LPII - Programação Concorrente - 2025.2

Prof. Carlos Eduardo Batista

Exercício Prático

Entrega por email: bidu @ ci . ufpb . br
Prazo: até 12h59 de 15/12/2025
O título do e-mail deve conter (substituir): "**[LPII-20252-E001] NOME DO ALUNO – MATRICULA**".
Arquivo de entrega deve anexar todos os códigos fonte em C/C++ dentro de um diretório nomeado "**MATRICULA_LPII-20252-E001**", comprimido em um arquivo ZIP ("**MATRICULA_LPII-20252-E001.zip**").
O NÃO ATENDIMENTO ÀS INSTRUÇÕES IMPLICARÁ NA NÃO CORREÇÃO DO EXERCÍCIO.
TRABALHO INDIVIDUAL - plágio será punido com a não correção do exercício.
Pontuação: até 4,0 pontos (para a primeira prova).

Contexto

Você vai implementar um programa que conta quantos números primos existem no intervalo **[2, N]**. Em seguida, você fará **duas versões** do programa:

1. **Sequencial (baseline)**
2. **Concorrente usando processos** (POSIX), com **IPC via pipes OU memória compartilhada**

Depois, você vai **medir desempenho e consumo de recursos** e **explicar** os resultados observados.

Objetivos de aprendizagem

- Criar e gerenciar processos (`fork`, `wait`/`waitpid`)
- Implementar paralelismo por **divisão de intervalo**
- Trocar/combinar resultados via **pipes** ou **shared memory**
- Medir **tempo**, **CPU**, **memória** e inferir gargalos
- Produzir uma **análise curta** baseada em evidências (métricas)

Parte A — Especificação do problema

Entrada

O programa deve receber por linha de comando:

- `N` (inteiro ≥ 2): limite superior do intervalo
- `P` (inteiro ≥ 1): número de processos worker (apenas no modo concorrente)
- `MODE`: `seq` ou `par`
- `IPC` (apenas no modo `par`): `pipe` ou `shm`
- (opcional) `--algo` com algoritmo de primalidade (`basic` obrigatório; extras opcionais)

Exemplos:

```
./primecount seq 5000000
./primecount par 5000000 4 pipe
./primecount par 5000000 8 shm
```

Saída

Seu programa deve imprimir uma linha única, em formato fácil de comparar (texto simples), contendo:

- modo (`seq`/`par`)
- N
- P (se aplicável)
- IPC (se aplicável)
- quantidade de primos
- tempo total (ms) medido internamente (ex.: `clock_gettime`)
- (opcional) tempo só do “miolo” de computação (sem setup)

Exemplo:

```
mode=par N=5000000 P=4 ipc=pipe primes=348513 time_ms=4123
```

Parte B — Implementação sequencial

Implemente:

- `count_primes_seq(N)` que percorre `i=2..N` e testa primalidade.

Teste de primalidade mínimo aceitável

- Método “básico”:
    - trate `n < 2`, `n == 2`, pares
    - teste divisores ímpares de `3..sqrt(n)`.

Observação: não é permitido usar bibliotecas prontas de primos/crivo. O foco é concorrência, mas você precisa de um método “honesto”.

Parte C — Implementação concorrente com processos

Modelo de execução

- Um processo **master** divide o intervalo `[2, N]` em `P` subintervalos.
- O master cria `P` processos **worker** com `fork()`.
- Cada worker conta quantos primos existem no seu subintervalo.
- O master agrega os resultados e imprime a saída.

Divisão de intervalos

Você deve particionar o trabalho para cobrir todo `[2, N]` sem sobreposição e sem buracos.

Sugestão:

- bloco `k` recebe `[Lk, Rk]` com tamanhos quase iguais.

IPC: escolha UMA das alternativas (ou implemente ambas para bônus)

Alternativa 1 — Pipes (pipe)

- O master cria um pipe por worker (ou outro desenho equivalente).
- Cada worker escreve sua contagem parcial em binário (ex.: `uint64_t`) e termina.
- O master lê de todos os pipes e soma.
- Fechar corretamente as pontas não usadas em cada processo (importante para não travar).

Alternativa 2 — Memória compartilhada (shm)

- Use `shm_open` + `ftruncate` + `mmap` **OU** SysV shared memory (qualquer uma aceita).
- A região deve conter um array `partial[P]` (ex.: `uint64_t partial_counts[P]`).
- Cada worker escreve em `partial[id]` e termina.
- O master espera todos (`wait`/`waitpid`) e soma o array.

Requisitos de robustez (obrigatórios)

- Validar argumentos (N, P, modo, IPC).
- Tratar falhas de `fork`, `pipe`, `mmap`, etc.
- Não deixar processos zumbis.
- Não vazar shared memory (se usar `shm_open`, lembre de `shm_unlink`).

Parte D — Medição de desempenho e consumo de recursos

Você vai comparar **seq vs par**.

1) Medir tempo (obrigatório)

- Meça tempo interno com `clock_gettime(CLOCK_MONOTONIC, ...)` ou equivalente.
- Rode pelo menos **3 vezes** cada configuração e reporte um valor representativo (ex.: menor, média, etc. — escolha e diga qual).

2) Medir recursos (obrigatório)

Você deve reportar, pelo menos:

- **Tempo de wall-clock**
- **Tempo de CPU (user+sys)**
- **Memória máxima (resident set size / max RSS)**

Ferramenta sugerida (Linux):

```
/usr/bin/time -v ./primecount seq 5000000
/usr/bin/time -v ./primecount par 5000000 4 pipe
/usr/bin/time -v ./primecount par 5000000 4 shm
```

Você deve copiar do output do `time -v` (ou ferramenta equivalente):

- `User time (seconds)`
- `System time (seconds)`
- `Elapsed (wall clock) time`
- `Maximum resident set size (kbytes)`

  Se usar outra forma (ex.: `perf stat`, `top`, `ps`), tudo bem, desde que explique claramente como coletou.

Parte E — Análise e explicação

Entregar uma explicação curta (pode ser no final do relatório) respondendo:

1. **Speedup observado**:

$$\text{speedup} = \frac{T_{seq}}{T_{par}}$$

   - O speedup aumentou com P? Até onde?

2. **Por que o speedup não é linear?**
   Discuta pelo menos **dois** fatores possíveis, escolhidos com base nas suas medições:

   - overhead de `fork()`/setup
   - overhead de IPC (pipe vs shm)
   - balanceamento de carga (intervalos com custo diferente)
   - limites do hardware (número de núcleos, escalonamento, turbo)
   - custo de `sqrt`/divisões e impacto de cache/branching
   - custo do sistema (tempo em sys, criação/fechamento de descritores)
   - contenção indireta (competição por CPU/cache/memória)

3. **Consumo de memória**:

   - Por que a memória muda (ou não muda) do seq para o par?
   - `pipe` e `shm` se comportaram diferente? Por quê?

4. (Se comparar pipe vs shm) **Por que o sys time muda?**

   - Pipes tendem a usar mais chamadas de sistema/transferência kernel↔user do que “escrever em RAM compartilhada”.

Entregáveis

1. Código-fonte (com `Makefile` ou comando de compilação claro)
2. Executável `primecount`
3. Um relatório curto `relatorio.txt` ou `README.md` contendo:
    - como compilar e rodar
    - tabela com resultados (seq e pelo menos 2 configurações par)
    - speedup calculado
    - sua análise (itens da Parte E)

Critérios de avaliação

- **(0,7)** Sequencial correto e robusto
- **(1,5)** Concorrente correto (processos + IPC) e sem zumbis/travamentos
- **(0,8)** Medição + comparação (tempo e recursos) + explicação coerente com dados
- **Bônus (até +1,0)** Implementar **pipe e shm** e comparar os dois com análise clara

---

Dicas práticas

- Use `uint64_t` para contagens.
- Pule pares: teste apenas ímpares (exceto o 2).
- Para checar corretude rapidamente: use Ns pequenos e compare seq vs par (devem dar o mesmo número).
- Comece com uma implementação simples e correta; depois otimize.

Casos de teste mínimos

- `N=2` → 1 primo
- `N=10` → 4 primos (2,3,5,7)
- `N=100` → 25 primos
- Para performance: `N` na ordem de milhões (depende do lab)