MULTIPROCESSOR PROGRAMMING, DV2597/DV2606 — LAB 1: THREADS PROGRAMMING

Håkan Grahn
Blekinge Institute of Technology

2022 fall
updated: 2022-10-07

The objective of this laboratory assignment is to introduce basic pthreads programming, and give some experience of how work and data partitioning as well as communication and synchronization impact the performance of parallel applications on a shared address-space computer.

The laboratory assignments should be conducted, solved, implemented, and presented in groups of two students! It is ok to do them individually, but groups of more than two students are not allowed.

Home Assignment 1. Preparations

Read through these laboratory instructions and do the **home assignments**. The purpose of the home assignments is to do them *before* the lab sessions to prepare your work and make more efficient use of the lab sessions.

Your are expected to have acquired the following knowledge before the lab sessions:

- You are supposed to have good programming experience and have good knowledge of C / C++ programming.
- You are supposed to have basic knowledge about working in a Unix/Linux environment.
- You should be familiar with common operating systems concepts and issues, see for example [3].

Read the following sections in the course book [2]:

- Chapters 2.1-2.4, 2.7 (Parallel Computer Architecture)
- Chapter 3 (Parallel Programming Models)
- Chapters 4.1-4.2 (Performance Analysis of Parallel Programs)
- Chapter 6.1 (Programming with Pthreads)
- Chapters 8.1, 8.3, 8.5 (Algorithms for Systems of Linear Equations)

End of home assignment 1.

Home Assignment 2. Plagiarism and collaboration

You are encouraged to work in groups of two. Groups larger than two are not accepted.

Discussions between laboratory groups are positive and can be fruitful. It is normally not a problem, but watch out so that you do not cross the border to cheating. For example, you are *not allowed* to share solution approaches, solutions to the different tasks, source code, output data, results, etc.

The submitted solution(s) to the laboratory assignments and tasks, should be developed by the group members only. You are not allowed to copy code from somewhere else, i.e., neither from your student fellows nor from the internet or any other source. The only other source code that is allowed to use is the one provided with the laboratory assignment.

End of home assignment 2.

1 Introduction

A computer system consists of one or several processors. Today, most processors are so called multicore processors, i.e., the processor contains several cpu cores internally. In larger systems, several multicore processors are connected either through a shared memory or as nodes in a distributed system.

In this laboratory assignment we will work with shared-memory multiprocessors, i.e., all processor cores communicate through a shared-memory model, We will study how we can divide the work of a program using threads, how they share data and information, and how we synchronize between threads. Further, focus will be how to parallelize a single program, i.e., divide the work among several processor cores, in order to achieve high performance.

The lab is divided into two parts:

- **Part 1:** An introduction / tutorial part (Section 4) covering basic concepts and basic programming of pthreads (POSIX threads). This part is mainly intended for those students who are not familiar with pthreads from previous courses. If you are very familiar with pthreads since before, you may skip some of these tasks as you see fit. No tasks in this part need to be submitted.
- **Part 2:** A parallel programming part (Section 5) covering how to parallelize two algorithms in order to achieve high performance. **The tasks in this part are to be submitted**.

The rationale behind using pthreads is that pthreads is a common approach to implement high-performance parallel applications on shared address-space computers. The thread model in many programming languages, e.g., C++ and Java, is highly influenced of how pthreads work. On shared-memory computers, data communication is implicit but may still impact the performance. Further, the synchronization of parallel threads and the protection of shared data may also have a significant impact on performance. In this lab, the goal is to parallelize and optimize the programs so they efficiently utilize the available processor cores.

The programs in this lab assignment are tested on Ubuntu Linux, and thus work on our lab server. However, they should also work on any Linux distribution you may have at home. The source code files and programs necessary for the laboratory are available on Canvas.

If nothing else is said/written, you shall compile your programs using:

```
gcc -O2 -o output_file_name sourcefile1.c ... sourcefileN.c -lpthread
```

When you compile and link a parallel program written using pthreads, you shall add "`-lpthread`" to the compile command in order to link the pthreads library to your application. All commands are written in a Linux terminal or console window.

The computer that we will use in the laboratory is called *jane.lab.bth.se* and is a Linux/Ubuntu server with a 16-core processor and 64GB main memory. You will have a new user account on the machine. The computer, i.e., *jane*, is not openly accessible from outside BTH. In order to login to the machine, you need to either (i) login from a computer already on the BTH network, e.g., a computer in the lab room, or (ii) first login remotely to BTH using SSH or VPN, and then login on *jane*.

The source code files for the laboratory are available on Canvas.

2 Examination and grading

Note: Present and discuss your solutions with a teacher / lab assistant when all tasks are done, i.e., before submitting on Canvas.

When you have discussed your solutions orally, prepare and submit a `tar`-file or `zip`-file containing:

- **Source code:** The source-code for working solutions to the tasks in Section 5. Specifically, you shall submit your well-commented source code for **Task 10** and **Task 11**.
- Corresponding `Makefile`(s), or a text-file describing how to compile the respective projects / tasks.
- **Written report:** You should write a short report (approximately 2-3 pages) describing, *for each task*, your implementations, the answers to the questions in the tasks, and your measurements (results), as outline below. The format of the report must be pdf.

- **Implementation:** A general description of your parallel implementations, i.e., you should describe how you have partitioned the work between the threads and cpus, how the data structures are organized, etc.
- **Measurements:** You should provide execution times for two cases: (i) the sequential version of the application, and (ii) the parallel version of the application running on 16 cpus (i.e., cores), using as many threads as you like.
  The data set sizes for the tasks are: Gaussian elimination using a 2048×2048 matrix, and Quicksort using an array with $64 * 2^{20}$ items.
  Further, analyze and discuss your results.

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

The examiner may contact you within a week, if he/she needs some oral clarifications on your code or report. In this case, all group members must be present at that oral occasion.

3 Parallelism and performance

Threads and processes are often used to introduce concurrency, i.e., the ability to do multiple things at the same time. There can be many reasons for that, e.g., handling independent requests to a server or handling different tasks such as spell checking and auto backup at the same time in a word processing application. However, another important reason to introduce concurrency is performance. When a concurrent program execute on multiple cores, it is said to execute in parallel. Ideally, if a parallel program execute on two cores it could run twice a fast, i.e., we can have a speedup of 2. Speedup is defined as:

$$Speedup = \frac{T_{sequential}}{T_{parallel}} \tag{1}$$

where $T_{sequential}$ is the execution time of sequential program and $T_{parallel}$ is the execution time of parallel program.

Unfortunately, it is often very hard to get as high speedup as we have cores running the parallel program. When we have as high speedup as we have cores, i.e., speedup of 4 on four cores, we call it *linear speedup*. However, in most cases the speedup is sub-linear, i.e., less than the ideal one. There are many reasons for that, e.g., sequential parts of the application, synchronization overhead, scheduling overhead, etc. In the following sections we will study two different applications, *matrix multiplication* and *Quicksort*. The first one is relatively easy to get close to linear speedup, while the second one is much harder.

When measuring the execution time, it looks a bit different if you have `bash` och `tcsh` as default shell in your terminal / console. In the lab room (G332) `tcsh` is default, but in a standard Ubuntu/Linux distribution `bash` is default. You usually can find out which shell you run by executing the command (if the environment variable `SHELL` is set correctly, which is unfortunately not always done):

```
echo $SHELL
```

If you are running `bash`, you should measure the execution time with:

```
time command
```

The value of "real" is the wall clock time that we are interested in.

If you are running `tcsh`, you should measure the execution time with:

```
/usr/bin/time command
```

The value of "elapsed" is the wall clock time that we are interested in.

4 Threads programming tutorial

Threads are different to processes in how they share resources such as memory and files. A process can contain one or several threads. For example, a sequential program is executed in one process with only one single executing thread. In this laboratory we will use POSIX Threads, or pthreads for short.

4.1 Basic pthreads programming

Home Assignment 3. Thread creation

Read the manual pages for the following functions:

- `pthread_create()`
- `pthread_exit()`
- `pthread_join()`

Especially, make sure you understand the different arguments (input parameters) to the different functions.

End of home assignment 3.

In the first thread example, see Listing 1 (`pthreadcreate.c`), we will study how threads are created and terminated.

Task 1. `pthread_create()` example

Compile and execute the program in Listing 1 (`pthreadcreate.c`). Remember the `-lpthread` flag to the compiler.

Make sure you understand what it does and why.

What does the program print when you execute it?

End of task 1.

Sometimes we would like to send arguments (input parameters) to a thread we create, and sometimes we also want the thread to return some data. A tricky thing is that a thread can only have *one* argument and that argument should be of type `void*`.

In the example in Listing 2 (`pthreadcreate2.c`), we show an example of how we can pass arguments to a thread when we create the thread. Two data items, `id` and `numThreads` are put in a `struct`, and then a `void*` pointer is set to point to that struct, and finally, the `void*` pointer is passed to the thread function. In the child threads, the input parameter is typecasted back to the `struct` type, and then we can use the different values passed to the child thread.

Task 2. `pthread_create()` example 2

Compile and execute the program in Listing 2 (`pthreadcreate2.c`).

Make sure you understand what it does and why.

Why do we need to create a new `struct threadArgs` for each thread we create?

End of task 2.

When we want to return values from a child thread, we cannot simply execute a `return` statement as in a normal function, since the thread function must have return type `void`. A solution is to return values to the parent thread through the same `struct` as the input parameters were passed. In this situation, we cannot allow the child to deallocate the `struct` where the arguments are passed to the child thread, as in Task 2 (Listing 2). Instead, the parent thread is responsible for both allocating and deallocating the argument `struct`.

Task 3. `pthread_create()`, how to handle return values

In Listing 3 (`pthreadcreate3.c`), we provide a program where the parent thread allocates (and deallocates) an array of `struct`s of type `threadArgs`, one `struct` for each child thread. The `threadArgs struct` passed to each child thread can then be used to pass values from the child thread to the parent thread.

Your task is to:

- Add a new variable in the struct `threadArgs`, called `squaredId`.
- In the child thread, take the thread id, square it, and return the value. Use the `threadArgs struct` for communicating values between the parent and the child.
- In the main program, when all threads have terminated, print the squared id for each thread (this value should be returned by each thread).

End of task 3.

4.2 Thread communication and synchronization

Since all threads in the same process share memory, they can easily communicate with each other by reading and writing data to variables in the shared memory. However, as in the case for processes, we need to protect the shared variables from simultaneous modification by different threads. A mechanism the achieve this protection is mutex variables. Therefore, we will in this part of the laboratory study how mutex synchronization works.

Home Assignment 4. Thread synchronization

Read the manual pages for the following functions:

- `pthread_mutex_init()`
- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`

End of home assignment 4.

Task 4. Bank account

In Listing 4 (`bankaccount.c`) you find a simple simulation of a bank account where deposits and withdrawals are done. Each thread does either 1000 deposits (odd thread id ⇒ calls `deposit()`) or 1000 withdrawals (even thread id ⇒ calls `withdraw()`).

If everything goes well, the saldo at the end of the execution should be 0, given an even number of threads.

Execute the program a number of times (with an even number of threads, e.g., 8, 16, 32, and 64 threads).

Does the program execute correctly? Why/why not?

End of task 4.

The problem that occur is that two (or more) threads try to read and update the same variable at the same time, and their accesses to the shared variable (in this case `bankAccountBalance`) are interleaved. When updating a variable, a thread reads the value of the variable from the memory, updates the value (a local operation in a processor register), and finally writes the new value back to memory. When two threads do this concurrently, the execution may look like the one in Figure 1 and is called a *race condition*. Note that this situation may occur also on single-core processors, e.g., if an interrupt and process context switch happens between line 1 and 2. Programs with data races may have unpredictable behavior, and can cause bugs that are very hard to find and reproduce (so called "Hiesenbugs").

Time	Thread 1 (on cpu1)		Thread 2 (on cpu2)	
1	proc_reg = mem_var	(reg_value == 1)		
2			proc_reg = mem_var	(reg_value == 1)
3	proc_reg++	(reg_value == 2)	proc_reg++	(reg_value == 2)
4	mem_var = proc_reg	(mem_value == 2)		
5			mem_var = proc_reg	(mem_value == 2)

Figure 1: Possible execution when two threads try to update the same variable `mem_var` concurrently, assuming that `mem_var==1` before the execution. The situation is referred to as a *race condition*. The correct value of the code sequence should be 3, not 2.

The solution to the problem in Figure 1 is to enclose accesses to shared variable in critical sections. Then we can guarantee mutual exclusion, i.e., only one thread at the time can access and modify a shared variable.

Task 5. Bank account, correction

Correct the program in Listing 4 (`bankaccount.c`) by introducing proper synchronizations on shared variables.

The shared variable(s) that are protected and accessed in critical sections should be locked as short time as possible, i.e., you are supposed to keep the critical sections are short os possible.

End of task 5.

4.3 Parallelization of a simple application

We start by parallelizing a simple application, *matrix multiplication*. A sequential version of the program is found in the file `matmulseq.c`, see Listing 5.

Task 6. Parallel matrix multiplication

Compile and execute the program `matmulseq.c`.

How long time did it take to execute the program?

We will now parallelize the matrix multiplication program using threads. Parallelize the program according to the following assumptions:

- Parallelize only the matrix multiplication, but not the initialization.
- A new thread shall be created for each row to be calculated in the matrix.
- The row number of the row that a thread shall calculate, shall be passed to the new thread as a parameter to the new thread.
- The main function shall wait until all threads have terminated before the program terminates.

Compile and link your parallel version of the matrix multiplication program. Don't forget to add `-lpthread` when you link your program. We will now measure how much faster the parallel program is as compared to the sequential one.

Execute the program and measure the execution time.

How long was the execution time of your parallel program?

Which speedup did you get (as compared to the execution time of the sequential version,
where $Speedup = T_{sequential}/T_{parallel}$)?

End of task 6.

A performance liming factor when writing and executing parallel applications is if there is any sequential execution path in the parallel program. For example, critical sections introduce sequential parts in a program, although they are necessary for the correctness of the program. In our matrix multiplication example, we have a sequential part in the initialization of the matrices.

Task 7. Parallelization of the initialization

Parallelize the initialization of the matrices also, i.e., the function `init_matrix`. Use one thread to initialize each of the rows in the matrices `a` and `b`. Compile, link, and execute your program.

How fast did the program execute now?

Did the program run faster or slower now, and why?

End of task 7.

In the previous tasks we have had a large number of threads (1024) to perform the matrix multiplication. An alternative would be to let each thread calculate several rows in the resulting matrix. This will then result in a larger granularity, i.e., more work is done per thread. An advantage is then that the scheduling overhead will decrease (fewer threads to administrate and schedule), but a disadvantage is that the amount of parallelism is lower which may result in lower speedup.

Task 8. Changing granularity

Rewrite your program so it only creates as many threads as there are processors to execute on, e.g., when we run the program on 16 processors (cores), we create 16 threads that each calculates $1024/16 = 256$ rows in the matrix. Hardcode the number of threads to use. Execute your program and measure the execution time.

Which is the execution time and speedup for the application now?

Do these execution times differ from those in Task 6? If so, why? If not, why?

Does it make any difference now if `init_matrix` is parallelized or not?

End of task 8.

4.4 Effects of false sharing

In a multiprocessor with shared memory, the different threads communicate by reading and writing to shared variables. The main memory is divided into blocks and these blocks can be moved closer to the processor by copying them to the cache memory. In a multiprocessor can several processors have a local copy of the same memory block. In order to guarantee that the memory content is consistent when one processors updates its copy, all other copies are invalidated upon the write. The mechanism that handles this is cancelled a cache coherence protocol.

The invalidation of cached memory blocks as a result of writes can have unexpected effects on the performance. In the file `false_sharing.c` (see Listing 6) you find a program with three global variables, `a`, `b`, `c`. These variables are updated a number of times of three concurrent threads, i.e., thread 1 (`inc_a`) updates variable `a`, thread 2 (`inc_b`) updates variable `b`, and thread 3 (`inc_c`) updates variable `c`.

Task 9. False sharing example

NOTE: Compile the program with `gcc` only, i.e., **no** optimization at all! Optimization removes the interesting effects in this simple example.

How fast did the program run (execution time)?

Rewrite the program so that the variable `a` is local in the function `inc_a`.

How fast did the program run this time?

Rewrite the program so that the variable `b` is local in the function `inc_b`.

How fast did the program run this time?

Finally, rewrite the program so that the variable `c` is local in the function `inc_c`.

How fast did the program run this time?

Which variables were probably allocated in the same memory block?

Did the execution time decrease when the last variable (`c`) became local? Why/why not?

End of task 9.

5 Thread assignments

This section presents the tasks to be solved and submitted for for grading for this lab assignment. You are going to parallelize two sequential algorithms using pthreads, i.e., Gaussian elimination (Section 5.1) and Quicksort (Section 5.2).

5.1 Gaussian elimination

The problem to be solved in this task is to implement a parallel version of Gaussian elimination using pthreads. The algorithm is described in Section 8.3 in [1] (presented as Algorithm 8.4) and Section 8.1 in [1]. You shall initialize the matrix `A`, and the vectors `y` and `b` with reasonable values (done by default).

```
1.   procedure GAUSSIAN_ELIMINATION ($A$, $b$, $y$)
2.   begin
3.      for $k := 0$ to $n - 1$ do              /* Outer loop */
4.      begin
5.         for $j := k + 1$ to $n - 1$ do
6.            $A[k, j] := A[k, j]/A[k, k]$;  /* Division step */
7.         $y[k] := b[k]/A[k, k]$;
8.         $A[k, k] := 1$;
9.         for $i := k + 1$ to $n - 1$ do
10.        begin
11.           for $j := k + 1$ to $n - 1$ do
12.              $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$; /* Elimination step */
13.           $b[i] := b[i] - A[i, k] \times y[k]$;
14.           $A[i, k] := 0$;
15.        endfor;          /* Line 9 */
16.     endfor;             /* Line 3 */
17.  end GAUSSIAN_ELIMINATION
```

Figure 2: Serial Gaussian elimination algorithm (Algorithm 8.4 in [1]).

The code for a sequential implementation of Gaussian elimination is found in Appendix A (Listing 7). The code is based on Algorithm 8.4 in Section 8.3 in [1].

Task 10. Parallel Gaussian elimination

You are supposed to do the following:

- Write a parallel implementation of gaussian elimination using pthreads.
- Measure the speedup of your parallel version on 16 cpus (i.e., cpu cores) using a 2048×2048 matrix.
- The resulting pthreads implementation shall have a speedup of at least 1.5 (on 16 cpus) over the sequential version.

End of task 10.

5.2 Parallel Quicksort implementation

The final application we shall parallelize is *Quicksort*. We saw earlier in the laboratory exercise that matrix multiplication is relatively easy to parallelize using data decomposition and extracting so called loop parallelism.

In contrast, Quicksort fits well for recursive decomposition, i.e., when doing a recursive call we create a new thread for that recursive call. Challenges here are, most related to work granularity vs thread overhead, e.g., how many threads shall you create, which is the smallest sequential chunk (i.e., when shall you stop creating new threads and execute the recursive call sequentially), etc.

A sequential version of Quicksort is found in the file `qsortseq.c`, see Listing 8.

Task 11. Parallel Quicksort

Write a parallel version of Quicksort using pthreads.

Measure and compare the execution times for

(i) the sequential version given to you,
(ii) your parallel version running with 4, 8, and 16 threads, and finally
(iii) your parallel version running with as many threads as you like.

Further, calculate the speedup of your parallel application in cases (ii) and (iii).

Your application shall have a speedup of at least 4 on 16 cpus (cores).

End of task 11.

References

[1] A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Introduction to Parallel Computing, 2nd ed.," *Addison-Wesley*, 2003, ISBN 0-201-64865-2.

[2] T. Rauber and G. Rünger, "Parallel Programming for Multicore and Cluster Systems, 2nd ed.," *Springer*, 2013, ISBN 978-3-642-37800-3.

[3] Andrew S. Tanenbaum and Herbert Bos, "Modern Operating Systems, 4th ed," *Pearson Education Limited*, ISBN-10: 0-13-359162-X, 2015.

A Program listings

Listing 1. `pthreadcreate.c`

```
#include <stdio.h>
#include <pthread.h> // pthread types and functions

void* child() {
    printf("This is the child thread.\n");
}

int main(int argc, char** argv) {
    pthread_t thread; // struct for child−thread info
    // spawn thread:
    pthread_create(&thread, // the handle for it
        NULL, // its attributes
        child, // the function it should run
        NULL); // args to that function

    printf("This is the parent (main) thread.\n");
    pthread_join(thread, NULL); // wait for child to finish
    return 0;
}
```

Listing 2. `pthreadcreate2.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct threadArgs {
    unsigned int id;
    unsigned int numThreads;
};

void* child(void* params) {
    struct threadArgs *args = (struct threadArgs*) params;
    unsigned int childID = args−>id;
    unsigned int numThreads = args−>numThreads;
    printf("Greetings from child #%u of %u\n", childID, numThreads);
    free(args);
}

int main(int argc, char** argv) {
    pthread_t* children; // dynamic array of child threads
    struct threadArgs* args; // argument buffer
    unsigned int numThreads = 0;
    // get desired # of threads
    if (argc > 1)
        numThreads = atoi(argv[1]);
    children = malloc(numThreads * sizeof(pthread_t)); // allocate array of handles
    for (unsigned int id = 0; id < numThreads; id++) {
        // create threads
        args = malloc(sizeof(struct threadArgs));
        args−>id = id;
        args−>numThreads = numThreads;
        pthread_create(&(children[id]), // our handle for the child
            NULL, // attributes of the child
            child, // the function it should run
            (void*)args); // args to that function
    }
    printf("I am the parent (main) thread.\n");
    for (unsigned int id = 0; id < numThreads; id++) {
        pthread_join(children[id], NULL );
    }
    free(children); // deallocate array
    return 0;
}
```

Listing 3. `pthreadcreate3.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct threadArgs {
    unsigned int id;
    unsigned int numThreads;
};

void* child(void* params) {
    struct threadArgs *args = (struct threadArgs*) params;
    unsigned int childID = args->id;
    unsigned int numThreads = args->numThreads;
    printf("Greetings from child #%u of %u\n", childID, numThreads);
}

int main(int argc, char** argv) {
    pthread_t* children; // dynamic array of child threads
    struct threadArgs* args; // argument buffer
    unsigned int numThreads = 0;
    // get desired # of threads
    if (argc > 1)
        numThreads = atoi(argv[1]);
    children = malloc(numThreads * sizeof(pthread_t)); // allocate array of handles
    args = malloc(numThreads * sizeof(struct threadArgs)); // args vector
    for (unsigned int id = 0; id < numThreads; id++) {
        // create threads
        args[id].id = id;
        args[id].numThreads = numThreads;
        pthread_create(&(children[id]), // our handle for the child
            NULL, // attributes of the child
            child, // the function it should run
            (void*)&args[id]); // args to that function
    }
    printf("I am the parent (main) thread.\n");
    for (unsigned int id = 0; id < numThreads; id++) {
        pthread_join(children[id], NULL );
    }
    free(args); // deallocate args vector
    free(children); // deallocate array
    return 0;
}
```

Listing 4. `bankaccount.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Shared Variables
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
double bankAccountBalance = 0;

void deposit(double amount) {
    bankAccountBalance += amount;
}

void withdraw(double amount) {
    bankAccountBalance -= amount;
}

// utility function to identify even-odd numbers
unsigned odd(unsigned long num) {
    return num % 2;
}

// simulate id performing 1000 transactions
void do1000Transactions(unsigned long id) {
    for (int i = 0; i < 1000; i++) {
        if (odd(id))
            deposit(100.00); // odd threads deposit
        else
            withdraw(100.00); // even threads withdraw
    }
}

void* child(void* buf) {
    unsigned long childID = (unsigned long)buf;
    do1000Transactions(childID);
    return NULL;
}

int main(int argc, char** argv) {
    pthread_t *children;
    unsigned long id = 0;
    unsigned long nThreads = 0;
    if (argc > 1)
        nThreads = atoi(argv[1]);
    children = malloc( nThreads * sizeof(pthread_t) );
    for (id = 1; id < nThreads; id++)
        pthread_create(&(children[id-1]), NULL, child, (void*)id);
    do1000Transactions(0); // main thread work (id=0)
    for (id = 1; id < nThreads; id++)
        pthread_join(children[id-1], NULL);
    printf("\nThe final account balance with %lu threads is $%.2f.\n\n", nThreads, bankAccountBalance);
    free(children);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Listing 5. `matmulseq.c`

```
/***************************************************************************
 *
 * Sequential version of Matrix−Matrix multiplication
 *
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define SIZE 2048

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void
init_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
            /* Simple initialization, which enables us to easy check
             * the correct answer. Each element in c will have the same
             * value as SIZE after the matmul operation.
             */
            a[i][j] = 1.0;
            b[i][j] = 1.0;
        }
}

static void
matmul_seq()
{
    int i, j, k;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            c[i][j] = 0.0;
            for (k = 0; k < SIZE; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

static void
print_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++)
            printf(" %7.2f", c[i][j]);
        printf("\n");
    }
}

int
main(int argc, char **argv)
{
    init_matrix();
    matmul_seq();
    //print_matrix();
}
```

Listing 6. `falsesharing.c`

```
/***************************************************************************
 *
 * false_sharing.c
 *
 * A simple program to show the performance impact of false sharing
 *
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define LOOPS (2000*1000000)

volatile static int a, b, c;

static void *
inc_a(void *arg)
{
    int i;
    //int a;
    printf("Create inc_a\n");
    while (i++ < LOOPS) a++;
    pthread_exit(0);
}

static void *
inc_b(void *arg)
{
    int i;
    //int b;
    printf("Create inc_b\n");
    while (i++ < LOOPS) b++;
    pthread_exit(0);
}

static void *
inc_c(void *arg)
{
    int i;
    //int c;
    printf("Create inc_c\n");
    while (i++ < LOOPS) c++;
    pthread_exit(0);
}

int
main(int argc, char **argv)
{
    int rc, t;
    pthread_t tid_a, tid_b, tid_c;

    rc = pthread_create(&tid_a, NULL, inc_a, (void *)t);
    rc = pthread_create(&tid_b, NULL, inc_b, (void *)t);
    rc = pthread_create(&tid_c, NULL, inc_c, (void *)t);
    pthread_join(tid_a, NULL);
    pthread_join(tid_b, NULL);
    pthread_join(tid_c, NULL);
}
```

Listing 7. `gaussianseq.c`

```
/***************************************************************************
 *
 * Sequential version of Gaussian elimination
 *
 ***************************************************************************/

#include <stdio.h>

#define MAX_SIZE 4096

typedef double matrix[MAX_SIZE][MAX_SIZE];

int N;  /* matrix size  */
int maxnum;  /* max number of element*/
char *Init;  /* matrix init type */
int PRINT;  /* print switch  */
matrix A;  /* matrix A  */
double b[MAX_SIZE]; /* vector b */
double y[MAX_SIZE]; /* vector y */

/* forward declarations */
void work(void);
void Init_Matrix(void);
void Print_Matrix(void);
void Init_Default(void);
int Read_Options(int, char **);

int
main(int argc, char **argv)
{
    int i, timestart, timeend, iter;

    Init_Default();  /* Init default values */
    Read_Options(argc,argv); /* Read arguments */
    Init_Matrix();  /* Init the matrix */
    work();
    if (PRINT == 1)
        Print_Matrix();
}

void
work(void)
{
    int i, j, k;

    /* Gaussian elimination algorithm, Algo 8.4 from Grama */
    for (k = 0; k < N; k++) { /* Outer loop */
        for (j = k+1; j < N; j++)
            A[k][j] = A[k][j] / A[k][k]; /* Division step */
        y[k] = b[k] / A[k][k];
        A[k][k] = 1.0;
        for (i = k+1; i < N; i++) {
            for (j = k+1; j < N; j++)
                A[i][j] = A[i][j] - A[i][k]*A[k][j]; /* Elimination step */
            b[i] = b[i] - A[i][k]*y[k];
            A[i][k] = 0.0;
        }
    }
}

void
Init_Matrix()
{
    int i, j;

    printf("\nsize = %dx%d ", N, N);
    printf("\nmaxnum = %d \n", maxnum);
    printf("Init = %s \n", Init);
    printf("Initializing matrix...");

    if (strcmp(Init,"rand") == 0) {
        for (i = 0; i < N; i++){
            for (j = 0; j < N; j++) {
                if (i == j) /* diagonal dominance */
                    A[i][j] = (double)(rand() % maxnum) + 5.0;
                else
                    A[i][j] = (double)(rand() % maxnum) + 1.0;
            }
        }
    }
    if (strcmp(Init,"fast") == 0) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                if (i == j) /* diagonal dominance */
                    A[i][j] = 5.0;
                else
                    A[i][j] = 2.0;
            }
        }
    }

    /* Initialize vectors b and y */
    for (i = 0; i < N; i++) {
        b[i] = 2.0;
        y[i] = 1.0;
    }

    printf("done \n\n");
    if (PRINT == 1)
        Print_Matrix();
}

void
Print_Matrix()
{
    int i, j;

    printf("Matrix A:\n");
    for (i = 0; i < N; i++) {
        printf("[");
        for (j = 0; j < N; j++)
            printf(" %5.2f,", A[i][j]);
        printf("]\n");
    }
    printf("Vector b:\n[");
    for (j = 0; j < N; j++)
        printf(" %5.2f,", b[j]);
    printf("]\n");
    printf("Vector y:\n[");
    for (j = 0; j < N; j++)
        printf(" %5.2f,", y[j]);
    printf("]\n");
    printf("\n\n");
}

void
Init_Default()
{
    N = 2048;
    Init = "rand";
    maxnum = 15.0;
    PRINT = 0;
}

int
Read_Options(int argc, char **argv)
{
    char *prog;

    prog = *argv;
    while (++argv, −−argc > 0)
        if (∗∗argv == '−')
            switch ( ∗++∗argv ) {
                case 'n':
                    −−argc;
                    N = atoi(∗++argv);
                    break;
                case 'h':
                    printf("\nHELP: try sor −u \n\n");
                    exit(0);
                    break;
                case 'u':
                    printf("\nUsage: gaussian [−n problemsize]\n");
                    printf(" [−D] show default values \n");
                    printf(" [−h] help \n");
                    printf(" [−I init_type] fast/rand \n");
                    printf(" [−m maxnum] max random no \n");
                    printf(" [−P print_switch] 0/1 \n");
                    exit(0);
                    break;
                case 'D':
                    printf("\nDefault: n = %d ", N);
                    printf("\n Init = rand" );
                    printf("\n maxnum = 5 ");
                    printf("\n P = 0 \n\n");
                    exit(0);
                    break;
                case 'I':
                    −−argc;
                    Init = ∗++argv;
                    break;
                case 'm':
                    −−argc;
                    maxnum = atoi(∗++argv);
                    break;
                case 'P':
                    −−argc;
                    PRINT = atoi(∗++argv);
                    break;
                default:
                    printf("%s: ignored option: −%s\n", prog, ∗argv);
                    printf("HELP: try %s −u \n\n", prog);
                    break;
            }
}
```

Listing 8. `qsortseq.c`

```
/∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗
 ∗
 ∗ Sequential version of Quick sort
 ∗
 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗/

#include <stdio.h>
#include <stdlib.h>

#define KILO (1024)
#define MEGA (1024∗1024)
#define MAX_ITEMS (64∗MEGA)
#define swap(v, a, b) {unsigned tmp; tmp=v[a]; v[a]=v[b]; v[b]=tmp;}

static int ∗v;

static void
print_array(void)
```

```
{
    int i;
    for (i = 0; i < MAX_ITEMS; i++)
        printf("%d ", v[i]);
    printf("\n");
}

static void
init_array(void)
{
    int i;
    v = (int *) malloc(MAX_ITEMS*sizeof(int));
    for (i = 0; i < MAX_ITEMS; i++)
        v[i] = rand();
}

static unsigned
partition(int *v, unsigned low, unsigned high, unsigned pivot_index)
{
    /* move pivot to the bottom of the vector */
    if (pivot_index != low)
        swap(v, low, pivot_index);

    pivot_index = low;
    low++;

    /* invariant:
     * v[i] for i less than low are less than or equal to pivot
     * v[i] for i greater than high are greater than pivot
     */

    /* move elements into place */
    while (low <= high) {
        if (v[low] <= v[pivot_index])
            low++;
        else if (v[high] > v[pivot_index])
            high--;
        else
            swap(v, low, high);
    }

    /* put pivot back between two groups */
    if (high != pivot_index)
        swap(v, pivot_index, high);
    return high;
}

static void
quick_sort(int *v, unsigned low, unsigned high)
{
    unsigned pivot_index;

    /* no need to sort a vector of zero or one element */
    if (low >= high)
        return;

    /* select the pivot value */
    pivot_index = (low+high)/2;

    /* partition the vector */
    pivot_index = partition(v, low, high, pivot_index);

    /* sort the two sub arrays */
    if (low < pivot_index)
        quick_sort(v, low, pivot_index-1);
    if (pivot_index < high)
        quick_sort(v, pivot_index+1, high);
}

int
main(int argc, char **argv)
{
    init_array();
    //print_array();
    quick_sort(v, 0, MAX_ITEMS-1);
    //print_array();
}
```