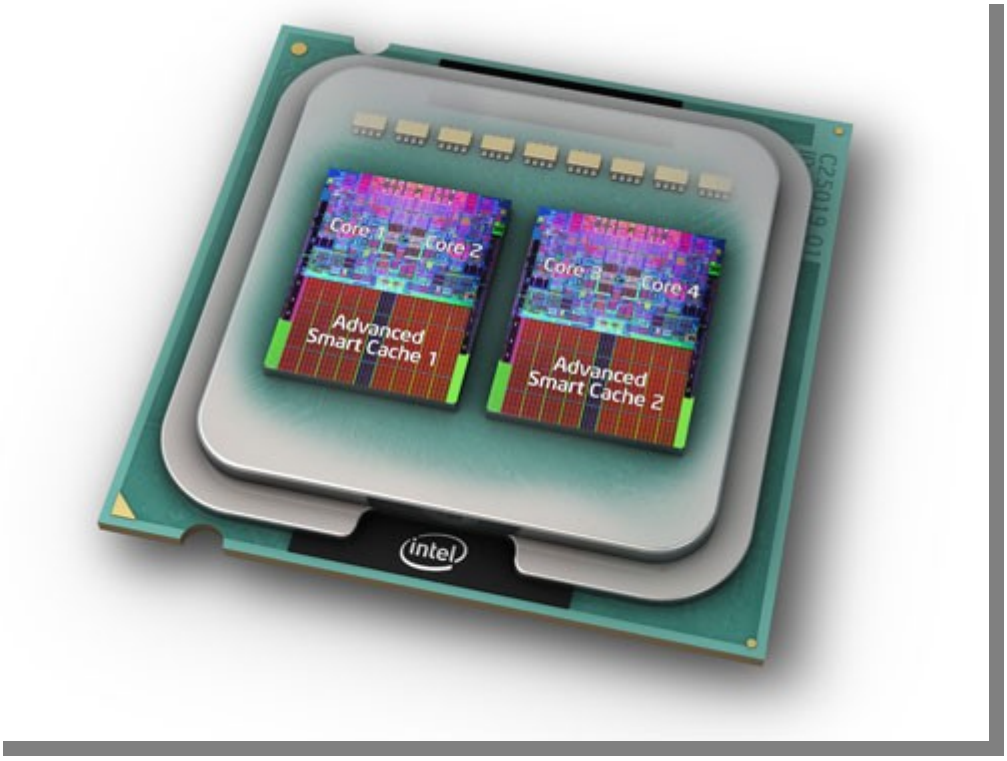
UNIVERSIDADE DE PERNAMBUCO

Multicore



Arquitetura de Computadores

Professor: Sérgio Murilo Maciel Fernandes

Discente: Iago Alves da Silva

Conceitos Necessários

- Cache
- L1
- L2
- L3
- Mapeamento de memória
- Direto: mais simples
- Associativo: sobrescrita > procurar próximo slot
- Thread
- MIMD – multiple instructions multiple data

Novos Conceitos

- L1
- L1i – mapeamento direto
- L1d – associativo
- Mutex – MUTual Exclusion
- CPU x Núcleo x Thread x Soquete
- Soquete
- Núcleo(s)
- Thread(s)
- CPU(s)
- SMT – simultaneous multithreading

Visões

Chuck Moore [...] suggested computers should be like cellphones, using a variety of specialty cores to run modular software scheduled by a high-level applications programming interface.
[...] Atsushi Hasegawa, a senior chief engineer at Renesas, generally agreed. He suggested the cellphone's use of many specialty cores working in concert is a good model for future multi-core designs.
[...] Anant Agarwal, founder and chief executive of startup Tilera, took the opposing view. He said multi-core chips need to be homogeneous collections of general-purpose cores to keep the software

model simple.

Bibliotecas

- OpenMP – C/C++ e Fortran
- Padroniza os últimos 20 anos de arquitetura SMP (shared-memory parallel)
- <thread> e <pthread> – C/C++11 (ou mais recente)

Na prática

0.sh

Como o S.O. vê os núcleos

Resumo dos núcleos

Topologia do computador (soquete, cache, cores etc)

1

Threads executando em paralelo

2

Exibindo thread executando em processadores de forma aleatória

Execução com restrição, por S.O., de núcleo (.sh)

3

Identificação de thread

4

Execução de threads com restrição, por código, de núcleo

5

Fpchurn: cálculos de ponto flutuante (seno, exponencial)

Seno

Acumulador: soma simples

Acumulador: soma usando biblioteca C/C++

UnrollAcum4

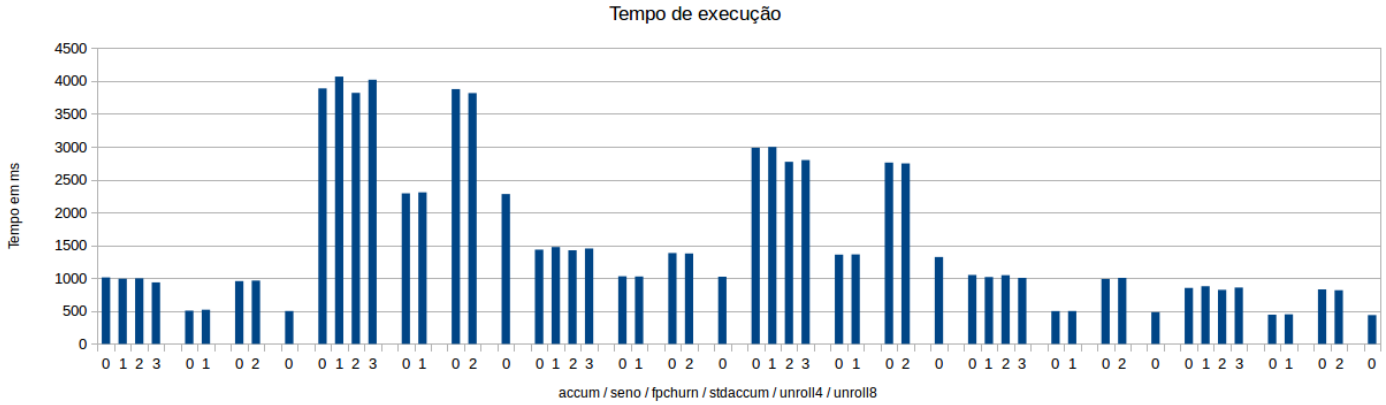
UnrollAcum8

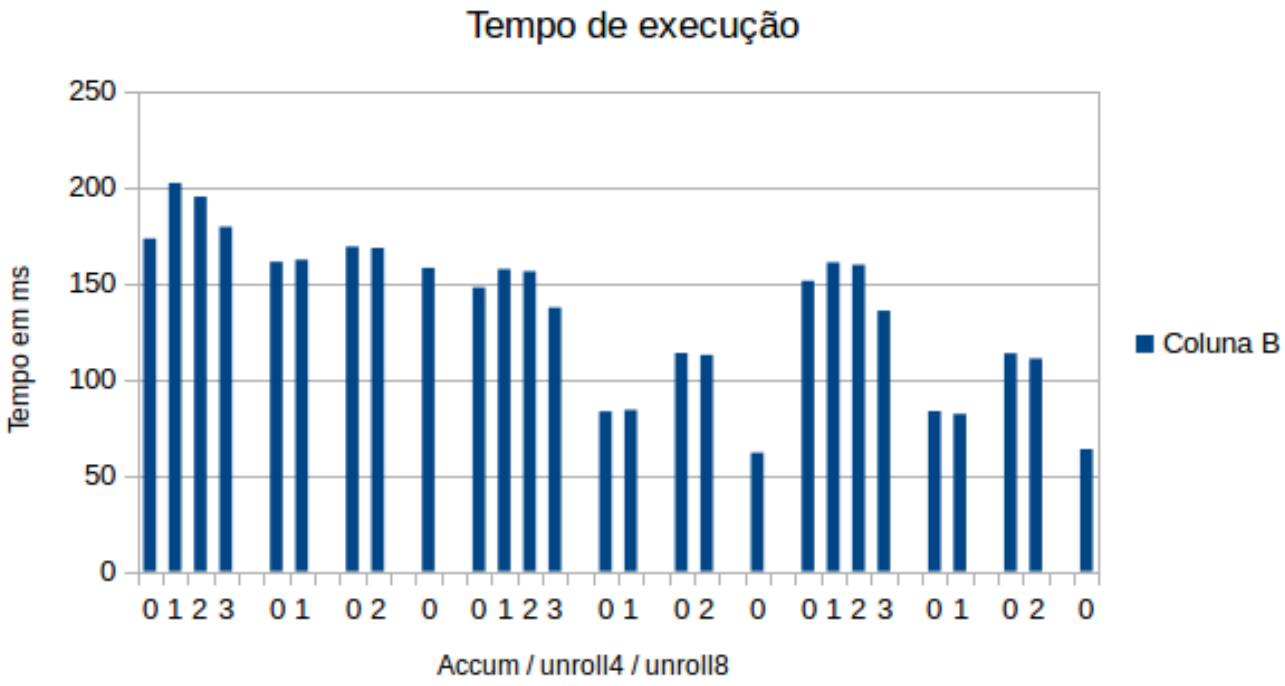
4 CPUs (concorrência em núcleo)

2 CPUs (sem concorrência em núcleo)

2 CPUs (concorrência em núcleo)

1 CPU (sem concorrência em núcleo)





Referências:

https://www.cs.cmu.edu/~fp/courses/15213-s06/lectures/27-multicore.pdf

http://www.eetimes.com/document.asp?doc_id=1167932

* http://www.ti.com/lit/an/sprab27b/sprab27b.pdf

http://www-usr.inf.ufsm.br/~reis/publicacoes/wsl2007_gomp.pdf

* http://eli.thegreenplace.net/2016/c11-threads-affinity-and-hyperthreading/

Anexos

Todo o código fonte e documentos estão disponibilizados em:
https://github.com/iagows/arquitetura_mestrado

Arquivo 0.sh

```
#!/bin/bash
clear
read -p "LISTANDO OS NÚCLEOS [Enter]"
cat /proc/cpuinfo
read -p "RESUMO DOS NÚCLEOS [Enter]"
lscpu
read -p "Topologia do computador [Enter]"
lstopo
read -p "fim"
```

Assembly.sh

```
#!/bin/bash
objdump -d main.o > dump.assembly
nano dump.assembly
read -p "[Enter]"
```

restrict_cpu_OS.sh

```
#!/bin/bash
taskset -c 1,2 ./2
```

Arquivo 1: main.cpp

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

#include <cstdlib>

int main(int argc, char** argv)
{
   //thread por CPU
```

```
    (void)argc;
    (void)argv;
    unsigned num_cpus = std::thread::hardware_concurrency();
    std::cout << "Lançando " << num_cpus << " threads\n";

    //Um mutex para acessar o std::cout de muitas threads
    std::mutex iomutex;
    std::vector<std::thread> threads(num_cpus);
    for(unsigned i=0; i<num_cpus; ++i)
    {
        threads[i] = std::thread([&iomutex, i]
        {
            {
                //travando só para o momento do uso do cout
                std::lock_guard<std::mutex> iolock(iomutex);
                std::cout << "Thread #" << i << " está rodando\n";
            }
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        });
    }

    for(auto& t:threads)
    {
        t.join();
    }

    return EXIT_SUCCESS;
}
```

Arquivo 2: main.cpp

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <mutex>
#include <sched.h>
#include <thread>
#include <vector>

#include <cstdlib>

int main(int argc, char **argv)
{
    (void)argc;
    (void)argv;
```

```
   //thread associada a um CPU

   constexpr unsigned num_threads = 4;
   //acesso exclusivo ao cout
   std::mutex iomutex;
   std::vector<std::thread> threads(num_threads);
   for(unsigned i=0; i<num_threads; ++i)
   {
      threads[i] = std::thread([&iomutex, i]
      {
         while(1)
         {
            {
               //protegendo a área do cout
               std::lock_guard<std::mutex> iolock(iomutex);
               std::cout << "Thread #" << i << ": na CPU " << sched_getcpu() << "\n";
            }

            std::this_thread::sleep_for(std::chrono::milliseconds(900));
         }
      });

   }

   for(auto& t: threads)
   {
      t.join();
   }
   return EXIT_SUCCESS;
}
```

Arquivo 3: main.cpp

```
#include <cstdlib>
#include <algorithm>
#include <chrono>
#include <iostream>
#include <mutex>
#include <pthread.h>
#include <thread>
#include <vector>

int main(int argc, char *argv[])
{
   (void)argc;
   (void)argv;
```

```
    //vendo ID das threads e os controladores nativos (OS)

    std::mutex iomutex;
    std::thread t = std::thread([&iomutex]
    {
        {
            std::lock_guard<std::mutex> iolock(iomutex);
            std::cout << "Thread: minha ID = " << std:: this_thread::get_id() << "\n"
                      << "PThread: minha ID= " << pthread_self() << "\n";
        }
    });
    {
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << "Lançou t: id = " << t.get_id() << "\n"
                  << "Controlador nativo = " << t.native_handle() << "\n";
    }

    t.join();

    std::cout << "\nO id é o mesmo. Então podemos localizar a thread"
              << "\ntanto pelo ID da thread (c++) como pelo ID da pthread (SO)." << std::endl;
    return EXIT_SUCCESS;
}
```

Arquivo 4: main.cpp

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <mutex>
#include <pthread.h>
#include <thread>
#include <vector>
#include <cstdlib>

int main(int argc, char *argv[])
{
    (void)argc;
    (void)argv;

    //Configurando a afinidade com a CPU por código
    constexpr unsigned num_threads = 4;

    //mutex para acessar o cout
    std::mutex iomutex;
    std::vector<std::thread> threads(num_threads);
```

```
    for(unsigned i =0; i<num_threads; ++i)
    {
        threads[i] = std::thread([&iomutex, i]
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(20));
            while(1)
            {
                {
                    //travando o mutex só enquanto usa o cout
                    std::lock_guard<std::mutex> iolock(iomutex);
                    std::cout <<"Thread #" << i << ": na CPU " << sched_getcpu() << "\n";
                }
                std::this_thread::sleep_for(std::chrono::milliseconds(900));
            }
        });
        //criando um objeto cpu_set_t representando os CPUs. Limpando a lista
        //e marcando só CPU i como o 'set' (conjunto)
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_SET(i, &cpuset);
        int rc = pthread_setaffinity_np(threads[i].native_handle(),
                                        sizeof(cpu_set_t), &cpuset);
        if(rc != 0)
        {
            std::cerr << "Erro ao criar a pthread_setaffinity_np: " << rc << "\n";
        }
    }

    for(auto& t: threads)
    {
        t.join();
    }
    std::cout << "\nA thread fica sempre ligada ao mesmo CPU especificado";
    return EXIT_SUCCESS;
}
```

Arquivo 5: main.cpp

```
#include <cstdlib>

#include <algorithm>
#include <chrono>
#include <cmath>
#include <iostream>
#include <mutex>
#include <pthread.h>
```

```
#include <random>
#include <thread>
#include <vector>

//tipo "função de carga de trabalho". Recebe um vetor e uma referência para um tipo FLOAT que será o resultado
using WorkLoadFunc = std::function<void(const std::vector<float>&, float&)>;

//reduzindo as chamadas do cronômetro
using hires_clock = std::chrono::high_resolution_clock;
using duration_ms = std::chrono::duration<double, std::milli>;

std::mutex iomutex;

void workload_fpchurn(const std::vector<float>& data, float& result)
{
   constexpr size_t NUM_ITERS = 10*1000*1000;
   auto t1 = hires_clock::now();
   float rt = 0;
   for(size_t i = 0; i < NUM_ITERS; ++i)
   {
      float item = data[i];
      float l = std::log(item);
      if(l>rt)
      {
         l = std::sin(l);
      }
      else
      {
         l = std::cos(l);
      }

      if(l>rt - 2.0)
      {
         l = std::exp(l);
      }
      else
      {
         l = std::exp(l+1.0);
      }

      rt++;
   }
   result = rt;
   {
      auto t2 = hires_clock::now();

      std::lock_guard<std::mutex> iolock(iomutex);
```

```
        std::cout << __func__ << " [CPU " << sched_getcpu() << "]:\n";
        std::cout << " elapsed: " << duration_ms(t2-t1).count() << " ms\n";
    }
}

void workload_sin(const std::vector<float>& data, float& result)
{
    auto t1 = hires_clock::now();
    float rt = 0;
    for(size_t i=0, total = data.size(); i<total; ++i)
    {
        rt+= std::sin(data[i]);
    }
    result = rt;

    {
        auto t2 = hires_clock::now();
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << __func__ << " [cpu " << sched_getcpu() << "]:\n";
        std::cout << " itens processados: " << data.size() << "\n";
        std::cout << " tempo passado: " << duration_ms(t2-t1).count() << " ms\n";
        std::cout << " resultado: " << result << "\n";
    }
}

void workload_accum(const std::vector<float>& data, float& result)
{
    auto t1 = hires_clock::now();
    float rt = 0;
    for(size_t i=0, total = data.size(); i< total; ++i)
    {
        /*
         * Num x86-64 isso pode gerar um loop de ADDs de data.size comprimento,
         * tudo somando no mesmo registrador xmm. Se compilado com -0fast
         * (-ffast-math), o compilador vai tentar realizar otimizações FP inseguras e
         * vai vetorizar o loop em um de data.size/4 ADDs.
         * Isso pode mudar a ordem em que os floats são adicionados, o que é inseguro
         * visto que adição FP é não associativa
        */
        rt+= data[i];
    }
    result = rt;

    {
        auto t2 = hires_clock::now();
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << __func__ << " [cpu " << sched_getcpu() << "]\n";
        std::cout << " itens processados: " << data.size() << "\n";
```

```
        std::cout << " tempo passado: " << duration_ms(t2-t1).count() << " ms\n";
        std::cout << " resultado: " << result << "\n";
    }
}

void workload_unrollaccum4(const std::vector<float>& data, float& result)
{
    auto t1 = hires_clock::now();
    if(data.size() % 4 != 0)
    {
        std::cerr << "ERRO em " << __func__ << ": data.size " << data.size() << "\n";
    }

    float rt0 = 0;
    float rt1 = 0;
    float rt2 = 0;
    float rt3 = 0;
    for(size_t i = 0; i<data.size(); i+=4)
    {
        /*
         * Esse UNROLL é faz uma quebra manual das dependências de um
         * único registrador xmm (o da função anterior). Deve ser mais rápido
         * porque ADDs distintos farão suas somas em registradores separados.
         * Mas também é inseguro pelo mesmo problema da função anterior
         */
        rt0 += data[i];
        rt1 += data[i+1];
        rt2 += data[i+2];
        rt3 += data[i+3];
    }
    result = rt0+rt1+rt2+rt3;

    {
        auto t2 = hires_clock::now();
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << __func__ << " [cpu " << sched_getcpu() << "]:\n";
        std::cout << " itens processados: " << data.size() << "\n";
        std::cout << " tempo passado: " << duration_ms(t2-t1).count() << " ms\n";
        std::cout << " resultado: " << result << "\n";
    }
}

void workload_unrollaccum8(const std::vector<float>& data, float& result) {
    auto t1 = hires_clock::now();
    if (data.size() % 8 != 0) {
        std::cerr
                << "ERROR in " << __func__ << ": data.size " << data.size() << "\n";
    }
```

```
    float rt0 = 0;
    float rt1 = 0;
    float rt2 = 0;
    float rt3 = 0;
    float rt4 = 0;
    float rt5 = 0;
    float rt6 = 0;
    float rt7 = 0;
    for (size_t i = 0; i < data.size(); i += 8) {
        rt0 += data[i];
        rt1 += data[i + 1];
        rt2 += data[i + 2];
        rt3 += data[i + 3];
        rt4 += data[i + 4];
        rt5 += data[i + 5];
        rt6 += data[i + 6];
        rt7 += data[i + 7];
    }
    result = rt0 + rt1 + rt2 + rt3 + rt4 + rt5 + rt6 + rt7;

    {
        auto t2 = hires_clock::now();
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << __func__ << " [cpu " << sched_getcpu() << "]:\n";
        std::cout << "  itens processados: " << data.size() << "\n";
        std::cout << "  tempo passado: " << duration_ms(t2 - t1).count() << " ms\n";
        std::cout << "  resultado: " << result << "\n";
    }
}

void workload_stdaccum(const std::vector<float>& data, float& result)
{
    auto t1 = hires_clock::now();
    result = std::accumulate(data.begin(), data.end(), 0.0f);

    {
        auto t2 = hires_clock::now();
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << __func__ << " [cpu " << sched_getcpu() << "\n";
        std::cout << " itens processados: " << data.size() << "\n";
        std::cout << " tempo passado: " << duration_ms(t2-t1).count() << " ms\n";
        std::cout << "resultado: " << result << "\n";
    }
}

/**
 * @brief make_input_array: Cria um vetor preenchido com N floats
 * distribuídos uniformemente (-1.0, 1.0)
```

```
 * @param N
 * @return
 */
std::vector<float> make_input_array(int N)
{
   std::random_device rd;
   std::mt19937 gen(rd());
   std::uniform_real_distribution<float> dis(-1.0f, 1.0f);

   std::vector<float> vf(N);
   for(size_t i=0, total = vf.size(); i < total; ++i)
   {
      vf[i] = dis(gen);
   }
   return vf;
}

/**
 * @brief do_not_optimize: Essa função pode ser usada para marcar memória
 * que não deve ser otimizada, ainda assim não vai gerar código
 * @param p
 */
void do_not_optimize(void *p)
{
   asm volatile("" : : "g"(p):"memory");
}

void pin_thread_to_cpu(std::thread& t, int cpu_num)
{
   cpu_set_t cpuset;
   CPU_ZERO(&cpuset);
   CPU_SET(cpu_num, &cpuset);

   int rc = pthread_setaffinity_np(t.native_handle(), sizeof(cpu_set_t), &cpuset);
   if(rc !=0)
   {
      std::cerr << "Erro ao chamar pthread_setaffinity_np: " << rc << "\n";
   }
}

int main(int argc, char *argv[])
{
   //separando números grandes com vírgula
   std::cout.imbue(std::locale(""));

   //chamando da linha de comando:
   //argv[0] nome do programa
   //argv[1] nome da workload e cpu no argv seguinte (argv[2])
```

```
    //argv[3] nome da workload e cpu no argv seguinte (argv[4])
    //etc

    int num_workloads = argc /2;

    std::vector<float> results(num_workloads);
    do_not_optimize(results.data());

    std::vector<std::thread> threads(num_workloads);

    constexpr size_t INPUT_SIZE = 100*1000*1000;
    auto t1 = hires_clock::now();

    /*
     * Alocando e inicializando um array de entrada extra - não usado pelos workloads.
     * Isso é para garantir que nenhum dos dados de trabalho (working) fique na cache L3
     * (que é bem grandinha), o que poderia dar uma vantagem injusta  para um dos workloads.
     * As camadas mais baixas de cache são muito pequenas e o tempo da vantagem de pré-
carregamento
     * pode ser descartado
     */
    std::vector<std::vector<float>> inputs(num_workloads+1);
    for(int i=0; i<num_workloads; ++i)
    {
       inputs[i] = make_input_array(INPUT_SIZE);
    }

    std::cout << "Criados " << num_workloads +1 << " arrays de entrada"
            << "; em: " <<duration_ms(hires_clock::now()-t1).count() << " ms\n";
    for(int i=1; i<argc; i+=2)
    {
       WorkLoadFunc func;
       std::string workload_name = argv[i];
       if(workload_name == "fpchurn")
       {
          func = workload_fpchurn;
       }
       else if(workload_name == "sin")
       {
          func = workload_sin;
       }
       else if(workload_name == "accum")
       {
          func = workload_accum;
       }
       else if(workload_name == "unrollaccum4")
       {
          func = workload_unrollaccum4;
```

```
        }
        else if(workload_name == "unrollaccum8")
        {
            func = workload_unrollaccum8;
        }
        else if(workload_name == "stdaccum")
        {
            func = workload_stdaccum;
        }
        else
        {
            std::cerr << "Workload desconhecida: " << argv[i] << "\n";
            return EXIT_FAILURE;
        }

        int cpu_num;
        if(i+1 >= argc)
        {
            cpu_num = 0;
        }
        else
        {
            cpu_num = std::atoi(argv[i+1]);
        }

        {
            std::lock_guard<std::mutex> iolock(iomutex);
            std::cout << "Chamando workload '" << workload_name << "' na CPU "
                    << cpu_num << "\n";
        }

        int nworkload = i/2;
        threads[nworkload] = std::thread(func, std::cref(inputs[nworkload]),
                                         std::ref(results[nworkload]));
        pin_thread_to_cpu(threads[nworkload], cpu_num);
    }

    /*
     * Todas as threads foram lançadas ao mesmo tempo, agora é só esperar acabar.
     */
    for(auto& t:threads)
    {
        t.join();
    }

    return EXIT_SUCCESS;
}
```