Uma comparação entre sistemas multiprocessados de memória compartilhada e de troca de mensagens baseada em CSP

Dalmon Ian Martins de Oliveira
Bacharelado em Ciência da Computação
Departamento de Informática – UFPR

12 de agosto de 2021

Resumo

Sistemas multiprocessados têm como objetivo melhorar a performance de programas utilizando paralelismo. Para o funcionamento correto do sistema, é necessário que os processadores se comuniquem e se coordenem. Este trabalho compara dois modelos de multiprocessamento: memória compartilhada e sincronização com semáforos; e rede de processadores com memória distribuída e sincronização por *rendezvous*. Foi implementado um simulador da arquitetura MIPS que permite extrair informações dos programas nele executados para a análise e comparação da performance dos modelos. O trabalho conclui comentando os resultados dos experimentos.

1 Introdução

Sistemas multiprocessados têm como objetivo melhorar a performance de programas utilizando paralelismo. A possibilidade do paralelismo advém da natureza concorrente dos programas executados nesse tipo de sistema. Devido à concorrência, é necessário que os processadores se comuniquem e se coordenem para o funcionamento correto do sistema.

São comparados dois modelos de multiprocessamento: (i) memória compartilhada, interconexão entre os elementos por barramento, sincronização com semáforos (modelo base); e (ii) rede de processadores com memória distribuída e sincronização por *rendezvous* (modelo CSP). A comparação parte da execução de dois problemas clássicos de sincronização (produtor-consumidor e banquete dos filósofos), e da análise dos contadores de performance para cada um dos modelos. A principal motivação do trabalho é investigar a utilização de um modelo de programação baseado em troca de mensagens com a comunicação explícita.

No modelo base, o barramento utilizado é implementado com uma estrutura de dados que garante que todos os processadores que tentem acessá-lo eventualmente consigam realizar esse acesso. Os semáforos utilizados são semáforos contadores, implementados utilizando instruções padrão do conjunto MIPS.

No modelo CSP, a rede consiste em nós de processamento dispersos em uma malha retangular. A troca de mensagens e sincronização entre os processadores foi implementada como uma extensão do conjunto de instruções do MIPS. A transmissão de mensagens é realizada por todos os nós de processamento.

Na comparação entre os modelos, para o problema produtor-consumidor foi observado que o modelo CSP apresenta vantagens sobre o modelo base em relação a tamanho do código da solução e performance. Essa situação se inverte para o problema do banquete dos filósofos. Um equívoco foi cometido na implementação dos problemas no modelo CSP, imitar o modelo base com a sincronização por semáforos. Além de resultados contraintuitivos, até mesmo os resultados positivos são afetados pelo equívoco.

Este trabalho é similar a [5], embora tenha sido desenvolvido de forma independente.

Este trabalho é organizado da seguinte forma: a Seção 2 introduz e define os conceitos utilizados para a compreensão das descrições do simulador e dos experimentos; a Seção 3 escreve o simulador e os experimentos; a Seção 4 contém uma argumentação informal sobre a equivalência dos mecanismos de sincronização; a Seção 5 mostra os códigos utilizados no experimento; a Seção 6 apresenta, analisa e comenta o resultado dos experimentos; a Seção 7 faz uma comparação geral entre os modelos considerando os resultados das simulações; e a Seção 8 conclui o trabalho.

2 Definições

Esta seção define e descreve conceitos utilizados no trabalho. A seção inicia apresentando conceitos essenciais para o entendimento da concorrência e os mecanismos necessários para a solução de problemas do tipo. Em seguida são apresentados a noção de execução paralela, o tipo de sistema computacional que utiliza dos conceitos anteriores para obter melhor performance e como a memória desses sistemas podem ser implementadas. Logo após, são introduzidas linguagens para a descrição e programação de problemas de concorrência. A seção finaliza com breves descrições dos processadores que inspiraram e são utilizados no trabalho.

2.1 Eventos e processos

Considere o seguinte trecho de código:

Programa 1: Implementação da Formula de Bhaskara

```
Bhaskara() {
    a = 1;                      /* E0 */
    b = 2;                      /* E1 */
    c = 3;                      /* E2 */

    bb = b * b;                 /* E3 */
    aux = 4 * a * c;            /* E4 */

    delta = bb - aux;           /* E5 */

    sqrt_delta = sqrt(delta);   /* E6 */

    x0 = b + sqrt_delta;        /* E7 */
    x1 = b - sqrt_delta;        /* E8 */

    print(x0, x1);              /* E9 */
}
```

Eventos são um conjunto de instruções de máquina que executam alguma computação, no caso do exemplo, os eventos (denotados E_n) realizam atribuições, operações aritméticas e imprimem valores de variáveis. Tem-se então que o Programa 1 pode ser descrito como o conjunto de eventos:

$$Bhaskara = \{E_0, E_1, E_2, E_4, E_5, E_6, E_7, E_8, E_9\}. \tag{1}$$

Processos são uma sequência de eventos [8]. Seja o operador ";" indicador da execução sequencial dos eventos, o operador "." indicador o término da execução. Executando os eventos do exemplo em ordem numérica, tem-se então o seguinte processo:

$$E_0; E_1; E_2; E_3; E_4; E_5; E_6; E_7; E_8; E_9; . \tag{2}$$

A ordem de execução dos eventos no processo não é totalmente arbitrária. Dada a definição da relação *acontece antes* em [8], denotada como "$\rightarrow$", os eventos do subconjunto $\{E_0, E_1, E_2\}$

podem ser executados em qualquer ordem, desde de que ocorram antes dos eventos do subconjunto $\{E_3, E_4\}$, que por sua vez, também podem ser executados em qualquer ordem. Aplicando este conceito ao exemplo, sua descrição em termos da relação acontece antes é:

$$\{E_0, E_1, E_2\} \rightarrow \{E_3, E_4\} \rightarrow E_5 \rightarrow E_6 \rightarrow \{E_7, E_8\} \rightarrow E_9\,. \tag{3}$$

A ideia de granularidade aplicada aos conceitos de eventos e processos dependem da sua utilidade para a análise: é possível subdividir os eventos do conjunto *Bhaskara* (Equação 1) de forma que os novos eventos contenham exatamente uma instrução de máquina cada; Também é possível agrupar os eventos do conjunto *Bhaskara*, como por exemplo, ao agrupar todos os eventos do subconjunto $\{E_0, E_1, E_2\}$ em um novo evento E_{attr}.

2.2 Concorrência

Dados dois eventos (a, b), esses eventos são *concorrentes* quando eles podem ocorrer em qualquer ordem [8]:

$$a \not\rightarrow b \land b \not\rightarrow a. \tag{4}$$

Se os eventos não satisfazem essa definição, eles são chamados de *eventos sequenciais* porque uma execução fora de ordem produz resultados errados. Eventos concorrentes são independentes, *i.e*, não há relação de causalidade entre os eventos, em contraste com eventos sequenciais, que caso executados fora de ordem, faz com que o resultado da execução do processo possivelmente seja incorreto.

A *ordenação dos eventos* indica o momento da execução de um evento em relação aos demais. Devido à relação acontece antes, pode-se afirmar que um processo é um conjunto parcialmente ordenado [8]. Na Equação 3 têm-se que $E_3 \rightarrow E_5$, $E_4 \rightarrow E_5$ e E_3, E_4 concorrentes. Há mais de uma sequência de eventos que satisfaz as condições:

$$...\, E_3; E_4; E_5;\; ... \tag{5}$$

$$...\, E_4; E_3; E_5;\; ... \tag{6}$$

Consequentemente, concorrência refere-se também à composição dos eventos do processo. Pode-se então rearranjar o texto do Programa 1 para obter um processo equivalente ao apresentado na Equação 2:

$$E_0; E_2; E_4; E_1; E_3; E_5; E_6; E_7; E_8; E_9;\; . \tag{7}$$

2.3 Sincronização

Eventos concorrentes podem ser executados fora de ordem, já que são independentes, mas os efeitos colaterais da execução dos eventos podem afetar os demais eventos concorrentes, alterando o resultado da computação. Para garantir o resultado esperado, algumas soluções como semáforos e sincronização por *rendezvous* foram propostas já na década de 1960 [4].

2.3.1 Condição de corrida

Condição de corrida é a condição de múltiplos processos que acessam e manipulam os mesmos dados concorrentemente [1], o resultado da execução dos processos é determinada então pela ordenação os eventos dos processos envolvidos.

2.3.2 Região crítica

Dado um sistema computacional executando múltiplos processos, e os processos compartilham de um recurso em comum, a *região crítica* é o conjunto de eventos que manipulam o recurso compartilhado.

Para evitar condições de corrida, as regiões críticas dos processos envolvidos devem satisfazer as seguintes condições: (i) apenas um processo executa sua região crítica (*exclusão mútua*); (ii) nenhum processo espera indefinidamente para entrar na região crítica; e (iii) nenhum processo fora da região crítica pode bloquear os demais processos.

2.3.3 Operações atômicas

Operações atômicas ou *operações indivisíveis* são operações que não interrompem ou interferem umas com as outras. Para que uma operação seja atômica, dois eventos a e b envolvidos na operação, devem satisfazer a seguinte condição:

$$a \to b \oplus b \to a\,. \tag{8}$$

O operador "$\oplus$" denota a operação *ou exclusivo.*

2.3.4 Semáforos

Os *semáforos* foram propostos por Dijkstra [4], e utilizam o princípio da exclusão mútua para sincronizar os processos concorrentes. Sejam "`sem`" uma variável de tipo inteiro inicializada com um valor maior ou igual a zero, e `P()`, `V()` duas funções. `P(sem)` decrementa o valor do semáforo em um. Se o valor resultante for maior que zero, então a execução prossegue normalmente. Caso contrário, o processo espera até que recurso seja liberado, *i.e*, até que o semáforo seja incrementado para um valor maior que zero. `V(sem)` incrementa o valor do semáforo em um. `P()`, `V()` são operações atômicas.

2.3.5 Rendezvous

Rendezvous é um mecanismo de sincronização no qual os processos envolvidos aguardam uns aos outros em um determinado ponto de suas execuções até que todos os demais processos também atinjam esse ponto. Dados dois processos A, B, dois eventos $a_x, a_r \in A$, dois eventos b_y, $b_r \in B$, a_r, b_r são os eventos de sincronização, a_x, b_y são eventos quaisquer de seus respectivos processos que executam antes de a_r, b_r. O processo A faz *rendezvous* com o processo B quando:

$$a_r \to b_r \wedge a_x \to a_r \wedge b_y \to b_r\,. \tag{9}$$

2.3.6 Inanição e impasse

Starvation ou *inanição* é a condição de um processo que necessita de um recurso para continuar sua execução, mas, o processo nunca consegue acessar o recurso.

Deadlock ou *impasse* é um caso específico da inanição: um processo está em espera para acessar a região critica e depende de algum outro processo para liberar o acesso, mas o segundo processo nunca o libera.

2.4 Execução paralela

Execução paralela é a execução simultânea dos eventos de um processo, ou de diversos processos. Note que concorrência e paralelismo não são a mesma coisa. Concorrência diz respeito à composição dos eventos dos processos ao passo que paralelismo é a execução simultânea desses eventos. A partir desse ponto, é considerado que "execução paralela" e *paralelismo* são sinônimos.

Paralelismo em dados consiste em explorar as características dos dados processados de forma a permitir que os processadores operem sobre blocos de dados simultaneamente. Por exemplo, o cálculo dos elementos do vetor b, obtido pela multiplicação dos elementos do vetor a por uma constante K, pode ser definido por:

$$b_i = \sum_{i=0}^{n} a_i K \, . \tag{10}$$

Nota-se que não há dependências entre o cálculo dos elementos individuais de b, o que permite que múltiplos processadores calculem elementos do vetor resultado sem interferir com a computação efetuada pelos outros processadores.

Paralelismo no nível de tarefa consiste em quebrar o problema computacional em diferentes processos e atribuir esses processos a diferentes processadores. Com base no exemplo da Equação 10, dado um sistema com p processadores, sejam[1] $n = |a|$; $p < n$ e p múltiplo de n; $i \in [0..n)$; e $j \in [0..p)$, pode-se dividir a tarefa de multiplicação em p processos:

$$proc_j : b_i = \sum_{i=j(n/p)}^{(j+1)(n/p)} a_i K \tag{11}$$

Esse é o tipo de paralelismo normalmente explorado por sistemas multiprocessados, e é o foco deste trabalho.

2.5 Multiprocessamento

Um *sistema multiprocessado* é definido neste trabalho como um sistema computacional que possui mais de um processador.

O objetivo de um sistema multiprocessado é melhorar algum parâmetro relacionado à performance (tempo de execução, consumo de energia, vazão dos dados, ou alguma combinação destes) dos processos executados. Em um sistema implementado adequadamente, esse objetivo é cumprido dada a habilidade inata de paralelismo do sistema computacional e da independência dos eventos concorrentes nele executados.

Sistemas multiprocessados se apresentam em várias formas e tamanhos: Desde meados da primeira década do século XXI, o público geral tem acesso a circuitos integrados com mais de um processador, e com os avanços na tecnologia de fabricação cada vez mais processadores podem ser incluídos num circuito integrado.

2.6 Organização de memória

Este trabalho define como *organização de memória* o conceito de como a memória é visualizada do ponto de vista lógico pelos programadores do sistema.

2.6.1 Memória compartilhada

O modelo de organização de memória compartilhada consiste nos processadores acessando uma memória unificada do ponto de vista do programador. A comunicação entre os processadores acontece na memória por meio de operações de *load* e *store*: a operação *load* lê um valor da memória e a operação *store* armazena um valor na memória. Essas operações são o equivalente lógico das instruções `load` e `store`.

[1] Notação: um intervalo em $\mathbb{Z}$ é representado por $[a..b]$ se a e b pertencem ao intervalo (fechado-fechado), ou $(a..b)$ se a e b não pertencem ao intervalo (aberto-aberto). Um intervalo pode ser fechado-aberto ($[a..b)$) ou aberto-fechado ($(a..b]$).

2.6.2 Memória distribuída

O modelo de organização de memória distribuída consiste em um conjunto de processadores, cada um com sua memória privativa. A comunicação entre os processadores ocorre por meio de operações de *send* (envia um dado à outro processador) e *receive* (recebe dado de algum processador). Do ponto de vista lógico, cada processador acessa com *load*s e *store*s somente a sua memória privativa.

2.7 Communicating Sequential Processes

Communicating Sequential Processes (CSP) é uma linguagem formal para descrever a concorrência entre processos em sistemas computacionais. Este trabalho emprega a versão de CSP descrita no trabalho original de C.A.R Hoare [6].

Um programa em CSP comporta-se de forma similar a pseudocódigo, com construções que permitem a descrição da concorrência no sistema: (i) um comando que indica a execução concorrente dos processos; (ii) um comando de entrada; (iii) um comando de saída; e (iv) um comando de seleção de alternativas. Esse texto utiliza a mesma notação do original [6] para definir os comandos em questão, embora corte alguns detalhes para fins de brevidade e altere alguns dos símbolos utilizados.

2.7.1 Comando de execução paralela

O comando de execução paralela especifica a execução concorrente dos processos que o constituem. Os processos não compartilham estado entre si. O comando completa quando todos os processos constituintes finalizam suas execuções.

Programa 2: Definição do comando paralelo

```
<comando paralelo> ::= [<processo>{||<processo>}]
```

O comando abaixo indica a execução concorrente dos processos P e Q:

```
[ P || Q ]
```

2.7.2 Comandos de entrada e saída

Os comandos de entrada e de saída especificam a comunicação entre dois processos.

Programa 3: Definição do comando de entrada e saída

```
<comando de entrada> ::= <fonte>?<variável destino>
<comando de saída> ::= <destino>!<expressão>
<fonte> ::= <processo>
<destino> ::= <processo>
```

A comunicação ocorre quando um processo Q executa um comando de entrada cuja fonte é o processo P, o processo P executa um comando de saída cujo destino é o processo Q e o tipo do dado enviado pelo processo P é do mesmo tipo da variável de destino do processo Q. O seguinte comando ilustra essa situação. A variável `str` é do tipo `string`.

```
[P -- Q!"hello, world" || Q -- str:string; P?str ]
```

O comando de entrada falha caso a fonte não exista. O comando de saída falha caso o destino não exista ou o tipo de dado enviado não corresponda com o tipo da variável de destino. Os comandos de entrada e saída sincronizam os processos envolvidos por *rendezvous*.

2.7.3 Comando de seleção de alternativas

O comando de seleção de alternativas é uma estrutura de execução condicional na qual uma determinada lista de comandos só é executada quando a respectiva condição (*guarda*) é satisfeita.

Programa 4: Definição do comando de seleção de alternativas

```
<seleção> ::= [<comando de guarda>{#<comando de guarda>}]
<comando de guarda> ::= <guarda>-><lista de comandos>
<guarda> :: = <lista de guardas>|<lista de guardas>;<comando de entrada>
<lista de guardas> :: = <elemento da guarda>{;<elemento da guarda>}
<elemento da guarda> ::= <expressão booliana>|<declaração>
```

A diferença deste comando com relação a outras estruturas de mesmo tipo como `if...else` ou `switch` é que a ordem de execução das guardas é arbitrária. Se a guarda escolhida falhar, outra é escolhida até que se esgotem as alternativas. Caso todas as guardas falhem o comando de seleção de alternativas falha. Por exemplo, para selecionar entre três alternativas:

```
[x>y->PROCESSA_XY # P?a->PROCESSA_A # Q?b->PROCESSA_B]
```

Os comandos de guarda são:

```
x > y -> PROCESSA_XY /* CG0 */
P ? a -> PROCESSA_A  /* CG1 */
Q ? b -> PROCESSA_B  /* CG2 */
```

O comando de seleção de alternativas arbitrariamente seleciona um comando de guarda. Por exemplo, seja CG_0 o primeiro comando a ser executado e a guarda `x > y` é avaliada. Supondo que falhe, o comando de seleção de alternativas escolhe CG_2, mas o comando de entrada da guarda `Q ? b` não está pronto. Por fim, o comando de seleção de alternativas seleciona CG_1, cuja guarda `P ? a` está pronta, então a lista de comandos indicada por `PROCESSA_A` é executada.

2.8 occam

Por ser uma linguagem formal para a descrição de sistemas, CSP não se adéqua enquanto linguagem de programação. Em 1982 foi introduzida a linguagem de programação *occam*, que implementa os conceitos definidos em CSP [2]. A sintaxe da linguagem é diferente da que é usada em CSP, mas como implementa as mesmas estruturas já vistas anteriormente, essa não é discutida neste trabalho.

2.9 Transputer

O *Transputer* é uma implementação em silício do modelo de computação da linguagem occam. O objetivo dos projetistas era explorar o decrescente custo de fabricação de circuitos integrados para implementar sistemas multiprocessados. Este trabalho faz uso da arquitetura do transputer descrita em [9].

O transputer diferencia-se das demais arquiteturas da época devido ao seu conjunto mínimo de instruções (16 instruções, com um número maior em versões mais recentes), memória integrada e facilidade de conectar-se a outros transputers.

Os transputers se conectam por par de links, um par para cada direção da comunicação (Norte, Sul, Leste, Oeste). A comunicação ponto-a-ponto emprega um protocolo serial. Por ser uma comunicação ponto-a-ponto, múltiplos transputers podem comunicar-se concorrentemente, aumentando a quantidade de dados que o sistema pode processar em paralelo.

2.10 MIPS

MIPS é uma arquitetura de processadores do tipo RISC, desenvolvido no início da década de 1980 [7]. Suas características principais são: (i) os dados são operados dentro dos registradores;

(ii) os únicos tipos de instrução que alteram a memória são as instruções `load` e `store`; e (iii) as instruções têm tamanho fixo de 32 bits.

3 Experimentos

Para efetuar esse trabalho foi implementado um simulador de sistemas multiprocessados. Dois modelos de sistemas multiprocessados são simulados, um com memória compartilhada, interconexão dos processadores por barramento e sincronização por semáforos, e outro com memória distribuída, interconexão dos processadores ponto-a-ponto (formando uma malha retangular) e sincronização por *rendezvous*. Os experimentos efetuados nesse trabalho consistem na execução de dois problemas clássicos de sincronização.

3.1 Problemas clássicos de sincronização

Os problemas escolhidos para a comparação de desempenho dos dois modelos são pares *produtor-consumidor* e o *banquete dos filósofos*. Esses problemas clássicos são normalmente utilizados para testar os mecanismos de sincronização [1].

3.1.1 Produtor-consumidor

O processo produtor produz algum dado que é consumido pelo processo consumidor. Após o processo produtor produzir o dado, este dado é inserido em uma área de transferência (*buffer*) na qual o processo consumidor o remove. O *buffer* é implementado como uma fila circular e pode armazenar até `MAXELEM` elementos.

A inserção dos elementos do *buffer* ocorre no fim na fila. A remoção dos elementos dá-se no início da fila. Para garantir que os apontadores do início e fim da fila sejam consistentes, os processos não devem acessar o *buffer* concorrentemente.

Caso o *buffer* esteja cheio, o processo produtor aguarda até que seja liberada uma posição para a inserção de um novo elemento. Caso o *buffer* esteja vazio, o processo consumidor aguarda até que algum elemento seja inserido.

Este trabalho explora duas variações do problema. Na primeira há um processo produtor e C consumidores, com $C \geq 1$. Na segunda versão são dois produtores e C consumidores, $C \geq 1$.

3.1.2 Banquete dos filósofos

No problema do *banquete dos filósofos*, têm-se 5 instâncias de um processo de tipo *filósofo* que realizam duas ações, comer e pensar.

Para comer, os filósofos sentam-se em uma mesa circular com 5 pratos, 5 garfos (cada garfo posicionado entre dois pratos) e no meio da mesa há um prato com macarrão que é (magicamente) reposto a todo momento. Para retirar e comer sua porção, o filósofo utiliza o garfo que está a sua esquerda mais o garfo que está a sua direita. Após comer (e somente após comer), o filósofo devolve os dois garfos à mesa, levanta-se, e vai realizar a ação de pensar até que sinta fome novamente e repita o processo.

O problema consiste em como os filósofos vão organizar-se à mesa de forma a que todos tenham acesso aos dois garfos e que nenhum filósofo morra de fome.

3.2 Modelo base

O modelo base consiste em um sistema com $p > 1$ processadores, uma única memória compartilhada uniformemente entre os processadores por meio de um barramento. Apenas um processador obtém acesso ao barramento por ciclo da simulação. A sincronização entre os processos utiliza semáforos.

3.3 Modelo CSP

O modelo CSP consiste de p processadores conectados através de uma malha retangular de enlaces ponto-a-ponto, tal que $linhas \times colunas = p$. Cada processador possui memória privativa. A sincronização entre os processos utiliza os comandos CSP de entrada, saída e seleção de alternativas.

Os links de comunicação são análogos aos encontrados no transputer, mas o formato de mensagem é diferente. O primeiro e segundo campos da mensagem indicam o destinatário e o remetente. O terceiro campo é o dado da mensagem, um inteiro de 32 bits. Dois campos indicam a quantidade de saltos através da mensagem na rede e se a mensagem é uma confirmação de recebimento de uma mensagem com dados válidos (ACK). Uma mensagem é inserida na rede pelo comando de saída, e esta é removida por um comando de entrada ou seleção de alternativas.

3.4 Simulador

O simulador está programado em linguagem C e implementa um subconjunto das instruções da arquitetura MIPS. O subconjunto escolhido foi o mínimo necessário para a execução dos experimentos, que foram compilados para a arquitetura utilizando um cross-compilador GCC na versão 10.2.0-r5.

A arquitetura MIPS foi escolhida devido seu uso no curso de Arquitetura de Computadores da graduação da UFPR. Para a simulação do modelo de multiprocessamento de memória distribuída, foram implementadas interfaces seriais que emulam as encontradas no transputer.

Apesar de simular múltiplos processadores, o simulador executa em uma *thread*, o que garante que os resultados de múltiplas execuções de um experimento sejam determinísticos, além de facilitar a depuração do simulador. O código fonte do simulador pode ser encontrado em [3].

3.4.1 Processadores

Das estruturas requeridas da especificação da arquitetura MIPS32 são implementadas 42 instruções do conjunto de instruções, o registrador *program counter* (PC), o banco de registradores de uso geral (GPR) com 32 registradores e o par de registradores HI e LO. Não são implementados o Coprocessador 0 (CP0), interrupções, exceções, caches e gerenciamento de memória. Das estruturas opcionais, é implementado o Coprocessador 2 (CP2). Contadores de performance são implementados para obter parâmetros da execução dos experimentos, esses contadores são específicos do simulador e não fazem parte da arquitetura MIPS.

Este trabalho utiliza a notação “`GPR[reg]`” para se referir ao registrador do GPR com o número de registrador “`reg`”.

Todos os processadores iniciam sua execução no primeiro ciclo da simulação. O PC inicial da execução do programa é indicado no executável. Por ser um sistema multiprocessado, cada processador recebe em sua criação um número identificador (ID), que é armazenado em `GPR[K0]`.

O processador finaliza sua execução ao encontrar uma instrução `wait`. Quando todos os processadores do sistema finalizam suas execuções, o simulador finaliza sua execução com sucesso. Caso algum erro ocorra, o simulador imprime uma mensagem de erro e finaliza sua execução.

3.4.2 Memória

A memória é um vetor de *bytes*. Para facilitar a escrita e a leitura dos dados, o simulador permite indexar a memória em *bytes* e em palavras de 4 *bytes*.

3.4.3 Barramento

No modelo base (com memória compartilhada), apenas um processador acessa o barramento a cada ciclo. Para fazer com que todos os processadores eventualmente acessem a memória, a

arbitragem pelo acesso ao barramento é implementado com uma estrutura de dados FIFO (*first-in-first-out*). A implementação dessa FIFO garante que nenhum processador fique a esperar eternamente pelo acesso ao barramento. Um processador ganha acesso ao barramento caso todas requisições anteriores já tenham sido atendidas.

Dado um sistema com p processadores, o processador que deseja utilizar a memória para a operação *load* ou *store* primeiro faz uma requisição ao barramento, que consiste em enfileirar o ID do processador em uma fila circular de tamanho p, caso o ID já se encontre na fila, este não é enfileirado novamente.

O barramento possui um variável booliana de controle (`used`) que indica se o barramento foi utilizado no ciclo atual. Se `used` tiver valor 0, o controlador do barramento então verifica o elemento inicial da fila, caso o valor seja o mesmo valor do ID do processador requerente, este é desenfileirado e processador ganha o acesso ao barramento e executa a operação de acesso à memória.

Caso `used` tenha valor 1, a fila está cheia ou o elemento inicial da fila tem valor diferente do ID do processador requerente, o processador não ganha acesso ao barramento e não executa a operação de acesso à memória, devendo tentar novamente no próximo ciclo (seu PC não é incrementado).

A função de requisição de acesso ao barramento permite a execução de apenas um processador por vez, e o pseudocódigo do Programa 5 indica a implementação. A variável global `bus_queue` é a fila FIFO do barramento.

Programa 5: Implementação da requisição de acesso ao barramento

```
busacc(ID) {
    enqueue(bus_queue, ID);       /* enfileira ID */

    if (used) {                   /* barramento utilizado? */
        return FAILURE;           /*   repete op. no próx. ciclo */
    }

    if (head(bus_queue) != ID) { /* ID é o elem. inicial da fila? */
        return FAILURE;           /*   caso não, repete operação */
    }

    dequeue(bus_queue);           /* desenfileira ID */

    used = 1;                     /* utiliza barramento */

    return SUCCESS;               /* ID ganha acesso */
}
```

3.4.4 Comunicação ponto-a-ponto

Para a comunicação ponto-a-ponto, cada processador do sistema simulado possui quatro pares de links (Norte, Sul, Leste, Oeste) que encaminham as mensagens na rede. Os links são modelados por uma fila com as mensagens e representam a comunicação em apenas um sentido.

Outra estrutura da comunicação é a *caixa de mensagens*, na qual são depositadas as mensagens que o processador recebe. A caixa de mensagens contém um vetor das mensagens, cujos índices representam o remetente.

O conjunto formado pelos links, processador e caixa de mensagens recebe o nome de *nó da rede*, cuja ilustração é a Figura 1. A cada ciclo do simulador, as mensagens saltam do nó atual para algum dos nós vizinhos. Do ponto de vista do processador, o envio ou recebimento das mensagens é realizado apenas pelos comandos CSP. O simulador implementa, para cada nó, as máquinas de estado que controlam o tráfego de mensagens pela rede.

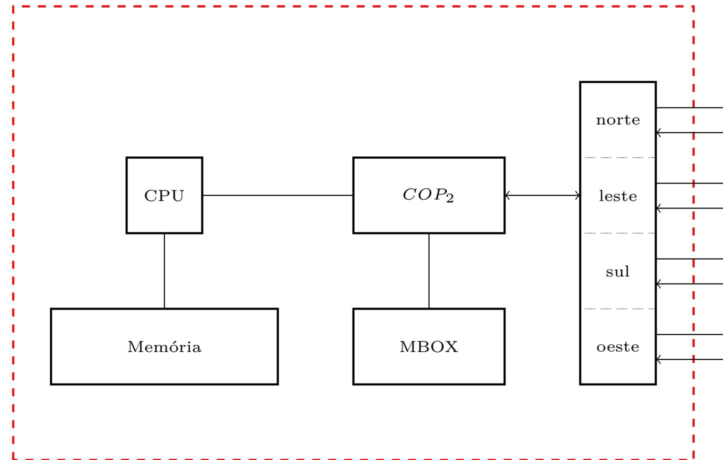

Figura 1: Diagrama de blocos do nó de processamento

As mensagens não são encaminhadas para o nó vizinho caso o link de destino da mensagem não tenha espaço para acomodar a mensagem. Quando isso acontece, é dito que a mensagem foi postergada.

Para transmitir a mensagem, compara-se o ID do nó atual (`cid`) com o nó destino (`dest`). Caso `cid` < `dest` e os nós pertençam a linhas diferentes, a mensagem é enviada ao nó do norte; se os nós pertencem a mesma linha, a mensagem é enviada para o nó do leste. O caso `cid` > `dest` é análogo ao caso anterior, porém enviando as mensagens para os nós do sul e oeste, respectivamente. Caso `cid` = `dest`, a mensagem é enviada à caixa de mensagens do nó. Esse algoritmo de roteamento evita a congestão da rede e o pseudocódigo que o implementa é indicado no Programa 6.

Programa 6: Algoritmo de roteamento das mensagens na malha 2D

```
guidance(cid, dest) {
    if (cid < dest) {
        if (sameline(cid, dest)) { /* ID atual é oeste do destino? */
            return EAST;
        }
        return NORTH;              /* ID atual é sul do destino? */
    }

    if (cid > dest) {
        if (sameline(cid, dest)) { /* ID atual é leste do destino? */
            return WEST;
        }
        return SOUTH;              /* ID atual é norte do destino? */
    }

    return MBOX;                   /* mensagem alcançou o destino */
}
```

3.4.5 Implementação dos semáforos

A arquitetura MIPS possui duas instruções para a implementação de operações atômicas: `ll` (*load-linked*) e `sc` (*store-conditional*). O par de instruções implementa a operação *read-modify-write*, que consiste nos processadores de um sistema multiprocessado lerem um dado, modificá-lo e escrevê-lo de volta na memória, sem que o dado seja acessado por outro(s) processador(es).

O processador que executa a instrução `ll` reserva um endereço, indicando o início da operação

read-modify-write e lê o conteúdo desse endereço para o para um registrador específico que mantém o endereço reservado. A operação termina com sucesso quando o mesmo processador executa a instrução `sc` nesse endereço. A operação atômica falha se alguma outra instrução do tipo *store* (de qualquer processador do sistema) é executada no endereço reservado.

O pseudocódigo nos programas 7 e 8 mostra a implementação das instruções `ll` e `sc` no simulador. Os parâmetros `rt` e `addr` são o número do registrador destino do dado e o endereço a ser lido, respectivamente. A função `invall(arr, addr)` invalida todas as instâncias do endereço `addr` no vetor `arr`, para indicar aos demais processadores a violação da atomicidade. A variável global `resaddr` é um vetor que armazena os endereços reservados pelos processadores que efetuam uma operação *read-modify-write*.

Programa 7: Implementação da instrução `load-linked (ll)`

```
ll(ID, rt, addr) {
    resaddr[ID] = addr; /* inicia sequência atômica */

    GPR[rt] = Mem[addr];
}
```

Programa 8: Implementação da instrução `store-conditional (sc)`

```
sc(ID, rt, addr) {
    if (resaddr[ID] == addr) {  /* operação é atômica? */
        Mem[addr] = GPR[rt];
        GPR[rt] = 1;            /* sucesso */
        invall(resaddr, addr);
    } else {                    /* atomicidade violada */
        GPR[rt] = 0;            /* falha */
        resaddr[ID] = INVALID;
    }
}
```

Do ponto de vista do programador do sistema, as operações de semáforos `P()` e `V()`, são implementadas como indicado nos programas 9 e 10. A implementação das operações utiliza *espera ocupada* para bloquear os processadores, e as funções `P()` e `V()` são chamadas `Sem_P` e `Sem_V`, respectivamente. Os semáforos são semáforos contadores.

Programa 9: Implementação da operação `P()`

```
Sem_P:
    ll    $t0, 0($a0)     # tenta acesso ao semáforo
    addiu $t0, $t0, -1    # $t0 < 0, repete tentativa
    bltz  $t0, Sem_P
    nop
    sc    $t0, 0($a0)     # tenta atualizar semáforo
    beqz  $t0, Sem_P      # $t0 >= 0, falhou
    nop
    jr    $ra
    nop
```

Programa 10: Implementação da operação `V()`

```
Sem_V:
    ll     $t0, 0($a0)     # tenta acesso ao semáforo
    addiu  $t0, $t0, 1
    sc     $t0, 0($a0)     # tenta atualizar semáforo
    beqz   $t0, Sem_V      # sc falhou?  Repete
    nop
    jr     $ra
    nop
```

3.4.6 Implementação dos comandos CSP

Os comandos CSP que o simulador implementa são os de entrada, saída e seleção de alternativas. Os comandos são implementados utilizando o coprocessador 2 (CP2). A interface consiste em 4 registradores, indicados na Tabela 1, juntamente com 3 operações do CP2, indicados na Tabela 2.

Tabela 1: Registradores do CP2

Nº do registrador	Nº de seleção	Função
0	0	processador correspondente
0	1	dado enviado/recebido
0	2	status
0	5	número de alternativas

Tabela 2: Operações do CP2

Nº da operação	Função	Apelido
2	Comando de entrada	INPUT
3	Comando de saída	OUTPUT
4	Comando de seleção de alternativas	ALT

O texto utiliza a notação "`CP2[reg][sel]`" para se referir ao registrador do CP2 com o número de registrador "`reg`", e número de seleção "`sel`".

As operações `INPUT`, `OUTPUT` e `ALT` modificam `CP2[0][2]` (*status*) para indicar a operação atual. Enquanto o registrador *status* for diferente de zero, o processador é bloqueado pelo controlador de rede (*stall*) e não pode executar a próxima instrução. O registrador *status* é utilizado pelo controlador da rede para identificar o estado da operação de comunicação atual.

3.4.6.1 Comando de entrada O comando de entrada comporta-se como foi definido na Seção 2.7.2. Para implementar o comando de entrada o programador escreve o ID do processador fonte em `CP2[0][0]` (*corr*), executa a operação `INPUT` e lê de `CP2[0][1]` (*data*) o dado recebido. Não há verificação de tipo. O código do Programa 11 mostra a implementação do comando em uma função que pode ser chamada pelo programador.

Programa 11: Implementação do comando de entrada

```
C2_input:
    mtc2 $a0, $0, 0 # corr
    cop2 2          # INPUT
    mfc2 $v0, $0, 1 # data
    jr   $ra
    nop
```

3.4.6.1.1 Implementação no simulador Ao identificar a operação `INPUT`, o controlador da rede verifica se a mensagem do processador correspondente está na caixa de mensagens; se a mensagem não chegou, o processador fica bloqueado até o próximo ciclo.

Caso a mensagem tenha chegado, o controlador de rede cria uma mensagem ACK e tenta a enviar ao processador correspondente. Se o enlace estiver ocupado, o controlador repete a operação no próximo ciclo. Se o ACK foi enviado com sucesso, o controlador retira a mensagem da caixa de mensagens, os dados da mensagem são escritos em *data* e por fim, escreve o valor zero em *status*, o que desbloqueia o processador que efetuou a operação `INPUT` (libera o *stall*).

O pseudocódigo do Programa 12 indica a implementação do controle. `mbox` é a caixa de mensagens do nó, `insmsg(dir, msg)` insere a mensagem `msg` no link de saída do nó na direção `dir`, a função `newack(to, from)` cria uma mensagem ACK a ser enviada para o nó `to` pelo nó `from`.

Programa 12: Implementação do controle do comando de entrada

```
input(mbox, *status, corr, ID, *data) {
    Direction   dir;

    struct msg *msg, *ack;

    msg = mbox[corr];
    if (!msg) {             /* se não chegou nova msg, */
        return TRYAGAIN;    /*   stall */
    }

    /* cria ACK */
    ack = newack(corr, ID);

    dir = guidance(ID, corr);
    if (insmsg(dir, ack) == FAILURE) {
        return TRYAGAIN;  /* se enlace ocupado, */
    }                     /*   stall */

    /* retira mensagem */
    mbox[corr] = NULL;
    *data = msg->data;

    *status = 0;          /* fim da execução do comando */

    return SUCCESS;
}
```

3.4.6.2 Comando de saída O comando de saída é o mesmo definido na Seção 2.7.2. Para implementar o comando de saída o programador escreve o ID do processador destino em *corr*, escreve o dado a ser enviado em *data* e executa a operação `OUTPUT`. Não há verificação de tipo. O

código do Programa 13 mostra a implementação do comando como uma função.

Programa 13: Implementação do comando de saída

```
C2_output:
        mtc2 $a0, $0, 0 # corr
        mtc2 $a1, $0, 1 # data
        cop2 3          # OUTPUT
        jr   $ra
        nop
```

3.4.6.2.1 Implementação no simulador Ao identificar a operação `OUTPUT`, o controlador de rede verifica se há uma nova mensagem a ser enviada. Se esse for o caso, o controlador cria uma nova mensagem copiando os valores de *corr*, o ID do processador e de *data* para os campos de destinatário, remetente e dados (respectivamente) da nova mensagem e tenta enviar a mensagem. Caso não consiga enviá-la, o controlador tenta novamente no próximo ciclo. Ao enviar a mensagem com sucesso, o controlador atualiza um bit de *status* para indicar que a mensagem foi enviada.

Caso a mensagem já tenha sido enviada, o controlador verifica a caixa de mensagens e procura o ACK do processador correspondente. Caso esteja na caixa de mensagens, o ACK é removido e o valor zero é escrito em *status*. Se o ACK não chegou nesse ciclo, nenhum estado é alterado e o controlador tenta novamente no próximo ciclo. Se a mensagem recém chegada não for um ACK, tem-se uma situação de *deadlock*. O pseudocódigo indicado no Programa 14 mostra a implementação do controle. A função `newmsg(to, from, data)` cria uma mensagem para o nó `to` do nó `from` com dados `data`.

Programa 14: Implementação do controle do comando de saída

```
output(mbox, *status, corr, ID, data) {
    Direction   dir;

    struct msg *msg;

    if (sentmsg(st)) {                    /* já enviou mensagem? */
        msg = mbox[corr];
        if (!msg) {                       /* se não chegou ACK, */
            return TRYAGAIN;              /*   stall */
        } else if (!msg->ack) {           /* se não é ACK então */
            return DEADLOCK;              /*   deadlock */
        }
        *status = 0;
        return SUCCESS;
    }

    msg = newmsg(corr, ID, data);         /* nova mensagem */

    dir = guidance(ID, corr);
    if (insmsg(dir, msg) == FAILURE) { /* enlace ocupado? */
        return TRYAGAIN;                  /*   stall */
    }

    setsentflag(status);                  /* mensagem enviada */

    return TRYAGAIN;                      /* stall para esperar o ACK */
}
```

3.4.6.3 Comando de seleção de alternativas O comando de seleção de alternativas difere do apresentado na Seção 2.7.3, pois a única expressão válida para as guardas são comandos de entrada com mais uma cláusula de escape. Para implementar o comando de seleção de alternativas o programador escreve em *corr* o endereço do vetor de alternativas (um vetor de números inteiros de 32 bits em que cada elemento do vetor é um processador fonte de um comando de entrada), escreve em `CP2[0][5]` (*nalt*) o tamanho do vetor de alternativas, executa a operação `ALT` e lê de *data* o dado recebido. Não há verificação de tipo. O Programa 15 mostra a implementação.

Programa 15: Implementação do comando de seleção de alternativas

```
C2_alt:
        mtc2  $a0, $0, 0  # corr
        mtc2  $a1, $0, 5  # nalt
        cop2  4           # ALT
        mfc2  $t0, $0, 1  # data
        sw    $t0, 0($a2)
        addiu $v0, $k1, 0
        jr $ra
        nop
```

3.4.6.3.1 Implementação no simulador Ao identificar a operação `ALT`, o controlador de rede busca na caixa de mensagens por alguma mensagem de algum dos processadores indicados no vetor de alternativas. Caso alguma dessas mensagem tenha chegado, o controlador da rede carrega *data* com os dados da mensagem, remove a mensagem da caixa de mensagens, envia ao processador correspondente uma mensagem ACK, carrega `GPR[K1]` com o ID da alternativa selecionada e carrega *status* com o valor zero. Se o último elemento do vetor de alternativas for o número -1, a operação completa com sucesso, mesmo que nenhum dado tenha sido recebido – essa alternativa é chamada cláusula *default*. Se nenhuma mensagem chegou na caixa de mensagens, o controlador bloqueia o processador que tentará novamente no próximo ciclo.

A operação `ALT` possui uma estrutura de controle que é interna ao controlador de rede. A cada ciclo, o controlador começa a verificação da caixa de mensagens a partir do índice `start` (variável privada de cada nó, inicializada com valor zero), incrementando o índice de forma circular. Os valores da variável `start` são mantidas entre as execuções da operação. Esse algoritmo tenta garantir que todos os enlaces sejam pesquisados pelo do controlador.

O pseudocódigo indicado no Programa 16 mostra a implementação do controle da operação. A variável global `netsize` é o número de nós na rede, `belongs(arr, val)` indica se o valor `val` pertence ao vetor `arr`.

Espaço em branco proposital.

Programa 16: Implementação do controle do comando seleção de alternativas

```
alt(*mbox, *start, *status, *corr, ID, ncl, *data) {
    int defaults, found, done, sel;

    if (corr[ncl - 1] == -1) {       /* default presente? */
        defaults = 1;
    }

    /* seleciona alternativa */
    sel = *start;
    found = 0;
    while (!done) {
        if ((mbox[sel] != NULL) && belongs(corr, sel)) {
            done = found = 1;
        } else {
            sel = (sel + 1) % netsize;
            if (sel == start) {
                done = 1;
            }
        }
    }

    if (!found) {                    /* selecionou alternativa? */
        if (defaults) {              /*   caso não, default presente? */
            return SUCCESS;          /*     sucesso */
        } else {                     /*   senão, stall */
            return TRYAGAIN;
        }
    }

    *start = (*start + 1) % netsize; /* atualiza round-robin */

    /* completa comando de entrada */
    input(mbox, status, sel, ID, data);
    GPR[K1] = sel;
    return SUCCESS;
}
```

3.4.7 Temporização das instruções

Dado um sistema com p processadores, as instruções que realizam operações de *load* e *store* demoram entre 1 e p ciclos para completar devido a implementação do controle do barramento, que impede inanição.

As instruções que implementam os comandos de entrada e seleção de alternativas somente completam quando a mensagem com o dado é recebida pelo nó. No melhor caso, a instrução completa em um ciclo (a mensagem já estava na caixa de mensagens do nó). No caso geral, seja $w \geq 0$ o tempo de espera pelo processador fonte para executar o comando de saída correspondente e $m \geq 1$ o tempo esperando a transmissão da mensagem na rede, w e m contabilizados com base na execução da instrução. O tempo de execução em ciclos é:

$$w + m \tag{12}$$

A instrução que implementa o comando de saída somente completa quando o nó recebe uma mensagem ACK. Seja $y \geq 0$ o tempo de espera no processador destino para a execução do comando de entrada ou seleção de alternativas correspondente, y é contabilizado a partir da

execução da instrução que implementa o comando de saída. O tempo de execução (em ciclos) de um comando de saída é:

$$2 \times m + y\,. \tag{13}$$

A instrução completa após a transmissão da mensagem até o processador destino, mais o tempo de espera pela execução do comando de entrada correspondente, mais do tempo da transmissão da mensagem ACK de volta ao processador que executou o comando de saída. No melhor caso (o processador se comunica com um vizinho que já executou o comando de entrada correspondente), a instrução demora dois ciclos para completar.

As demais instruções completam em um ciclo.

4 Equivalência entre os mecanismos de sincronização

Para uma comparação efetiva dos modelos é necessário verificar se esses são equivalentes. Uma prova formal está fora do escopo deste trabalho, mas uma argumentação informal que os modelos são de fato equivalentes é apresentada nesta seção.

O argumento para a equivalência se baseia em dois fatos: (i) é possível implementar os comandos CSP com semáforos no modelo base; e (ii) é possível implementar as operações do semáforo no modelo CSP.

4.1 Implementação dos comandos CSP no modelo base

Para emular os comandos CSP no modelo base, cada um dos processos necessita de um semáforo para realizar o *rendezvous* e de uma caixa de mensagens para receber os dados. Sejam as variáveis globais `rndz[NPROC]` um vetor de semáforos de tamanho `NPROC` (número de processadores do sistema) inicializado com 0 e `mbox[NPROC]` um vetor de inteiros de 32 bits. Os elementos dos vetores `rndz` e `mbox` representam o semáforo e caixa de mensagem de um processo. A função `processor_id()` retorna o ID do processador.

Seja `PROC_i` um processo que efetua o comando de entrada. Para realizar o *rendezvous* são necessários dois semáforos: um no qual o processo `PROC_i`, que está executando o comando se bloqueie, e outro para que o processo correspondente (`PROC_j`) também se bloqueie. Ao executar o comando de entrada, `PROC_i` libera o processo `PROC_j` que executa o comando de saída correspondente, e bloqueia a si próprio. Após o processo `PROC_j` liberar `PROC_i`, é retornado o valor contido na caixa de mensagens. O pseudocódigo do Programa 17 indica a implementação.

Programa 17: Implementação do comando de entrada no modelo base

```
EQV_input(from) {
    int id;

    id = processor_id();

    Sem_V(&rndz[from]); /* libera quem vai escrever o dado */
    Sem_P(&rndz[id]);     /* bloqueia */
    return mbox[id];    /* retorna o dado */
}
```

Seja `PROC_o` o processo que executa o comando de saída. `PROC_o` bloqueia a si próprio até ser liberado pelo processo que executa o comando de entrada correspondente (`PROC_m`), quando então `PROC_o` escreve o dado na caixa de mensagens e libera `PROC_m`. O pseudocódigo do Programa 18 indica a implementação.

Programa 18: Implementação do comando de saída no modelo base

```
EQV_output(to, data) {
    Sem_P(&rndz[processor_id()]; /* bloqueia */
    mbox[to] = data;             /* envia dado */
    Sem_V(&rndz[to]);            /* libera quem vai ler o dado */
}
```

Para o comando de seleção de alternativas é necessário empregar um mecanismo de tipo *round-robin* para garantir justiça na seleção das alternativas. Uma variável global `rridx[NPROC]`, um vetor de inteiros de 32 bits de inicializado com -1, usada para implementar o *round-robin*. Cada elemento do vetor armazena o índice que o processo utiliza no vetor de alternativas. Esse índice é incrementado a cada chamada ao comando de seleção de alternativas. Após incrementar o índice do *round-robin*, o processo atual efetua um comando de entrada para o processo indicado no elemento do vetor de alternativas. Ao completar, o comando retorna o ID do processo que foi selecionado.

Programa 19: Implementação do comando de seleção de alternativas no modelo base

```
EQV_alt(*clauses, ncl, *data) {
    int id;

    id = processor_id();

    rridx[id] = (rridx[id] + 1) % ncl;      /* atualiza round-robin */
    *data = EQV_input(clauses[rridx[id]]); /* lê dado */
    return clauses[rridx[id]];             /* id do processador atendido */
}
```

4.2 Implementação das operações do semáforo no modelo CSP

Para emular as operações do semáforo utilizando os comandos CSP, são necessários dois processos: `PROC_P` e `PROC_V`. Os processos que utilizam o semáforo devem executar `C2_output(PROC_P, DONTCARE)` e `C2_output(PROC_V, DONTCARE)` para realizar as operações `P()` e `V()` respectivamente. O valor do output é descartado, por isso o valor enviado aos processos (`DONTCARE`) é irrelevante.

O processo `PROC_P` é responsável por manter o valor do semáforo. Seja o vetor de inteiros de 32 bits `procs[]` de tamanho `np` o vetor de alternativas contendo os IDs dos processos que utilizam o semáforo. Após inicializar o semáforo, o processo executa as seguintes ações infinitamente: (i) caso valor do semáforo seja maior que 0, seleciona um processo que executou a operação `P()` e decrementa o valor; e (ii) caso valor do semáforo igual a 0, aguarda o processo `PROC_V` para incrementar o valor. O pseudocódigo no Programa 20 indica a implementação.

Espaço em branco proposital.

Programa 20: Implementação da operação `P()` no modelo CSP

```
PROC_P(initval, *procs, np) {
    int val;
    int ret, ign;

    val = initval;                       /* inicializa semáforo */

    for (;;) {
        if (val > 0) {
            ret = C2_alt(procs, np, &ign); /* seleciona processo  */
            --val;                         /* decrementa semáforo */
        } else {
            C2_input(PROC_V);              /* aguarda incremento  */
            ++val;                         /* incrementa semáforo */
        }
    }
}
```

O processo `PROC_V` é responsável por indicar o incremento do valor do semáforo. A execução do processo consiste em atender uma solicitação de algum dos processos que executaram a operação `V()` e indicar o incremento ao processo `PROC_P`. O pseudocódigo no Programa 21 indica a implementação.

Programa 21: Implementação da operação `V()` no modelo CSP

```
PROC_V(*procs, np) {
    int ret, ign;

    for (;;) {
        ret = C2_alt(procs, np, &ign); /* seleciona processo     */
        C2_output(PROC_P, INCREMENT);  /* incremento do semáforo */
    }
}
```

5 Código dos experimentos

Foram implementados 6 programas na linguagem C, cada um deles sendo uma solução para os problemas clássicos que nos interessam.

Como o simulador representa um sistema altamente idealizado, para aproximar a simulação de um sistema realista, há um laço de espera ocupada, no qual os processos efetuam trabalho computacional, para além das operações de comunicação e sincronização.

5.1 Produtor-consumidor

O produtor e o consumidor se comunicam através de um vetor de números inteiros de 32 bits que implementa uma fila circular com `MAXELEM = 16` elementos.

O processo produtor produz um conjunto de inteiros representados em 32 bits, na faixa de 0 a `MAXVAL - 1`, `MAXVAL = 1024`. Cada inteiro é inserido no *buffer* pelo produtor e após produzir todos os valores, o processo produtor finaliza.

O processo consumidor remove um inteiro de 32 bits do *buffer* e realiza alguma operação (consumo) para dispender tempo. Quando não há mais dados a serem consumidos, o processo consumidor termina. Como descrito na Seção 3.1.1, o problema é modelado com um produtor e $C \geq 1$ consumidores.

5.1.1 Modelo base

São utilizados três semáforos na solução: (i) `sem_lock` trava o acesso ao *buffer*, inicializado com valor um; (ii) `sem_full` trava o produtor quando o *buffer* está cheio, inicializado com valor zero; e (iii) `sem_empty` trava os consumidores que encontrarem o *buffer* vazio, inicializado com valor zero. As variáveis `nfull` e `nempty` indicam quantos processos estão presos em `sem_full` e `sem_empty`, respectivamente. A variável `ct` indica quantos elementos há no *buffer*. A variável `producing` indica se o produtor está produzindo novos dados. Todas essas variáveis são globais.

O produtor produz o dado e acessa a região crítica para inserir o dado no *buffer*. Caso haja espaço disponível no *buffer*, o produtor enfileira o dado e verifica se há algum consumidor que está preso esperando o *buffer*, liberando-o quando for o caso. Caso o *buffer* esteja cheio, o produtor deve esperar que algum consumidor o libere. Após terminar a produção dos dados, o produtor libera todos os processos consumidores que por ventura estejam esperando o *buffer* e termina sua própria execução. O pseudocódigo do Programa 22 indica a implementação.

Programa 22: Implementação do produtor no modelo base

```
producer() {
    int  val, flag;

    for (val = 0; val < MAXVAL; ++val) {
        busywait(PRODUCER_WAIT);   /* produz */
        Sem_P(&sem_lock);
        if (ct < MAXELEM) {        /* espaço disponível? */
            enqueue(val); ++ct;    /* enfileira valor */
            if (nempty > 0) {      /* consumidor bloqueado? */
                Sem_V(&sem_empty); /*   libera-o */
                --nempty;
            }
            flag = SUCCESS;
        } else {                   /* senão, buffer lotado */
            nfull = 1; flag = FULL;
        }
        Sem_V(&sem_lock);
        if (flag == FULL) {        /* buffer lotado? */
            Sem_P(&sem_full);      /* bloqueia a si próprio */
            Sem_P(&sem_lock);      /* após liberação */
            enqueue(val); ++ct;    /*   enfileira valor */
            Sem_V(&sem_lock);
        }
    }

    Sem_P(&sem_lock);
    producing = 0;                 /* produção terminou */
    while (nempty > 0) {           /* libera consumidores em espera */
        Sem_V(&sem_empty);
        --nempty;
    }
    Sem_V(&sem_lock);
}
```

Ao acessar a região crítica, os processos consumidores verificam se há dados no *buffer*, caso haja, o processo remove o dado do *buffer*, consome o dado e libera o produtor caso o *buffer* esteja cheio. Caso não haja dados no *buffer*, o processo verifica se o produtor terminou sua execução, caso ele tenha terminado, o consumidor termina sua execução. Caso o produtor não tenha finalizado, o consumidor deve esperar até que o produtor insira algum dado no *buffer*. O pseudocódigo do Programa 23 mostra a implementação.

Programa 23: Implementação do consumidor no modelo base

```
consumer() {
    int  item, flag;

    flag = SUCCESS;
    while (flag != END) {
        Sem_P(&sem_lock);
        if (ct > 0) {                  /* buffer c/ elem? */
            flag = SUCCESS;
            item = dequeue();
            if (nfull) {
                Sem_V(&sem_full);
                nfull = 0;
            }
        } else if (!producing) {       /* fim produção? */
            item = -1;
            flag = END;                /*   termina */
        } else {                       /* buffer vazio */
            ++nempty;
            flag = EMPTY;
        }
        Sem_V(&sem_lock);
        switch (flag) {
        case SUCCESS:
            busywait(CONSUMER_WAIT); /* consome */
            break;
        case EMPTY:                  /* buffer vazio? */
            Sem_P(&sem_empty);       /*   bloqueia */
            break;
        }
    }
}
```

5.1.2 Modelo CSP

Além dos processos produtor e consumidor, a implementação no modelo CSP utiliza um processo que enfileira e desenfileira os dados (processo enfileirador). Essa solução foi escolhida para permitir o compartilhamento do *buffer*. No modelo base, o *buffer* é compartilhado entre os processadores na memória. Como no modelo CSP a memória é distribuída, se o *buffer* for compartilhado utilizando a memória, cada processador necessita de uma cópia coerente do *buffer*, o que dificulta a implementação da lógica do programa. Ao criar o processo enfileirador que mantém o *buffer*, todos os demais processos necessitam comunicar-se somente com ele.

O processo produtor envia os dados produzidos ao processo enfileirador. Ao término da produção o produtor envia o valor -1 ao processo enfileirador, indicando o fim da produção e termina sua execução. O pseudocódigo do Programa 24 indica a implementação.

Programa 24: Implementação do produtor (CSP)

```
producer() {
    int val;

    for (val = 0; val < MAXVAL; ++val) {
        busywait(PRODUCER_WAIT);    /* produz */
        C2_output(BUFFER, val);     /* envia dados */
    }

    C2_output(BUFFER, -1);          /* indica término */
}
```

O processo consumidor envia ao processo enfileirador uma requisição indicando a intenção de consumir um dado. Após a requisição ser atendida, o consumidor recebe o dado do processo enfileirador. Caso o dado seja -1, o consumidor termina sua execução. O pseudocódigo do Programa 25 indica a implementação.

Programa 25: Implementação do consumidor (CSP)

```
consumer() {
    int item;

    item = 0;
    while (item >= 0) {              /* enquanto não terminar */
        C2_output(BUFFER, WANTDATA); /*   pedido dos dados */
        item = C2_input(BUFFER);     /*   recebe dados */
        busywait(CONSUMER_WAIT);     /*   consome */
    }
}
```

O processo enfileirador recebe o dado do produtor e o enfileira no *buffer*. Logo após, o processo enfileirador verifica se recebeu alguma requisição dos processos consumidores. Caso haja, o processo enfileirador remove um dado do *buffer* e envia ao processo consumidor correspondente. Ao receber o valor -1 do produtor, o processo enfileirador envia o restante dos elementos do *buffer* aos consumidores, envia aos consumidores o valor -1 e termina sua execução. O pseudocódigo do Programa 26 indica a implementação. A variável `consumers` indica o vetor de alternativas, de tamanho `NCONSUMERS + 1`, para acomodar todos os IDs dos processos consumidores mais a clausula *default*.

Espaço em branco proposital.

Programa 26: Implementação do processo enfileirador (CSP)

```
buffer() {
    int data, req, ret, nc;

    /* enquanto produtor não finalizar */
    for (;;) {
        if (ct < MAXELEM) {
            data = C2_input(PRODUCER); /* recebe dado */
            if (data < 0) {            /* fim produtor? */
                goto REMITEMS;         /*   esvazia buf */
            } else {                   /* senão, */
                enqueue(data);         /*   enfileira */
                ++ct;
            }
        }
        /* verifica requisições */
        ret = C2_alt((int *) &consumers,
                     NCONSUMERS + 1, &req);
        if (ret >= 0) {                /* nova requisição? */
            C2_output(ret, dequeue()); /*   atente */
            --ct;
        }
    }

REMITEMS:
    /* desenfileira e envia restante do buffer */
    while (ct > 0) {
        ret = C2_alt((int *) &consumers,
                     NCONSUMERS, &req);
        C2_output(ret, dequeue());
        --ct;
    }

    /* indica fim aos consumidores */
    for (nc = NCONSUMERS; nc > 0; --nc) {
        ret = C2_alt((int *) &consumers,
                     NCONSUMERS, &req);
        C2_output(ret, -1);
    }
}
```

5.1.3 Análise qualitativa

O tamanho do código do modelo CSP é aproximadamente 75% do tamanho do código do modelo com semáforos, mesmo que a solução no modelo CSP empregue um processo a mais.

Na opinião do autor, o código do modelo CSP é de fácil compreensão quando comparado ao modelo base, pois as abstrações do modelo de programação do CSP provêm sincronizações explícitas de forma concisa e clara – CSP é expressivo, conciso e elegante. Além disso a complexidade das abstrações é relativamente pequena, por isso elas são implementadas como funções de uma biblioteca que pode ser utilizada pelo programador do sistema simulado. *Como não existe almoço grátis*, a complexidade do modelo CSP fica escondida na implementação do *hardware* dos controladores das interfaces da rede/malha.

5.2 Produtor-consumidor (versão 2)

Essa versão é análoga a apresentada na Seção 5.1, mas a diferença está na modelagem com dois produtores e $C \geq 1$ consumidores. Os produtores, `PRODUCER0` e `PRODUCER1`, produzem números pares e ímpares respectivamente.

5.2.1 Modelo base

As variáveis e estruturas de dados descritas na Seção 5.1.1 são as mesmas utilizadas nessa versão. A diferença é na variável `producing`, que agora é um vetor com dois elementos, cada elemento indicando o status dos processos produtores.

A solução assume que `PRODUCER0` é um ID par e `PRODUCER1` é um ID ímpar, para fins de simplicidade. Após determinar qual é o processo que está sendo executado, a lista de números a serem produzidos é dividida em dois e a produção ocorre de forma análoga ao mostrado no Programa 22. O pseudocódigo do Programa 27 indica a implementação.

A implementação dos processos consumidores é análoga ao Programa 23; a diferença é que o consumidor só finaliza caso ambos os processos produtores tenham finalizado. O pseudocódigo do Programa 28 indica a implementação.

Programa 27: Implementação da versão 2 do produtor (modelo base)

```
producer() {
    int  val, pid, flag;

    pid = processor_id();
    /* val = (pid == PRODUCER0)? 0 : 1 */
    for (val = pid & 1; val < MAXVAL; val += 2) {
        busywait(PRODUCER_WAIT);
        Sem_P(&sem_lock);
        if (ct < MAXELEM) {
            enqueue(val); ++ct;
            if (nempty > 0) {
                Sem_V(&sem_empty);
                --nempty;
            }
            flag = SUCCESS;
        } else {
            ++nfull; flag = FULL;
        }
        Sem_V(&sem_lock);
        if (flag == FULL) {
            Sem_P(&sem_full);
            Sem_P(&sem_lock);
            enqueue(val); ++ct;
            Sem_V(&sem_lock);
        }
    }

    Sem_P(&sem_lock);
    producing[pid & 1] = 0; /* PRODUCERX finalizou */
    while (nempty > 0) {
        Sem_V(&sem_empty);
        --nempty;
    }
    Sem_V(&sem_lock);
}
```

Programa 28: Implementação da versão 2 do consumidor (modelo base)

```
consumer() {
    int  item, flag;

    flag = SUCCESS;
    while (flag != END) {
        Sem_P(&sem_lock);
        if (ct > 0) {
            item = dequeue();
            if (nfull > 0) {
                Sem_V(&Sem_full);
                --nfull;
            }
            flag = SUCCESS;
        } else if (!producing[0] &&
                   !producing[1]) { /* produtores finalizaram? */
            item = -1;
            flag = END;
        } else {
            ++nempty; flag = EMPTY;
        }
        Sem_V(&sem_lock);
        switch (flag) {
        case SUCCESS:
            busywait(CONSUMER_WAIT);
            break;
        case EMPTY:
            Sem_P(&sem_empty);
            break;
        }
    }
}
```

5.2.2 Modelo CSP

A produção de valores é análoga ao demonstrado no Programa 24. Após determinar qual é o ID do processo, a lista de valores a serem produzidos é dividida em dois e os produtores enviam os dados produzidos ao processo enfileirador até que não hajam mais elementos a serem produzidos. O Programa 29 indica a implementação. A implementação dos processos consumidores é idêntica ao Programa 25.

Programa 29: Implementação da versão 2 do produtor (CSP)

```
producer(void) {
    int val;

    val = (processor_id() == PRODUCER1)? 1 : 0;
    while (val < MAXVAL) {
        busywait(PRODUCER_WAIT); /* "produzindo" */
        C2_output(BUFFER, val);
        val += 2;
    }

    C2_output(BUFFER, -1);
}
```

O processo enfileirador é análogo ao demonstrado no Programa 26, considerando que há mais de um produtor. A variável `producers` indica o vetor de alternativas com tamanho `NPRODUCERS` para acomodar todos os IDs dos produtores.

Programa 30: Implementação da versão 2 processo enfileirador (CSP)

```
buffer() {
    int data, req, ret, nc, np;

    np = NPRODUCERS;
    while (np > 0) { /* enquanto tem produtor produzindo */
        if (ct < MAXELEM) {
            /* seleciona produtor */
            ret = C2_alt((int *) &producers,
                         NPRODUCERS, &data);
            if (data < 0) { /* produtor terminou? */
                --np;       /*  decrementa produtores em execução */
                continue;
            } else {        /* senão enfileira */
                enqueue(data); ++ct;
            }
        }
        ret = C2_alt((int *) &consumers,
                     NCONSUMERS + 1, &req);
        if (ret >= 0) {
            C2_output(ret, dequeue()); --ct;
        }
    }

    while (ct > 0) {
        ret = C2_alt((int *) &consumers,
                     NCONSUMERS, &req);
        C2_output(ret, dequeue()); --ct;
    }

    for (nc = NCONSUMERS; nc > 0; --nc) {
        ret = C2_alt((int *) &consumers,
                     NCONSUMERS, &req);
        C2_output(ret, -1);
    }
}
```

5.2.3 Análise qualitativa

O tamanho do código no modelo CSP é aproximadamente 82% do tamanho do código para o modelo base. A discussão da Seção 5.1.3 é igualmente válida aqui. Implementar a versão com dois produtores é uma tarefa trivial e não altera a semântica do problema.

5.3 Banquete dos filósofos

Como a mesa na qual os filósofos se sentam é circular, caso todos os filósofos estejam sentados, todos conseguem levantar o garfo que está a sua esquerda. Mas ao tentar levantar o garfo que está a sua direita, esse já foi levantado pelo filósofo vizinho, gerando uma situação de inanição.

Uma solução para esse problema é garantir que em nenhum momento todos os filósofos estejam a mesa. Um garçom é introduzido na solução, cuja a tarefa é garantir que no máximo 4 dos 5 filósofos estejam sentados na mesa ao mesmo tempo.

Na versão descrita na Seção 3.1.2, o problema não tem um término definido, os filósofos comem e pensam para sempre. Para obter uma simulação finita, cada filósofo tem um número finito de ideias durante sua execução. Após pensar (em espera ocupada) o número de ideias é decrementado e o filósofo se senta a mesa. Quando o garçom permitir que o filósofo se sente para comer, e após levantar ambos os garfos, o filósofo come (também em espera ocupada), devolve os garfos à mesa e se levanta. Caso o filósofo não tenha mais nenhuma ideia para pensar, ele termina sua execução. Do contrário, as ações de pensar e comer são repetidas.

5.3.1 Modelo base

A constante `PHILOS` indica a quantidade de filósofos do problema. A constante `IDEAS` indica a quantidade de ideias em que cada filósofo deve pensar.

Os garfos são implementados como um vetor de semáforos (`fork`) de tamanho `PHILOS`, inicializado com valor 1. O garfo a esquerda do filósofo é indicado pelo elemento do vetor cujo índice é o ID do processador que executa essa instância do processo filósofo. O garfo a direita é o elemento do vetor cujo o índice é `(ID + 1) % PHILOS`. Essas são variáveis globais.

O garçom é implementado com um semáforo (`footman`[2]) inicializado com valor `PHILOS - 1`. O pseudocódigo no Programa 31 mostra a implementação da solução.

Programa 31: Implementação do banquete dos filósofos (modelo base)

```
philosopher() {
    unsigned int id, fid, ideas;

    id = processor_id();
    fork[id] = 1;                /* inicializa semáforo */

    fid = (id + 1) % PHILOS;     /* garfo da direita */
    for (ideas = IDEAS; ideas > 0; --ideas) {
        busywait(THINK);         /* pensa */
        Spin_lock(&footman);     /* senta a mesa */
        Spin_lock(&fork[id]);    /* levanta talheres */
        Spin_lock(&fork[fid]);
        busywait(EAT);           /* come */
        Spin_unlock(&fork[fid]); /* devolve talheres */
        Spin_unlock(&fork[id]);
        Spin_unlock(&footman);   /* levanta da mesa */
    }
}
```

5.3.2 Modelo CSP

O código do modelo CSP emprega 3 classes de processos: filósofo, talher e garçom, e cada processo é alocado a um processador. A solução emprega uma topologia específica da malha que é formada por 2 linhas com 6 processadores. Os filósofos são os processadores da primeira linha com os IDs 0, 1, 3, 4 e 5. Os talheres são os processadores da segunda linha com IDs 6, 7, 9, 10 e 11. O garfo à esquerda do filósofo é seu vizinho imediato na malha. O garçom executa no processador 2 e o processador 8 não é utilizado.

O código do processo filósofo é análogo ao do Programa 31. O vetor de inteiros `fork` de tamanho `PHILOS` contém os IDs dos processadores contendo o garfo que os filósofos devem utilizar. Cada processador que executa um filósofo possui uma cópia (*read-only*) desse vetor.

[2] A tradução literal de *footman* é lacaio, mas o autor traduziu livremente como garçom por ser uma metáfora menos obtusa.

Após pensar, o filósofo indica ao garçom que vai *sentar-se* a mesa. Assim que estiver sentado, o filósofo indica aos talheres que irá *levantá-los* e quanto obtiver os dois, come sua refeição. Após comer, o filósofo indica aos talheres que *os devolveu a mesa* e indica ao garçom que irá *levantar-se* da mesa. Ao terminar sua execução, o filósofo indica aos talheres e ao garçom seu término. O pseudocódigo indicado no Programa 32 mostra a implementação.

Programa 32: Implementação do filósofo (CSP)

```
philosopher(ID) {
    int ideas, ownfork, nbrfork;

    /* ID dos garfos da esquerda e direita */
    ownfork = fork[ID];
    nbrfork = fork[(ID + 1) % PHILOS];

    for (ideas = IDEAS; ideas > 0; --ideas) {
        busywait(THINK);              /* pensa */
        C2_output(FOOTMAN, SITDOWN); /* senta a mesa */
        C2_output(ownfork, PICKUP);  /* levanta talheres */
        C2_output(nbrfork, PICKUP);
        busywait(EAT);                /* come */
        C2_output(nbrfork, PUTDOWN); /* devolve talheres */
        C2_output(ownfork, PUTDOWN);
        C2_output(FOOTMAN, GETUP);   /* levanta da mesa */
    }

    /* finaliza */
    C2_output(FOOTMAN, DONE);
    C2_output(ownfork, DONE);
    C2_output(nbrfork, DONE);
}
```

Os processos do tipo talher representam os garfos que os filósofos utilizam, definindo quais filósofos tem acesso ao garfo e em qual estado o garfo se encontra. O vetor constante de inteiros `philo` contém os IDs dos processadores nos quais executam os filósofos. Cada processador que executa um processo garfo possui uma cópia (*read-only*) desse vetor. Apenas os filósofos imediatamente à esquerda e direita do garfo têm permissão para acessá-lo. Qualquer outro filósofo que tente acessar aquele garfo entra em *deadlock*. Os IDs dos filósofos com permissão de acesso ao garfo estão no vetor de alternativas `allowed`. A variável `np` indica quantos filósofos com permissão de acesso estão executando.

O processo garfo inicia sua execução no estado *na mesa*. Enquanto há filósofos com permissão de acesso executando, o garfo verifica se algum filósofo quer realizar alguma ação. Se a ação for levantar talher, o garfo transaciona para o estado *levantado* e espera até que o filósofo que o levantou o retorne à mesa. Se o garfo está na mesa e a ação indicar o término da execução do filósofo, `np` é decrementado. O pseudocódigo do Programa 33 indica a implementação.

Espaço em branco proposital.

Programa 33: Implementação dos talheres (CSP)

```
cutlery(ID) {
    int np, aux, ret, action;

    int allowed[2];

    /* seleciona filósofos com permissão de acesso */
    allowed[0] = philo[ID];
    if ((aux = ID - 1) < 0) {
        allowed[1] = philo[PHILOS - 1];
    } else {
        allowed[1] = philo[aux];
    }

    np = 2;
    while (np > 0) {
        /* estado atual: na mesa */
        ret = C2_alt((int *) allowed, 2, &action); /* seleciona filósofo */
        /* qual ação o filósofo executou? */
        switch (action) {
        case PICKUP:                               /* levantou */
            /*
             * estado atual: levantado
             * espera filósofo devolver talher
             */
            while (C2_input(ret) != PUTDOWN);
            break;
        case DONE:                                 /* terminou */
            --np;
            break;
        }
    }
}
```

O processo garçom controla o acesso à mesa, que é modelada com a seguinte estrutura de dados:

```
static struct {
    int seated;      /* número de filósofos sentados */
    int arr[PHILOS]; /* vetor de alternativas */
} table;
```

A mesa está lotada quando há `PHILOS - 1` filósofos sentados a mesa. Enquanto há filósofos executando e a mesa não estiver lotada, o garçom seleciona um dos filósofos e verifica qual ação ele vai realizar: (i) se a ação for a de sentar-se a mesa, o garçom insere o ID do filósofo no vetor `table.arr` e incrementa a quantidade de filósofos sentados a mesa (função `sitdown(philoID)`); (ii) se a ação for de levantar-se da mesa, o garçom remove o ID do filósofo no vetor `table.arr` e decrementa a quantidade de filósofos sentados a mesa, (função `getup(philoID)`); e (iii) se a ação for a indicação do término de execução, o número de filósofos executando é decrementado.

Se a mesa estiver lotada, o garçom apenas verifica as ações dos filósofos que estão sentados, esperando que algum deles se levante. O garçom finaliza sua execução quando não há mais filósofos a se sentar. O pseudocódigo do Programa 34 indica a implementação do garçom.

Programa 34: Implementação do garçon (CSP)

```
footman() {
    int np, ret, action;

    np = PHILOS;

    table.seated = 0;

    while (np > 0) {
        if (isfull(table)) { /* mesa lotada */
            /* recebe ação de algum filósofo sentado */
            ret = C2_alt((int *) table.arr,
                         table.seated, &action);
            /* se ação for levantar-se, libera a mesa */
            if (action == GETUP) {
                getup(ret);
            }
        } else {            /* tem espaço a mesa */
            /* recebe ação dos filósofos */
            ret = C2_alt((int *) philo,
                         PHILOS, &action);
            /* qual foi a ação tomada? */
            switch (action) {
            case SITDOWN:   /* sentar */
                sitdown(ret);
                break;
            case GETUP:     /* levantar */
                getup(ret);
                break;
            case DONE:      /* terminar */
                --np;
                break;
            }
        }
    }
}
```

5.3.3 Análise qualitativa

O tamanho do código do modelo CSP é aproximadamente 5 vezes maior que o tamanho do código do modelo com semáforos.

No modelo base, as ações dos filósofos são implícitas na sincronização por semáforos. A natureza explícita do CSP implica em um código maior para a solução escolhida, porque os garfos e o garçom são processos separados, e cada processo deve conter a lógica de controle de acesso ao recurso que representa.

Outro problema está relacionado à implementação do simulador. Como o subsistema de interrupções do MIPS não foi modelado, implementar mais de um processo por processador torna-se algo de difícil execução. Por conta dessa característica, dado que 11 processos são empregados na solução (5 filósofos, 5 garfos e 1 garçom), é necessário um sistema com 12 processadores, uma vez que o simulador emprega somente redes 2D regulares, o que desperdiça um processador.

O subsistema de interrupções não foi implementado por duas razões: (i) a principal motivação do trabalho é explorar o paralelismo do sistema, portanto a execução dos processos deve acontecer em diferentes processadores; e (ii) para endereçar múltiplos processos que executam

num mesmo processador, um esquema mais complexo de nomeação de processos é exigido. Devido a questões de prazo, o modelo mais simples, com um único processo que executa em um processador, foi escolhido.

6 Resultados da simulação

Os programas discutidos na Seção 5 foram compilados e executados no simulador. Um mesmo programa é executado diversas vezes com diferentes *perfis de simulação.* Cada perfil de simulação indica o período de espera ocupada dos processos, de forma que o simulador represente diferentes condições de operação.

Os resultados coletados das simulações do modelo base são: (i) tempo de execução do programa em ciclos; (ii) taxa de utilização do barramento; (iii) taxa de falhas de acesso ao barramento; (iv) número de ciclos tomando-se a média dos processadores (global) em que os processos ficam bloqueados no(s) semáforo(s); e (v) taxa de falhas no acesso ao(s) semáforo(s).

Os resultados coletados das simulações do modelo CSP são: (i) tempo de execução do programa em ciclos; (ii) taxa de utilização da rede; (iii) número médio global dos ciclos em espera para completar a comunicação.

Todos os gráficos apresentados são do tipo linear-logaritmo, indicando que o eixo vertical é apresentado em escala linear enquanto o eixo horizontal é apresentado em escala logarítmica.

Os resultados e discussão da versão 2 do produtor-consumidor é muito similar à discussão da Seção 6.1. Para evitar a repetição, os resultados e a discussão estão no Apêndice A.

6.1 Produtor-consumidor

Os 4 perfis de simulação utilizados são: (i) *static* indica que a espera ocupada é um valor constante de 10 ciclos para a produção e 13 ciclos para o consumo dos dados, tempos que foram escolhidos selecionando os valores de uma faixa que resultam num baixo tempo de execução; (ii) *low* indica que o tempo de produção varia de 2 a 16 ciclos e o tempo de consumo de 1 a 1024 ciclos; (iii) *high* indica que o tempo de produção e consumo variam de 1 a 1024 ciclos; e (iv) *high-p* indica que o tempo de produção varia de 1 a 1024 ciclos e o tempo de consumo varia de 2 a 16 ciclos.

6.1.1 Modelo base

A análise do comportamento do sistema no modelo base parte da seguinte hipótese: considerando-se os tempos de produção e consumo (definidos pelos perfis), incrementar o número de processadores faz com que as execuções das instruções de memória fiquem *espalhados* no tempo, reduzindo a quantidade de acessos simultâneos ao barramento. Porém, ao introduzir mais processadores no sistema, a competição pelo barramento cresce de tal forma que o espalhamento torna-se irrelevante.

6.1.1.1 Tempo de execução A Figura 2 apresenta o número de ciclos da execução do programa, no qual o eixo horizontal indica a quantidade de processadores no sistema e o eixo vertical indica a quantidade de ciclos para finalizar a execução. O comportamento geral dos perfis é o mesmo, o tempo de execução cresce linearmente com a quantidade de processadores do sistema. A exceção da observação de crescimento é o perfil *low* com P variando de 2 a 8 processadores. A razão para tal é apresentada na Seção 6.1.1.

Espaço em branco proposital.

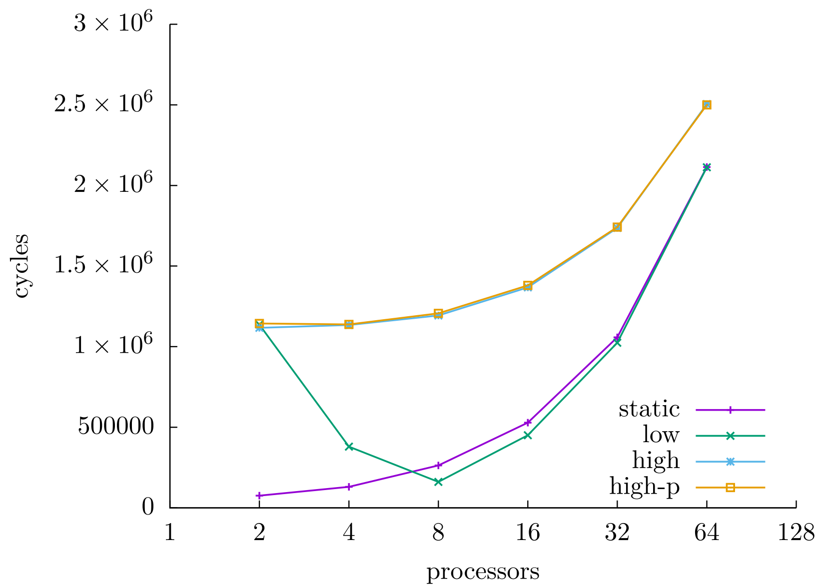

Figura 2: Tempo da execução do programa produtor-consumidor (modelo base)

6.1.1.1.1 Análise qualitativa O perfil *low* varia o tempo de produção entre 2 a 16 ciclos e o tempo de consumo entre 1 e 1024 ciclos. A primeira hipótese para o comportamento do perfil *low* no intervalo de 2 a 8 processadores é a de que o espalhamento das instruções de acesso a memória permite que a produção e consumo dos dados sejam *intercalados* de forma que os processos envolvidos tenham oportunidade de acessar o recurso compartilhado (barramento) sem competição, fazendo com que o sistema execute o programa em menos tempo. Uma segunda hipótese é, considerando-se que o tempo de produção assume valores em uma faixa menor que os tempos de consumo, o produtor consegue atender à demanda dos consumidores.

Outra característica do sistema é que os perfis *high* (com tempos de produção e consumo variando de 1 a 1024 ciclos) e *high-p* (com tempo de produção variando de 1 a 1024 ciclos e o tempo de consumo variando de 2 a 16 ciclos) são similares. Para o perfil *high*, os tempos de produção e consumo estão na mesma faixa de valores, assim o produtor opera na mesma velocidade dos consumidores, reduzindo a taxa de inserção dos dados no *buffer*. No perfil *high-p*, tem-se a situação inversa do perfil *low*, o tempo de produção assume uma grande variedade de valores e o tempo de consumo é menor e varia pouco. Com isso, o produtor não consegue atender à demanda dos consumidores, e ademais, os consumidores estão competindo constantemente pelo *buffer*.

Espaço em branco proposital.

6.1.1.2 Utilização do barramento A Figura 3 apresenta a taxa de utilização do barramento. O eixo horizontal indica a quantidade de processadores e o eixo vertical indica a utilização do barramento em pontos percentuais. Os perfis se comportam da mesma forma: ao incrementar o número de processadores do valor inicial em um fator de 2 (ou 3 no caso do perfil *low*), a utilização do barramento atinge o valor mínimo observado. Para quantidades maiores de processadores, a utilização do barramento cresce rapidamente de forma assintótica em direção aos 100%.

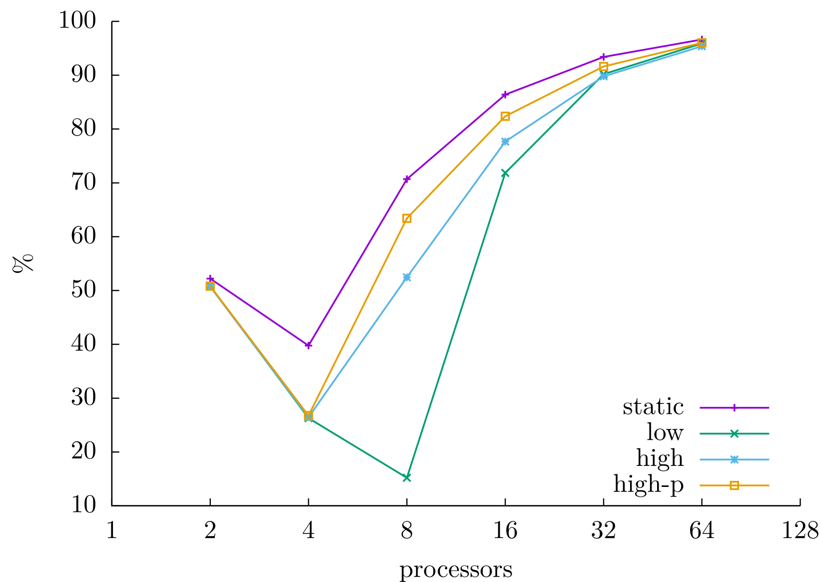

Figura 3: Utilização do barramento pelo programa produtor-consumidor (modelo base)

6.1.1.2.1 Análise qualitativa O gráfico das taxas de utilização do barramento reflete diretamente a hipótese do espalhamento das instruções de memória. Num sistema com 2 processadores, todos os perfis apresentam, grosso modo, a mesma taxa de utilização do barramento (aproximadamente 50%). Ao incrementar a quantidade de processadores do sistema, o espalhamento das instruções de acesso a memória faz com que a utilização do barramento caia para o valor mínimo observado. Os casos em que as taxas de utilização do barramento crescem para um valor maior ou igual que a taxa de utilização para 2 processadores indicam que a competição pelo barramento supera as vantagens que o espalhamento das instruções de memória poderia oferecer.

Espaço em branco proposital.

6.1.1.3 Falhas de acesso ao barramento

A Figura 4 apresenta a taxa de falhas ao acessar o barramento. O eixo horizontal indica a quantidade de processadores e o eixo vertical indica a taxa de falhas. Todos os perfis se comportam de forma similar, crescendo assintoticamente em direção aos 100%. Uma vez que a utilização do barramento se aproxima dos 70%, a grande maioria dos acessos à memória demora mais de um ciclo para completar.

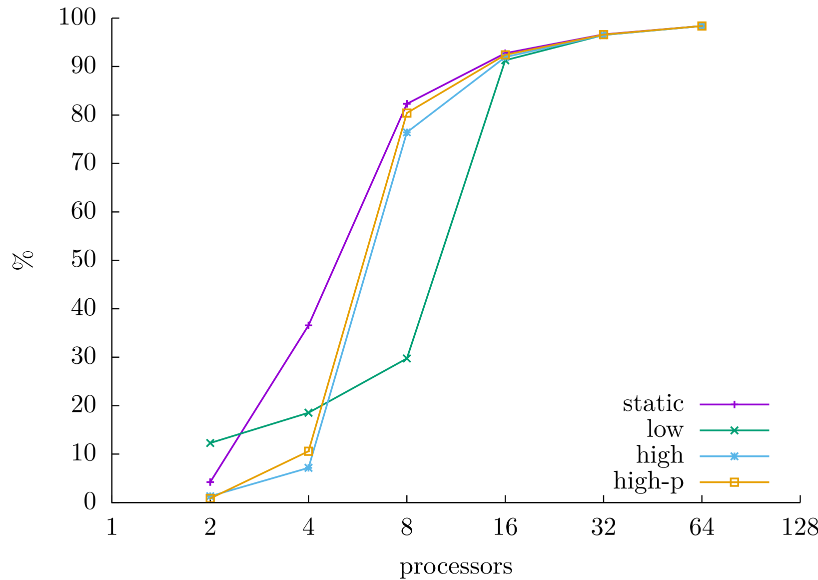

Figura 4: Taxa de falhas ao acessar o barramento do produtor-consumidor (modelo base)

6.1.1.3.1 Análise qualitativa

O gráfico de falhas de acesso ao barramento complementa os resultados no gráfico de taxas de utilização do barramento, pois reforça a observação de que, a partir de um certo número de processadores no sistema, a competição supera as vantagens do espalhamento das instruções de memória. A quantidade de processadores (P) no sistema cuja taxa de falhas de acesso a memória está acima dos 70%, é o mesmo P cuja taxa de utilização do barramento cresce em relação ao valor mínimo observado.

6.1.1.4 Tempo de espera para acessar o semáforo

A Figura 5 apresenta o número de ciclos expostos na espera para acessar a região crítica. Em um sistema com P processadores o diagrama de caixas mostra a somatória do tempo que cada processo dispendeu para cada operação `P()` de todos os experimentos com os 4 perfis. O eixo horizontal indica a quantidade de processadores no sistema e o eixo vertical indica quantidade de ciclos em espera. O comportamento geral dos dados é um crescimento linear com poucos valores discrepantes. Os diagramas de caixa exibem a mesma tendência das curvas dos perfis no gráfico de tempo de execução da Figura 2. Nota-se que os tempos de execução do programa e os tempos em que os processadores do sistema esperam para acessar a região critica estão na mesma faixa de valores.

Espaço em branco proposital.

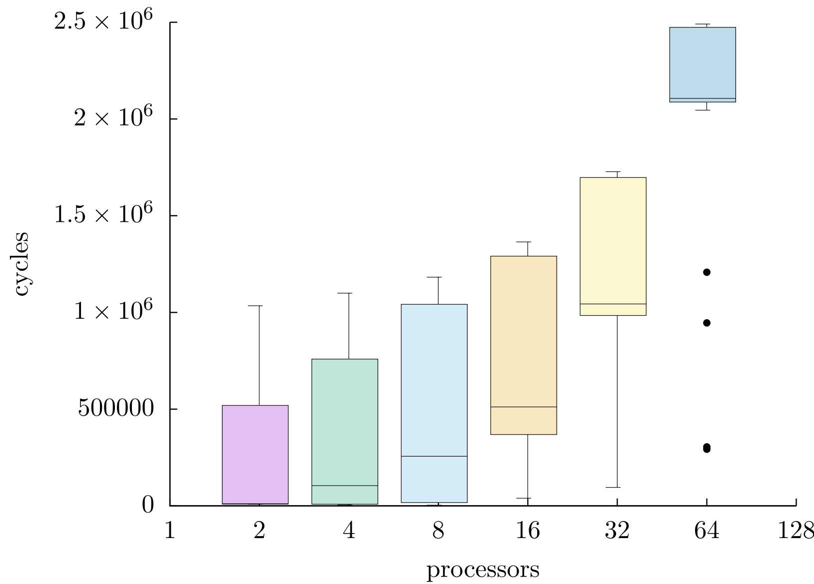

Figura 5: Tempo que um sistema aguarda o acesso aos semáforos (modelo base)

6.1.1.4.1 Análise qualitativa Os diagramas de caixa exibem a mesma tendência das curvas dos perfis no gráfico de tempo de execução porque o que limita a velocidade da execução de um programa em qualquer sistema é seu componente mais lento. No caso do modelo base, o gargalo é o barramento. Isso significa que na medida em que cresce o número de processadores, na maior parte de execução, cada processador dispende mais tempo esperando para acessar a região crítica (a competição pelo barramento afeta diretamente o andamento das operações *read-modify-write*) do que trabalhando na produção ou no consumo dos dados. Uma observação que pode corroborar a afirmação é que, considerando o perfil *high* com 2 processadores, e supondo que o tempo de produção e consumo seja sempre 1024 ciclos (pior caso), têm-se que os tempos totais da produção e consumo seriam $2 \times 1024 \times 1024 = 2.097.152$ ciclos, valor inferior a um quarto do tempo de execução observado.

6.1.1.5 Falhas das operações *read-modify-write* A Figura 6 apresenta a quantidade de operações *read-modify-write* que falharam num sistema com P processadores. O eixo horizontal indica a quantidade de processadores do sistema e o eixo vertical indica a o número de operações *read-modify-write* que falharam. As operações *read-modify-write* indicam as tentativas de acesso aos 3 semáforos empregados na solução (`lock`, `full` e `empty`). Para um sistema com P processadores, os dados dos perfis são combinados e representados por um diagrama de caixa. O comportamento dos dados é assintótico, crescendo em direção ao valor 1300 com poucos valores bem discrepantes. A assíntota parece indicar que quanto maior o número de processadores do sistema, o número de falhas das operações *read-modify-write* se aproxima da quantidade de valores produzidos, e que a competição pelo barramento faz com que os acessos aos semáforos sejam postergados por dezenas de ciclos. Dada a escala do gráfico, o diagrama de caixa para dois processadores é tão pequeno que se torna imperceptível, com variação entre 0 e 5 ciclos. Outra característica relevante do gráfico é a altura das caixas, que para sistemas com 4 e 8 processadores as caixas são altas, indicando grande variação entre os dados observados, em contraste com sistemas com 2 processadores ou com mais de 8 processadores, que apresentam variação dos dados substancialmente menor.

Espaço em branco proposital.

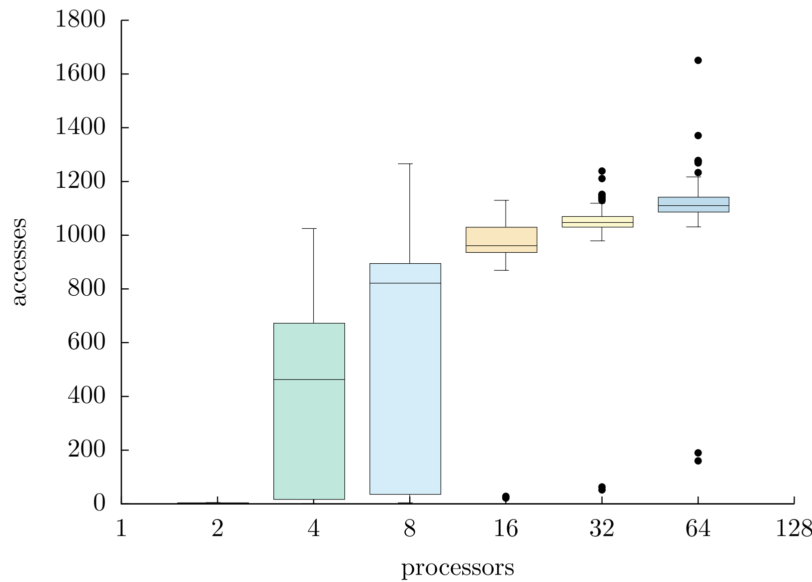

Figura 6: Número de operações *read-modify-write* que falharam (modelo base)

6.1.1.5.1 Análise qualitativa O gráfico mostra que ao aumentar a quantidade de processadores no sistema para um valor maior que 2, a tendência é que ocorra uma falha da operação de *read-modify-write* por elemento produzido. Porém, há uma discrepância entre a quantidade de elementos produzidos (1024) e o valor aproximado da assíntota (1300). Isso ocorre pois há 3 semáforos na solução. O semáforo `lock` regula acesso à região crítica (inserção e remoção dos dados no *buffer*) sendo o provável responsável pela maioria das falhas observadas, já que é o semáforo acessado mais vezes durante a execução do programa. O semáforo `full` bloqueia o produtor quando o *buffer* está cheio, e o semáforo `empty` bloqueia os consumidores quando o *buffer* está vazio. Com o aumento da competição pelo barramento, inevitavelmente as operações *read-modify-write* para esses semáforos falharão com maior frequência.

A altura do diagrama de caixas para sistemas com 4 e 8 processadores indica a eficácia dos perfis no espalhamento das instruções de acesso à memória. Alguns perfis têm um espalhamento de forma a alcançarem uma quantidade de falhas menores que 100, enquanto outros têm uma quantidade de falhas que supera o valor 1000. Isso ocorre pois sistemas com 4 e 8 processadores são os sistemas com os menores valores observados para a utilização do barramento

Para sistemas com 2 processadores, a competição é pequena o suficiente de modo que algum espalhamento garante que não haverá competição pelos semáforos. Para sistemas com $P \geq 16$, a situação se inverte, e há tanta competição pelo barramento, que não importa o espalhamento escolhido, as chances de haver alguém competindo pelo semáforo naquele instante são muito próximas de 100%.

6.1.2 Modelo CSP

É importante notar que, como a solução emprega um processo a mais (processo enfileirador), o número mínimo de processadores do sistema é de 3 processadores, e para as quantidades de processadores maiores há um consumidor a menos quando comparado ao modelo base.

Duas hipóteses guiam a análise: (i) de forma análoga ao modelo base, dados os tempo de produção e consumo, as instruções de comunicação são espalhadas no tempo; e (ii) comparando-se a largura de banda da rede e do barramento, o modelo CSP permite espalhamento no espaço, uma vez que há mais caminhos disponíveis para a comunicação entre os processadores.

Uma característica importante do modelo CSP é a localização do processo produtor em relação ao processo enfileirador. O produtor deve ser vizinho do processo enfileirador, de forma

a reduzir o tempo de comunicação entre esses dois processos. Uma vez que o produtor apenas se comunica com o processo enfileirador, é vantajoso que o produtor resolva a comunicação com o processo enfileirador o mais rápido possível, de forma a liberar o processo enfileirador para a comunicação com os consumidores.

6.1.2.1 Tempo de execução A organização do gráfico da Figura 7 é análoga a Figura 2, apresentando o número de ciclos da execução. As curvas dos perfis *static* e *high-p* são constantes. As curvas dos perfis *low* e *high* decrescem até se aproximar dos valores encontrados nos perfis *static* e *high-p* respectivamente.

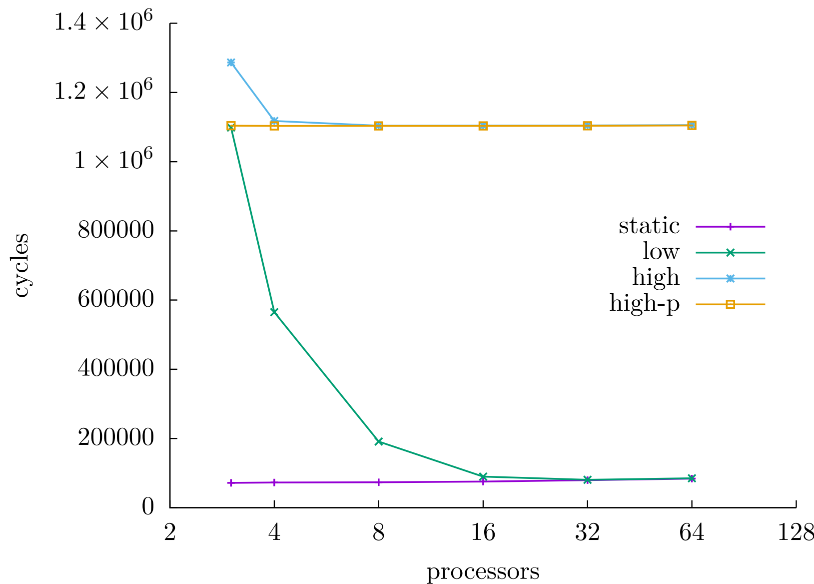

Figura 7: Tempo de execução do programa produtor-consumidor (modelo CSP)

Após alcançarem o valor mínimo observado do perfil, as curvas crescem muito lentamente, o que faz parecer que são constantes. A Tabela 3 apresenta os valores dos tempos de execução, de forma a facilitar a visualização do crescimento das curvas.

Tabela 3: Tempo de execução do produtor-consumidor (modelo CSP)

	P					
perfil	3	4	8	16	32	64
static	7.179e+04	7.282e+04	7.356e+04	7.567e+04	7.949e+04	8.410e+04
low	1.099e+06	5.649e+05	1.914e+05	8.986e+04	8.056e+04	8.513e+04
high	1.287e+06	1.117e+06	1.104e+06	1.104e+06	1.105e+06	1.106e+06
high-p	1.104e+06	1.103e+06	1.103e+06	1.103e+06	1.104e+06	1.105e+06

Esse comportamento ocorre pois o sistema do modelo CSP é limitado pelo processo mais lento de cada perfil, e a variação dos valores é devida aos custos de transmissão das mensagens. Apenas o perfil *low* apresenta características que permitem que haja a variação do tempo de execução.

6.1.2.1.1 Análise qualitativa Nas curvas da Figura 7 o perfil *low* indica condições que permitem que o tempo de execução decresça para $P \leq 32$, e o perfil *high* varia o tempo de execução em uma taxa menor para $P \leq 8$. A partir de um certo ponto os dois perfis apresentam o mesmo comportamento: um crescimento linear com uma taxa muito baixa. Essa taxa é

associada aos custos de comunicação, que crescem com o tamanho da rede. O limite inferior é o tempo de execução do processo mais lento para o perfil.

Produtor: Sejam *producing* a média do tempo de produção e $wait_p$ a média do tempo em espera para completar as operações de envio dos dados para o processo enfileirador. O tempo médio do produtor é:

$$T_{\text{prod}} = 1024 \times producing \times wait_p \tag{14}$$

O número de elementos a serem produzidos não varia, *producing* é constante entre diversas execuções. Apenas $wait_p$ varia consideravelmente, de acordo com o gráfico do lado direito na Figura 10, cujo o comportamento é similar ao gráfico de tempo de execução da Figura 7.

Processo enfileirador: Considerando que *enqueue* e *dequeue* são as médias do tempo de enfileiramento e desenfileiramento dos dados respectivamente, $wait_{\text{rec}}$ a média do tempo esperando para receber um item do produtor, $wait_{\text{alt}}$ a média do tempo esperando para completar o comando de seleção de alternativas, e $wait_{\text{snd}}$ a média do tempo em espera enviando o dado ao consumidor, o tempo médio do processo enfileirador é:

$$T_{\text{buf}} = 1024 \times (wait_{\text{rec}} + enqueue + \ wait_{\text{alt}} + dequeue + wait_{\text{snd}}) \tag{15}$$

Novamente, o número de elementos processados é 1024. Os valores de $wait_x$ são indicados nas curvas dos gráficos da Figura 12.

O gráfico do canto superior esquerdo indica $wait_{\text{rec}}$ e os valores não variam muito, ficando em torno do valor 1000 para os perfis *high*, e *high-p* e 1 para os perfis *static* e *low*.

O gráfico do canto superior direito indica $wait_{\text{alt}}$, que é uma curva menos regular que os demais, porém varia apenas entre 1 e 3 ciclos, que é um valor desprezível se comparado aos demais.

O gráfico do canto inferior indica o valor de $wait_{\text{snd}}$, que varia de 3 a 14 ciclos.

O fator que determina o tempo de execução no processo enfileirador é $wait_{\text{rec}}$ nos perfis *high* e *high-p* e $wait_{\text{snd}}$ para os demais perfis.

Consumidor: Seja *nc* o número de consumidores. Em média cada consumidor recebe uma quantidade de elementos igual a:

$$elem = 1024 \div nc \tag{16}$$

Sejam então $wait_{\text{req}}$ a média do tempo esperando para completar a requisição de pedido dos dados para o processo enfileirador, $wait_{\text{rsp}}$ a média do tempo esperado para receber os dados do processo enfileirador e *consuming* o tempo consumindo os dados. O tempo médio de um consumidor é:

$$T_{\text{cons}} = elem \times (wait_{\text{req}} + wait_{\text{rsp}} + consuming) \tag{17}$$

O número de elementos processados diminui ao introduzir mais processadores no sistema, e *consuming* é estável entre as execuções dos perfis. Os valores de $wait_{cn}$ estão indicados nas curvas da Figura 14. O gráfico da direita indica $wait_{\text{req}}$, que cresce rapidamente para os perfis *high* e *high-p* e devagar para os perfis *static* e *low*. O gráfico da direita indica $wait_{\text{rsp}}$ e decresce de 24 a 23 ciclos, o que é desprezível. O fator que determina o tempo de execução no consumidor é $wait_{\text{req}}$, mas como o número de elementos por processar diminui quando P cresce, a quantidade de elementos acaba limitando o tempo do execução do consumidor.

Enfim o tempo de execução do produtor-consumidor é determinado por:

$$T_{\text{PC}} = \max(T_{\text{prod}}, T_{\text{buf}}, T_{\text{cons}}) \tag{18}$$

6.1.2.2 Utilização da rede A Figura 8 apresenta a taxa de utilização da rede. O eixo horizontal indica a quantidade de processadores e o vertical indica a taxa de ocupação da rede. O comportamento geral dos dados é crescer de forma assintótica em direção aos 100%. A exceção é o perfil *low*, que decresce no intervalo de 3 a 16 processadores. No intervalo de 16 a 64 processadores, o perfil *low* tem um comportamento similar dos demais.

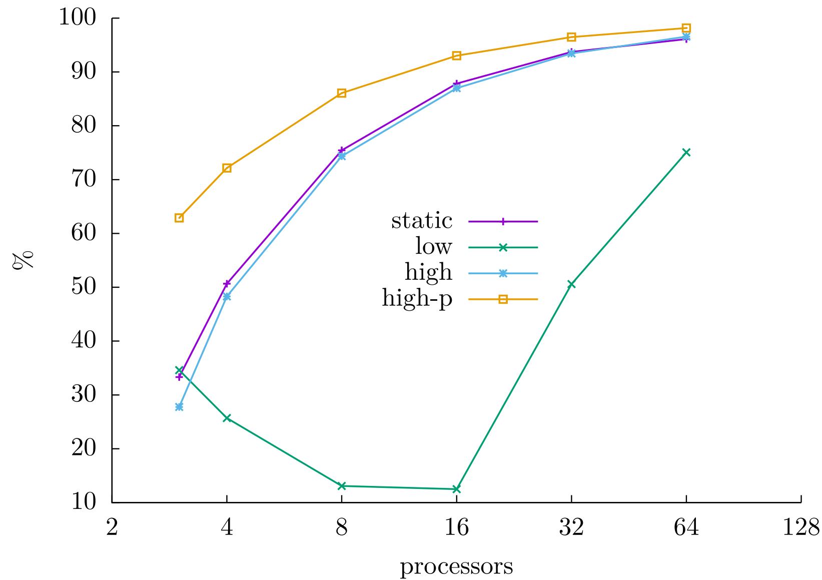

Figura 8: Utilização da rede pelo programa produtor-consumidor (modelo CSP)

6.1.2.2.1 Análise qualitativa As curvas das taxas de utilização de rede são análogas às curvas das taxas de utilização do barramento. Contudo, enquanto 8 processadores saturam o barramento, são necessários de 32 a 64 processos para saturar a rede.

Espaço em branco proposital.

6.1.2.3 Tempo em espera para completar a comunicação A Figura 9 apresenta a somatória dos ciclos que cada processo espera para completar a comunicação. O eixo horizontal indica a quantidade de processadores no sistema e o eixo vertical a quantidade de ciclos em espera. Os diagramas de caixa indicam a variação dos dados observados. A altura das caixas indica que há grande variação entre os tempos de espera dos processos, e que o tempo máximo de espera é limitado superiormente por um valor pouco acima dos 1×10^6 ciclos.

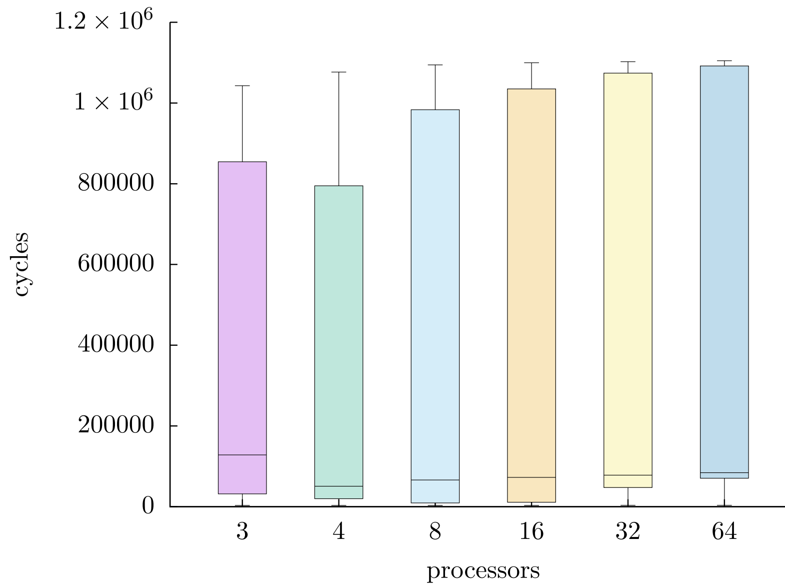

Figura 9: Tempo que um sistema aguarda na comunicação (modelo CSP)

6.1.2.3.1 Tempo de espera produtor A Figura 10 apresenta o tempo em ciclos que cada processo produtor espera para completar a comunicação. A figura da esquerda apresenta um diagrama de caixas análogo ao apresentado pela Figura 9. No intervalo de 3 a 16 processadores, o tempo em espera pela comunicação decresce rapidamente e a variação dos valores observados é inversamente proporcional a quantidade de processadores no sistema. No intervalo de 16 a 64 processadores o tempo de espera cresce lentamente. Na figura da direita é apresentado o gráfico que indica a média do tempo em espera por cada comando de saída que o produtor executa para o processo enfileirador, que segue a mesma tendência do gráfico da esquerda.

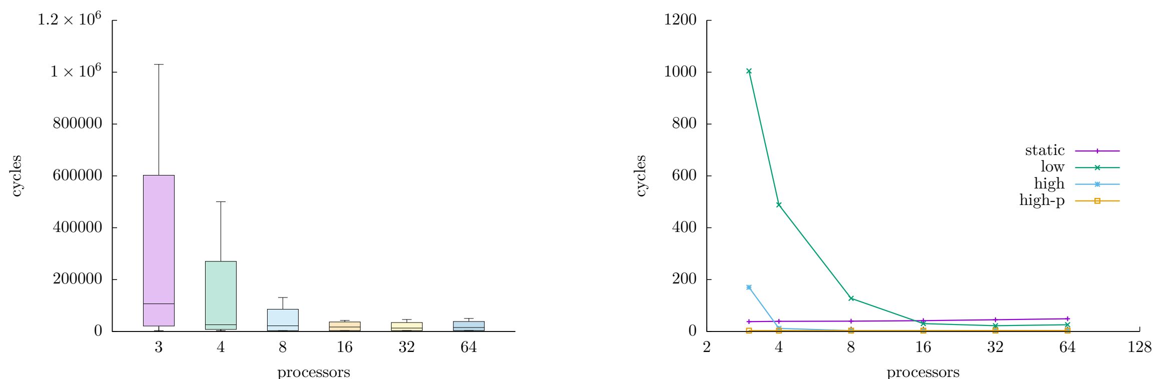

Figura 10: Tempo que o processo produtor aguarda na comunicação (modelo CSP)

6.1.2.3.2 Tempo de espera processo enfileirador

A Figura 11 apresenta o número de ciclos que cada processo enfileirador espera para completar a comunicação. É possível observar que, a partir de 8 processadores, os diagramas de caixa são muito similares e que o limite superior é de aproximadamente 1×10^6 ciclos. A hipótese para explicar esse limite é que a execução do processo enfileirador é limitada pela comunicação mais lenta. Para confirmar essa hipótese, a Figura 12 apresenta 3 gráficos: (i) no canto superior esquerdo o gráfico apresenta a média do tempo em espera do dado vindo do processo produtor – esse gráfico mostra a comunicação que mais demora para completar; (ii) no canto superior direito o gráfico apresenta a média do tempo em espera de cada comando de seleção de alternativas; e (iii) no canto inferior o gráfico apresenta a média do tempo de cada comando de saída pelo consumidor.

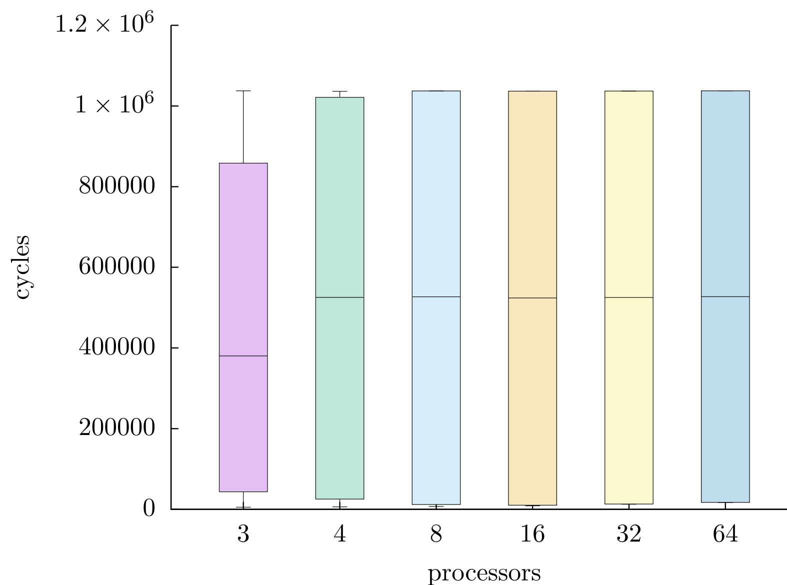

Figura 11: Tempo que um sistema aguarda na comunicação (modelo CSP)

Espaço em branco proposital.

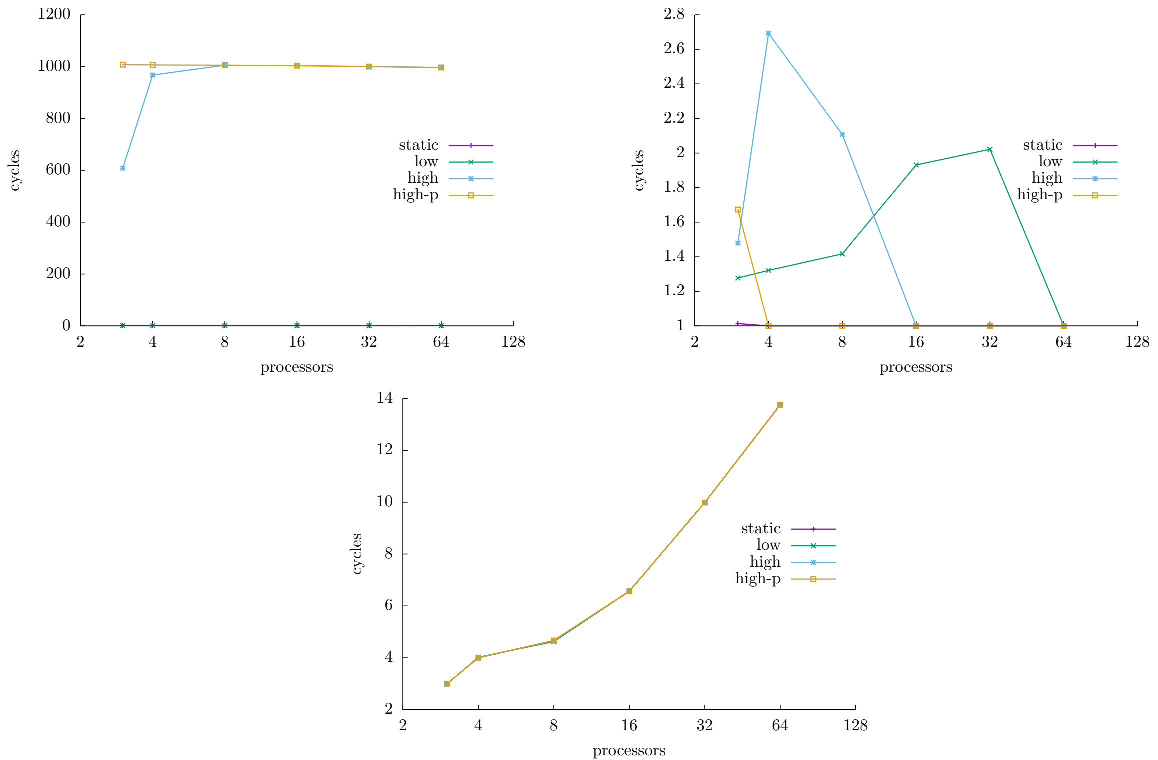

Figura 12: Média do tempo em espera da comunicação por comando CSP (processo enfileirador)

6.1.2.3.3 Tempo de espera consumidor A Figura 13 apresenta o tempo em ciclos que em média cada processo consumidor espera para completar a comunicação. Os diagramas de caixa também são limitados superiormente por um valor pouco acima de 1×10^6 ciclos. A altura das caixas cresce no intervalo de 3 a 16 processadores, e todos as caixas do diagrama são limitadas superiormente por um valor pouco acima de 1×10^6.

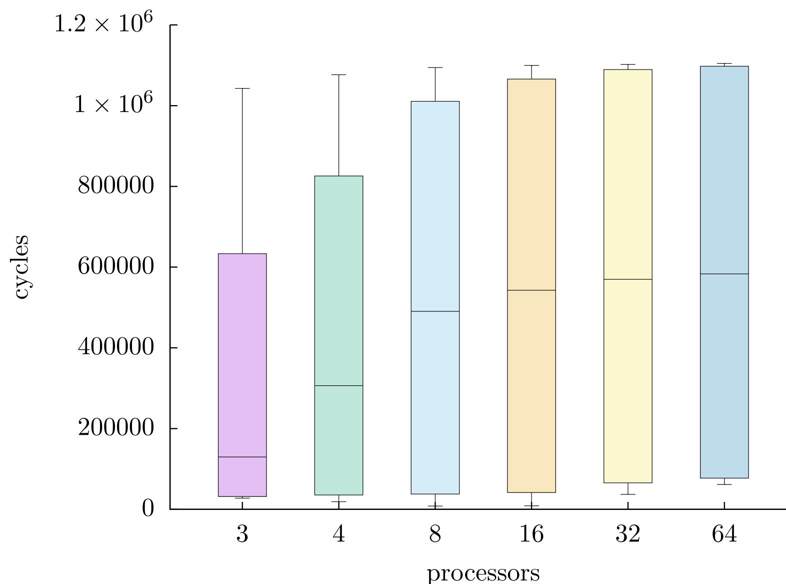

Figura 13: Tempo que um sistema aguarda na comunicação (modelo CSP)

A hipótese que explica esse comportamento é similar a apresentada dois parágrafos acima: o tempo de espera é limitado pela comunicação mais lenta. A Figura 14 apresenta dois gráficos: (i) o gráfico da esquerda mostra a média do tempo em espera da requisição para o processo

enfileirador – esse gráfico indica a comunicação que mais demora para completar; e (ii) o gráfico da direita mostra a média do tempo em espera para receber os dados do processo enfileirador.

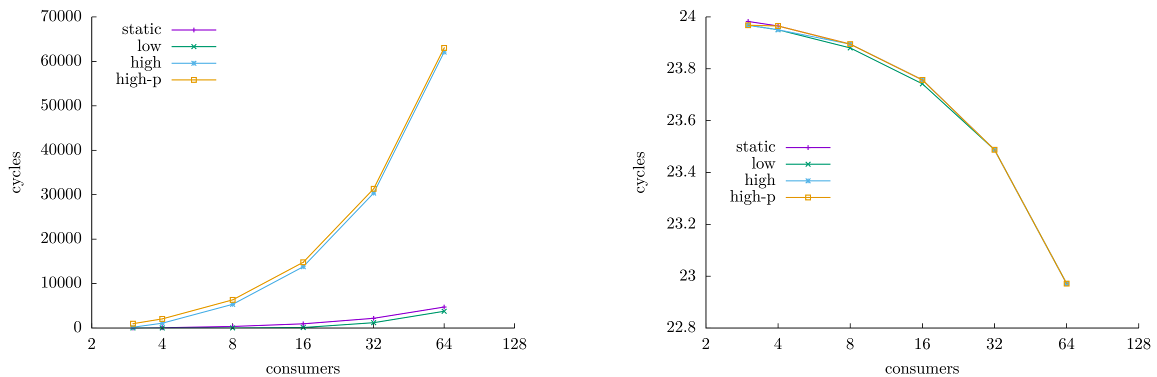

Figura 14: Média do tempo em espera na comunicação por comando CSP (consumidor)

6.1.2.3.4 Análise qualitativa O gráfico da Figura 9 indica que algum elemento da solução faz com que os tempos de espera para completar a comunicação não variem com o número de processadores do sistema. A explicação para esse comportamento é análoga àquela para o comportamento do tempo de execução. Ao comparar os tempos de execução do programa com os tempos em espera, observa-se que o programa dispende mais tempo na sincronização do que no trabalho de produção e consumo.

6.2 Banquete dos filósofos

Para o banquete dos filósofos, são 4 perfis de simulação utilizados: (i) *fixed ideas* indica que os filósofos pensam 10 ideias, o tempo da ação de pensar varia de 25 a 75 ciclos e o tempo comendo varia de 2 a 16 ciclos; (ii) *thinkers* indica que os filósofos pensam entre 2 a 16 ideias, o tempo da ação de pensar varia de 25 a 75 ciclos e o tempo comendo varia de 2 a 16 ciclos; (iii) *glutons* indica que os filósofos pensam entre 2 a 16 ideias, o tempo da ação de pensar varia de 2 a 16 ciclos e o tempo comendo varia de 25 a 75 ciclos; e (iv) *hungry thinkers* indica que os filósofos pensam entre 2 e 16 ideias, o tempo da pensar e comer varia de 25 a 75 ciclos.

6.2.1 Modelo base

A análise do comportamento do sistema no modelo base parte da hipótese do espalhamento das instruções de acesso à memória no tempo, como discutido na Seção 6.1.1.

Espaço em branco proposital.

6.2.1.1 Tempo de execução A Figura 15 apresenta o tempo de execução do programa, no qual o eixo horizontal indica a quantidade de processadores do sistema e o eixo vertical indica o número de ciclos para finalizar a execução. O perfil *thinkers* apresenta o menor tempo de execução devido a variação da quantidade de ideias de cada filósofo.

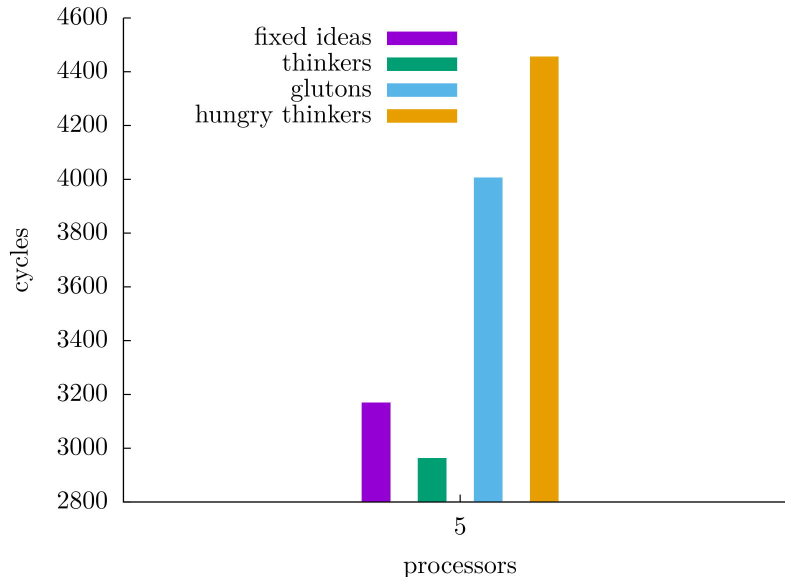

Figura 15: Tempo da execução do programa produtor-consumidor (modelo base)

6.2.1.1.1 Análise qualitativa O tempo de execução do perfil *thinkers* é o mínimo observado, e é importante notar que os tempos para pensar (entre 25 e 75 ciclos) e comer (entre 2 e 16 ciclos) são os mesmos do perfil *fixed ideas*. A diferença entre os perfis é a quantidade de ideias que cada filósofo pensa: 10 ideias no perfil *fixed ideas* e entre 2 e 16 ideias para o perfil *thinkers*. Isso indica que a média da quantidade de ideias entre os filósofos é menor que 10 no perfil *thinkers*, assim como em todos os demais perfis em que a quantidade de ideias é variável, uma vez que esse valor é definido por um gerador de números pseudoaleatórios cujos estados iniciais são os mesmos para todos os perfis.

Para o perfil *glutons* o tempo de execução está relacionado ao tempo dispendido para comer (entre 25 e 75 ciclos) uma vez que o tempo para pensar fica entre 2 e 16 ciclos. Primeiramente o filósofo pensa um tempo muito curto para ir logo comer, e para comer é necessário que o filósofo acesse a região crítica ficando nela por mais tempo comendo.

Para o perfil *hungry thinkers*, os tempos de pensar e comer são os mesmos, entre 25 e 75 ciclos, o que faz com que esse seja o perfil que mais demora para completar sua execução.

6.2.1.2 Utilização do barramento A Figura 3 apresenta a taxa de utilização do barramento. O eixo horizontal indica a quantidade de processadores e o eixo vertical indica a utilização do barramento em pontos percentuais.

6.2.1.2.1 Análise qualitativa O perfil *fixed ideas* apresenta a maior taxa de utilização do barramento pois a média de quantidade de ideias dos filósofos é maior que nos demais perfis, como discutido acima.

O perfil *glutons* apresenta a maior taxa de utilização do barramento entre os perfis que variam o número de ideias, pelos mesmos motivos discutidos na seção de tempo de execução. A competição pela região crítica é algo parecido com a brincadeira de dança das cadeiras: um filósofo pensa as ideias o mais rápido possível para sentar-se logo à mesa – quando um filósofo

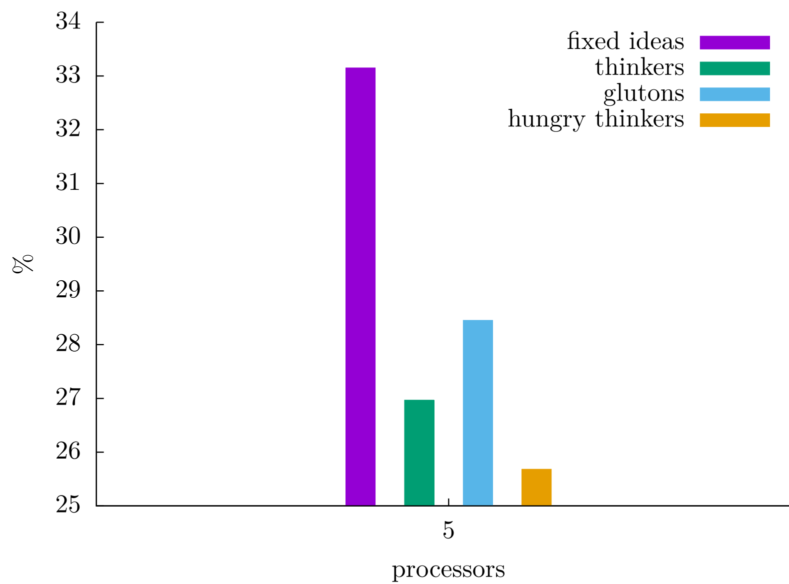

Figura 16: Utilização do barramento pelo programa produtor-consumidor (modelo base)

tenta acessar a região crítica, há uma grande probabilidade de que algum outro filósofo também esteja a tentar o acesso.

6.2.1.3 Falhas de acesso ao barramento A Figura 17 apresenta a taxa de falhas ao acessar o barramento. O eixo horizontal indica a quantidade de processadores e o eixo vertical indica a taxa de falhas.

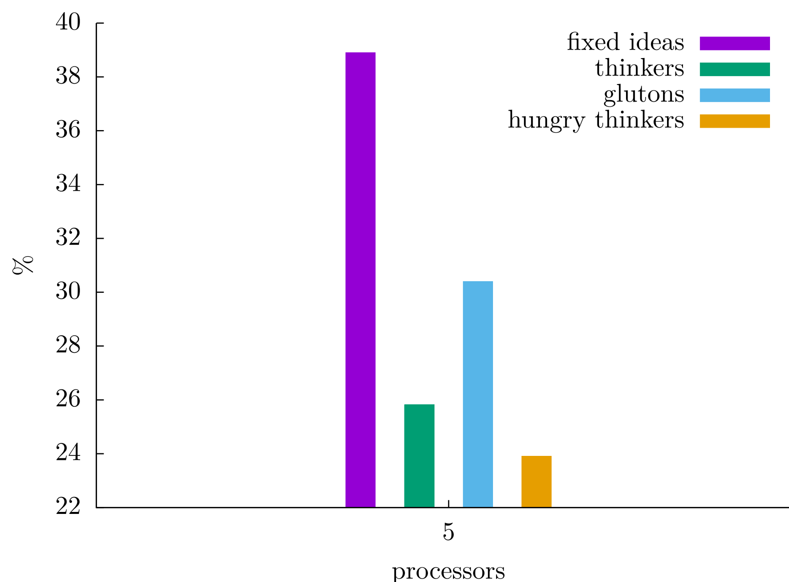

Figura 17: Taxa de falhas ao acessar o barramento do produtor-consumidor (modelo base)

6.2.1.4 Tempo de espera para acessar o barramento A Tabela 4 apresenta número de ciclos que o sistema aguarda para obter o semáforo, que é a soma do tempo que um processador dispende para operação `P()` durante sua execução. Desse ponto em diante, a condição que descreve todos os processos de todos os perfis é chamada de *global*. Observa-se que o mínimo de ciclos em espera global é 181 ciclos no processador 2 no perfil *hungry thinkers*, e o máximo

global é 1764 no processador 3 do perfil *glutons*. A mediana dos valores observados é 712,5.

Tabela 4: Tempo que um sistema com *P* processadores aguarda para acessar o semáforo (modelo base)

	P				
perfil	0	1	2	3	4
fixed ideas	544	750	607	527	442
thinkers	403	395	306	415	397
glutons	1428	1159	515	**1764**	1445
hungry thinkers	875	654	**181**	1130	484

6.2.1.4.1 Análise qualitativa Todos os processadores no perfil *thinkers* apresentam tempo em espera abaixo da mediana (712.5 ciclos) devido ao discutido na Seção 6.2.1.1.1.

Para o perfil *glutons* todos os processadores apresentam tempos de espera maiores que a mediana, com exceção do processador 2. O tempo em espera do processador 2 é menos da metade dos demais processadores do sistema, indicando que as condições de execução favorecem o processador 2. A hipótese que explica esse favorecimento é a de que o filósofo que executa no processador 2 é o que pensa menos, sempre chegando antes à mesa antes dos demais.

6.2.1.5 Falhas das operações *read-modify-write* A Tabela 5 apresenta a quantidade de operações *read-modify-write* que falham em cada processador do sistema. Observa-se que o mínimo de falhas global é 6 falhas para processador 2 do perfil *hungry thinkers* e o máximo global é 43 para o processador 3 do perfil *fixed ideas*. A mediana dos valores observados é 30,5.

Tabela 5: Número de operações *read-modify-write* que falham dado um sistema com *P* processadores (modelo base)

	P				
perfil	0	1	2	3	4
fixed ideas	18	30	28	**43**	36
thinkers	14	21	9	39	24
glutons	31	37	11	26	39
hungry thinkers	21	34	**6**	26	42

6.2.1.5.1 Falhas das operações *read-modify-write* O número de falhas observadas é pequeno quando comparado com os demais experimentos pois o banquete dos filósofos executa menos instruções, já que foi escolhido um número pequeno de ideias que cada filósofo pensa.

6.2.2 Modelo CSP

Desse ponto em diante, o processo que executa no processador 2 é o chamado de garçom, os processos que executam nos processadores `0`, `1`, `3`, `4` e `5` são chamados filósofos e os processos que executam nos processadores `6`, `7`, `9`, `10` e `11` são os chamados garfos.

A análise do comportamento do sistema no modelo base parte da hipótese que a implementação escolhida para o banquete dos filósofos no modelo CSP, que imita a implementação do modelo base, é o fator responsável pelo comportamento observado.

6.2.2.1 Tempo de execução A organização do gráfico da Figura 18 é análoga a Figura 15, apresentando o tempo de execução em ciclos. O gráfico indica que todos os tempos de execução são os mesmos para todos os perfis, e isso tem está relacionado com a maneira que o problema foi implementado na versão CSP. Mais detalhes são dados na Seção 6.2.2.

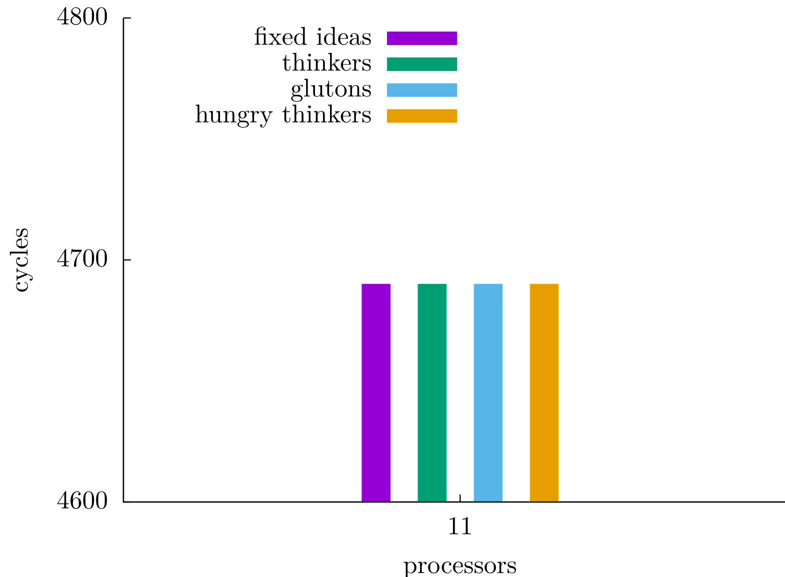

Figura 18: Tempo de execução do banquete dos filósofos (modelo CSP)

A Tabela 6 apresenta os tempos de execução de cada processador utilizado no sistema. O garçom apresenta o maior tempo de execução, devido as atribuições do processo, que organiza o acesso à mesa. O tempo médio entre os filósofos é 4705 ciclos e o tempo médio entre os garfos é 4647 ciclos.

Tabela 6: Tempo de execução dos processadores no banquete dos filósofos (modelo CSP)

	P										
perfil	0	1	2	3	4	5	6	7	9	10	11
fixed ideas	4613	4639	4830	4663	4792	4818	4781	4456	4513	4716	4769
thinkers	4613	4639	4830	4663	4792	4818	4781	4456	4513	4716	4769
glutons	4613	4639	4830	4663	4792	4818	4781	4456	4513	4716	4769
hungry thinkers	4613	4639	4830	4663	4792	4818	4781	4456	4513	4716	4769

6.2.2.2 Utilização da rede A Figura 19 apresenta a taxa de utilização da rede. O eixo horizontal indica a quantidade de processadores do sistema e o eixo vertical indica a taxa de utilização da rede em pontos percentuais. De forma similar ao gráfico da Figura 18, todos os perfis apresentam a mesma taxa de utilização, também devido a forma que o programa foi implementado. Mais detalhes são dados na Seção 6.2.2.

Espaço em branco proposital.

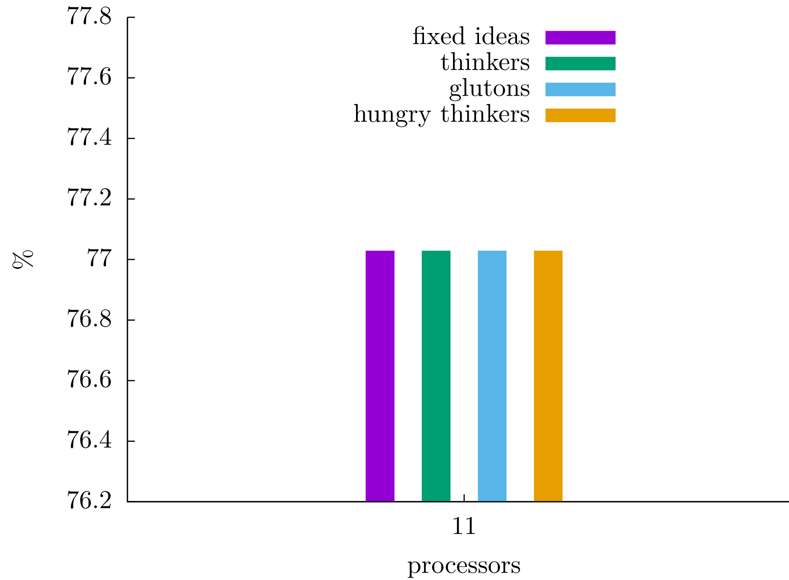

Figura 19: Utilização da rede pelo programa produtor-consumidor (modelo CSP)

6.2.2.3 Tempo em espera para completar a comunicação A Tabela 7 apresenta o tempo em espera para completar a comunicação. Análogo ao apresentado na execução do banquete dos filósofos no modelo CSP, todos os valores observados são idênticos. O garçom apresenta o menor tempo de espera, indicando que a administração dos lugares da mesa leva mais tempo que atender as requisições dos filósofos que querem se sentar. O tempo médio de espera dos filósofos é 4103 ciclos e o tempo médio de espera dos garfos é 4057 ciclos.

Tabela 7: Tempo que um sistema aguarda na comunicação (modelo CSP)

	processador										
perfil	0	1	2	3	4	5	6	7	9	10	11
fixed ideas	4024	4043	123	4061	4185	4204	4206	3871	3922	4120	4166
thinkers	4024	4043	123	4061	4185	4204	4206	3871	3922	4120	4166
glutons	4024	4043	123	4061	4185	4204	4206	3871	3922	4120	4166
hungry thinkers	4024	4043	123	4061	4185	4204	4206	3871	3922	4120	4166

6.2.2.4 Análise qualitativa Como comentado na Seção 5.3.3, uma característica da implementação utilizada no modelo base (Programa 31) é que as operações nos semáforos refletem diretamente as ações dos filósofos: `P()` representa sentar-se a mesa e levantar o garfo, `V()` representa devolver o garfo e levantar-se da mesa. A tradução desse programa para o modelo CSP envolve transformar cada semáforo (5 garfos e um garçom) em um processo. As operações de comunicação do modelo CSP não refletem diretamente as ações dos filósofos, então é necessário que os processos do tipo garfo e garçom implementem a semântica necessária, que para o modelo CSP mostrou-se ineficiente.

6.2.2.4.1 Garçom Seja $comm_p$ a taxa em pontos percentuais do tempo em espera pela comunicação em relação ao tempo total de execução de um processo p. A taxa $comm_{\text{foot}}$ é:

$$comm_{\text{foot}} = 123 \div 4830 \tag{19}$$

Tem-se então que $comm_{\text{foot}} \approx 2.5\%$, logo 97.5% do tempo de execução o garçom está executando a lógica necessária para que a implementação do problema no modelo CSP seja equiparável

à do modelo base. O garçom tem o comportamento de um semáforo contador inicializado com valor 4, implementado pelo Programa 34. Nota-se que o código utilizado no modelo CSP é mais prolixo que no modelo base, já que precisa explicitar os estados do processo e utiliza uma lista que indica quais processos têm acesso à região crítica, cujo o custo de processamento é maior do que para o contador do semáforo.

6.2.2.4.2 Filósofos Seja $comm_{\text{philos}}$ a taxa em pontos percentuais da média do tempo de espera pela comunicação em relação à media do tempo de execução dos filósofos. Então a taxa $comm_{\text{philos}}$:

$$comm_{\text{philos}} = 4103.4 \div 4705 \tag{20}$$

Tem-se então que $comm_{\text{philos}} \approx 87\%$, logo 13% do tempo de execução do filósofo é para comer e pensar. O filósofo deve se comunicar com o garçom duas vezes, uma para indicar que vai se sentar à mesa para comer e outro para indicar que levantou-se da mesa para pensar. Dada a descrição do tempo de execução do comando de saída dada na Seção 3.4.7 (Equação 13), sejam $T_{\text{p!q}}$ o tempo de execução do comando de saída entre dois processos p e q, y o tempo de espera no processador destino para a execução do comando de entrada ou seleção de alternativas correspondente e m o tempo esperando a transmissão da mensagem na rede. Sejam também *phil* um filósofo qualquer e $foot$ o garçom. O tempo para realizar as ações de sentar-se a mesa e levantar-se da mesa é:

$$T_{\text{phil!foot}} = 2 \times m + y \tag{21}$$

Como discutido em 6.2.2.4.1, o garçom dispende muito tempo para poder atender cada requisição dos filósofos – y é um valor demasiadamente grande para cada comando de saída do filósofo para o garçom. Observando a saída de depuração do simulador, cada ação de sentar e levantar-se da mesa dispende em média mais de 200 ciclos em espera.

O tempo em espera pelos garfos depende de se o garfo já está levantado pelo vizinho ou não. Levantar e devolver os garfos à mesa é implementado como um comando de saída para o processo que representa o garfo.

Sejam *phil* e *nbr* dois filósofos, e *phil* é o filósofo que deseja utilizar os garfos e *nbr* algum dos seus vizinhos, $fork$ o garfo que *phil* e *nbr* compartilham e E_{nbr} o tempo restante para que *nbr* termine de comer e devolva $fork$ à mesa. Caso os garfos não tenham sidos levantados pelos vizinhos de *phil* (melhor caso), o tempo de espera de *phil* para levantar o garfo (T_{pickup}) é:

$$T_{\text{pickup}} = T_{\text{phil!fork}} \tag{22}$$

Caso o garfo tenha sido levantado pelo vizinho, o tempo para *phil* levantar o garfo é:

$$T_{\text{pickup}} = E_{\text{nbr}} + T_{\text{nbr!fork}} + T_{\text{phil!fork}} \tag{23}$$

Para devolver o garfo à mesa, o tempo é aquele da equação 22, uma vez que após comer basta devolver o garfo, tem nenhum outro estado do garfo que não o permita voltar à mesa.

6.2.2.4.3 Garfo O garfo no modelo base é um semáforo contador inicializado em 1, cujo a tradução semântica para o modelo CSP é mais simples que a implementada para o garçom. O garfo no modelo CSP apenas aguarda que algum dos filósofos o levante e o devolva a mesa, e isso é implementado como comandos de entrada para os filósofos. Dada a definição do tempo de execução do comando de entrada e seleção de alternativas na Equação 12, seja $T_{p\,?\,q}$ o tempo de execução de um comando de entrada ou seleção de alternativas entre dois processos p e q. Sejam também $fork$ um garfo e *phil* um filósofo que utiliza o garfo. O tempo que o processo garfo aguarda entre cada utilização por um filósofo (T_{forkutil}) é:

$$T_{\text{forkutil}} = 2 \times T_{fork\,?\,phil} \tag{24}$$

Finalmente, assim como apresentado na Equação 18, o tempo de execução é limitado pelo processo mais lento, que nesse caso é o garçom. Na implementação no modelo CSP há uma cadeia de dependências: o garfo espera o filósofo, o filósofo espera o garçom e o garçom demora para atender os filósofos.

Nota-se também que o garçom do modelo CSP faz o papel do barramento do modelo base, mas diferente do barramento, uma estrutura em hardware, o garçom faz o mesmo trabalho em software, o que faz com que o mesmo trabalho em hardware seja executado em mais ciclos.

6.2.2.4.4 Distribuição dos processos na rede Outra característica que foi cogitada como razão do comportamento do banquete dos filósofos no modelo CSP foi o mapeamento dos processos na rede. Um outro mapeamento foi testado na mesma topologia de 2 linhas com 6 processadores: o processo garçom foi executado no processador `0`; os filósofos executados nos processadores `2`, `4`, `7`, `9` e `11`; e os garfos executam nos processos `1`, `3`, `5`, `8` e `10`. Nesse novo mapeamento cada filósofo está entre dois garfos, representando fielmente a descrição da mesa circular do problema. Ao executar o programa com essa nova condição, os resultados encontrados são muito similares aos descritos na Seção 6.2.2.

7 Comparação geral entre os modelos

Com a análise dos resultados da simulação do produtor-consumidor e banquete dos filósofos nos modelos base e CSP, descobriu-se que um equívoco foi cometido na implementação dos problemas no modelo CSP, que é imitar o gargalo do modelo base.

A razão para o modelo CSP imitar o modelo base parte de um princípio de metodologia: para comparar dois tipos de sistemas é necessário que hajam características relevantes e comuns a esses sistemas, neste caso, a habilidade de executar corretamente instâncias de dois problemas de sincronização. Imitar o texto dos programas do modelo base no modelo CSP é uma tentativa ingenua de explicitar a característica de comparação. Por isso, foram impostas restrições adicionais ao modelo CSP como um único processo enfileirador no produtor-consumidor e o processo garçom no banquete dos filósofos. Tal escolha impactou negativamente no desempenho do modelo CSP.

O modelo base é o mais pervasivo entre os modelos de execução de sistemas multiprocessados: é o modelo ensinado no curso de ciência da computação da UFPR e é o utilizado nos computadores pessoais que estão nas mãos do público geral. As análises dessa seção fazem uma comparação geral entre os modelos e aponta algumas soluções para esse problema.

7.1 Produtor-consumidor

Para o problema produtor-consumidor, o modelo CSP apresenta vantagens sobre o modelo base: o texto do programa no modelo CSP é mais simples e fácil de entender que no modelo base, e; tempo de execução é menor e cresce lentamente com o número de processadores.

Partindo do pressuposto de que é possível dividir a produção dos dados, a limitação do processo enfileirador pode ser contornada de duas formas: (i) removendo a limitação de apenas um produtor no sistema, pode-se aumentar a quantidade dos pares de processos produtor e enfileiradores, *e.g* um a cada linha da rede, distribuindo o gargalo pela rede; e (ii) remover todos os processos enfileiradores, de forma que cada produtor envia os dados apenas aos consumidores que são seus vizinhos, e assim a computação ocorre de forma totalmente distribuída.

7.2 Banquete dos filósofos

Para o problema do banquete dos filósofos, o modelo base apresenta vantagens sobre o modelo CSP. No modelo CSP o código é maior, mais complicado, gasta mais recursos computacionais e executa em um tempo maior que o modelo base.

Para reduzir a quantidade de processadores utilizados no modelo CSP, é possível alterar a forma de representação dos garfos. Os garfos podem ser representados como um vetor de inteiros (`fork[]`) no garçom. O elemento `fork[i]` representa o garfo do filósofo `i`, sendo `1` quando não está em uso e `0` caso contrário. Se a limitação de quais garfos o filósofo pode utilizar for removida (o filósofo pode usar qualquer garfo disponível), é possível que o tempo de execução no modelo CSP seja reduzido, já que na formulação original do problema, quando um filósofo está comendo os filósofos vizinhos estão bloqueados.

Remover o garçom exige que outra solução seja implementada. Por questões de prazo soluções alternativas não foram testadas.

7.3 Comparação geral

Enfim, pode-se dizer o que o modelo CSP apresenta vantagens sobre o modelo base quando o programa que está executando não depende de um recurso centralizado. Esta é uma forma não ortodoxa de se pensar, construir e programar sistemas multiprocessados.

É opinião do autor que o utilizar o modelo CSP é vantajoso e benéfico, pois o mecanismo de sincronização é mais simples e direto que semáforos, o paralelismo obtido é fruto direto do modelo computacional em que a concorrência entre os componentes do problema é explicitada. Uma ferramenta adicional para lidar com sistemas multiprocessados pode ser útil para quem lida com esse tipo de sistema.

O desenvolvimento do simulador e análise dos problemas foram tarefas mais árduas do que antecipado. O número de linhas de código do simulador não é um indicador preciso para o esforço dispendido na implementação do simulador, nem nos testes e medições do sistema simulado.

Trabalhos futuros podem cobrir outros tópicos como: (i) reimplementar as soluções no modelo CSP e incluir mais problemas de sincronização; (ii) implementar o subsistema de interrupções e a rede de forma que mais de um processo execute por processador; e (iii) automatizar a nomeação dos processos nos programas em C, de forma que o executável gerado seja carregado e executado pelo sistema de maneira eficiente.

8 Conclusão

Este trabalho compara dois modelos de multiprocessamento: modelo base com barramento e memória compartilhada, e o modelo CSP com memória distibuída. Foi implementado um simulador de sistemas multiprocessados com a arquitetura MIPS, que simula a execução de programas nos modelos. Os programas escolhidos para análise e comparação entre os modelos foram dois problemas clássicos de sincronização: produtor-consumidor e banquete dos filósofos.

Na análise do produtor-consumidor, o modelo CSP apresenta um código menor e mais simples que o do modelo base. Além disso, o tempo de execução no modelo CSP é menor que o do modelo base, e dependendo das condições de execução do problema, adicionar mais processadores ao modelo CSP provê uma melhora de performance significativa.

Na análise do banquete dos filósofos, o modelo base apresenta tamanho de código e tempo de execução menores que o modelo CSP. Ao traduzir a solução do modelo base para o modelo CSP, fez-se necessário a utilização de mais recursos computacionais e portanto foi necessário adicionar o código que controla esses recursos. O tempo de execução no modelo CSP é maior que o pior caso observado para o modelo base devido à solução implementada. Cometeu-se um equívoco ao implementar as soluções do modelo CSP: imitar o texto das implementações do modelo base, o que faz com que o mesmo gargalo do modelo base fosse replicado desnecessariamente no modelo CSP. Soluções para esse problema são discutidas, mas não foram implementadas.

Apesar dos problemas causados pelo equívoco, o autor acredita que o modelo CSP traz vantagens e benefícios frente ao modelo base, e que futuros trabalhos podem elucidar esse ponto.

Referências

[1] P. B. G. Abraham Silberschatz, Greg Gagne. *Operating System Concepts*. Wiley, 10th edition, 2018.

[2] G. Barrett. Occam 3 reference manual. Technical report, Technical report, Inmos Limited, March 1992. Available at: https://www.transputer.net/obooks/occref/oc3refman.pdf, 1992.

[3] D. I. M. de Oliveira. psim: processor simulator. `https://github.com/xDIMOx/psim`. Acessado: 2021-03-21.

[4] E. W. Dijkstra. The structure of the "the"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[5] M.-D. Feng, W.-F. Wong, and C.-K. Yuen. Compiling parallel lisp for a shared memory multiprocessor. Technical report, 1995.

[6] C. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, 1978.

[7] D. A. P. John L. Hennessy. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, 2006.

[8] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.*, pages 558–565, July 1978. 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). Also awarded an ACM SIGOPS Hall of Fame Award in 2007.

[9] C. Whitby-Strevens. The transputer. In *ISCA 1985*, 1985.

A Resultados da simulação produtor-consumidor (versão 2)

Os mesmos perfis apresentados na Seção 6.1 são empregados nos experimentos descritos nessa seção. Como essa versão emprega 2 produtores, o número mínimo de processadores no sistema é 4.

A.1 Modelo base

Como comentado na Seção 5.1.3 a versão 2 do produtor-consumidor não difere da primeira versão do problema dada a análise dos gráficos. A hipótese do espalhamento das instruções de memória na primeira versão do problema (Seção 6.1.1) é igualmente válida para a esta análise, mas outra hipótese, a de que como há mais produtores executando em paralelo, a demanda dos consumidores é suprida mais rapidamente que na primeira versão do produtor-consumidor, é considerada para a análise da segunda versão.

De forma geral, os comentários feitos na Seção 6.1.1 são igualmente válidos para a segunda versão.

A.1.0.1 Tempo de execução A organização do gráfico da Figura 20 é análoga à Figura 2. O tempo de execução é menor na versão 2 do produtor-consumidor e de forma similar ao observado em 6.1.1 o tempo de execução de todos os perfis crescem com a quantidade de processadores no sistema, a exceção é o perfil *low* decresce o tempo de execução no intervalo de 4 a 8 processadores.

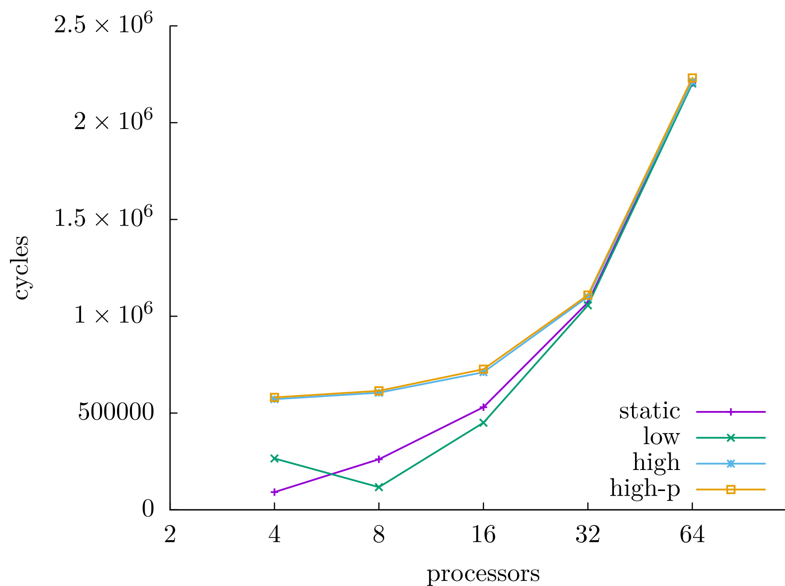

Figura 20: Tempo da execução do programa produtor-consumidor (modelo base)

A.1.0.1.1 Análise qualitativa A diferença notável entre as duas versões do produtor-consumidor é que a segunda versão possui o tempo de execução menor que a primeira. Isso pode ser explicado pela observação feita sobre o(s) produtor(res) suprir(em) a demanda dos consumidores. Com dois produtores, o *buffer* está sendo constantemente preenchido, o que faz com que os produtores sempre tenham acesso a um novo elemento cada vez que acessarem o *buffer* para o consumo. Além disso, como cada produtor produz uma quantidade menor de elementos, o que reduz seu tempo de execução.

A.1.0.2 Utilização do barramento A organização do gráfico da Figura 21 é análoga a Figura 3. Todos os perfis se comportam de forma similar, crescendo assintoticamente em direção aos 100%. A exceção é o *low* decresce taxa de utilização do barramento no intervalo de 4 a 8 processadores.

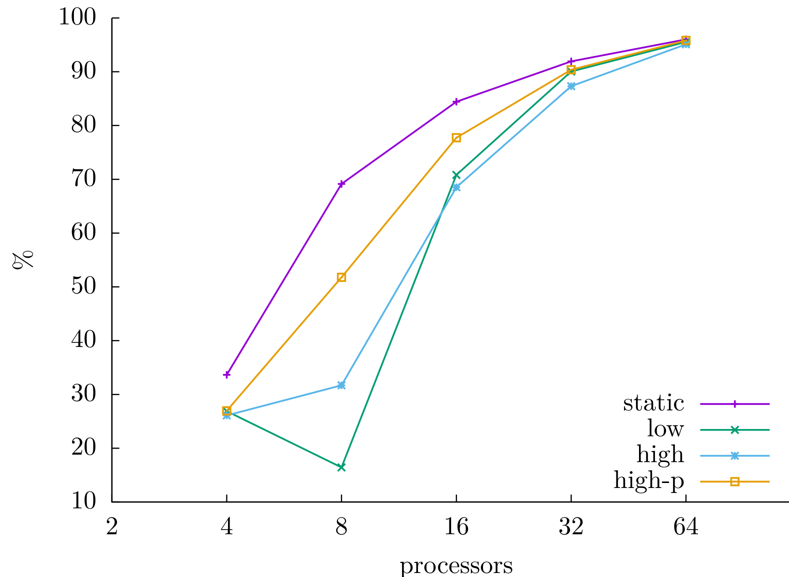

Figura 21: Utilização do barramento pelo programa produtor-consumidor (modelo base)

A.1.0.2.1 Análise qualitativa Analisando a Figura 3 os sistemas com P processadores no intervalo $4 \leq P \leq 64$ e comparando com a Figura 21 no mesmo intervalo, observa-se que o comportamento dos sistemas é similar. A diferença está no perfil *low* que para a segunda versão do produtor-consumidor utiliza mais o barramento para sistemas com $P \geq 16$. Isso ocorre pois há mais um produtor competindo pelo recurso e os tempos de produção (entre 2 e 16 ciclos) acabam não sendo o suficiente para espalhar as instruções de acesso a memória, o contraponto é perfil *high*, os tempos de produção (entre 1 e 1024 ciclos) conseguem espalhar melhor as instruções de acesso a memória, reduzindo o uso do barramento.

Espaço em branco proposital.

A.1.0.3 Falhas de acesso ao barramento A organização do gráfico da Figura 22 é análoga a Figura 4. Todos os perfis se comportam de forma similar, crescendo assintoticamente em direção aos 100%.

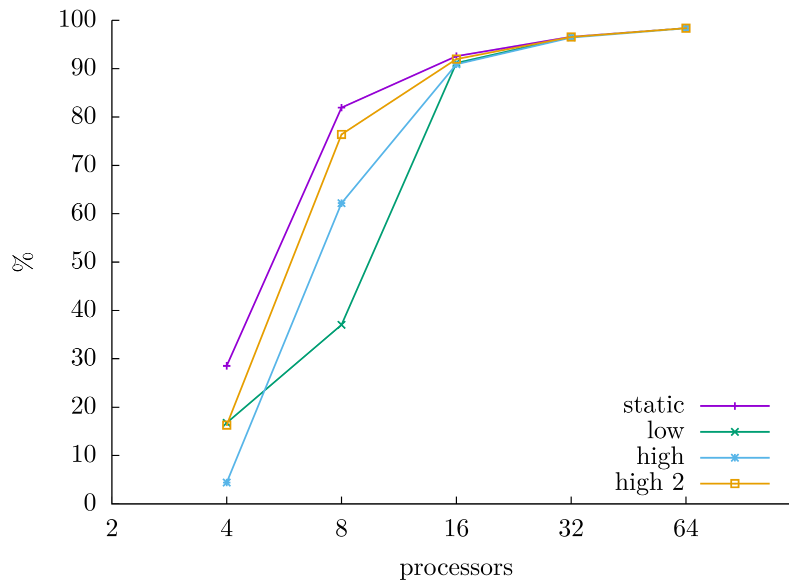

Figura 22: Taxa de falhas ao acessar o barramento do produtor-consumidor (modelo base)

A.1.0.3.1 Análise qualitativa Analisando a Figura 4 no intervalo $4 \leq P \leq 64$ e comparando com a Figura 22 no mesmo intervalo, observa-se que o comportamento dos sistemas é similar.

A.1.0.4 Tempo de espera para acessar o semáforo A organização do gráfico da Figura 23 é análoga a Figura 5. Nota-se que os tempos de execução do programa e os tempos em que os processadores do sistema esperam para acessar a região critica estão na mesma faixa de valores. Na versão 2 do produtor-consumidor há menor variação no tempo em espera para acessar o barramento e maior quantidade de valores discrepantes quando comparado com a primeira versão do problema.

Espaço em branco proposital.

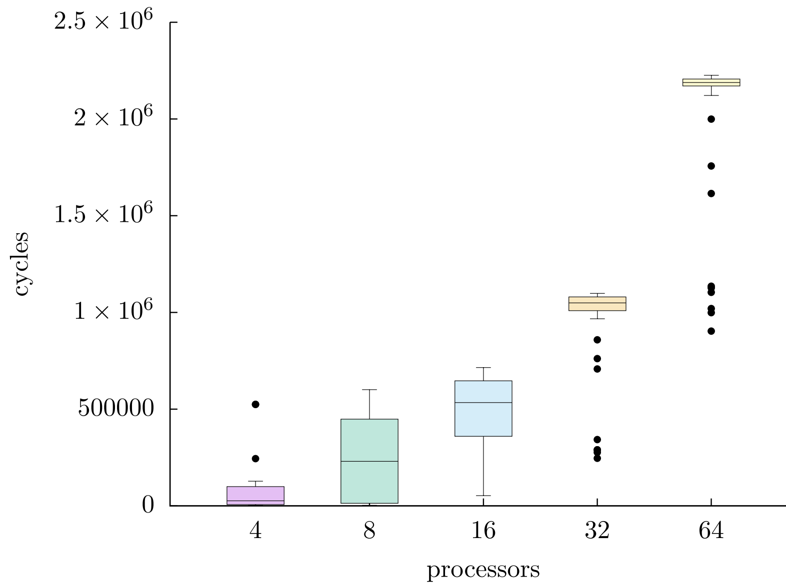

Figura 23: Tempo que um sistema aguarda o acesso aos semáforos (modelo base)

A.1.0.4.1 Análise qualitativa Analisando a Figura 4 no intervalo $4 \leq P \leq 64$ e comparando com a Figura 22 no mesmo intervalo, observa-se que o comportamento dos sistemas é similar. Na versão 2 do produtor-consumidor os blocos do diagrama indicam um tempo menor de espera e uma menor variação entre os tempos esperando para acessar a região crítica. Os valores discrepantes são em maioria tempos menores que os observados, e em sistemas com $P \geq 32$.

A.1.0.5 Falhas das operações *read-modify-write* A organização do gráfico da Figura 24 é análoga a Figura 6. O comportamento dos dados é assintótico, crescendo em direção ao valor 1300 com vários valores discrepantes, a exceção é o número de falhas para 8 processadores, que tem valor máximo de aproximadamente 2000 falhas e nenhum valor discrepante.

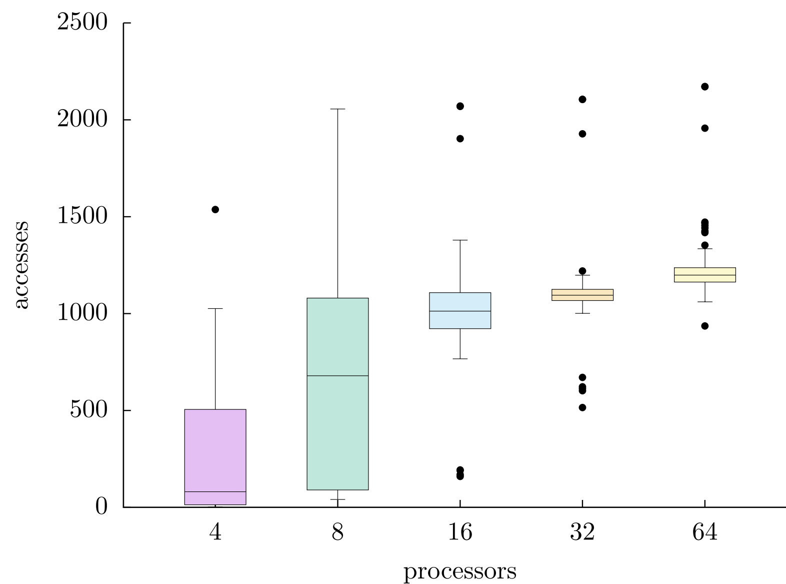

Figura 24: Número de operações *read-modify-write* que falharam (modelo base)

A.1.0.5.1 Análise qualitativa

Analisando a Figura 6 no intervalo $4 \leq P \leq 64$ e comparando com a Figura 24 no mesmo intervalo, observa-se que o comportamento dos sistemas é similar. Observa-se que na versão 2 do produtor-consumidor há mais falhas das operações de *read-modify-write*, devido a maior competição pelo barramento.

A.2 Modelo CSP

As hipóteses para o modelo CSP apresentadas na Seção 6.1.2 do espalhamento das execuções das instruções de comunicação e espalhamento no espaço são igualmente válidas para esta análise. Assim como na primeira versão do produtor-consumidor, o processo enfileirador é vizinho dos produtores.

A.2.0.1 Tempo de execução

A organização do gráfico da Figura 25 é análoga a Figura 7. Na versão 2 do produtor-consumidor os tempos de execução são menores que na primeira versão. Os tempos dos perfis *high* e *high-p* são a metade do tempo visto no gráfico da Figura 7. A hipótese que explica para esse comportamento é: como o tempo de produção desses perfis é alto (entre 1 e 1024 ciclos), a execução dos dois produtores da versão 2 se intercala de forma que o tempo de execução diminua. Para os perfis *static* e *low*, cujo os tempos de produção variam menos, essa vantagem não aparece, e apresentam curvas similares ao observado no gráfico da Figura 7

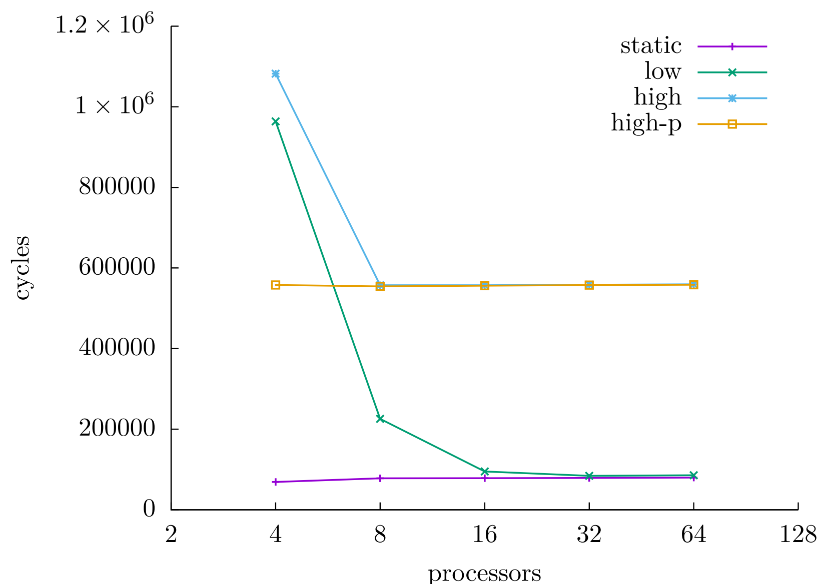

Figura 25: Tempo de execução do programa produtor-consumidor versão 2 (modelo CSP)

Após alcançarem o valor mínimo observado do perfil, as curvas crescem muito lentamente, o que faz parecer que são constantes. A Tabela 8 apresenta os valores dos tempos de execução, de forma a facilitar a visualização do crescimento das curvas.

Tabela 8: Tempo de execução do produtor-consumidor (modelo CSP)

	P				
perfil	4	8	16	32	64
static	6.912e+04	7.815e+04	7.841e+04	7.893e+04	8.000e+04
low	9.637e+05	2.256e+05	9.513e+04	8.439e+04	8.561e+04
high	1.082e+06	5.572e+05	5.572e+05	5.584e+05	5.596e+05
high-p	5.579e+05	5.544e+05	5.562e+05	5.574e+05	5.586e+05

A.2.0.1.1 Análise qualitativa Os comentários feitos na Seção 6.1.2 para essa métrica são igualmente válidos. A principal diferença entre a primeira e segunda versão do produtor-consumidor é que a segunda versão apresenta tempo de execução menor que a primeira.

A.2.0.2 Utilização da rede A organização do gráfico da Figura 26 é análoga a Figura 8. O comportamento geral dos dados é crescer de forma assintótica em direção aos 100%. A exceção é o perfil *low*, que decresce no intervalo de 3 a 16 processadores. No intervalo de 16 a 64 processadores, o perfil *low* tem o mesmo comportamento dos demais.

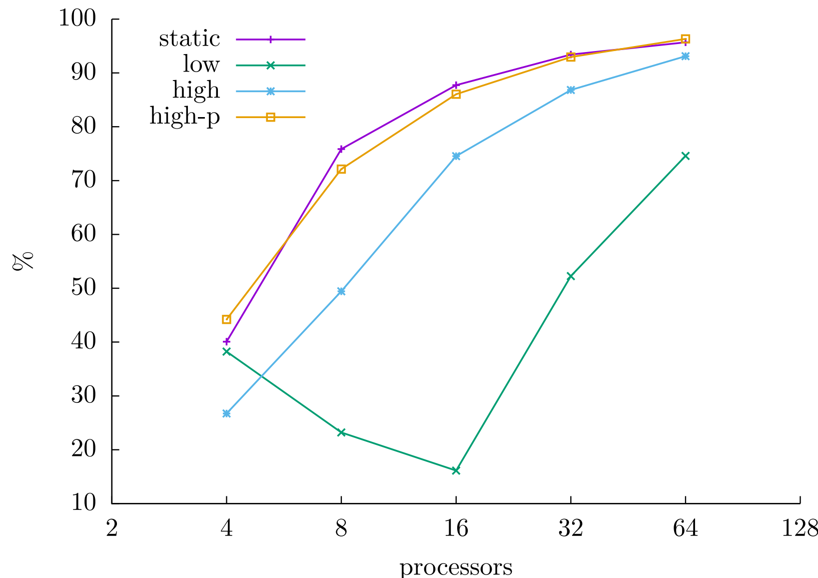

Figura 26: Utilização da rede pelo programa produtor-consumidor (modelo CSP)

A.2.0.2.1 Análise qualitativa Os gráficos das figuras 8 e 26 são análogos, apresentando o mesmo comportamento. Na segunda versão do produtor-consumidor, para sistemas com $P = 4$ processadores, a utilização da rede é menor quando comparada a primeira versão, pois (i) a rede é pequena; (ii) as instruções de comunicação estão espalhadas na dimensão de tempo; e (iii) os processos estão organizados de forma que haja menos competição na rede. Outra característica da segunda versão é que o perfil *low* decresce a utilização da rede no intervalo $P \leq 16$, o que ocorre devido ao motivo (ii) apresentado anteriormente.

A.2.0.3 Tempo em espera para completar a comunicação A organização do gráfico da Figura 27 é análoga a Figura 9. A tendência observável é um valor abaixo dos 600 mil ciclos, exceto para sistemas com 2 processadores, cujo o tempo de comunicação pode ultrapassar os 1×10^6 ciclos.

A.2.0.3.1 Tempo de espera produtor A organização do gráfico da Figura 28 é análoga a Figura 10 apresentando o tempo em ciclos que cada processo produtor espera para completar a comunicação. A figura da esquerda apresenta um diagrama de caixas análogo ao apresentado pela Figura 27. No intervalo de 3 a 16 processadores, o tempo em espera pela comunicação decresce rapidamente e há menor variação dos valores observados quanto maior a quantidade de processadores no sistema. No intervalo de 16 a 64 processadores o tempo de espera cresce lentamente. Na figura da direita é apresentado o gráfico que indica a média do tempo em espera por cada comando de saída que o produtor executa para o processo enfileirador, que segue exatamente a mesma tendência do gráfico da esquerda.

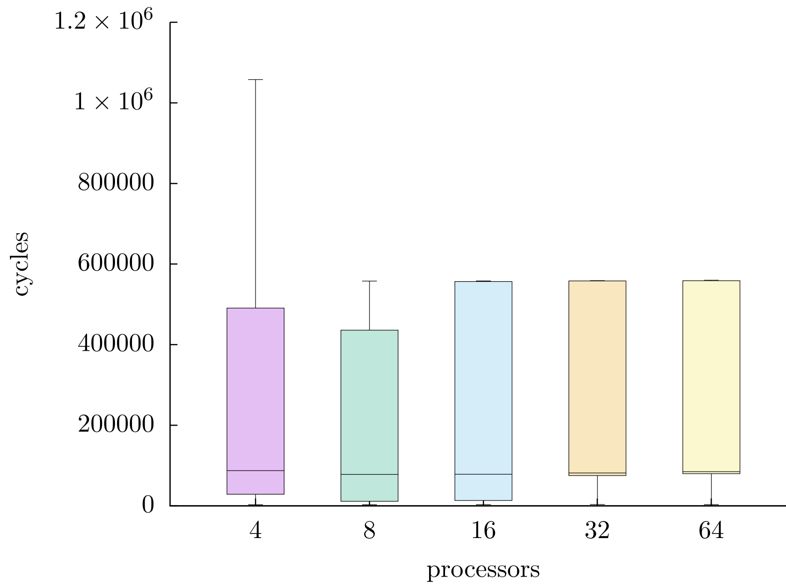

Figura 27: Tempo que um sistema aguarda na comunicação (modelo CSP)

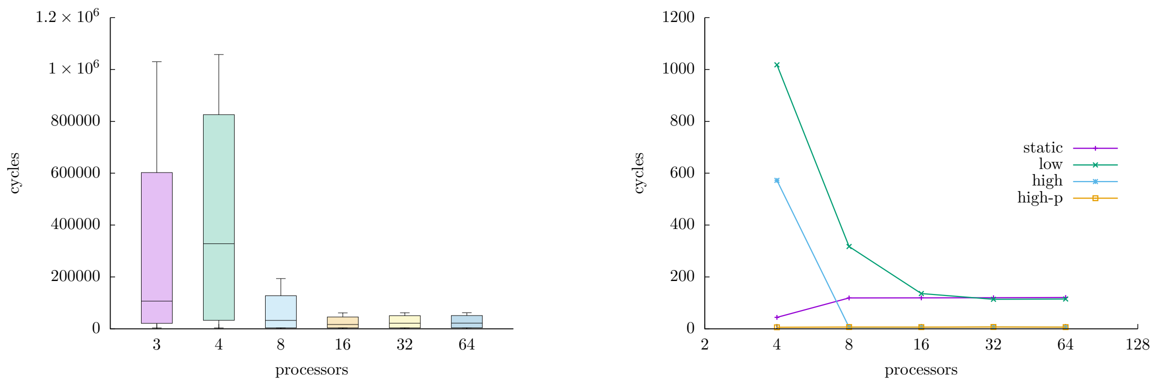

Figura 28: Tempo que o processo produtor aguarda na comunicação (modelo CSP)

A.2.0.3.2 Tempo de espera processo enfileirador A Figura 29 apresenta o tempo em ciclos que cada processo enfileirador espera para completar a comunicação e é análoga a Figura 11 . É possível observar que a partir de 8 processadores os diagramas de caixa são idênticos e que o limite superior de aproximadamente 5×10^5 ciclos está presente. A hipótese para explicar esse limite é a mesma apresentada na Seção 6.1.2. A Figura 30 apresenta 2 gráficos: (i) o gráfico da esquerda apresenta a média do tempo em espera por cada comando de seleção de alternativas; e (ii) o gráfico da direita presenta a média do tempo por cada comando de saída para o consumidor.

A.2.0.3.3 Tempo de espera consumidor A Figura 29 apresenta o tempo em ciclos que em média cada processo consumidor espera para completar a comunicação. Os diagramas de caixa são limitados superiormente por um valor por volta de 5.5×10^5 ciclos. A altura das caixas crescem no intervalo de 4 a 8 processadores, e a hipótese que explica esse comportamento é a mesma apresentada na Seção 6.1.2. A Figura 14 apresenta dois gráficos: (i) o gráfico da esquerda apresenta a média do tempo em espera de cada comando de saída executado para o processo enfileirador, esse gráfico apresenta a comunicação que mais demora para completar; e (ii) o gráfico da direita apresenta a média do tempo em espera para executar cada comando de entrada para o processo enfileirador.

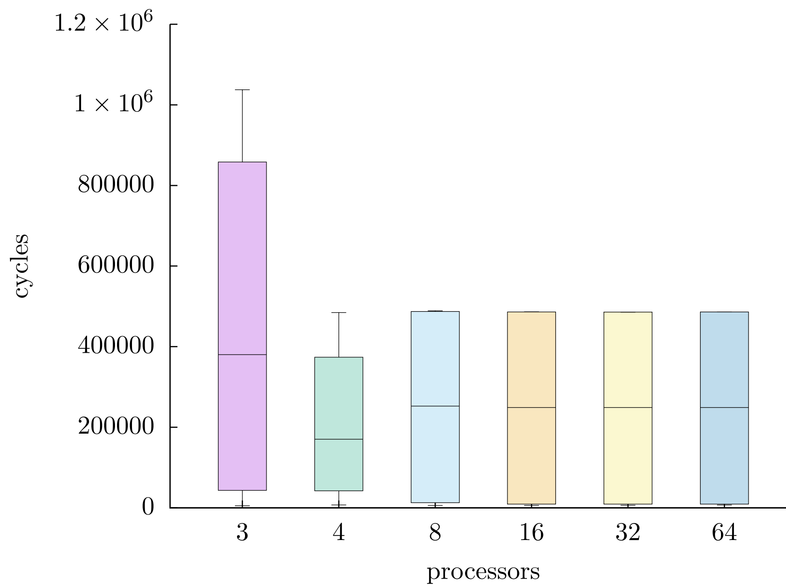

Figura 29: Tempo que um sistema aguarda se comunicando (modelo CSP)

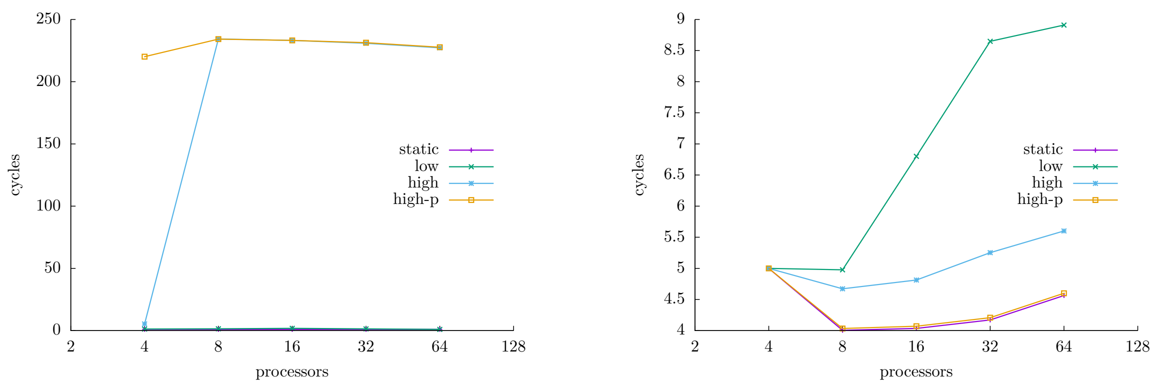

Figura 30: Média do tempo em espera da comunicação por comando CSP (processo enfileirador)

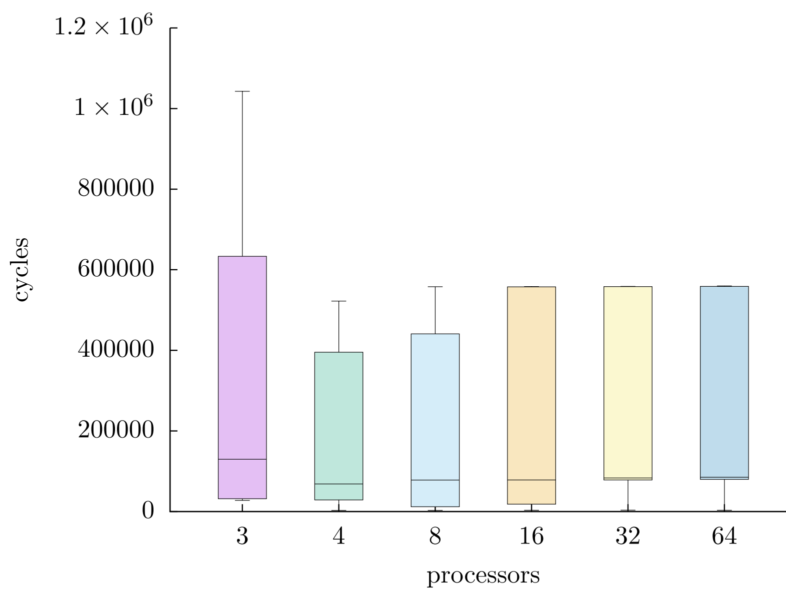

Figura 31: Tempo que um sistema com *P* processadores aguarda se comunicando (modelo CSP)

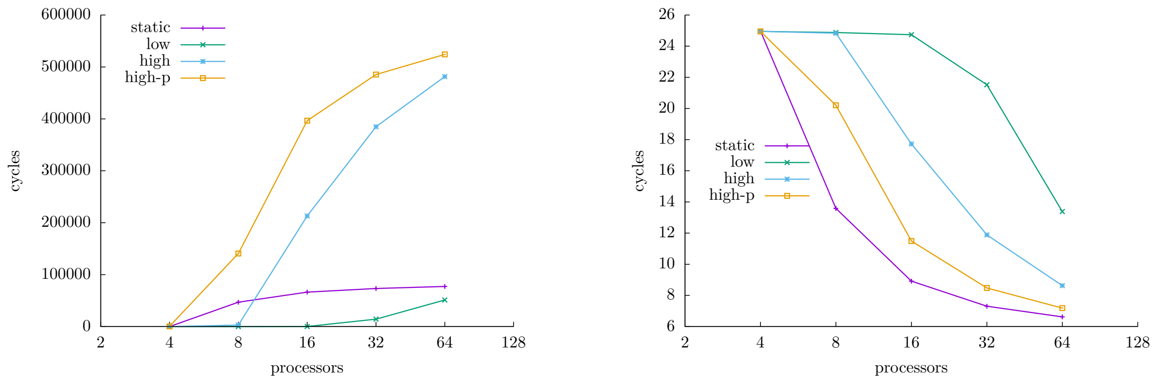

Figura 32: Média do tempo em espera da comunicação por comando CSP (consumidor)

A.2.0.3.4 Análise qualitativa Os gráficos das figuras 9 e 27 são análogos. A segunda versão apresenta tempos de espera menores quando comparados a primeira versão do produtor-consumidor. Sistemas com 4 processadores apresentam tempo em espera maior que os demais, pois os perfis *low* e *high* apresentam tempos de execução acima de 1×10^6 ciclos.