# Parallel Programming


Jan Faigl

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


Lecture 07

**BE5B99CPL – C Programming Language**



## Overview of the Lecture







# 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
  - Hardware based – true hardware parallelism of multiprocessor systems
  - Software based – pseudo-parallelism
- Pseudo-parallelism – A program with parallel constructions may run in pseudo-parallel environment on single or multi-processor systems





## Motivation Why to Deal with Parallel Programming

- Increase computational power
  - Having multi-processor 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





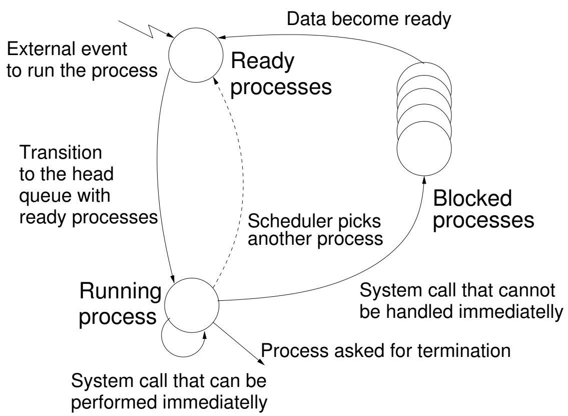
## Process – Executed Program

- Process is an executed program running in a dedicated memory space
- Process is an entity of the Operating System (OS) that is scheduled for an 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 OS by an identifier, e.g., PID
- Scheduler of the OS manage running processes to be allocated to the available processors





## Process States








## Multi-processor Systems

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





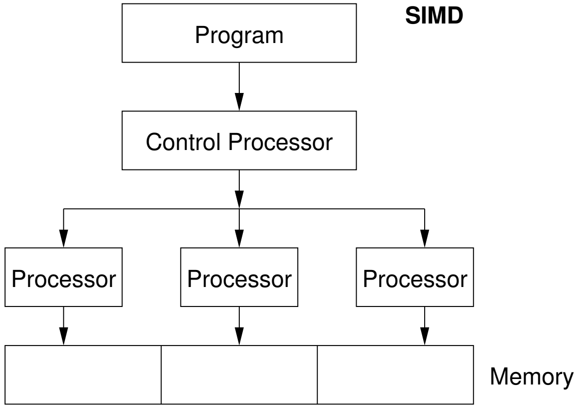
## Possible Architectures for Parallel Executions

- 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, 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 memory

    *E.g., computational grids*

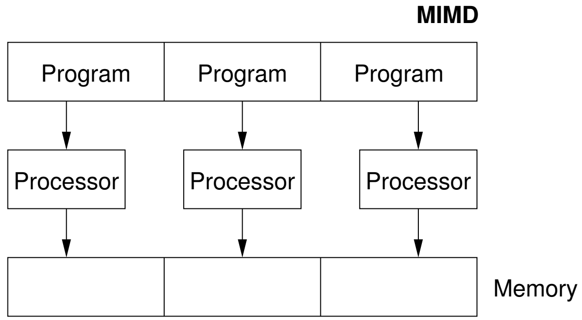
## SIMD – Single-Instruction, Multiple-Data







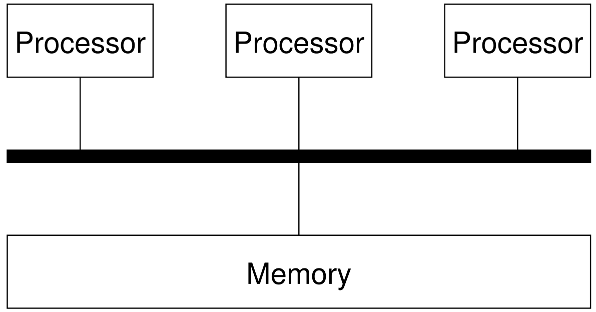

## MIMD – Multiple-Instruction, Multiple-Data








## Systems with Shared Memory

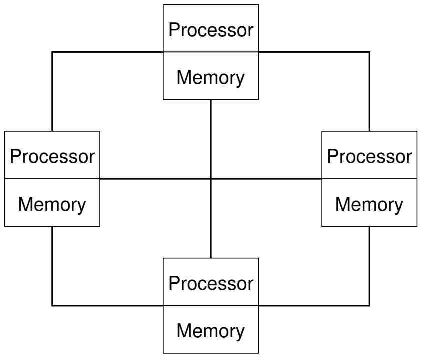



- Processors communicate using shared memory space
- Processors may also synchronize their activities, i.e., grant exclusive access to the memory





## Systems with Distributive Memory




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





## 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 processes synchronization
- 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





## Parallel Processing and Programming Languages

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





## Example of Parallel Processing Realized by Compiler 1/2

### Example – Array Multiplication

```
#include <stdlib.h>

#define SIZE 30000000

int main(int argc, char *argv[])
{
   int i;
   int *in1 = (int*)malloc(SIZE*sizeof(int));
   int *in2 = (int*)malloc(SIZE*sizeof(int));
   int *out = (int*)malloc(SIZE*sizeof(int));
   for (i = 0; i < SIZE; ++i) {
      in1[i] = i;
      in2[i] = 2 * i;
   }
   for (i = 0; i < SIZE; ++i) {
      out[i] = in1[i] * in2[i];
      out[i] = out[i] - (in1[i] + in2[i]);
   }
   return 0;
}
```





## Example of Parallel Processing Realized by Compiler 2/2

### Example 1

```
icc compute.c
time ./a.out

real 0m0.562s
user 0m0.180s
sys  0m0.384s
```

### Example 2

```
icc -msse compute.c; time ./a.out
compute.c(8) : (col. 2) remark: LOOP WAS
      VECTORIZED.

real 0m0.542s
user 0m0.136s
sys  0m0.408s
```

### Example 3

```
icc -parallel  compute.c; time ./a.out
compute.c(12) : (col. 2) remark: LOOP WAS AUTO-
      PARALLELIZED.

real 0m0.702s
user 0m0.484s
sys  0m0.396s
```





## Example – Open MP – Matrix Multiplication 1/2

- Open Multi-Processing (OpenMP) - application programming interface for multi-platform shared memory multiprocessing

http://www.openmp.org

- We can instruct compiler by macros for parallel constructions
  - Example of parallelization over the outside loop for *i* variable

```
void multiply(int n, int a[n][n], int b[n][n], int c[n][n])
{
   int i;
#pragma omp parallel private(i)
#pragma omp for schedule (dynamic, 1)
   for (i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         c[i][j] = 0;
         for (int k = 0; k < n; ++k) {
            c[i][j] += a[i][k] * b[k][j];
         }
      }
   }
}
```

lec07/demo-omp-matrix.c

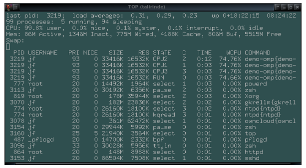
*Squared matrices of the same dimensions are used for simplicity.*

## Example – Open MP – Matrix Multiplication 2/2

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

```
gcc -std=c99 -O2 -o demo-omp demo-omp-matrix.c -fopenmp
./demo-omp 1000
Size of matrices 1000 x 1000 naive
        multiplication with O(n^3)
c1 == c2: 1
Multiplication single core 9.33 sec
Multiplication  multi-core 4.73 sec

export OMP_NUM_THREADS=2
./demo-omp 1000
Size of matrices 1000 x 1000 naive
        multiplication with O(n^3)
c1 == c2: 1
Multiplication single core 9.48 sec
Multiplication  multi-core 6.23 sec
```



lec07/demo-omp-matrix.c





## Languages with Explicit Support for Parallelism

- It has support for 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 how new process is created – the most important is
  - Does the parent process stops 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





## Parallelism – Statement Level

### Example – parbegin-parend block

**parbegin**
$S_1$;
$S_2$;
...
$S_n$
**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

```
for i = 1 to n doparalel {
   for j = 1 to n do {
      c[i,j] = 0;
      for k = 1 to n do {
         c[i,j] = c[i,j] + a[i,k]*b[k,j];
} } }
```

Parallel execution of the outer loop over all *i*. *E.g., as the OpenMP in C*





## 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*





## 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 variable and data at the moment of the call

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

```
if (fork() == 0) {
   /* code executed by the child process */
} else {
   /* code executed by the parent process */
}
```

*E.g., fork() in C*

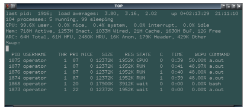




## Example – fork()

```
#define NUMPROCS 4
for (int i = 0; i < NUMPROCS; ++i) {
   pid_t pid = fork();
   if (pid == 0) {
      compute(i, n);
      exit(0);
   } else {
      printf("Child %d created\n", pid);
   }
}
printf("All processes created\n");
for (int i = 0; i < NUMPROCS; ++i) {
   pid_t pid = wait(&r);
   printf("Wait for pid %d return: %d\n",
     pid, r);
}
void compute(int myid, int n)
{
   printf("Process myid %d start
        computing\n", myid);
   ...
   printf("Process myid %d
        finished\n", myid);
}
```

lec07/demo-fork.c



```
clang  demo-fork.c && ./a.out
Child 2049 created
Process myid 0 start computing
Child 2050 created
Process myid 1 start computing
Process myid 2 start computing
Child 2051 created
Child 2052 created
Process myid 3 start computing
All processes created
Process myid 1 finished
Process myid 0 finished
Wait for pid 2050 return: 0
Process myid 3 finished
Process myid 2 finished
Wait for pid 2049 return: 0
Wait for pid 2051 return: 0
Wait for pid 2052 return: 0
```





## 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* $\begin{cases} S > 0 - S = S - 1 \\ \text{otherwise – suspend execution of the calling process} \end{cases}$
  - *Signal* $\begin{cases} \text{awake a waiting process if such process exists} \\ \text{otherwise} - S = S + 1 \end{cases}$
- Semaphores can be used to control access to shared resources
  - S < 0 – shared resource is in use; the process asks for the 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*





## Semaphores Implementation

- Operations with a semaphore must be atomic
  *The processor cannot be interrupted during execution of the operation*
- Machine instruction *TestAndSet* reads and stores a content of the addressed memory space and set 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





## Usage of Semaphores

- Semaphores can be utilized for defining **critical sections**
- Critical section is 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();
```

Process p waits for termination of the process q

## Example – Semaphore 1/4 (System Calls)

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

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* create or get existing set of semphores */
int semget(key_t key, int nsems, int flag);

/* atomic array of operations on a set of semphores */
int semop(int semid, struct sembuf *array, size_t nops);

/* control operations on a st of semaphores */
int semctl(int semid, int semnum, int cmd, ...);
```





## Example – Semaphore 2/4 (Synchronization Protocol)

- Example when the main (master) process waits for two other processes (slaves) become ready
  1. *Master* process suspend the execution and waits for two other processes *slaves* become ready
  2. *Slave* processes then wait to be released by the master process
- Proposed synchronization "protocol"
  - Define our way to synchronize the processes using the system semaphores
  - Slave process increments semaphore by 1
  - Slave process waits for the semaphore becomes 0 and then it is terminated
  - Master process waits for two slave processes and decrements the semaphore about 2
    - It must also ensure the semaphore value is not 0; otherwise slaves would be terminated prematurely
  - We need to use the atomic operations with the semaphore

lec07/sem-master.c lec07/sem-slave.c





## Example – Semaphore 3/4 (Master Process)

```c
int main(int argc, char* argv[])
{
   struct sembuf sem[2]; // structure for semaphore atomic operations
   int id = semget(1000, 1, IPC_CREAT | 0666); // create semaphore
   if (id != -1) {
      int r = semctl(id, 0, SETVAL, 0) == 0;

      sem[0].sem_num = 0;  // operation to acquire semaphore
      sem[0].sem_op = -2;  // once its value would be >= 2
      sem[0].sem_flg = 0;  // representing that two slaves are ready

      sem[1].sem_num = 0;  // the next operation in the atomic set
      sem[1].sem_op = 2;   // of operations increases the value of
      sem[1].sem_flg = 0;  // the semaphore about 2

      printf("Wait for semvalue >= 2\n");
      r = semop(id, sem, 2); // perform all operations atomically
      printf("Press ENTER to set semaphore to 0\n");
      getchar();
      r = semctl(id, 0, SETVAL, 0) == 0; // set the value of semaphore
      r = semctl(id, 0, IPC_RMID, 0) == 0; // remove the semaphore
   }
   return 0;
}
```

lec07/sem-master.c





## Example – Semaphore 4/4 (Slave Process)

```c
int main(int argc, char* argv[])
{
   struct sembuf sem;
   int id = semget(1000, 1, 0);
   int r;
   if (id != -1) {
      sem.sem_num = 0; // add the slave
      sem.sem_op = 1;  // to the "pool" of resources
      sem.sem_flg = 0;
      printf("Increase semafore value (add resource)\n");
      r = semop(id, &sem, 1);
      sem.sem_op = 0;
      printf("Semaphore value is %d\n", semctl(id, 0, GETVAL, 0));
      printf("Wait for semaphore value 0\n");
      r = semop(id, &sem, 1);
      printf("Done\n");
   }
   return 0;
}
```

lec07/sem-master.c

- The IPC entities can be listed by ipcs

```
clang sem-master.c -o sem-master
clang sem-slave.c -o sem-slave
```





## Issues with Semaphores

- The main issues are arising from a wrong usage
- Typical mistakes are:
  - 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*/
...
Wait(S1);
Wait(S2);
...
Signal(S2);
Signal(S1);
...
```

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





## Sensing Messages and Queues of Messages

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

### Example of System Calls

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/*Create a new message queue*/
int msgget(key_t key, int msgflg);

/* Send a message to the queue -- block/non-block (IPC_NOWAIT) */
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

/* Receive message from the queue -- block/non-block (IPC_NOWAIT) */
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

/* Control operations (e.g., destroy) the message queue */
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*





## Example – Messages Passing 1/4 (Synchronization, Master)

- Two processes are synchronized using messages
  1. The master process waits for the message from the slave process
  2. The master process informs slave to solve the task
  3. The slave process informs master about the solution
  4. The master processes sends message about termination

### Example of master process 1/2

```c
struct msgbuf {
   long mtype;
   char mtext[SIZE];
};

int main(int argc, char *argv[])
{
   struct msgbuf msg;
   int id = msgget(KEY, IPC_CREAT | 0666);
   int r;
   if (id != -1) {
```





## Example – Messages Passing 2/4 (Master)

### Example of master process 2/2

```c
      msg.mtype = 3; //type must be > 0
      printf("Wait for other process \n");
      r = msgrcv(id, &msg, SIZE, 3, 0);
      printf("Press ENTER to send work\n");
      getchar();
      strcpy(msg.mtext, "Do work");
      msg.mtype = 4; //work msg is type 4
      r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
      fprintf(stderr, "msgsnd r:%d\n",r);
      printf("Wait for receive work results\n",r);
      msg.mtype = 5;
      r = msgrcv(id, &msg, sizeof(msg.mtext), 5, 0);
      printf("Received message:%s\n", msg.mtext);
      printf("Press ENTER to send exit msg\n");
      getchar();
      msg.mtype = EXIT_MSG; //I choose type 10 as exit msg
      r = msgsnd(id, &msg, 0, 0);
   }
   return 0;
}
```

lec07/msg-master.c





## Example – Messages Passing 3/4 (Slave)

```c
int main(int argc, char *argv[])
{
   ...
   msg.mtype = 3;
   printf("Inform main process\n");
   strcpy(msg.mtext, "I'm here, ready to work");
   r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
   printf("Wait for work\n");
   r = msgrcv(id, &msg, sizeof(msg.mtext), 4, 0);
   printf("Received message:%s\n", msg.mtext);
   for (i = 0; i < 4; i++) {
      sleep(1);
      printf(".");
      fflush(stdout);
   } //do something useful
   printf("Work done, send wait for exit\n");
   strcpy(msg.mtext, "Work done, wait for exit");
   msg.mtype = 5;
   r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
   msg.mtype = 10;
   printf("Wait for exit msg\n");
   r = msgrcv(id, &msg, SIZE, EXIT_MSG, 0);
   printf("Exit message has been received\n");
```

lec07/msg-slave.c

## Example – Messages Passing 4/4 (Demo)

1. Execute the master process
2. Execute the slave process
3. Perform the computation
4. Remove the created message queue identified by the **msgid**

`#define KEY 1000`

`ipcrm -Q 1000`

```
% clang msg-master.c -o master
% ./master
Wait for other process
Slave msg received, press ENTER
     to send work msg

msgsnd r:0
Wait for receive work results
Received message:I'm going to
     wait for exit msg
Press ENTER to send exit msg

%ipcrm -Q 1000
%ipcrm -Q 1000
ipcrm: msqs(1000): : No such
     file or directory
%
```

```
% clang msg-slave.c -o slave
% ./slave
Inform main process
Wait for work
Received message:Do work
....done
Work done, send wait for exit
Wait for exit msg
Exit message has been received
%ipcs -q
Message Queues:
T ID    KEY  MODE     OWNER GROUP
q 65536 1000 -rw-rw- jf    jf
%
```

lec07/msg-master.c lec07/msg-slave.c





## Shared Memory

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

### Example of System Calls

```
/* obtain a shared memory identifier */
int shmget(key_t key, size_t size, int flag);

/* detach shared memory */
void* shmat(int shmid, const void *addr, int flag);

/*  detach shared memory */
int shmdt(const void *addr);

/*  shared memory control */
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

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





## Example – Shared Memory 1/4 (Write)

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

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SIZE 512

int main(int argc, char *argv[])
{
   char *buf;
   int id;
   if ((id = shmget(1000, SIZE, IPC_CREAT | 0666)) != -1)
    {
      if ( (buf = (char*)shmat(id, 0, 0)) ) {
         fgets(buf, SIZE, stdin);
         shmdt(buf);
      }
   }
   return 0;
}
```

lec07/shm-write.c





## Example – Shared Memory 2/4 (Read)

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

```
#include <sys/types.h>
#include <sys/shm.h>
#include <stdio.h>

#define SIZE 512

int main(int argc, char *argv[])
{
   int id;
   char *buf;
   if ((id = shmget(1000, 512, 0)) != -1) {
      if ((buf = (char*)shmat(id, 0, 0)) ) {
         printf("mem:%s\n", buf);
      }
      shmdt(buf);
   } else {
      fprintf(stderr, "Cannot access to shared memory!\n");
   }
   return 0;
}
```

lec07/shm-read.c





## Example – Shared Memory 3/4 (Demo)

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

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

```
% clang -o shm-write shm-write.c
% ./shm-write
Hello! I like programming in C!
```

```
% clang -o shm-read shm-read.c
% ./shm-read
mem:Hello! I like programming in
     C!

% ./shm-read
mem:Hello! I like programming in
     C!

% ipcrm -M 1000
% ./shm-read
Cannot access to shared memory!
```

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





## Example – Shared Memory 4/4 (Status)

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

```
after creating shared memory segment and before writing the text
m        65539         1000 --rw-rw-rw-   jf    jf    jf    jf
               1         512          1239         1239 22:18:48
     no-entry 22:18:48

after writing the text to the shared memory
m        65539         1000 --rw-rw-rw-   jf    jf    jf    jf
               0         512          1239         1239 22:18:48
     22:19:37 22:18:48

after reading the text
m        65539         1000 --rw-rw-rw-   jf    jf    jf    jf
               0         512          1239         1260 22:20:07
     22:20:07 22:18:48
```





## Massive parallelism using graphics cards

- Image rendering performed pixel-by-pixel can be easily parallelized
- Graphics Processing Units (GPU) has similar (or even higher) degree of integration with the main processors (CPU)
- They have huge number of parallel processors

  *E.g., GeForce GTX 1060 ∼ 1280 cores*
- The computational power can also be used in another applications
  - 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 |
| Radeon HD 4670 | 480 GigaFLOPs |

*Peak catalogue values*

- Main processors :

| | |
|---|---|
| Phenom X4 9950 (@2.6 GHz) | 21 GigaFLOPs |
| Core 2 Duo E8600 (@3.3 GHz) | 22 GigaFLOPs |
| Cure 2 Quad QX9650 (@3.3 GHz) | 35 GigaFLOPs |
| Cure 2 Quad QX9650 (@3.3 GHz) | 35 GigaFLOPs |
| Core i7 970 (@3.2 GHz) | 42 GigaFLOPs |

*Test linpack 32-bit*

- Is the reported power really achievable?

  *(float vs double)*
- How about other indicators

  *E.g., computational power / power consumption*
  - CSX700 has typical power consumption around 9W





## CUDA

- NVIDIA Compute Unified Device Architecture.
- Extension of the C to access to 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

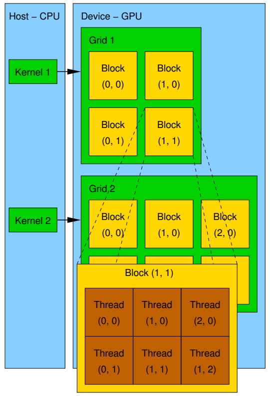
- Kernel (computation) is divided into blocks
- Each block represent a parallel computation of the part of the result
  *E.g., a part of the matrix multiplication*
- Each block consists of computational threads
- Parallel computations are synchronization within the block
- Blocks are organized into the **grid**
- Scalability is realized by dividing the computation into blocks

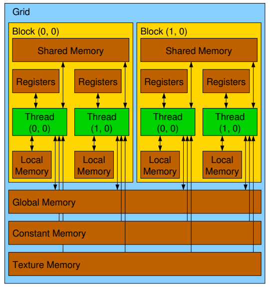
*Blocks may not be necessarily computed in parallel; based on the available number of parallel units, particular blocks can be computed sequentially*





# CUDA – Grid, Blocks, Threads, and Memory Access




- Access time to the memory
- Collisions for simultaneous access of several threads





# CUDA – Example – Matrix Multiplication 1/8

- NVIDIA CUDA SDK - Version 2.0, `matrixMul`
- Simple matrix multiplication
  - $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$
  - Matrices have identical dimensions $n \times n$
  - where $n$ is the multiple of the block size
- Comparison
  - naïve implementation in C ($3\times$ *for loop*),
  - naïve 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

## Naïve implementation

```
void simple_multiply(const int n,
      const float *A, const float *B, float *C)
{
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      float prod = 0;
      for (int k = 0; k < n; ++k) {
        prod += A[i * n + k] * B[k * n + j];
      }
      C[i * n + j] = prod;
    }
  }
}
```





# CUDA – Example – Matrix Multiplication 3/8

## Naïve implementation with transpose

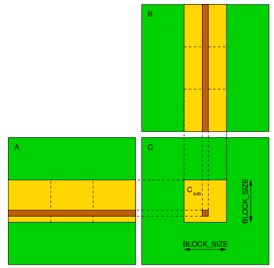
```
void simple_multiply_trans(const int n,
      const float *a,  const float *b, float  *c)
{
  float * bT = create_matrix(n);
  for (int i = 0; i < n; ++i) {
    bT[i*n + i] = b[i*n + i];
    for (int j = i + 1; j < n; ++j) {
      bT[i*n + j] = b[j*n + i];
      bT[j*n + i] = b[i*n + j];
    }
  }
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      float tmp = 0;
      for (int k = 0; k < n; ++k) {
        tmp += a[i*n + k] * bT[j*n + k];
      }
      c[i*n + j] = tmp;
    }
  }
  free(bT);
}
```





# 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

```
void cuda_multiply(const int n,
      const float *hostA, const float *hostB, float *hostC)
{
 const int size = n * n * sizeof(float);
 float *devA, *devB, *devC;

 cudaMalloc((void**)&devA, size);
 cudaMalloc((void**)&devB, size);
 cudaMalloc((void**)&devC, size);

 cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);
 cudaMemcpy(devB, hostB, size, cudaMemcpyHostToDevice);

 dim3 threads(BLOCK_SIZE, BLOCK_SIZE);  // BLOCK_SIZE == 16
 dim3 grid(n / threads.x, n /threads.y);

 // Call kernel function matrixMul
 matrixMul<<<grid, threads>>>(n, devA, devB, devC);

 cudaMemcpy(hostC, devC, size, cudaMemcpyDeviceToHost);

 cudaFree(devA);
 cudaFree(devB);
 cudaFree(devC);
}
```





# CUDA – Example – Matrix Multiplication 6/8

## CUDA implementation – kernel function

```
__global__ void matrixMul(int n, float* A, float* B, float* C) {
  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int aBegin = n * BLOCK_SIZE * by;  //beginning of sub-matrix in the block
  int aEnd = aBegin + n - 1; //end of sub-matrix in the block
  float Csub = 0;
  for (
       int a = aBegin, b = BLOCK_SIZE * bx;
       a <= aEnd;
       a += BLOCK_SIZE, b += BLOCK_SIZE * n
     ) {
     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE]; // shared memory within
     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE]; // the block
     As[ty][tx] = A[a + n * ty + tx]; // each thread reads a single element
     Bs[ty][tx] = B[b + n * ty + tx]; // of the matrix to the memory
     __syncthreads(); // synchronization, sub-matrix in the shared memory

     for (int k = 0; k < BLOCK_SIZE; ++k) { // each thread computes
        Csub += As[ty][k] * Bs[k][tx]; // the element in the sub-matrix
     }
     __syncthreads();
  }
  int c = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + n * ty + tx] = Csub; // write the results to memory
}
```





# 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);
}
```

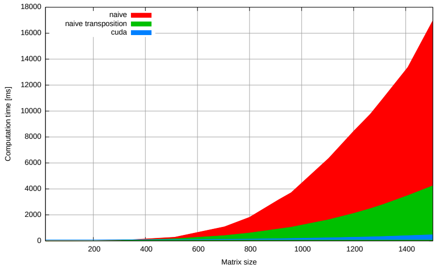
2. Compile the CUDA code to the C++ code

```
nvcc --cuda cuda_func.cu -o cuda_func.cu.cc
```

3. Compilation of the `cuda_func.cu.cc` file using standard compiler

## CUDA – Example – Matrix Multiplication 8/8

### Computational time (in milliseconds




| N | Naïve | Transp. | CUDA |
|---|---|---|---|
| 112 | 2 | 1 | 81 |
| 208 | 11 | 11 | 82 |
| 304 | 35 | 33 | 84 |

| N | Naïve | Transp. | CUDA |
|---|---|---|---|
| 704 | 1083 | 405 | 122 |
| 1104 | 6360 | 1628 | 235 |
| 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.





# Summary of the Lecture





## Topics Discussed

- Introduction to Parallel Programming
  - Ideas and main architectures
  - Program and process in OS
- Parallel processing
- Sychronization and Inter-Process Communication (IPC)
  - Semaphores
  - Messages
  - Shared memory
- Parallel processing on graphics cards
- Next: Multithreading programming