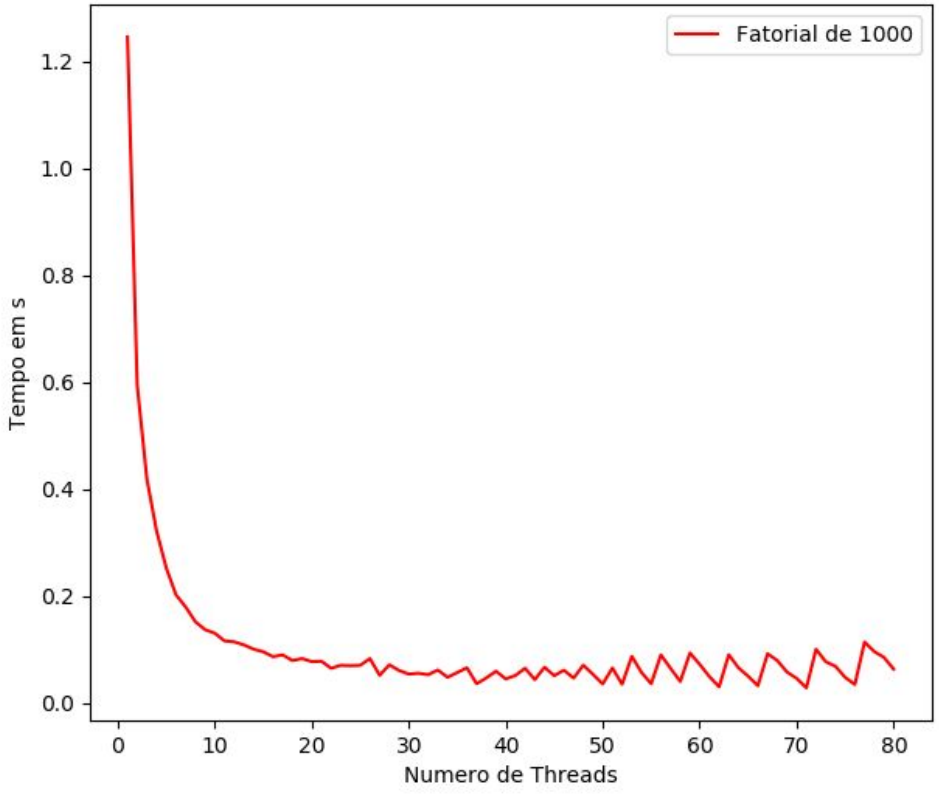
A main do código de thread pode ser vista abaixo:

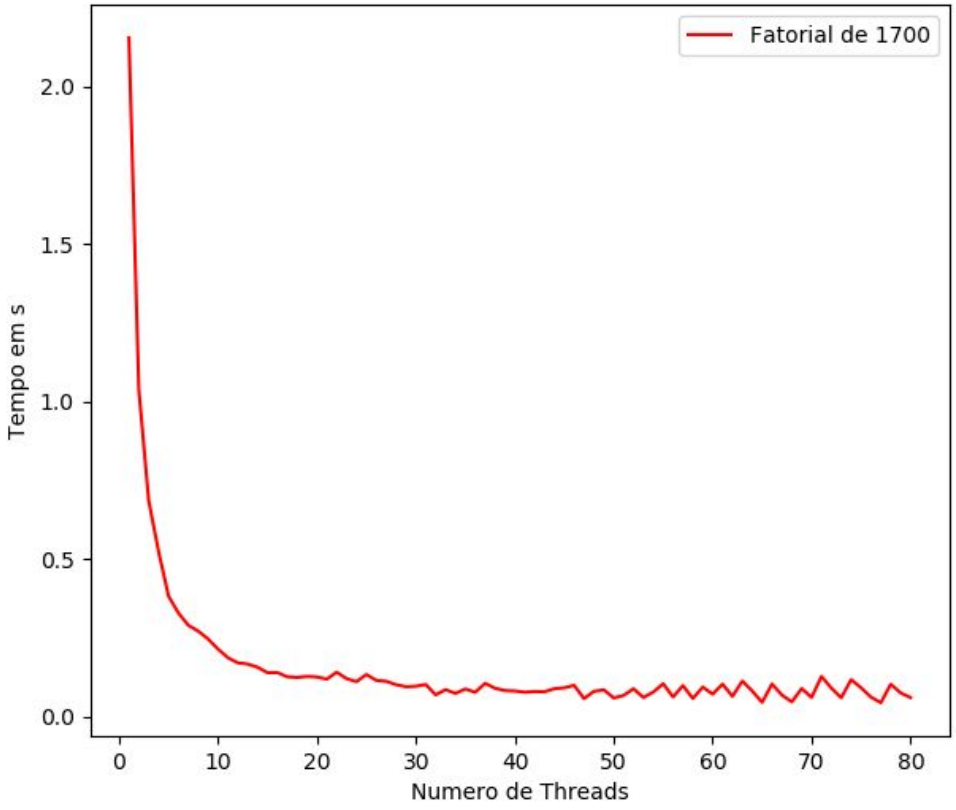
```c
int main(int argc, char *argv[])
{
    // Gets initial clock to save how many time programm needs to execute
    struct timeval begin, end;
    gettimeofday(&begin, 0);

    uint8_t threadCounter = 0;
    long double result = 1;cap

    // Checks if user entered with valid inputs
    if (validUserInput(argc, argv) == False)
    {
        exit(EXIT_FAILURE);
    }

    threadingStruct *thread = organizeStructForEveryThread(argvNumberThreads, argvFat);

    // Handle with threads
    handleThreads(thread, argvNumberThreads, argvFat);

    // Gets results from threads
    while(threadCounter < argvNumberThreads)
    {
        result = result * thread[threadCounter].result;
        threadCounter++;
    }
    free(thread);
    //printf("\nResultado do fatorial = %LF\n", result);

    gettimeofday(&end, 0);
    long seconds = end.tv_sec - begin.tv_sec;
    long microseconds = end.tv_usec - begin.tv_usec;
    double elapsedTime = seconds + microseconds*1e-6;
    saveToCsvFile(elapsedTime, argvNumberThreads, argvFat);

    return 0;
}
```

Resultados com Thread

Os gráficos abaixo demonstram o tempo de cálculo do fatorial para o número de threads.



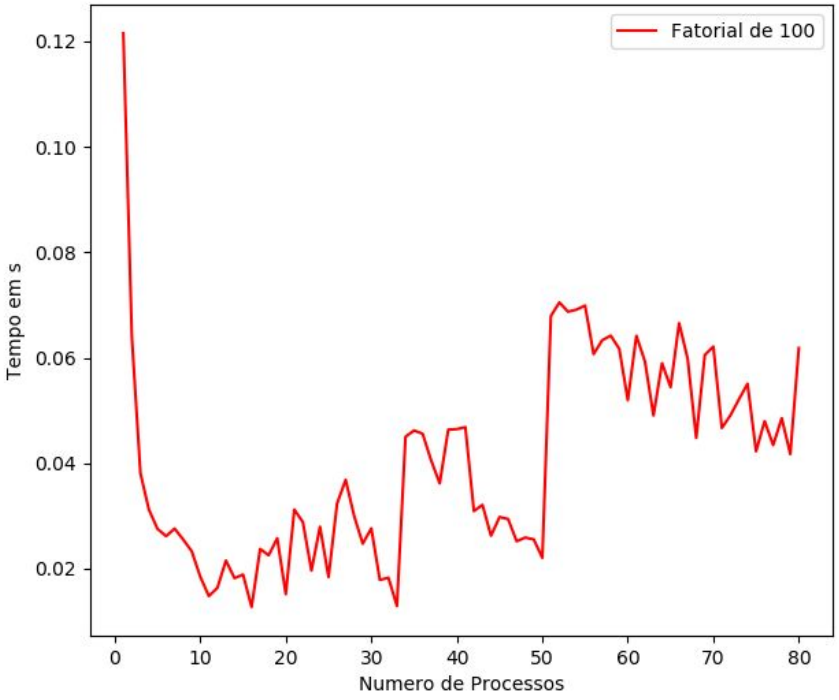


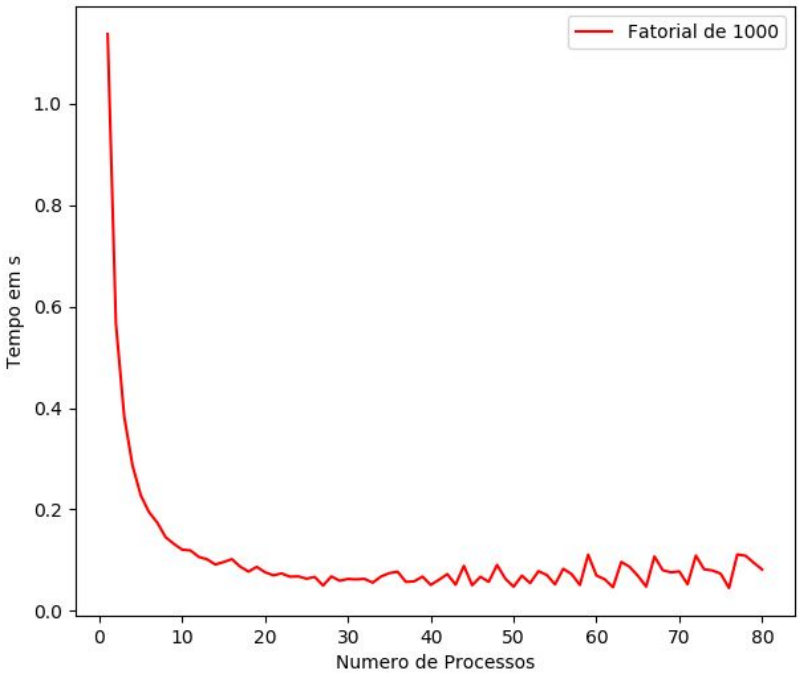


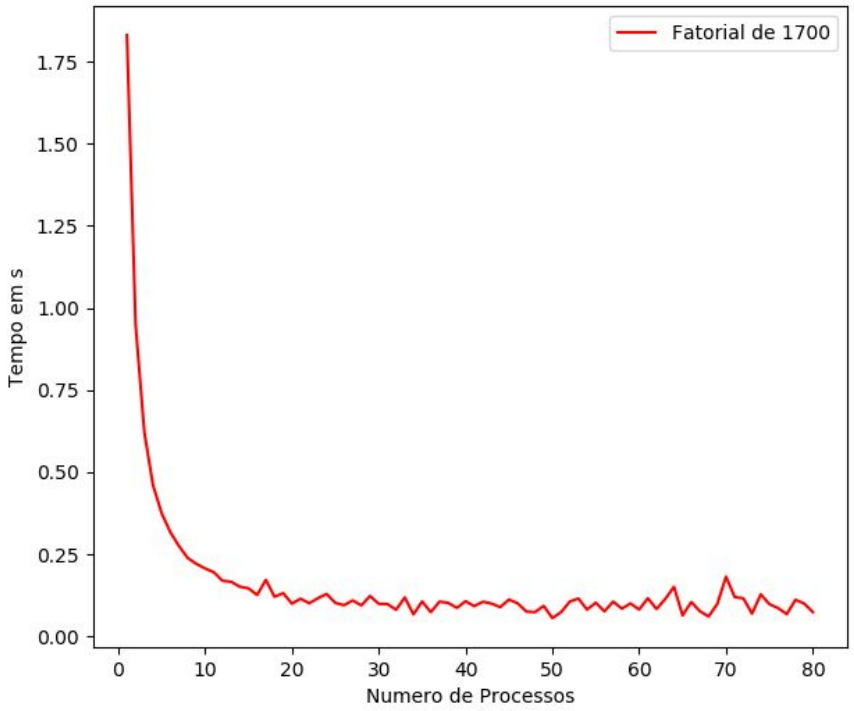
De acordo com os gráficos, quanto maior o número de threads, menos tempo o programa leva para executar a main. Isso se dá ao fato de que o número a ser calculado é dividido entre as threads e no fim a thread da main apenas multiplica os resultados das outras threads.

Resultados com Fork

Os gráficos abaixo demonstram o tempo de cálculo do fatorial para o número de processos.







De acordo com os gráficos, quanto maior o número de processos, menos tempo o programa leva para executar a main. Isso se dá ao fato de que o número a ser calculado é dividido entre os processos e no fim o processo pai apenas multiplica os resultados das outras threads.