UNISC
UNIVERSIDADE DE SANTA CRUZ DO SUL

Programação Paralela e Distribuída

Aula 6 – Estudo de Caso: RNC com PThreads

Professor Geancarlo Abich
geancarlo@unisc.br

Sumário

Na Aula Anterior

- Sincronização de Tarefas
- Mutex
- Concorrência
- Semáforo
- Atividades Práticas com Pthread

Conteúdo Prático da Disciplina

Definições para o restante da disciplina:

- Uso do sistema operacional Linux;
- Edição dos arquivos pelo gedit/geany ou editor de preferência;
- Compilação e execução por linha de comando via terminal (GCC/LLVM CLANG);
- Uso da linguagem C/C++ para todos os exercícios a serem implementados e entregues, bem como para todos os trabalhos solicitados como avaliação.

Quais serão nossos objetivos na disciplina?

- Apropriação e aplicação de POSIX threads (pthreads);
- Conhecimento e aplicação de OpenMP;
- Conhecimento de aplicação de MPI;
- União da aplicação de OpenMP com MPI.

Arquiteturas e Programas Paralelos

Principais APIs de Programação Paralela

- Multiprocessador
  - Threads (POSIX threads)
  - OpenMP

- Multicomputador/sistema distribuído
  - MPI (OpenMPI)

Sincronização de Tarefas

Apresentação do Problema

Como podemos ter concorrência em um processador single-core?

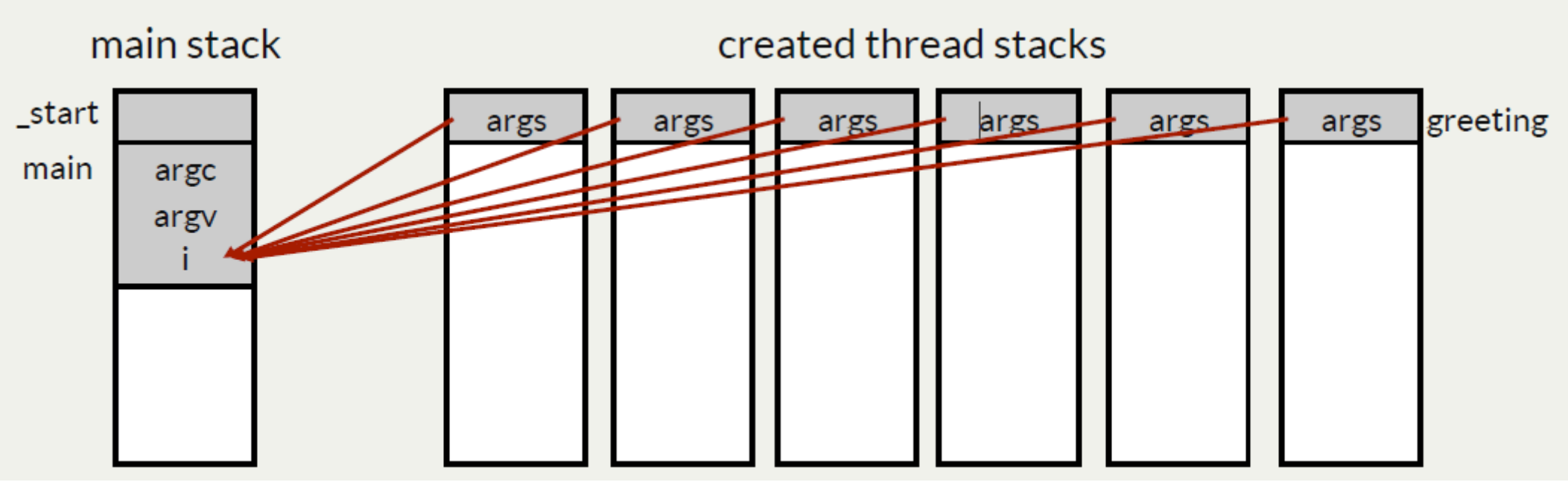
Clássico problema do jantar com Filósofos

- Há cinco filósofos em torno de uma mesa
- Um garfo é colocado entre cada filósofo
- Cada filósofo deve, alternadamente, refletir e comer
- Para que um filósofo coma, ele deve possuir dois garfos
  - Os dois garfos devem ser aqueles logo a sua esquerda e a sua direita
- Para pegar um garfo
  - Somente pode ser pego por um filósofo
  - Somente pode ser pego se não estiver em uso por nenhum outro filósofo
- Após comer, o filósofo deve liberar os garfos que utilizou
- Um filósofo pode segurar o garfo da direita ou esquerda assim que estiverem disponíveis
  - Mas só pode começar a comer quando ambos estiverem sob sua posse
- Quando estão cheios, eles largam os garfos na mesma ordem em que os pegaram e voltam a pensar por um tempo.
- Ao pensar, o filósofo guarda para si por algum tempo. Às vezes eles pensam por um longo tempo, e às vezes eles mal pensam.

Sincronização de Tarefas

Memória compartilhada pela Thread

Exemplo saudações de convidados:



Sincronização de Tarefas

Resumindo - Sincronização

Quando surge a necessidade de sincronização no contexto em que estamos?

Se duas ou mais threads quiserem compartilhar a mesma área de memória ao mesmo tempo.

E, de forma bem resumida, o que é sincronização?

É quando se evita que as threads citadas acima acessem a mesma área de memória ao mesmo tempo.

O que se garante com uma sincronização implementada corretamente?

Que o resultado produzido pelo programa paralelizado será o mesmo de sua versão sequencial, com diminuição do tempo de execução (quase sempre!)

E se ela não for bem empregada, adivinhem? DEADLOCK!! Alguém vai ter que morrer nessa história!

Sincronização de Tarefas

Resumindo - Mutex

É uma das ferramentas que temos para nos ajudar na sincronização.

São variáveis criadas na memória compartilhada e que ficam alternando entre os valores 0 e 1.

Se um mutex estiver em 0, ele será colocado para 1 e o processo poderá utilizar a região compartilhada (região ou seção crítica). Depois de executar as instruções da região crítica, o processo sai e o mutex volta a valer 0.

Se um mutex estiver em 1, o processo terá que esperar para utilizar a região crítica. Quando um processo liberar a região crítica, o mutex vai para 0 e o próximo acesso pode ocorrer pelo processo que conseguir obter o mutex.

Para cada região crítica que precisar ser protegida, um mutex deverá ser associado a ela.

Sincronização de Tarefas

Mutex - Tipos

Mutex padrão:

Se um thread segurando o bloqueio tentar bloqueá-lo novamente, o deadlock

Mutex Recursivo (*recursive_mutex*):

Um thread pode bloquear o mutex várias vezes e precisa desbloqueá-lo o mesmo número de vezes para liberar isso para outros threads

Mutex temporizado (*timed_mutex*):

Um thread pode tentar bloquear (*try_lock_for / try_lock_until)*: se o tempo passar, não usa bloqueio/não bloqueia

Acontece *deadlock* (concorrência) se o mesmo thread tentar bloquear várias vezes, como *mutex* padrão

Sincronização de Tarefas

Resumindo-Semáforos

Este padrão de “desvio de permissão” com sinalização é um padrão muito comum:

- Usa um contador, *mutex* e *condition_variable* para rastrear algumas permissões
- É um método seguro de thread para conceder permissão e aguardar permissão (também conhecida como suspensão)
- Mas é complicado precisar de 3 variáveis para implementar isso - existe uma maneira melhor?
- Um semáforo é um único tipo de variável que encapsula toda essa funcionalidade

Um semáforo é um tipo de variável que permite gerenciar uma contagem de recursos finitos.

- Você inicializa o semáforo com a contagem de recursos para começar
- Você pode solicitar permissão via *semaphore::wait*() - também conhecido como *waitForPermission*
- Você pode conceder permissão via *semaphore::signal*() - também conhecido como *grantPermission*
- Esta é uma definição padrão, mas não incluída nas bibliotecas padrão C++. Por quê?
- Talvez porque seja facilmente construído em termos de outras construções suportadas.

Sincronização de Tarefas

Primitivas na biblioteca Pthreads

Declara o mutex e já o inicializa

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Inicializa os valores do mutex

```
int  pthread_mutex_init(pthread_mutex_t  *mutex,  const  pthread_mutex-attr_t *mutexattr);
```

Impede o acesso de outros processos à região crítica

```
int pthread_mutex_lock(pthread_mutex_t *mutex); /*se região crítica estiver bloqueada, o processo chamador é suspenso (ficará esperando)*/
```

Libera o acesso à região crítica

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Destrói o controle dado pelo objeto mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Exercício Prático

1. Implemente um “hello world” e utilize a biblioteca “phtread.h” para execução em múltiplas threads usando C ou C++ (a critério do Aluno).
2. Compile e execute o hello world utilizando 4 threads através o Docker criado, para isso mapeie uma pasta “projetos” dentro do docker que vamos usar para testes na disciplina;
3. Escreva um código que cria duas threads:
   - A primeira thread calcula a soma dos números de 0 até N (passado por parâmetro);
   - A segunda thread calcula a multiplicação de 1 até M (passado por parâmetro).
   - As threads usam variáveis globais para armazenar o valor da soma e da multiplicação (soma e multiplicação devem ficar no mesmo código!).
   - Resultados finais são impressos no *main*.
4. Escreva um programa que conta o número máximo de threads que podem ser criadas por um processo. Inicialize com 1000000 de threads sendo criadas. Cada thread deve executar uma função que incrementa uma variável global compartilhada, cujo valor deve ser impresso no main ao final da execução de todas as threads.
5. Escrever um programa que cria duas threads, uma que deposita um valor num banco (conta) e outra que retira um valor desse mesmo banco (conta). Observe que deve ser criada somente uma thread para depósito e outra para saque. Além disso, ambas compartilham a variável global ‘saldo’, que deve ser inicializada com o valor 1000. A cada saque ou depósito, uma unidade deve ser incrementada ou decrementada da variável saldo. Ao final do main, escreva o valor do saldo.
6. Implemente o “Dining Philosophers Problem” presente nos exemplos dos slides (*Stanford University*) ou de outra fonte que contenha semáforos e mutex da biblioteca *pthread*.
7. Faça ambos os exemplos para os compiladores gcc e clang (via linha de comando ou makefile).
8. Compare os tempos de execução sequencial e paralela (para comparar, mande somar e multiplicar os mesmos números de elementos nos dois códigos).

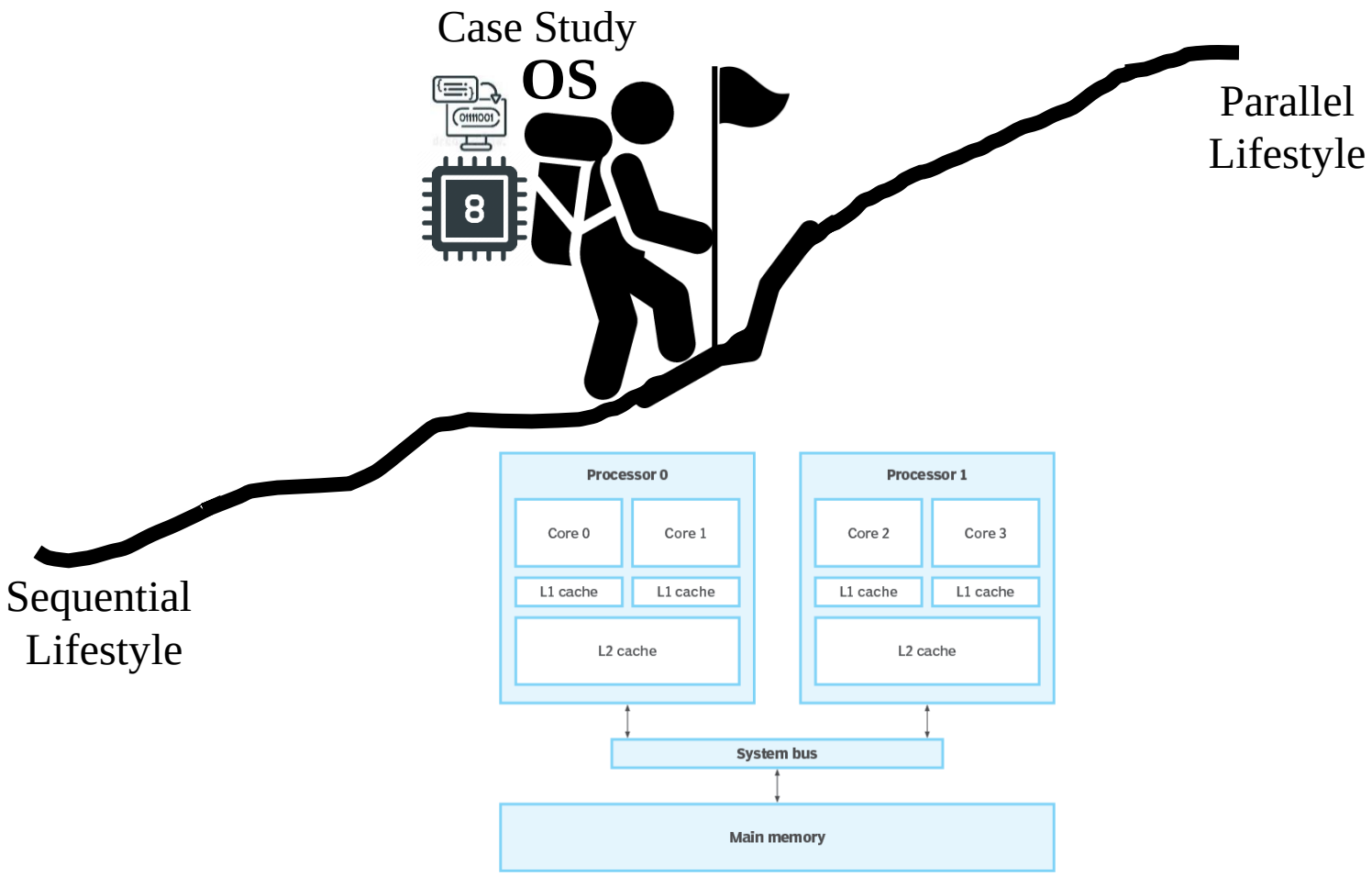
Observação: use preferencialmente Linux cuidando com o número de núcleos definidos na máquina virtual. Podem usar código.

Até hoje temos

Etapas Revisadas

Apresentação da Disciplina

Estudo de Caso Paralelismo



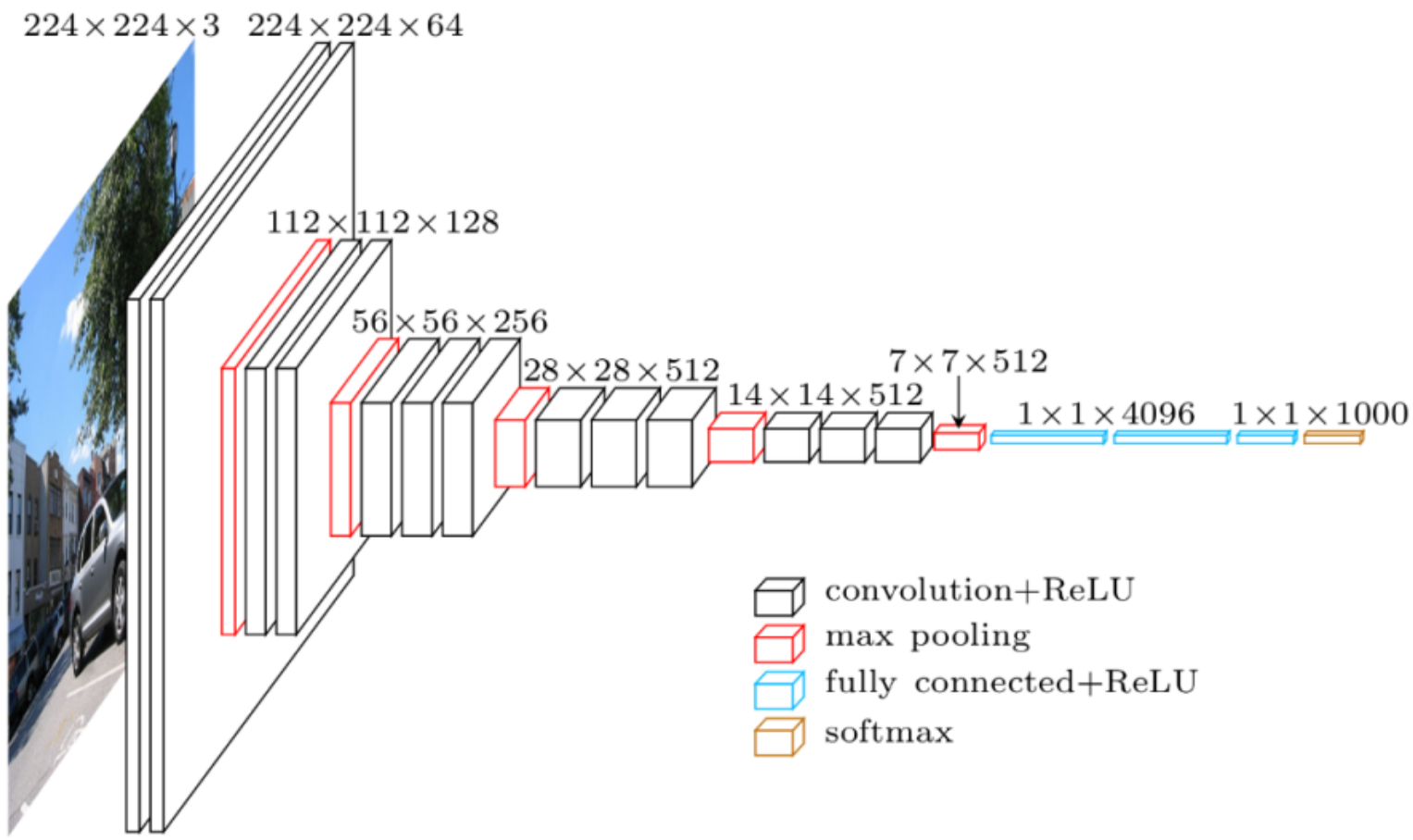
UNISC
UNIVERSIDADE DE SANTA CRUZ DO SUL

Na aula de Hoje...

Pthread vs OpenMP

Especificação do Trabalho 1

Rede Neural Convolucional (RNC) CIFAR-10 Paralela



Especificação do Trabalho 1

Rede Neural Convolucional (RNC) CIFAR-10 Paralela

Paralelizar a aplicação de rede neural convolucional (RNC) CIFAR-10 desenvolvida em C/C++ disponível dentro dos exemplos da biblioteca CMSIS-NN para microcontroladores Arm. Repositório CMSIS: https://github.com/ARM-software/CMSIS_5/tree/5.7.0/

Tarefas:

Etapa 1)

1. Extrair a RNC do código e compilar para a arquitetura da sua máquina host.
2. Calcular o tempo de execução da RNC na sua máquina Host.
3. Gerem 1000 entradas para a RNC (100 de cada tipo, podem usar python para isso) e alterem o código para suportar essa entrada de dados, ou seja, cada execução deve processar todas as entradas.

Etapa 2)

1. Paralelizar com Pthreads a execução da RNC para que cada inferência (detecção de padrões) execute em 1 thread.
2. Casos com pelo menos 1, 2, 3 e 4 threads (até o máximo de núcleos físicos da sua máquina host) e verifiquem quantas inferências são feitas por segundo para cada uma das configurações.
3. Utilizar 2 compiladores: GCC (v10 ou maior) e CLANG (v10 ou maior).

Prévia do Trabalho 1

RNC CIFAR-10 Paralela

Etapas a serem entregues:

1. Etapa1: Entradas geradas e versão sequencial 13/04 antes do início da Aula.
2. Etapa 2: Código com todas as versões + makefile e relatório 20/04.

Grupos:

Máximo de 4 pessoas por grupo.

Referências

https://man7.org/linux/man-pages/man7/pthreads.7.html

Leitura para auxílio no Trabalho:

ABICH, Geancarlo et al. Impact of Thread Parallelism on the Soft Error Reliability of Convolution Neural Networks. In: 2022 IEEE 13th Latin America Symposium on Circuits and System (LASCAS). IEEE, 2022. p. 1-4. (**MOODLE**)

LAI, Liangzhen; SUDA, Naveen; CHANDRA, Vikas. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. arXiv preprint arXiv:1801.06601, 2018. Available at: https://arxiv.org/abs/1801.06601

UNISC
UNIVERSIDADE DE SANTA CRUZ DO SUL

Programação Paralela e Distribuída

Aula 6 – Estudo de Caso: RNC com PThreads

Professor Geancarlo Abich
geancarlo@unisc.br