# Aplicação servidor-cliente para paralelismo de processamento usando máquina de estados

Luis Paulo Durão
Engenharia Electrotécnica e de Computadores
Faculdade de Engenharia da
Universidade do Porto
Portugal, Porto
Email: up201306126@fe.up.pt

Orlanda Alice Sousa
Engenharia Electrotécnica e de Computadores
Faculdade de Engenharia da
Universidade do Porto
Portugal, Porto
Email: up201304155@fe.up.pt

***Resumo*—Este projecto foi desenvolvido no âmbito da disciplina de Sistemas Distribuídos. O problema proposto foi a criação de uma aplicação cliente-servidor, em que o cliente fizesse pedidos ao servidor de algum processamento ou algoritmo mais complexos e que esse os atendesse usando paralelismo controlado por uma máquina de estados para uma utilização óptima dos seus recursos.**

## I. Palavras-chave

As palavras-chave deste projecto são:

- C++
- Cliente-servidor
- Multi-thread
- Paradigma de programação orientada a objectos
- Parallelism
- POSIX
- Sistemas Distribuídos
- TCP/IP

## II. Definição do problema

O projeto apresentado tem como objetivo desenvolver um servidor com operações de IO e de cálculo complexas, realizando implementações concorrentes baseadas em *multi-thread* e *state machine*. Na arquitetura criada, o cliente faz um pedido ao servidor e fica à espera da sua resposta. Por sua vez, o servidor pode responder a vários clientes ao mesmo tempo. Os canais estabelecidos foram baseados no protocolo TCP visto que estes possuem conexão e pretendemos garantir a ordem das mensagens. O servidor deve dar uma resposta rápida e eficiente aos clientes. No final do projeto deverá ser possível tirar conclusões acerca de quais as situações em que é vantajoso utilizar este tipo de aplicações em detrimento de outras.

## III. Descrição da solução

Para a criação da aplicação do lado do servidor, foi escolhida uma arquitectura modular organizada em classes. Para que tal fosse possível, foi usada a linguagem C++ (revisão C++17) e feito um esforço para a adaptação de todas as bibliotecas C para classes C++ através da criação de classes *wrap*. O diagrama UML do sistema encontra-se na figura 1 mas será também incluído como anexo para uma leitura mais fácil. De seguida irão ser explicados os grupos de classes mais importantes da implementação.

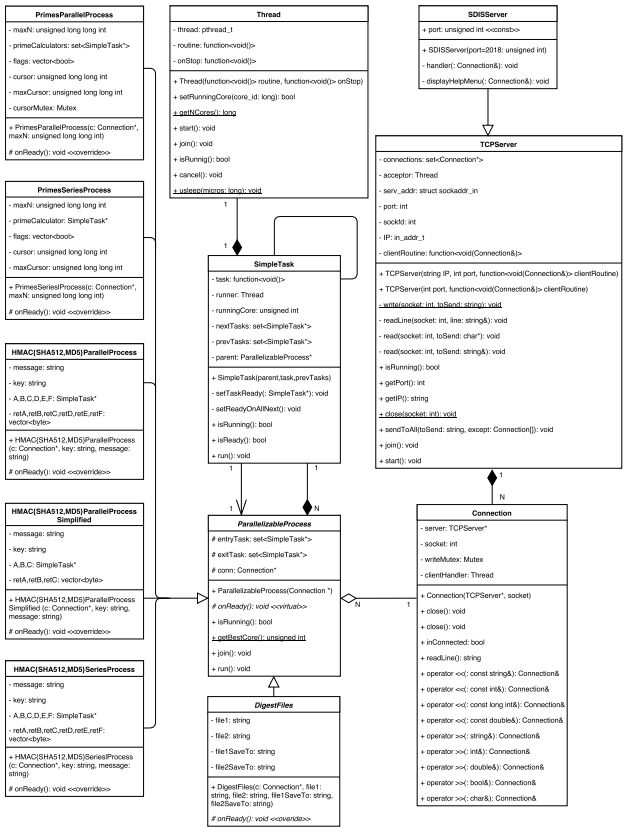



Figura 1. Diagrama UML da aplicação

### A. Estruturação dos Processos

Os diferentes processos são todos subclasses da mesma superclasse (*ParallelizableProcess*) abstracta que é usada para

que todos os processos sejam chamados pela sua superclasse (evitando a necessidade de adaptar o código sempre que uma nova classe é criada) e para agilizar a criação de novos processos.

A criação de novos processos foi desenhada de forma a ser indutiva e fácil. Para criar um novo processo, apenas se deve criar uma nova classe, herdar publicamente a classe *ParallelizableProcess*, implementar o método *onReady* e criar (no construtor) instâncias da classe *SimpleTask*, encadeando-as de modo a gerar um grafo direccionado (ou máquina de estados) que será automaticamente seguido de forma a executar todas as tarefas até ao fim. É importante que o grafo não tenha ciclos. O código criado não verifica se existem ciclos. É da inteira responsabilidade do programador de novos processos que o grafo criado seja válido.

A superclasse *ParallelizableProcess* é responsável por guardar a quantidade de *SimpleTask*'s que estão a correr em cada núcleo disponível do processador. Aquando da primeira chamada do método *run* da classe *SimpleTask*, para a escolha do núcleo onde a tarefa irá correr, é chamado o método estático *getBestCore* da classe *ParallelizableProcess* que retorna o núcleo com menos tarefas atribuídas.

### B. *Classes* Thread, Mutex *e* ThreadCondition

A tecnologia de *threading* usada foi a *POSIX*, *pthread*. Foi criada uma classe *wrap* da *pthread*, *Thread*, uma classe *wrap* da *pthread_mutex*, *Mutex*, e uma class *warp* da *pthread_cond*, *ThreadCondition*. Assim foi possível criar facilmente *threads*, *mutexes* e condições ao longo de todo o código.

A classe *Thread* criada recebe como argumentos dois objectos do tipo *std::function<void()>* (> C++11) no seu único construtor. O primeiro é obrigatório e representa a rotina da *thread*. O segundo é opcional e é garantidamente executado no final da execução da *thread*, quer esta tenha sido cancelada ou tenha terminado porque a rotina terminou. Os construtores de cópia de todas estas classes foram eliminados para que nunca existisse mais do que um *handler* do mesmo *pthread*, *pthread_mutex* ou *pthread_cond*. Se o programador tentar passar um objecto de qualquer uma destas classes a uma função por valor (em vez de por apontador ou por referência), o compilador retornará um erro pois nenhuma delas pode ser copiada. É importante também notar que foram tomadas precauções de forma a que os objectos destas classes pudessem ser usados em funções constantes e que as suas referências constantes possam ser usadas, dando a sinalização de constante (*const*) a todos os métodos que não alterem o estado do objecto.

A classe *Thread*, além de permitir cancelar a execução de uma *thread*, verificar se está a correr e fazer *join*, também permite escolher o núcleo onde essa mesma *thread* deverá correr (ou seja, definir a afinidade da *thread*). Esta classe tem também um método estático *usleep* que implementa uma forma segura de colocar a *thread* que o chama em pausa durante um determinado número de microsegundos, quer esta seja ou não uma *pthread* (pode ser chamada na main).

A classe *Mutex* permite bloquear (*lock*) e desbloquear (*unlock*) o *mutex* e saber se o mesmo está bloqueado.

A classe *ThreadCondition* permite colocar uma *thread* em espera de uma sinal, tanto durante um período de tempo ou ilimitadamente, fazer *signal* e *broadcast* (ambos os métodos são análogos aos métodos nativos da biblioteca *POSIX* com os mesmos nomes).

### C. *Interface de rede*

A comunicação com os clientes, como já antes mencionado, foi feita por sockets TCP/IP. A biblioteca usada foi a nativa do *Linux*, escrita em C. Por essa razão, foram criadas classes *wrap* para a utilização das mesmas.

A classe *TCPServer* contém um objecto da classe *Thread* que é responsável pela aceitação de novas ligações. Quando uma nova ligação é aceite, uma nova instância (objecto) da classe *Connection* é criada e guardada num *std::set<Connection*>* de conecções activas. Esta classe tem diversos métodos para o controlo do servidor mas, por motivos de falta de espaço, o leitor terá de remeter ao diagrama UML para saber quais são (é de notar que todos os nomes dos métodos são significativos do que fazem e que, por isso, conseguem facilmente explicar o que cada método faz). Esta classe recebe uma string (opcional) que deve conter o endereço onde o servidor irá ouvir, um inteiro (obrigatório) com a porta na qual vai esperar ligações e, mais importante, uma *std::function<void(Connection&)>* que é a rotina que contém o programa a ser corrido pelo cliente (como um menu para selecção de opções ou para fazer login).

A classe *SDISServer* é uma subclasse de *TCPServer* e foi criada para definir a rotina de cliente específica para este projecto. Define um método *handler* que contém o menu com as opções que o cliente tem para escolha do processo a correr e um menu de ajuda.

A classe *Connection* representa uma ligação aceite por um servidor da classe TCPServer. Implementa métodos de *IO* para a socket aberta usando os operadores « para a escrita de strings, inteiros, inteiros longos e valores de dupla precisão e » para a leitura de strings, inteiros, valores de dupla precisão, booleanos e caracteres.

## IV. Processos criados como exemplo

Após a criação da aplicação foram criados alguns exemplos de processos de forma a testar o sistema. Os mesmos encontram-se de seguida.

### A. *Cálculo de HMAC*

Um dos processos de exemplo de um algoritmo puramente de processamento foi o cálculo do HMAC *(Hash-based Message Authentication Code)*. Este algoritmo foi escolhido pois calcula concatenações de hashes sendo assim possível paralelizar algumas partes do mesmo.

Porque $HMAC = Hash((K \oplus opad)||Hash((K \oplus ipad)||m))$, é possível calcular cada um dos lados da primeira concatenação em paralelo. A paralelização criada está representada na figura 2.

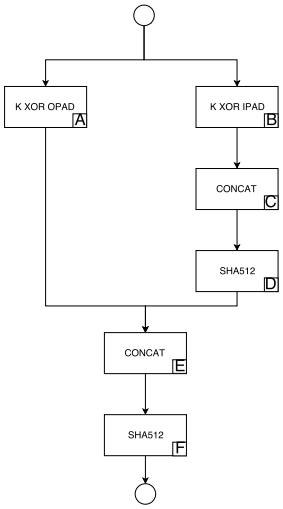



Figura 2. Diagrama de paralelização do HMAC

Foram realizadas várias versões deste teste, uma versão sem máquina de estados, uma versão com máquina de estados mas com processamento em série e por fim uma versão com maquina de estados e processamento em paralelo, com o objetivo de comparar os diferentes tempos de execução de cada caso. Foi também criada uma versão simplificada do processo representado na figura 2 em que foram unidos todos os passos em série num só. Por motivos de falta de espaço, os esquemas de todos os outros métodos usados para o cálculo do HMAC estarão incluídos em anexo.

### B. *Cálculo de números primos*

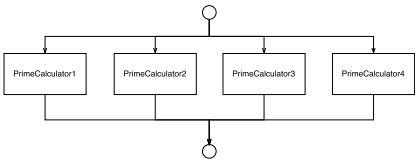



Figura 3. Diagrama de paralelização do cálculo de primos

Para o segundo conjunto de testes, foi implementado um algoritmo para o cálculo dos números primos menores que um dado número N , introduzido pelo utilizador, e seguindo a mesma lógica de testes realizada anteriormente, ou seja, sem máquina de estados e com máquina de estados com processamento em série e em paralelo para mais uma vez ser possível comparar os diferentes tempos de resposta. Este é outro exemplo de um processo puramente de processamento. O algoritmo usado foi o de Crivo de Eratóstenes. O diagrama de processamento pode ser encontrado na figura 3.

### C. *Algoritmo de criação de entropia em ficheiros*

Este exemplo usa IO e processamento. Lê 2 ficheiros em paralelo, escreve para ficheiros cada caracter somado ao seu caracter seguinte e faz o módulo para que a soma seja um caracter ASCII que possa ser imprimido (*fileOut[i]=(fileIn[i]+fileIn[i+1])%(126-32+1)+32*). O

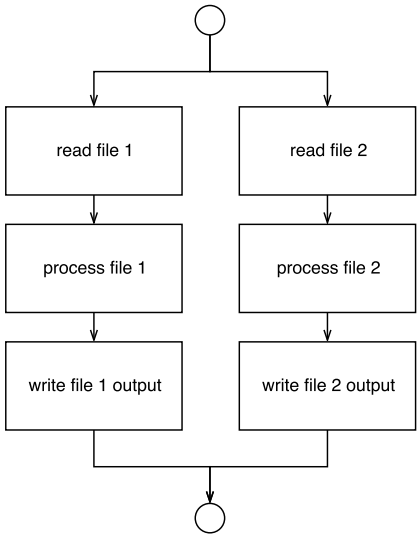



Figura 4. Diagrama de paralelização do exemplo com ficheiros

esquema completo está descrito na figura 4 mas porque já tinha sido testada uma situação do diagrama da figura 2 (e já tendo em conta os resultados do mesmo que só iremos discutir em seguida), todas as tarefas que estão em série foram concatenadas numa única tarefa, como descrito na figura 5

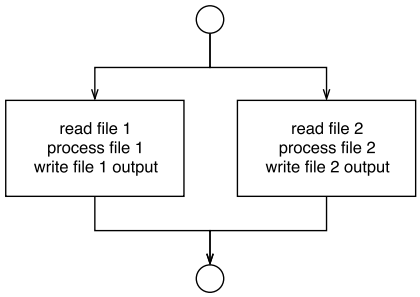



Figura 5. Diagrama de paralelização do exemplo com ficheiros simplificado

## V. Resultados

Neste capitulo vamos apresentar os resultados obtidos para os testes realizados. Cada medição de tempo apresentada de seguida é a média de 10 medições.

### A. *HMAC-SHA512*

Neste teste pretendeu-se comparar a performance do mesmo algoritmo implementado com diferentes níveis de paralelização. Foi comparada a performance do algoritmo sem usar máquina de estados, com máquina de estados em série, em paralelo com partição exagerada de tarefas e em paralelo sem tarefas em série (simplificado). O gráfico da figura 6 representa os resultados das medições feitas para cada implementação.

### B. *Cálculo dos números primos menores que um dado valor N*

Também neste teste se pretendeu comparar a performance do mesmo algoritmo para diferentes níveis de paralelização.

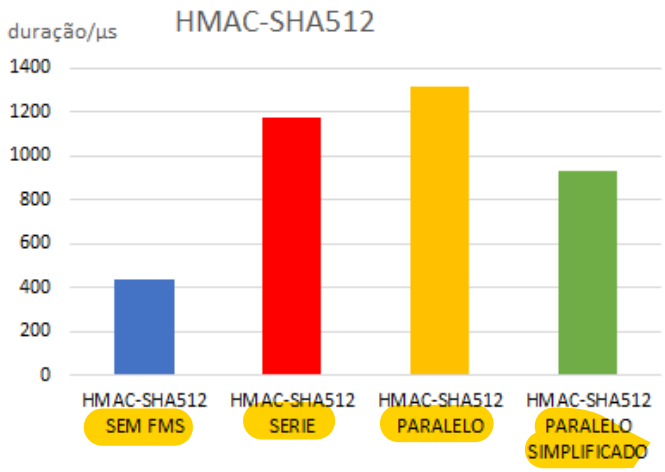



Figura 6. Diagrama de colunas do teste HMAC-SHA512

Foi comparada a performance do algoritmo sem usar máquina de estados, com máquina de estados em série e em paralelo. O gráfico da figura 7 representa os resultados das medições realizadas para cada implementação.

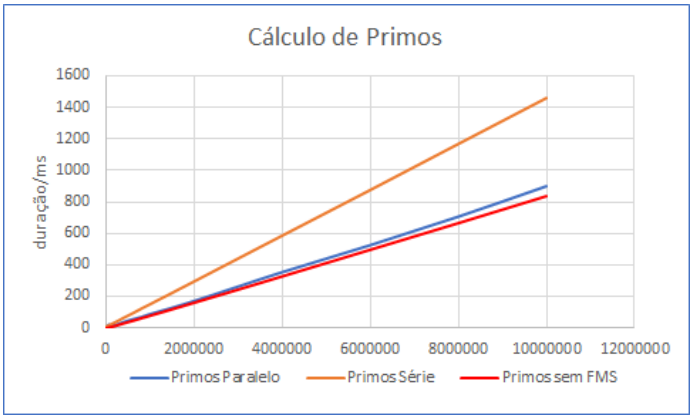



Figura 7. Gráfico dos tempos de resposta do teste para o cálculo dos números primos

### C. *Processamento de ficheiros*

Este teste é diferente pois, além de incluir leitura e escrita do disco, apenas tem uma implementação, o esquema da figura 5. Mesmo assim, temos mais liberdade neste teste do que nos outros pois podemos simular pedidos simultâneos de vários clientes e variar o tamanho dos dois ficheiros de entrada.

Num primeiro cenário, foi mantido constante o tamanho dos ficheiros a serem processados (10Kb), variando a quantidade de pedidos simultâneos. A escala vertical e a variação do número de pedidos são logarítmicas para que seja possível avaliar a variação dos tempos de resposta em grandes escalas e também o comportamento do sistema em números de pedidos mais pequenos. Em suma, a escala logarítmica permite-nos avaliar melhor os casos extremos. Os resultados deste cenário estão representados na figura 8.

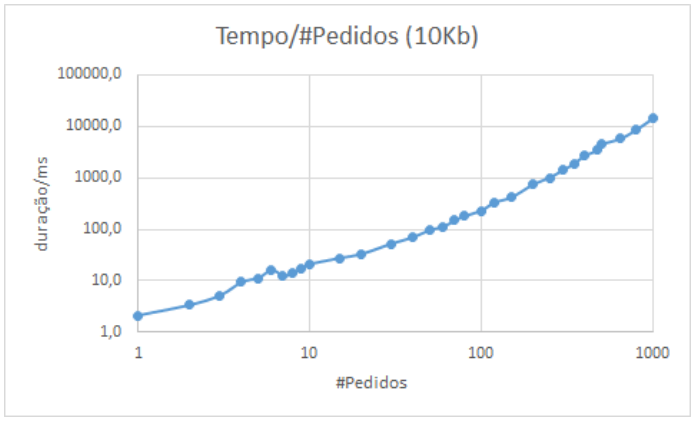



Figura 8. Gráfico dos tempos de resposta do teste de 2 ficheiros com 10Kb e número de processos variável

Num segundo cenário, foi mantido constante o número de pedidos simultâneos (10), variando o tamanho dos ficheiros. As escalas verticais e horizontais são também logarítmicas pelos mesmos motivos do segundo cenário. Os resultados deste cenário encontram-se na figura 9.

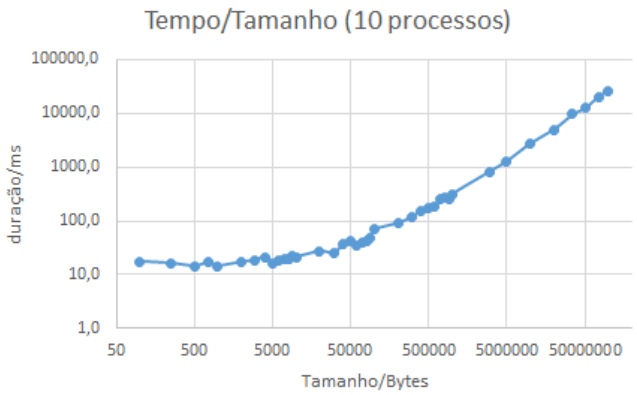



Figura 9. Gráfico dos tempos de resposta do teste de 2 ficheiros com 10 processos e tamanho variável

## VI. Análise crítica

Neste capítulo serão analisados os resultados dos testes realizados.

### A. *Análise da figura 6*

Nesta figura pode-se induzir que a utilização de máquina de estados em processos simples apenas adiciona *overhead* e torna os mesmos mais lentos.

Pode também verificar-se que a colocação de tarefas em série na máquina de estados é altamente desvantajosa, analisando as barras vermelha, amarela e verde. O processo simplificado (verde/sem séries) foi o mais rápido dos da máquina de estados enquanto que o método com a maior divisão de tarefas foi o método mais lento.

Com este teste foi possível concluir que a utilização da máquina de estados não é compensadora em casos em que a tarefa seja simples.

### B. Análise da figura 7

Neste gráfico pode-se verificar também que o uso de tarefas em série na máquina de estados é desvantajoso. É possível no entanto ver que a diferença entre não usar máquina de estado e usar é pequena neste caso, o que, fazendo um *educated guess*, se deve ao facto de que a variável que contém o número a ser avaliado a seguir é sincronizada entre as duas Threads para não existir escritas simultâneas na mesma variável. Este não é, portanto um bom exemplo de paralelização.

### C. Análise das figuras 8 e 9

Na figura 8 acontece o que é esperado: um crescimento linear do tempo de processamento com o número de pedidos simultâneos que resulta num tempo aproximadamente constante por pedido (considerando todos os pedidos iguais).

No entanto, o gráfico 9 mostra claramente uma região constante seguida de uma região linear. Isto é devido ao facto de que o tempo extra provocado pelo *overhead* não é desprezável quando os tempos úteis de processamento são curtos, o que acontece quando os ficheiros a serem processados são mais pequenos. A zona linear é quando o tempo extra provocado pelo *overhead* passa a ser inferior ao tempo útil de processamento. Nesta região, o tempo de processamento é linear com o tamanho do ficheiro, ou seja com o tempo útil de processamento.

## VII. Repartição das tarefas pelos elementos do grupo

A divisão das tarefas pelos elementos do grupo foi a seguinte:

**Luis Paulo Durão:**

- Escrita do código
- Descrição da Solução neste relatório
- Processos Criados como Exemplo neste relatório
- Resultados e Análise Crítica neste relatório
- Conclusões

**Orlanda Sousa:**

- Criação de todas as figuras deste relatório
- Realização dos testes e gráficos
- Criação do documento de anexos
- Definição do problema neste relatório

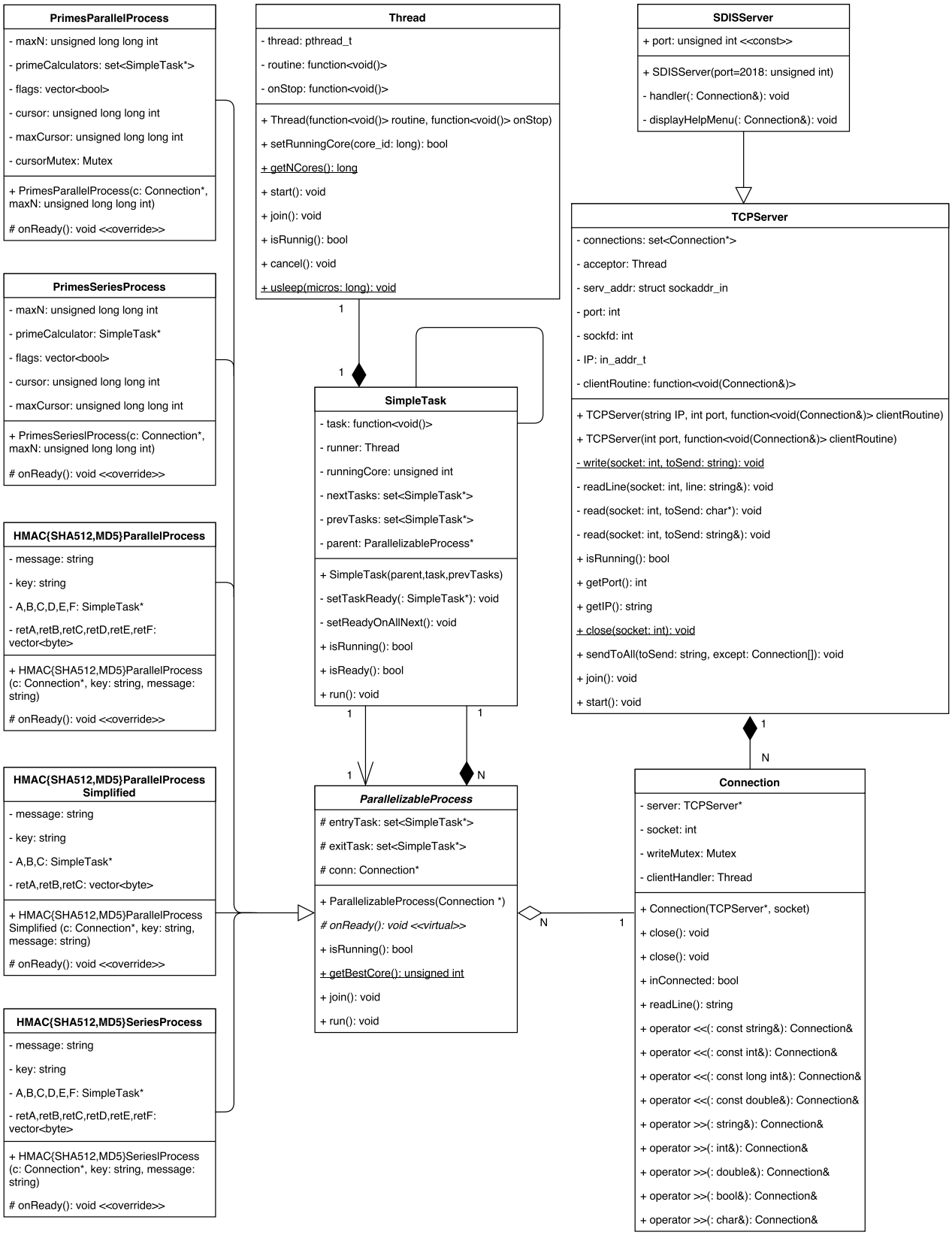
## VIII. Conclusão

Deste projecto foi possível retirar várias conclusões. Essas irão ser enumeradas de seguida:

1) A máquina de estados não deve ser usada para tarefas simples
2) Tarefas em série na máquina de estados são altamente penalizadoras no seu desempenho
3) O tempo de espera em situações com vários clientes é inferior se forem usados todos os núcleos disponíveis.
4) Através da avaliação da performance da máquina de estados com 1 (série) e 4 núcleos (paralelo) de processamento, é possível perceber que a tendência é que a máquina de estados fique mais rápida com o aumento do número de núcleos mas podemos também saber que o aumento da performance não será linear com o número de núcleos devido à lei de Amdahl.

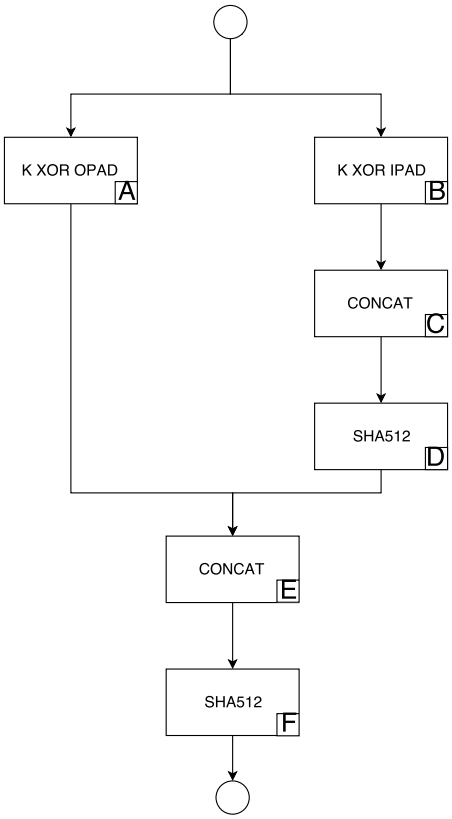
# Anexo 1

Diagrama UML da aplicação



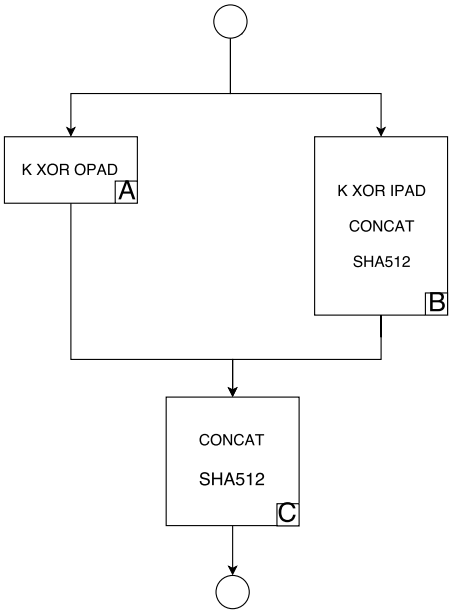
# Anexo 2

Diagrama de paralelização do HMAC



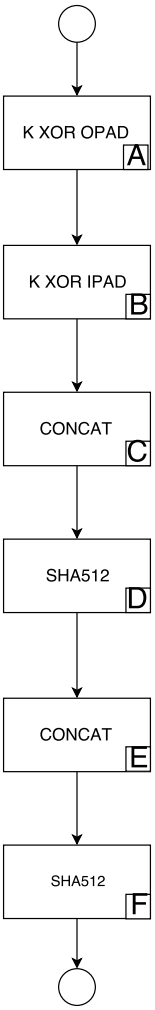

# Anexo 3

Diagrama de paralelização simplicado do HMAC



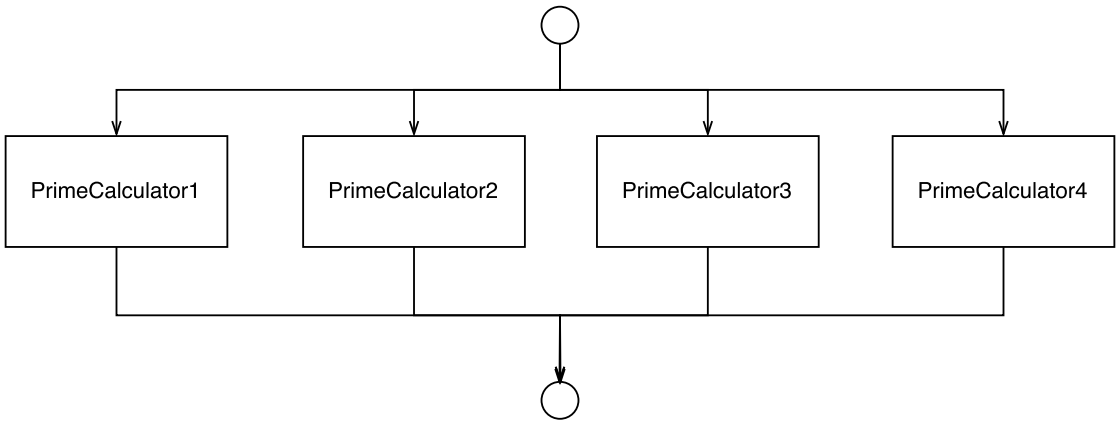
# Anexo 4

Diagrama em série do HMAC



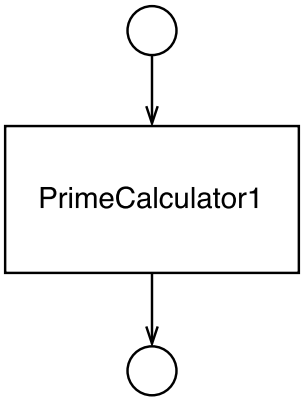

# Anexo 5

Diagrama de paralelização do cálculo de primos



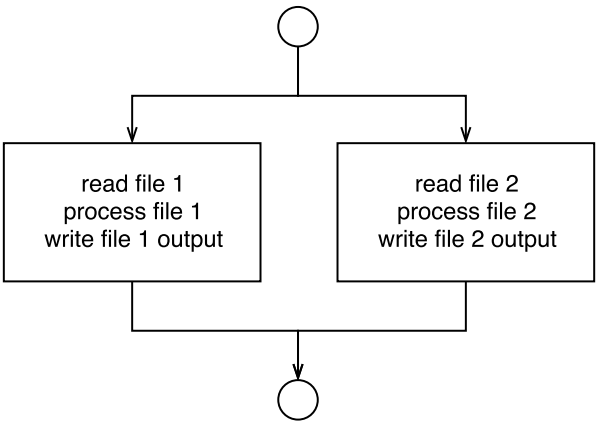
# Anexo 6

Diagrama em série do cálculo de primos



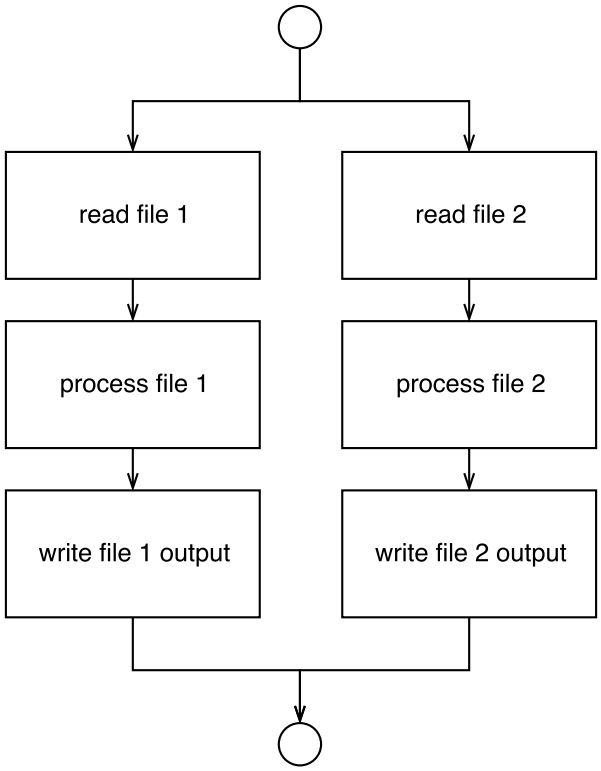

# Anexo 7

Diagrama de paralelização do exemplo com ficheiros simplificado



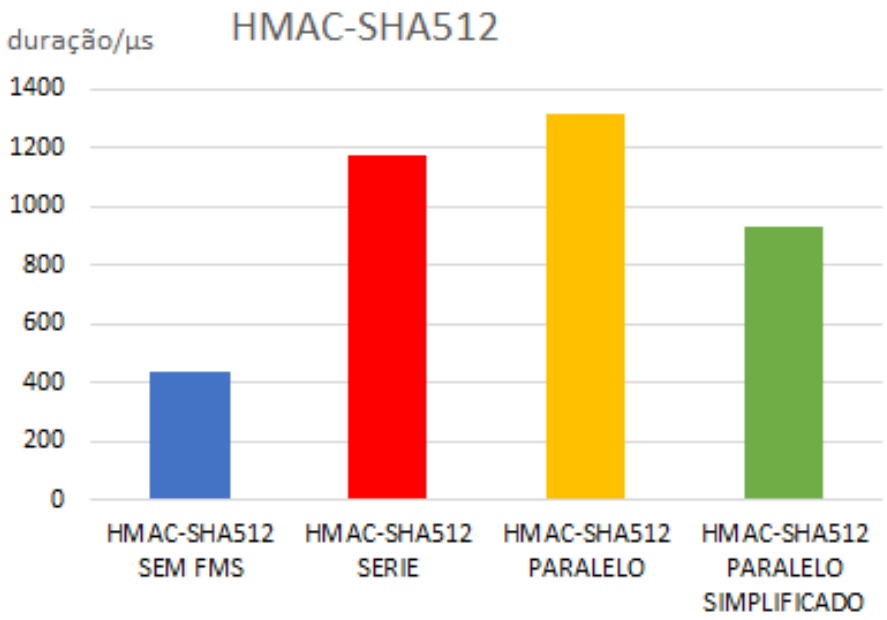
# Anexo 8

Diagrama de paralelização do exemplo com ficheiros



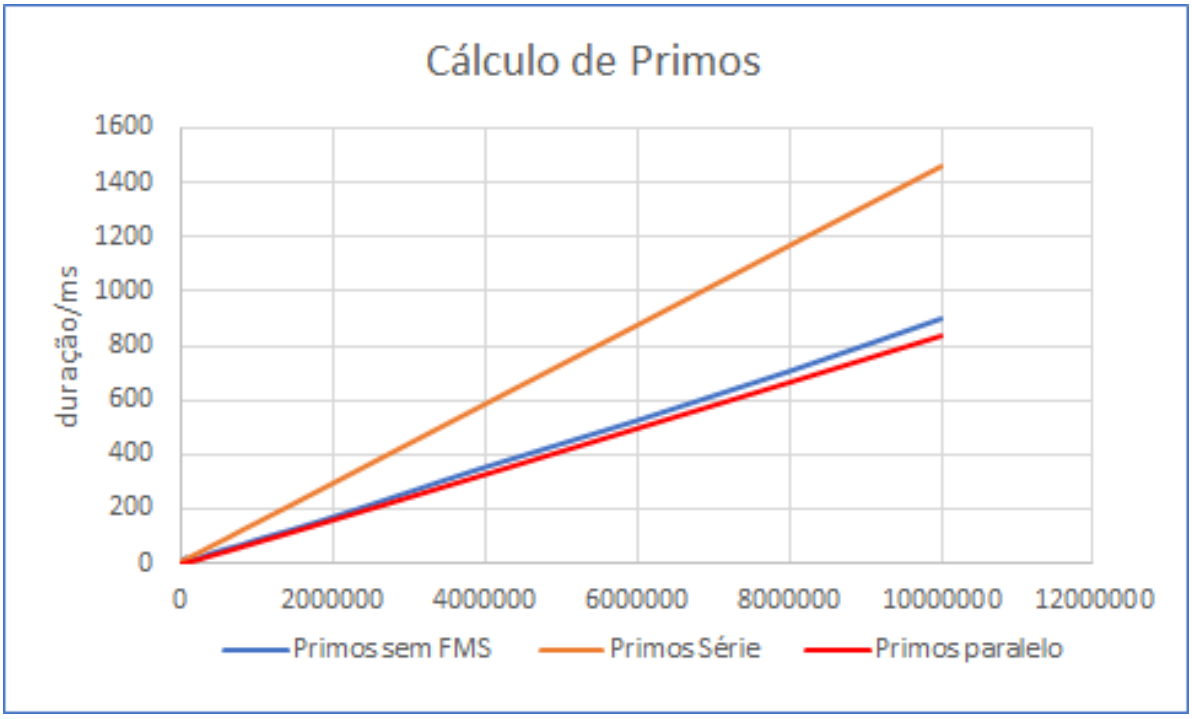

# Anexo 9

Diagrama de colunas do teste HMAC-SHA512



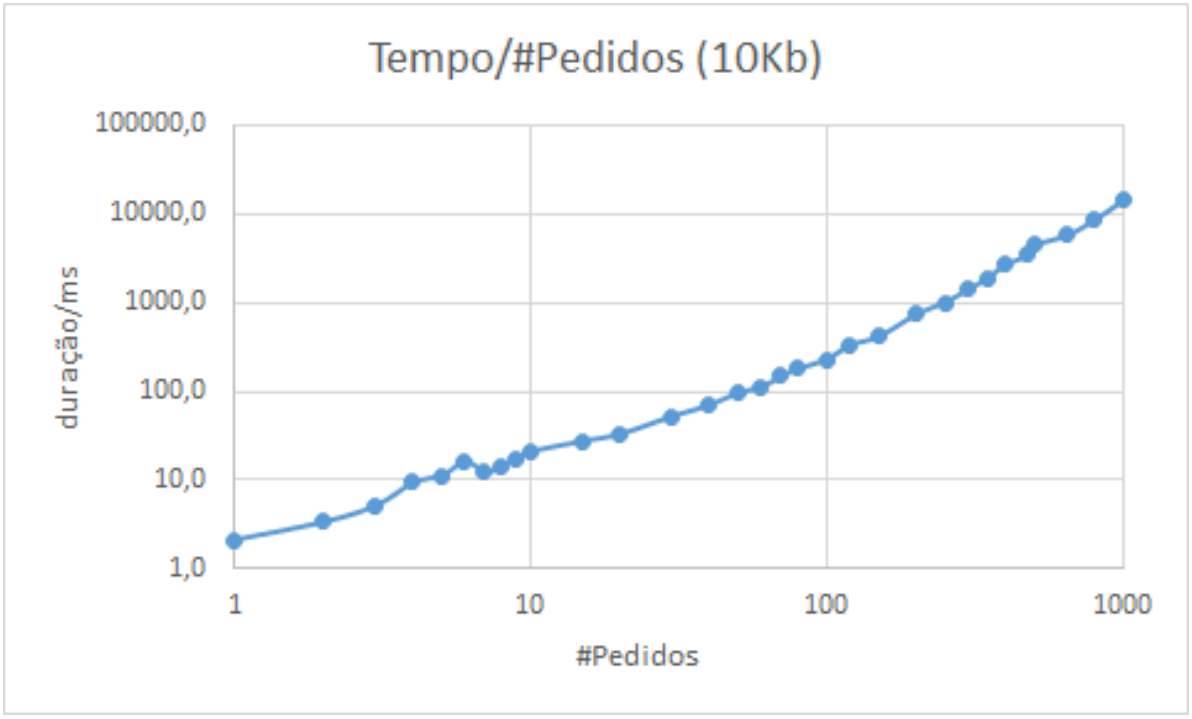
# Anexo 10

Gráfico dos tempos de resposta do teste para o cálculo dos números primos




# Anexo 11

Gráfico dos tempos de resposta do teste de 2 ficheiros com 10Kb e número de processos variável



# Anexo 12

Gráfico dos tempos de resposta do teste de 2 ficheiros com 10 processos e tamanho variável