

Parallel Programming

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 07

B3B36PRG – Programming in C

Overview of the Lecture

- Part 1 – Introduction to Parallel Programming

Introduction
Parallel Processing
Semaphores
Shared Memory
Messages

Part I

Part 1 – Introduction to Parallel Programming

Parallel Programming

- The idea of parallel programming comes from the 60s with the first multi-program and pseudo-parallel systems.
- Parallelism can be hardware or software-based.
 - Hardware-based – true hardware parallelism of multiprocessor systems.
 - Software based – pseudo-parallelism.
- Pseudo-parallelism – A program with parallel constructions may run in the pseudo-parallel environment on single or multiprocessor systems.

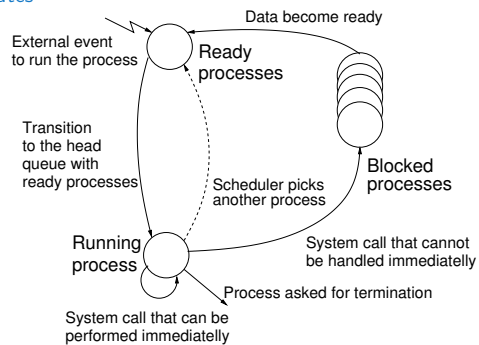
Motivation Why to Deal with Parallel Programming

- Increase computational power.
 - Having a multiprocessor system, we can solve the computational problem faster.
- Efficient usage of the computational power.
 - Even a running program may wait for data.
 - E.g., a usual program with user interaction typically waits for the user input.
- Simultaneous processing of many requests.
 - Handling requests from individual clients in client/server architecture.

Process – Executed Program

- Process is an executed program running in a dedicated memory space.
- Process is an entity of the Operating System (OS) that is scheduled for independent execution.
- Process is usually in one of three basic states:
 - Executing – currently running on the processor (CPU);
 - Blocked – waiting for the periphery;
 - Waiting – waiting for the processor.
- A process is identified in the OS by its identifier, e.g., Process Identifier PID.
- Scheduler of the OS manages running processes to be allocated to the available processors.

Process States



Multi-processor Systems

- Multiprocessor systems allow true parallelism.
- It is necessary to synchronize processors and support data communication.
 - Resources for activity synchronization.
 - Resources for communication between processors (processes).

Possible Architectures for Parallel Executions

- Control of individual instructions.
 - SIMD** – Single-Instruction, Multiple-Data – same instructions are simultaneously performed on different data.
 - “Processors” are identical and run synchronously.
 - E.g., “Vectorization” such as MMX, SSE, 3Dnow!, and AVX, AVX2, etc.
 - MIMD** – Multiple-Instruction, Multiple-Data – processors run independently and asynchronously.
- Memory Control Access.
 - Systems with shared memory – central shared memory.
E.g., multi-core CPUs.
 - Systems with distributed memory – each processor has its own memory.
E.g., computational grids.

Introduction Parallel Processing Semaphores Shared Memory Messages

SIMD – Single-Instruction, Multiple-Data

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 11 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

MIMD – Multiple-Instruction, Multiple-Data

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 12 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Systems with Shared Memory

- Processors communicate using shared memory space.
- Processors may also synchronize their activities, i.e., granting exclusive access to the memory.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 13 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Systems with Distributive Memory

- There is not a problem with exclusive access to the memory.
- It is necessary to address communication between the processors.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 14 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

The Role of the Operating System (OS)

- OS provides hardware abstraction layer – encapsulates HW and separates the user from the particular hardware architecture (true/pseudo parallelism).
- OS is responsible for the synchronization of running processes.
- OS provides user interfaces (system calls):
 - To create and destroy processes.
 - To manage processes and processors.
 - To schedule processors on available processors.
 - To control access to shared memory.
 - Mechanisms for inter-process communication (IPC).
 - Mechanisms for processes synchronization.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 15 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Parallel Processing and Programming Languages

- Regarding parallel processing, programming languages can be divided into languages w/o and without explicit support for parallelism.
 - Without explicit support for parallelism – possible mechanisms of parallel processing.
 - Parallel processing is realized by the compiler and operating system.
 - Parallel constructions are explicitly marked for the compiler.
 - Parallel processing is performed by OS system calls.
 - With explicit support for parallelism.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 17 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example of Parallel Processing Realized by Compiler 1/2

Example – Array Multiplication

```

1 #include "my_malloc.h"
2 #define SIZE 30000000
3 int main(int argc, char *argv[])
4 {
5     int i;
6     int *in1 = (int*)myMalloc(SIZE * sizeof(int));
7     int *in2 = (int*)myMalloc(SIZE * sizeof(int));
8     int *out = (int*)myMalloc(SIZE * sizeof(int));
9     for (i = 0; i < SIZE; ++i) {
10        in1[i] = i;
11        in2[i] = 2 * i;
12    }
13    for (i = 0; i < SIZE; ++i) {
14        out[i] = in1[i] * in2[i];
15        out[i] = out[i] - (in1[i] + in2[i]);
16    }
17    return 0;
18 }

```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 18 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example of Parallel Processing Realized by Compiler 2/2

Example 1

```

1 icc compute.c
2 time ./a.out
3
4 real 0m0.562s
5 user 0m0.180s
6 sys 0m0.384s

```

Example 2

```

1 icc -m32 compute.c; time ./a.out
2 compute.c(8) : (col. 2) remark: LOOP WAS VECTORIZED.
3
4 real 0m0.542s
5 user 0m0.136s
6 sys 0m0.408s

```

Example 3

```

1 icc -parallel compute.c; time ./a.out
2 compute.c(12) : (col. 2) remark: LOOP WAS AUTO-PARALLELIZED.
3
4 real 0m0.702s
5 user 0m0.484s
6 sys 0m0.396s

```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 19 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – OpenMP – Matrix Multiplication 1/2

- Open Multi-Processing (OpenMP) – application programming interface for multi-platform shared memory multiprocessing. <http://www.openmp.org>
- We can instruct the compiler by macros for parallel constructions, such as parallelization over the outside loop for the *i* variable.

```

1 void multiply(int n, int a[n][n], int b[n][n], int c[n][n])
2 {
3     int i;
4     #pragma omp parallel private(i)
5     #pragma omp for schedule (dynamic, 1)
6     for (i = 0; i < n; ++i) {
7         for (int j = 0; j < n; ++j) {
8             c[i][j] = 0;
9             for (int k = 0; k < n; ++k) {
10                c[i][j] += a[i][k] * b[k][j];
11            }
12        }
13    }
14 }

```

lec07/demo-omp-matrix.c
Squared matrices of the same dimensions are used for simplicity.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 20 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – OpenMP – Matrix Multiplication 2/2

- Comparison of matrix multiplication with 1000x 1000 matrices using OpenMP on iCore5 (2 cores with HT).

```

1 gcc -std=c99 -O2 -o demo-omp demo-omp-matrix.c
2 ./demo-omp 1000
3 Size of matrices 1000 x 1000 naive
4   multiplication with 0(n^3)
5 c1 == c2: 1
6 Multiplication single core 9.33 sec
7 Multiplication multi-core 4.73 sec
8
9 export OMP_NUM_THREADS=2
10 ./demo-omp 1000
11 Size of matrices 1000 x 1000 naive
12   multiplication with 0(n^3)
13 c1 == c2: 1
14 Multiplication single core 9.48 sec
15 Multiplication multi-core 6.23 sec
  
```

Use, e.g., top program for a list of running processes/threads.

```

lec07/demo-omp-matrix.c
  
```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 21 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Languages with Explicit Support for Parallelism

- It has support for the creation of new processes.
 - Running process creates a copy of itself.
 - Both processes execute the identical code (copied).
 - The **parent** process and **child** process are distinguished by the process identifier (PID).
 - The code segment is explicitly linked with the new process.
- Regardless of how a new process is created, the most important is the relation to the parent process execution and memory access.
 - Does the parent process stop its execution till the end of the child process?
 - Is the memory shared by the child and parent processes?
- Granularity of the processes – parallelism ranging from the level of the instructions to the parallelism of programs.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 22 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Parallelism – Statement Level

Example – parbegin-parend block

```

parbegin
S1;
S2;
...
Sn
parend
  
```

- Statement S_1 are S_n executed in parallel.
- Execution of the main program is interrupted until all statements S_1 to S_n are terminated.
- Statement S_1 are S_n executed in parallel.

Example – doparallel

```

1 for i = 1 to n doparallel {
2   for j = 1 to n do {
3     c[i,j] = 0;
4     for k = 1 to n do {
5       c[i,j] = c[i,j] + a[i,k]*b[k,j];
6     } }
  }
  
```

Parallel execution of the outer loop over all i .

E.g., OpenMP in C.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 23 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Parallelism – Procedure Level

- A procedure is coupled with the execution process.


```

...
procedure P;
...
PID  $x_{pid}$  = newprocess(P);
...
killprocess( $x_{pid}$ );
      
```

 - P is a procedure and x_{pid} is a process identifier.
- Assignment of the procedure/function to the process at the declaration


```

PID  $x_{pid}$  process(P).
      
```

 - The process is created at the creation of the variable x .
 - The process is terminated at the end of x or sooner.

E.g., Threads (pthreads) in C.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 24 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Parallelism – Program (Process) Level

- A new process can be only a whole program.
- A new program is created by a system call, which creates a complete copy of itself, including all variables and data at the moment of the call.

Example - Creating a copy of the process by fork system call

```

1 if (fork() == 0) {
2   /* code executed by the child process */
3 } else {
4   /* code executed by the parent process */
5 }
  
```

E.g., fork() in C

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 25 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – fork()

```

1 #define NUMPROCS 4
2 for (int i = 0; i < NUMPROCS; ++i) {
3   pid_t pid = fork();
4   if (pid == 0) {
5     compute(i, n);
6     exit(0);
7   } else {
8     printf("Child %d created\n", pid);
9   }
10 }
11 printf("All processes created\n");
12 for (int i = 0; i < NUMPROCS; ++i) {
13   pid_t pid = wait(&r);
14   printf("Wait for pid %d return: %d\n", pid, r);
15 }
16 void compute(int myid, int n)
17 {
18   printf("Process myid %d start computing\n", myid);
19   ...
20   printf("Process myid %d finished\n", myid);
21 }
  
```

Jan Faigl, 2025 lec07/demo-fork.c B3B36PRG – Lecture 07: Parallel Programming 26 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Semaphore

- E. W. Dijkstra – Semaphore is a mechanism to synchronize parallel processes with shared memory.
- Semaphore is an integer variable with the following operations.
 - InitSem** - initialization.
 - Wait**

```

{ If  $S > 0$  then  $S \leftarrow S - 1$  (resources are available, in this case, acquire one).
  Otherwise suspend execution of the calling process (wait for  $S$  become  $S > 0$ ).
      
```
 - Signal**

```

{ If there is a waiting process, awake it (let the process acquire one resource).
  Otherwise increase value of  $S$  by one, i.e.,  $S \leftarrow S + 1$  (release one resource).
      
```
- Semaphores can be used to control access to shared resources.
 - $S < 0$ - shared resource is in use. The process asks for access to the resources and waits for its release.
 - $S > 0$ - shared resource is available. The process releases the resource.

The value of the Semaphore can represent the number of available resources. Then, we can acquire (or wait for) k resources – wait(k): $S \leftarrow S - k$ for $S > k$, and also releases k resources – signal(k): $S \leftarrow S + k$.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 28 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Semaphores Implementation

- Operations with a semaphore must be atomic.

The processor cannot be interrupted during the execution of the operation.
- Machine instruction **TestAndSet** reads and stores the content of the addressed memory space and sets the memory to a non-zero value.
- During execution of the **TestAndSet** instructions, the processor holds the system bus, and access to the memory is not allowed for any other processor.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 29 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Usage of Semaphores

- Semaphores can be utilized for defining a **critical sections**.
- Critical sections are a part of the program where exclusive access to the shared memory (resources) must be guaranteed.

Example of critical section protected by a semaphore

```

InitSem(S,1);
Wait(S);
/* Code of the critical section */
Signal(S);
  
```

- Synchronization of the processes using semaphores.

Example of synchronization of processes.

```

/* process p */
...
InitSem(S,0)
Wait(S); ...
exit();

/* process q */
...
Signal(S);
exit();
  
```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 30 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Semaphore 1/4 (System Calls)

- Semaphore is an entity of the Operating System (OS).

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/sem.h>
4
5 /* create or get existing set of semaphores */
6 int semget(key_t key, int nsems, int flag);
7
8 /* atomic array of operations on a set of semaphores */
9 int semop(int semid, struct sembuf *array, size_t nops);
10
11 /* control operations on a st of semaphores */
12 int semctl(int semid, int semnum, int cmd, ...);

```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 31 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Semaphore 2/4 (Synchronization Protocol)

- Example when the main (primary) process waits for two other processes (secondary) to become ready.
 - Primary process suspend the execution and waits for two other secondary processes become ready.
 - Secondary processes then wait to be released by the primary process.
- Proposed synchronization "protocol".
 - Define our way to synchronize the processes using the system semaphores.
 - Secondary process increments semaphore by 1.
 - The secondary process waits for the Semaphore to become 0, and then it is terminated.
 - Primary process waits for two secondary processes and decrements the semaphore about 2.
 - It must also ensure the semaphore value is not 0; otherwise, secondary processes will be terminated prematurely.
 - We need to use the atomic operations with the Semaphore.


```

lec07/sem-primary.c  lec07/sem-secondary.c

```

The design of the communication/synchronization protocol is the most important task of the developer.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 32 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Semaphore 3/4 (Primary Process)

```

1 int main(int argc, char* argv[])
2 {
3     struct sembuf sem[2]; // structure for semaphore atomic operations
4     int id = semget(1000, 1, IPC_CREAT | 0666); // create semaphore
5     if (id != -1) {
6         int r = semctl(id, 0, SETVAL, 0) == 0;
7         sem[0].sem_num = 0; // operation to acquire semaphore
8         sem[0].sem_op = -2; // once its value will be >= 2
9         sem[0].sem_flg = 0; // representing two secondary processes are ready
10        sem[1].sem_num = 0; // the next operation in the atomic set
11        sem[1].sem_op = 2; // of operations increases the value of
12        sem[1].sem_flg = 0; // the semaphore about 2
13        printf("Wait for semvalue >= 2\n");
14        r = semop(id, sem, 2); // perform all (two) operations (in sem array) atomically
15        printf("Press ENTER to set semaphore to 0\n");
16        getchar();
17        r = semctl(id, 0, SETVAL, 0) == 0; // set the value of semaphore
18        r = semctl(id, 0, IPC_RMID, 0) == 0; // remove the semaphore
19    }
20    return 0;
21 }

```

lec07/sem-primary.c

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 33 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Semaphore 4/4 (Secondary Process)

```

1 int main(int argc, char* argv[])
2 {
3     struct sembuf sem;
4     int id = semget(1000, 1, 0);
5     int r;
6     if (id != -1) {
7         sem.sem_num = 0; // add the secondary process
8         sem.sem_op = 1; // to the "pool" of resources
9         sem.sem_flg = 0;
10        printf("Increase semaphore value (add resource)\n");
11        r = semop(id, &sem, 1);
12        sem.sem_op = 0;
13        printf("Semaphore value is %d\n", semctl(id, 0, GETVAL, 0));
14        printf("Wait for semaphore value 0\n");
15        r = semop(id, &sem, 1);
16        printf("Done\n");
17    }
18    return 0;
19 }

```

clang sem-primary.c -o sem-primary
clang sem-secondary.c -o sem-secondary
lec07/sem-secondary.c

- The IPC entities can be listed by `ipcs`.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 34 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Issues with Semaphores

- The main issues are arising from a wrong usage.
- Typical mistakes are as follows.
 - Wrongly identified a critical section.
 - Process may block by multiple calls of Wait(S).
 - E.g., the **deadlock** issues may arise from situations like.

Example – Deadlock

```

/* process 1*/          /* process 2*/
...                    ...
Wait(S1);              Wait(S2);
Wait(S2);              Wait(S1);
...                    ...
Signal(S2);            Signal(S1);
Signal(S1);            Signal(S2);
...                    ...

```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 35 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Shared Memory

- Labeled part of the memory accessible from different processes.
- OS service provided by system calls.

Example of System Calls

```

1 /* obtain a shared memory identifier */
2 int shmget(key_t key, size_t size, int flag);
3
4 /* attach shared memory */
5 void* shmat(int shmid, const void *addr, int flag);
6
7 /* detach shared memory */
8 int shmdt(const void *addr);
9
10 /* shared memory control */
11 int shmctl(int shmid, int cmd, struct shmid_ds *buf);

```

- OS manages information about the usage of shared memory.
- OS also manages permissions and access rights.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 37 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Shared Memory 1/4 (Write)

- Write a line read from `stdin` to the shared memory.

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define SIZE 512
6
7 int main(int argc, char *argv[])
8 {
9     char *buf;
10    int id;
11    if ((id = shmget(1000, SIZE, IPC_CREAT | 0666)) != -1) {
12        if ( (buf = (char*)shmat(id, 0, 0)) ) {
13            fgets(buf, SIZE, stdin);
14            shmdt(buf);
15        }
16    }
17    return 0;
18 }

```

See man fgets!
lec07/shm-write.c

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 38 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Shared Memory 2/4 (Read)

- Read a line from the shared memory and put it to the `stdout`.

```

1 #include <sys/types.h>
2 #include <sys/shm.h>
3 #include <stdio.h>
4 #define SIZE 512
5
6 int main(int argc, char *argv[])
7 {
8     int id;
9     char *buf;
10    if ((id = shmget(1000, 512, 0)) != -1) {
11        if ((buf = (char*)shmat(id, 0, 0)) ) {
12            printf("mem:%s\n", buf);
13        }
14        shmdt(buf);
15    } else {
16        fprintf(stderr, "Cannot access to shared memory!\n");
17    }
18    return 0;
19 }

```

lec07/shm-read.c

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 39 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Shared Memory 3/4 (Demo)

- Use `shm-write` to write a text string to the shared memory.
- Use `shm-read` to read data (string) from the shared memory.
- Remove shared memory segment.


```
ipcrm -M 1000
```
- Try to read data from the shared memory.


```

1 % clang -o shm-write shm-write.c
2 % ./shm-write
3 mem:Hello! I like programming in C!
4
5 % ./shm-read
6 mem:Hello! I like programming in C!
7
8 % ipcrm -M 1000
9 % ./shm-read
10 Cannot access to shared memory!

```

lec07/shm-write.c lec07/shm-read.c

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 40 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Shared Memory 4/4 (Status)

- A list of accesses to the shared memory using `ipcs` command.

```

1 after creating shared memory segment and before writing the text
2 m 65539 1000 --rw-rw-rw- jf jf jf jf 1 512
   1239 1239 22:18:48 no-entry 22:18:48
4 after writing the text to the shared memory
5 m 65539 1000 --rw-rw-rw- jf jf jf jf 0 512
   1239 1239 22:18:48 22:19:37 22:18:48
7 after reading the text
8 m 65539 1000 --rw-rw-rw- jf jf jf jf 0 512
   1239 1260 22:20:07 22:20:07 22:18:48

```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 41 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Sensing Messages and Queues of Messages

- Processes can communicate via messages sent/received to/from system messages queues.
- Queues are entities of the OS with defined system calls.

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/msg.h>
4
5 /* Create a new message queue */
6 int msgget(key_t key, int msgflg);
7
8 /* Send a message to the queue -- block/non-block (IPC_NOWAIT) */
9 int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
10
11 /* Receive message from the queue -- block/non-block (IPC_NOWAIT) */
12 int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
13
14 /* Control operations (e.g., destroy) the message queue */
15 int msgctl(int msqid, int cmd, struct msqid_ds *buf);

```

Another message-passing system can be implemented by a user library, e.g., using network communication.

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 43 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Messages Passing 1/4 (Synchronization, Primary)

- Two processes are synchronized using messages.
 - The **primary** process waits for the message from the **secondary** process
 - The primary process informs the secondary to solve the task.
 - The secondary process informs the primary about the solution.
 - The primary process sends a message about termination.

```

1 struct msgbuf {
2     long mtype;
3     char mtext[SIZE];
4 };
5
6 int main(int argc, char *argv[])
7 {
8     struct msgbuf msg;
9     int id = msgget(KEY, IPC_CREAT | 0666);
10    int r;
11    if (id != -1) {

```

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 44 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Messages Passing 2/4 (Primary)

```

12 msg.mtype = 3; //type must be > 0
13 printf("Wait for other process \n");
14 r = msgrcv(id, &msg, SIZE, 3, 0);
15 printf("Press ENTER to send work\n");
16 getchar();
17 strcpy(msg.mtext, "Do work");
18 msg.mtype = 4; //work msg is type 4
19 r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
20 fprintf(stderr, "msgsnd r: %d\n", r);
21 printf("Wait for receive work results\n", r);
22 msg.mtype = 5;
23 r = msgrcv(id, &msg, sizeof(msg.mtext), 5, 0);
24 printf("Received message: %s\n", msg.mtext);
25 printf("Press ENTER to send exit msg\n");
26 getchar();
27 msg.mtype = EXIT_MSG; //I choose type 10 as exit msg
28 r = msgsnd(id, &msg, 0, 0);
29 }
30 return 0;
31 }

```

lec07/msg-primary.c

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 45 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Messages Passing 3/4 (Secondary)

```

1 int main(int argc, char *argv[])
2 {
3     ...
4     msg.mtype = 3;
5     printf("Inform main process\n");
6     strcpy(msg.mtext, "I'm here, ready to work");
7     r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
8     printf("Wait for work\n");
9     r = msgrcv(id, &msg, sizeof(msg.mtext), 4, 0);
10    printf("Received message: %s\n", msg.mtext);
11    for (i = 0; i < 4; i++) {
12        sleep(1);
13        printf(".");
14        fflush(stdout);
15    } //do something useful
16    printf("Work done, send wait for exit\n");
17    strcpy(msg.mtext, "Work done, wait for exit");
18    msg.mtype = 5;
19    r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
20    msg.mtype = 10;
21    printf("Wait for exit msg\n");
22    r = msgrcv(id, &msg, SIZE, EXIT_MSG, 0);
23    printf("Exit message has been received\n");

```

lec07/msg-secondary.c

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 46 / 49

Introduction Parallel Processing Semaphores Shared Memory Messages

Example – Messages Passing 4/4 (Demo)

- Execute the primary process.
- Execute the secondary process.
- Perform the computation.
- Remove the created message queue identified by the `msgid`.

```

1 % clang msg-primary.c -o primary
2 % ./primary
3 Wait for other process
4 Worker msg received, press ENTER to send work msg
5 Received message:Do work
6 msgsnd r: 0
7 Work done, send wait for exit
8 Received message: I am going to wait for exit msg
9 Exit message has been received
10 %ipcs -q
11 Message Queues:
12 T ID KEY MODE OWNER GROUP
13 q 65536 1000 --rw-rw- jf jf
14 %

```

ipcrmq -Q 1000

lec07/msg-primary.c lec07/msg-secondary.c

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 47 / 49

Topics Discussed

Summary of the Lecture

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 48 / 49

Topics Discussed

- Introduction to Parallel Programming
 - Ideas and main architectures
 - Program and process in OS
- Parallel processing
- Synchronization and Inter-Process Communication (IPC)
 - Semaphores
 - Messages
 - Shared memory

Design the synchronization/communication between the processes.

- Next: Multithreading programming

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 49 / 49

Parallel Computing using GPU (optional)

Part III

Appendix

Jan Faigl, 2025 B3B36PRG – Lecture 07: Parallel Programming 50 / 49

Massive parallelism using graphics cards

- Image rendering performed pixel-by-pixel can be easily parallelized.
- Graphics Processing Units (GPU) have a similar (or even higher) degree of integration with the main processors (CPU).
- They have a huge number of parallel processors.
 - E.g., GeForce GTX 1060 ~ 1280 cores.*
- The computational power can also be used in another application.
 - Processing stream of data (SIMD instructions - processors).
 - GPGPU - General Purpose computation on GPU. <http://www.gpgpu.org>
 - OpenCL (Open Computing Language) – GPGPU abstract interface.
 - CUDA - Parallel programming interface for NVIDIA graphics cards. http://www.nvidia.com/object/cuda_home.html

Computational Power (2008)

- What is the reported processor computational power?
- Graphics (stream) processors.

CSX700	96 GigaFLOPs	
Cell	102 GigaFLOPs	
GeForce 8800 GTX	518 GigaFLOPs	(including texture units)
Radeon HD 4670	480 GigaFLOPs	
GeForce RTX 4060	15 110 GigaFLOPs	(2023)
- Main processors :

Phenom X4 9950 (@2.6 GHz)	21 GigaFLOPs	<i>Peak catalogue values.</i>
Core 2 Duo E8600 (@3.3 GHz)	22 GigaFLOPs	
Cure 2 Quad QX9650 (@3.3 GHz)	35 GigaFLOPs	<i>Test linpack 32-bit.</i>
Cure 2 Quad QX9650 (@3.3 GHz)	35 GigaFLOPs	
Cure i7 970 (@3.2 GHz)	42 GigaFLOPs	<i>(Float vs double)</i>
Core i9-13900 (@2.00–5.60 GHz)	846 GigaFLOPs	
- Is the reported power really achievable?
- How about other indicators?
 - CSX700 has typical power consumption around 9W. *E.g., computational power / power consumption.*

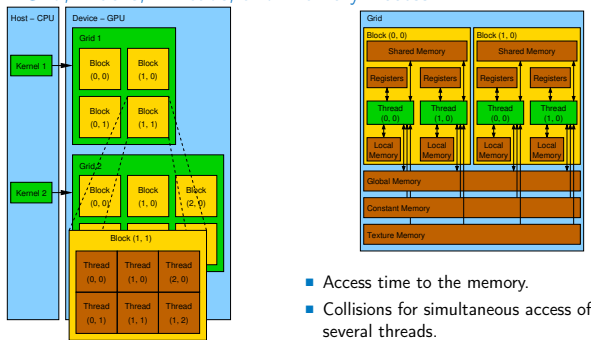
CUDA

- NVIDIA Compute Unified Device Architecture.
- Extension of the C to access the parallel computational units of the GPU.
- Computation (**kernel**) is executed by the GPU.
- Kernel is performed in parallel using available computational units.
- Host** - Main processor (process).
- Device** - GPU.
- Data must be in the memory accessible by the GPU.
 - Host memory → Device memory*
- The result (of the computation) is stored in the GPU memory.
 - Host memory ← Device memory*

CUDA – Computational Model

- Kernel (computation) is divided into blocks.
 - Each block represents a parallel computation of the part of the result. *E.g., a part of the matrix multiplication.*
- Each block consists of computational threads.
- Parallel computations are synchronization within the block.
- Blocks are organized into the **grid**.
- Scalability is realized by dividing the computation into blocks.
 - Blocks may not necessarily be computed in parallel. Based on the available number of parallel units, particular blocks can be computed sequentially.*

CUDA – Grid, Blocks, Threads, and Memory Access



CUDA – Example – Matrix Multiplication 1/8

- NVIDIA CUDA SDK - Version 2.0, matrixMul.
- Simple matrix multiplication.
 - $C = A \cdot B$.
 - Matrices have identical dimensions $n \times n$.
 - where n is the multiple of the block size.
- Comparison
 - naive implementation in C ($3 \times$ for loop),
 - naive implementation in C with matrix transpose.
 - CUDA implementation.
- Hardware
 - CPU - Intel Core 2 Duo @ 3 GHz, 4 GB RAM,
 - GPU - NVIDIA G84 (GeForce 8600 GT), 512 MB RAM.

CUDA – Example – Matrix Multiplication 2/8

Naive implementation

```

1 void simple_multiply(const int n,
2   const float *A, const float *B, float *C)
3 {
4   for (int i = 0; i < n; ++i) {
5     for (int j = 0; j < n; ++j) {
6       float prod = 0;
7       for (int k = 0; k < n; ++k) {
8         prod += A[i * n + k] * B[k * n + j];
9       }
10      C[i * n + j] = prod;
11    }
12  }
13 }
    
```

CUDA – Example – Matrix Multiplication 3/8

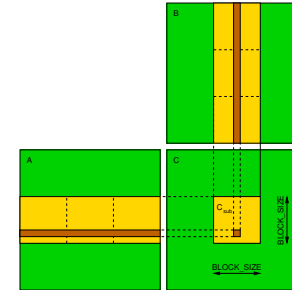
Naive implementation with transpose

```

1 void simple_multiply_trans(const int n, const float *a, const float *b, float *c)
2 {
3   float *bT = create_matrix(n);
4   for (int i = 0; i < n; ++i) {
5     bT[i*n + i] = b[i*n + i];
6     for (int j = i + 1; j < n; ++j) {
7       bT[i*n + j] = b[j*n + i];
8       bT[j*n + i] = b[i*n + j];
9     }
10  }
11  for (int i = 0; i < n; ++i) {
12    for (int j = 0; j < n; ++j) {
13      float tmp = 0;
14      for (int k = 0; k < n; ++k) {
15        tmp += a[i*n + k] * bT[j*n + k];
16      }
17      c[i*n + j] = tmp;
18    }
19  }
20  free(bT);
21 }
    
```

CUDA – Example – Matrix Multiplication 4/8

- CUDA – computation strategy
 - Divide matrices into blocks.
 - Each block computes a single sub-matrix C_{sub} .
 - Each thread of the individual blocks computes a single element of C_{sub} .



CUDA – Example – Matrix Multiplication 5/8

```

CUDA – Implementation – main function
1 void cuda_multiply(const int n, const float *hostA, const float *hostB, float *hostC)
2 {
3     const int size = n * n * sizeof(float);
4     float *devA, *devB, *devC;
5     cudaMalloc((void**)&devA, size);
6     cudaMalloc((void**)&devB, size);
7     cudaMalloc((void**)&devC, size);
8     cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);
9     cudaMemcpy(devB, hostB, size, cudaMemcpyHostToDevice);
10    dim3 threads(BLOCK_SIZE, BLOCK_SIZE); // BLOCK_SIZE == 16
11    dim3 grid(n / threads.x, n / threads.y);
12    // Call kernel function matrixMul
13    matrixMul<<<grid, threads>>>(n, devA, devB, devC);
14    cudaMemcpy(hostC, devC, size, cudaMemcpyDeviceToHost);
15    cudaFree(devA);
16    cudaFree(devB);
17    cudaFree(devC);
18 }

```

CUDA – Example – Matrix Multiplication 6/8

```

CUDA implementation – kernel function
1 __global__ void matrixMul(int n, float* A, float* B, float* C) {
2     int bx = blockIdx.x; int by = blockIdx.y;
3     int tx = threadIdx.x; int ty = threadIdx.y;
4     int aBegin = n * BLOCK_SIZE * by; //beginning of sub-matrix in the block
5     int aEnd = aBegin + n - 1; //end of sub-matrix in the block
6     float Csub = 0;
7     for (
8         int a = aBegin, b = BLOCK_SIZE * bx;
9         a <= aEnd;
10        a += BLOCK_SIZE, b += BLOCK_SIZE * n
11    ) {
12        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE]; // shared memory within
13        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE]; // the block
14        As[ty][tx] = A[a + n * ty + tx]; // each thread reads a single element
15        Bs[ty][tx] = B[b + n * ty + tx]; // of the matrix to the memory
16        __syncthreads(); // synchronization, sub-matrix in the shared memory
17        for (int k = 0; k < BLOCK_SIZE; ++k) { // each thread computes
18            Csub += As[ty][k] * Bs[k][tx]; // the element in the sub-matrix
19        }
20        __syncthreads();
21    }
22    int c = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
23    C[c + n * ty + tx] = Csub; // write the results to memory
24 }
25 }

```

CUDA – Example – Matrix Multiplication 7/8

- CUDA source codes.

Example – Dedicated source file cuda_func.cu

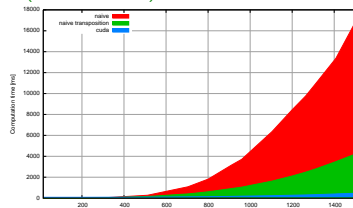
1. Declaration of the external function.


```
extern "C" { // declaration of the external function (cuda kernel)
void cuda_multiply(const int n, const float *A, const float *B, float *C);
}
```
2. Compile the CUDA code to the C++ code.


```
1 nvcc --cuda cuda_func.cu -o cuda_func.cc
```
3. Compilation of the cuda_func.cu.cc file using standard compiler.

CUDA – Example – Matrix Multiplication 8/8

Computational time (in milliseconds)



N	Naive	Transp.	CUDA	N	Naive	Transp.	CUDA
112	2	1	81	704	1083	405	122
208	11	11	82	1104	6360	1628	235
304	35	33	84	1264	9763	2485	308

■ Matlab 7.6.0 (R2008a):
 n=1104; A=rand(n,n); B=rand(n,n); tic; C=A*B; toc
 Elapsed time is 0.224183 seconds.