TÓPICOS ESPECIAIS – SISTEMAS EMBARCADOS
PROGRAMAÇÃO PARALELA E TEMPO REAL

PROF. JOSENALDE OLIVEIRA

josenalde.oliveira@ufrn.br

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - UFRN

Agregando o paradigma paralelo ao *toolbox dev*

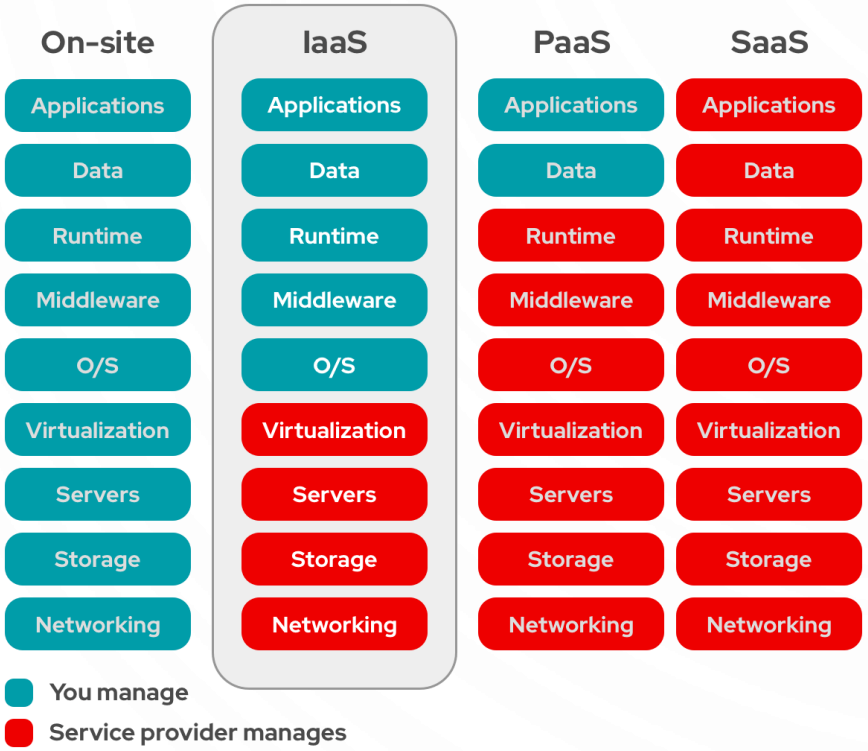
https://www.youtube.com/watch?v=8-uPCmHX3U0

Kilobots (Harvard)

FENDT Xaver

Objetivos gerais ao usar programação paralela

- Reduzir o tempo para solucionar um problema
- Resolver problemas mais complexos E de maior dimensão
  - Utilizar infraestrutura (processadores, memória) não local (IaaS/PaaS)
- Aumentar métricas típicas de HPC (*high performance computing*): FLOPS...



Exemplo: NPAD UFRN

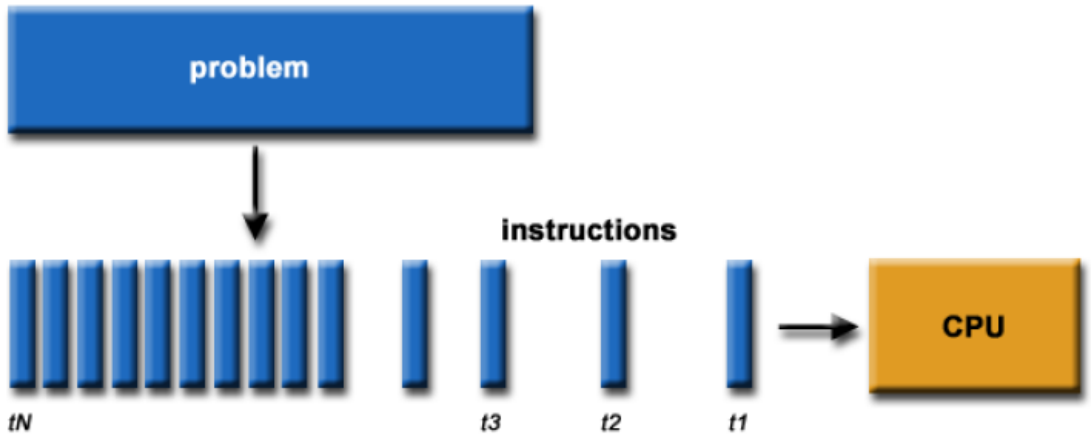
Originalmente simulações/cenários/predições:

- Físicos (astronomia)
- Climáticos (previsão do tempo etc.)
- Químicos e biológicos (genoma, reações)
- Geológicos (atividade sísmica etc.)
- Mecânica/Elétrica (resistência de materiais, placas)

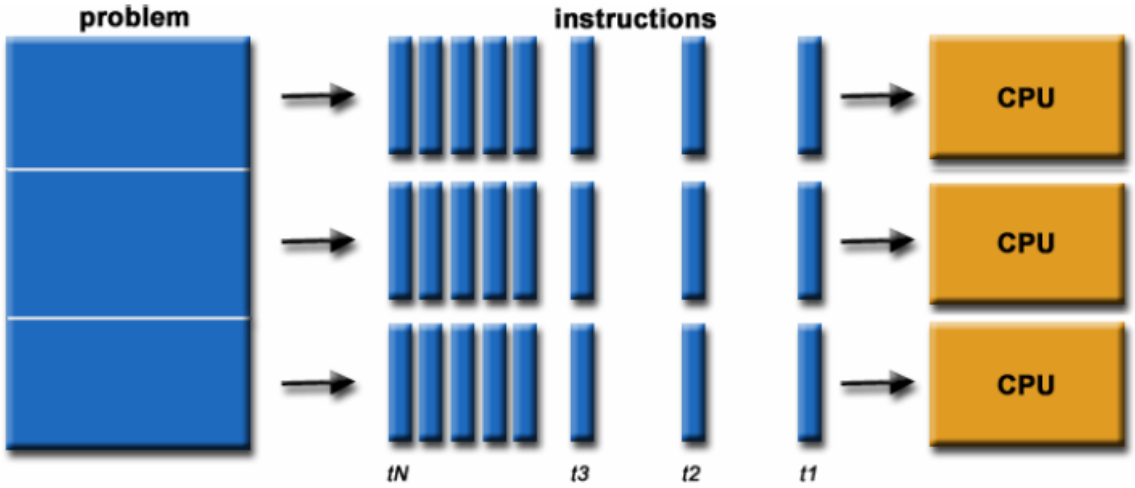
Atualmente, forte demanda por processamento rápido, variado e volumoso:

- Web (busca)
- Mineração em bases de dados (big data/spark/mapreduce)
- Realidades virtual, aumentada
- Diagnóstico e manutenção (médica, industrial etc.)
- Robótica autônoma (veículos etc.)

Programação sequencial X Programação paralela



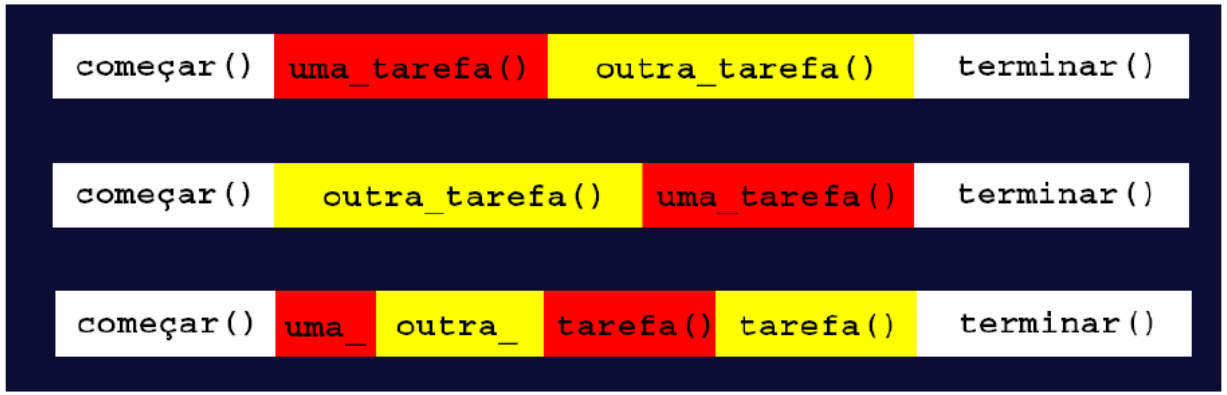
Sequencial



Paralelo: instruções sequenciais em cada nó de processamento, de forma concorrente
Introduz OVERHEAD

Concorrência X Paralelismo

Concorrência ou paralelismo potencial diz-se quando um programa possui **tarefas** (partes contíguas do programa) que podem ser executadas em qualquer ordem ou em instantes diferentes sem alterar o resultado final.



Fonte: Prof. Ricardo Rocha DCC/FCUP
Programação Paralela e Distribuída

Paralelismo: quando as tarefas de um programa são executadas em simultâneo em mais do que um processador.

Fonte: Prof. Ricardo Rocha DCC/FCUP
Programação Paralela e Distribuída

Concurrency

Concurrency happens whenever different parts of your program might execute at different times or out of order. In an embedded context, this includes:

- interrupt handlers, which run whenever the associated interrupt happens,
- various forms of multithreading, where your microprocessor regularly swaps between parts of your program,
- and in some systems, multiple-core microprocessors, where each core can be independently running a different part of your program at the same time.

Since many embedded programs need to deal with interrupts, concurrency will usually come up sooner or later, and it's also where many subtle and difficult bugs can occur. Luckily, Rust provides a number of abstractions and safety guarantees to help us write correct code.

Paralelismo implícito e explícito

Implícito: determinado pelo compilador, o qual infere regiões de paralelismo, atribui tarefas em paralelo, controla e sincroniza execução/comunicação (Exemplos: *g++ -ftree-parallelize-loops=N, #pragma omp parallel, API stream() Java, joblib Python*

- Mais fácil para o programador, porém sem garantia de eficiência a depender do problema

Explícito: cabe ao programador definir tarefas para execução em paralelo, distribuição de processadores, inserir pontos de sincronização (Exemplos: #pragma c/c++, multiprocessing python, Java threads, Workers JS, MPI etc.)

Alguns exemplos:

- João Guilherme (FC@TADS 2023.1), QTimer C++: https://github.com/JoaoGuilhermeMA/Jogo-Matematica
  https://www.youtube.com/watch?v=ewFSIxyzC4A

Alguns exemplos:

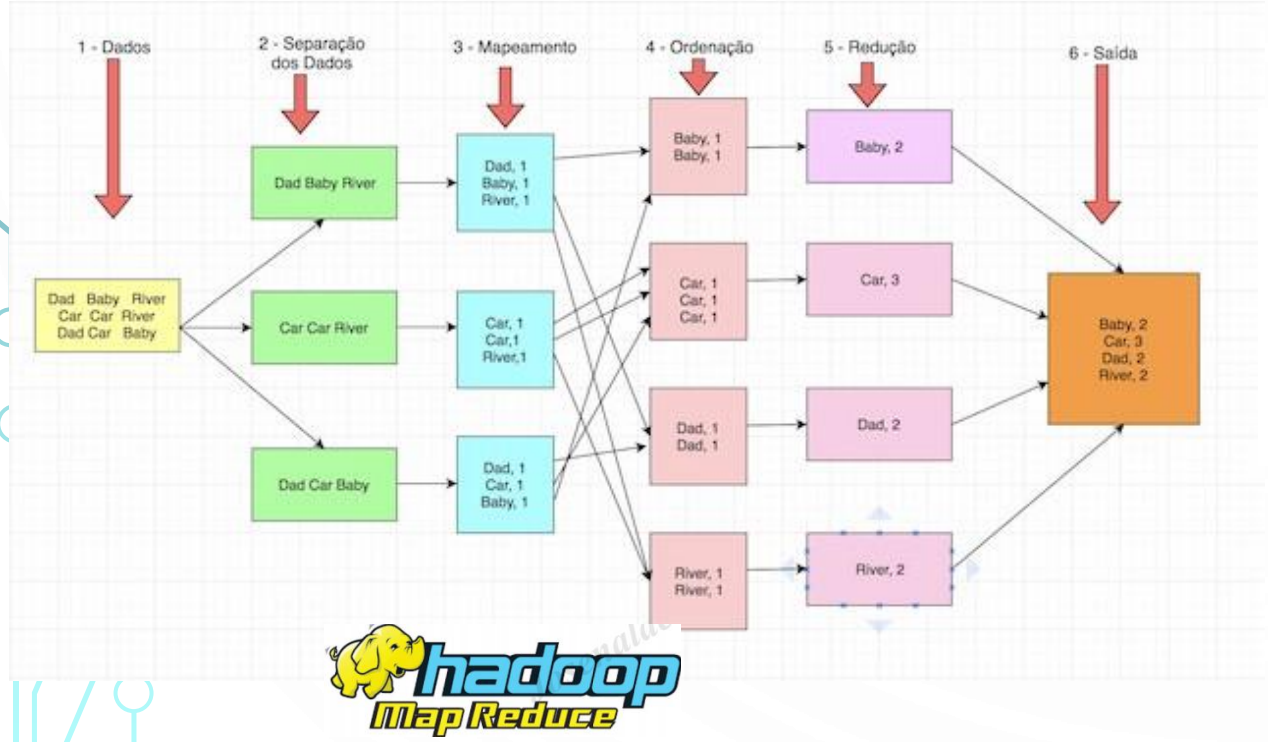
- José Vitor (FC@TADS 2023.1), Worker JS: https://github.com/NiangZd/jogoMatematico
  https://www.youtube.com/watch?v=XhadSDwSiNU

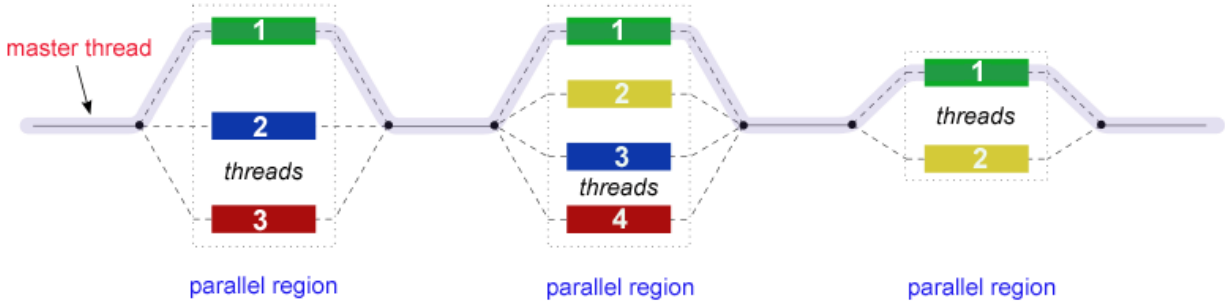
Aspectos sobre programação paralela

Ideia principal: dividir problema complexo em partes mais simples

- *computação numérica: multiplicação matricial*
- *manipulação de dados: mapreduce e spark:* ***MESMA IDEIA DO FORK - JOIN***

Dificuldades: *algoritmos, integração, codificação*



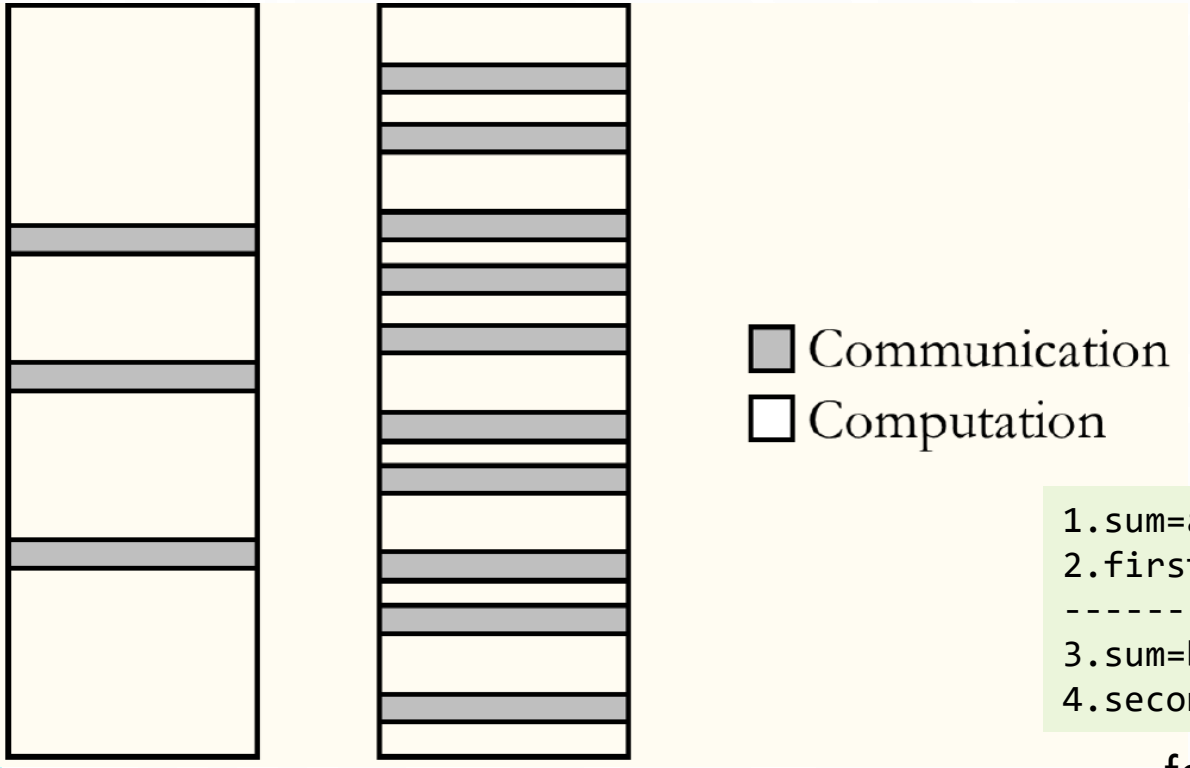


Aspectos sobre programação paralela

Escalabilidade

- altamente/fracamente: Problemas que são facilmente paralelizáveis

Granularidade: *coarse (grossa): mais computação que comunicação; fine (fina): mais comunicação*



Pouca dependência de outros processos

Muita dependência de outros processos

Bottlenecks:

- Dependência entre dados (flow dependence:fd)
  - Um processo aguardando outro
  - T1 lê posição na mem após T2 gravar

```
1.sum=a+1
2.first_term=sum*scale1
---------------------
3.sum=b+1
4.second_term=sum*scale2
```

```
1.first_sum=a+1
2.first_term=first_sum*scale1
-----------------------
3.second_sum=b+1
4.second_term=second_sum*scale2
```

fd(1,2); fd(3,4)

Se paralelizar, 2 e 3 anti-dependência (restringe concorrência)

Exemplo de altamente escalável: somatório

Seja um array x de tamanho n = {6, 4, 16, 10, 12, 14, 2, 8)

```
//o básico
sum = 0
for (i=0; i<n; i++) {
    sum += x[i];
}
```

Não há ordem definida
Associa e acumula

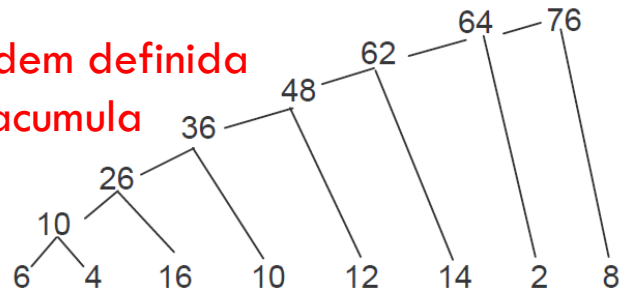


Figure 1.1. Summing in sequence. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) when adding them to an accumulation variable.

Forma 'comum' de pensar

Outra forma de 'ver' com pares de somas (par/ímpar):

$(x_0 + x_1) + (x_2 + x_3), (x_4 + x_5) + (x_6 + x_7), \ldots$

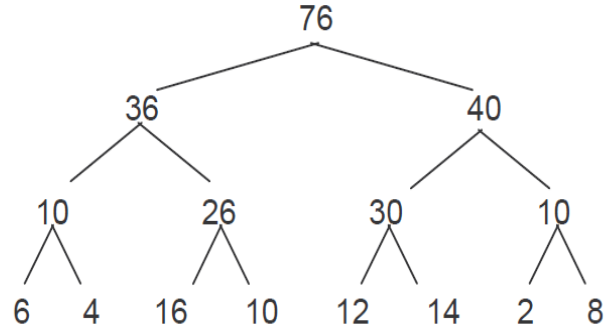


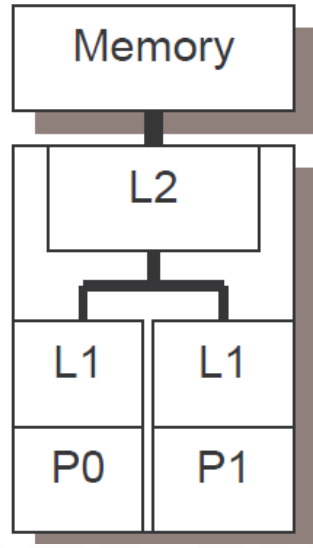
Figure 1.2. Summing in pairs. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) by (recursively) combining pairs of values, then pairs of results, etc.

1 processador: 7 somas na solução 1 e na solução 2!
P = N/2 processadores. Somas de mesmo nível em paralelo, sem dependências.

Exemplo: contar items numa lista

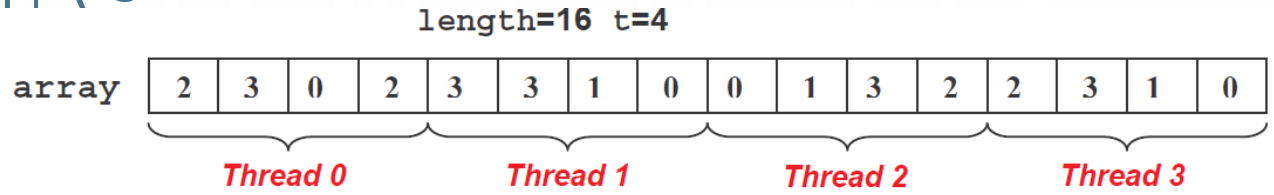
```
int *array;
int length;
int count;

int count3s() {
   int i;
   count = 0;
   for (i=0; i<length; i++) {
     if(array[i]==3) count++;
   }
   return count;
}
```

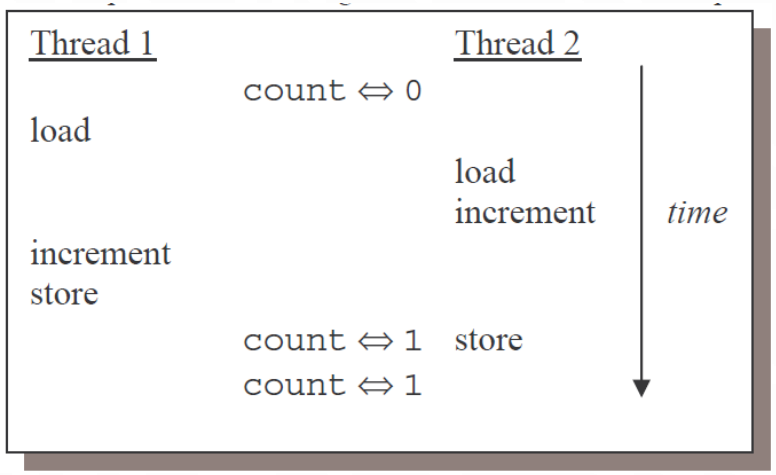
Solução 1: sequencial



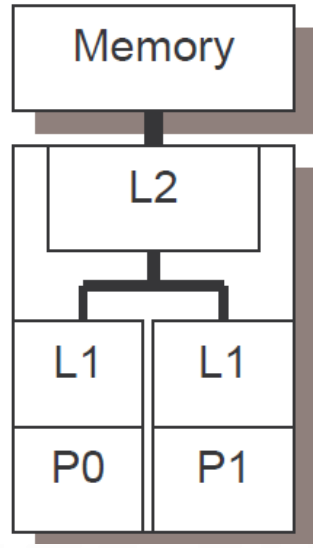
https://github.com/josenalde/parallel_programming_rtos/blob/main/src/count3s_s.cpp

Exemplo: contar items numa lista





Recurso compartilhado pode gerar 'bug'
Race condition: condição de corrida



Solução 2: multithread

```
1  int t;          /* number of threads */
2  int *array;
3  int length;
4  int count;
5
6  void count3s ()
7  {
8     int i;
9     count = 0;
10    /* Create t threads */
11    for (i=0; i<t; i++)
12    {
13       thread_create (count3s_thread, i);
14    }
15
16    return count;
17 }
18
19 void count3s_thread (int id)
20 {
21   /* Compute portion of the array that this thread should work on */
22    int length_per_thread = length/t;
23    int start = id * length_per_thread;
24
25    for (i=start; i<start+length_per_thread; i+)
26    {
27       if (array[i] == 3)
28       {
29          count++;
30       }
31    }
32 }
```

https://github.com/josenalde/parallel_programming_rtos/blob/main/src/pthread_count3s_1.cpp

Mais ‘gente’ trabalhando...

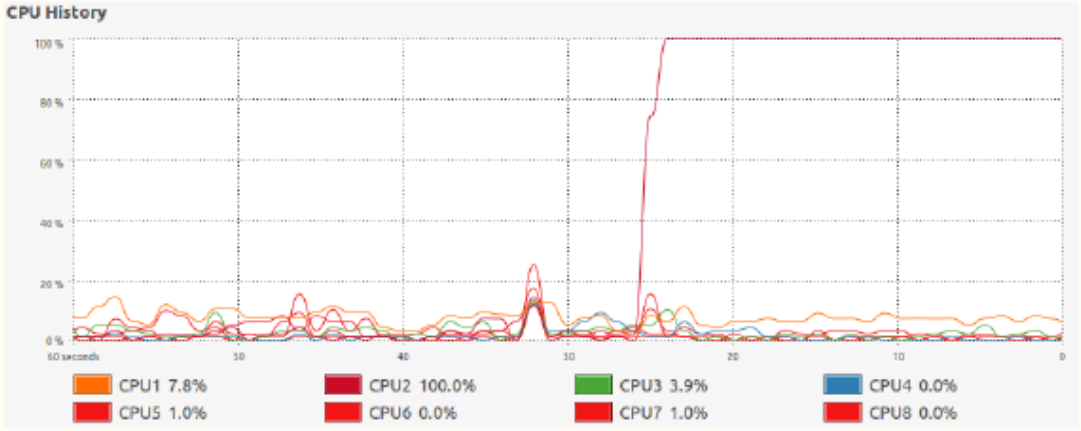


FIGURE 2: 8 Cores Used for Computation