LINFO2241 : Project 1 & 2 (v2.2)

Imagine you have a company with customers using one of your products, a client-server application. When looking at the performance of such an application, one is typically interested in two problems:

- What response time do my customers experience?
    - If the response time is high, what is the reason for this?
    - Is the CPU or the disk in the server too slow?
    - Or maybe the network connection?
    - What about memory usage?
    - …
- Assuming I get new customers in the future, how many customers can my application handle before the response time becomes too high?

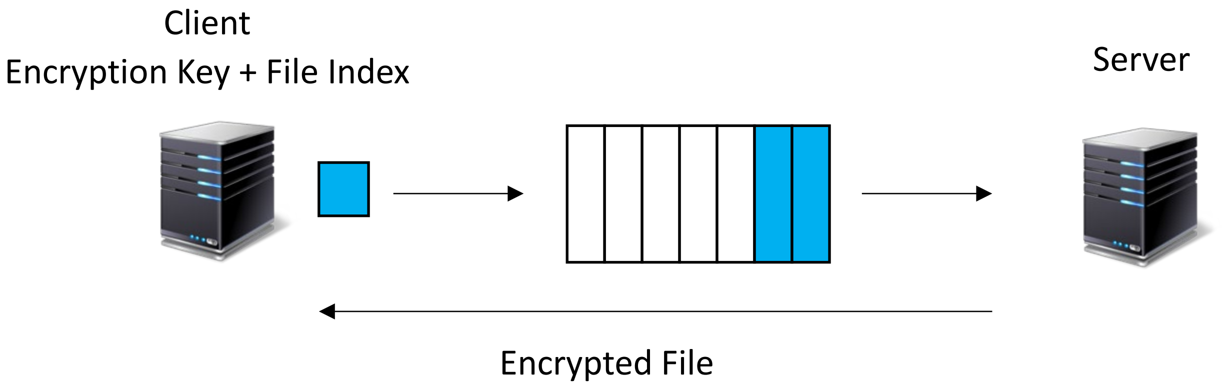


Figure 1: Client/Server schema. The client send a key and a file index to the server that encrypts a file and returns it.

In this project, you will analyze the performance (e.g. response time) of a simple client-server application that you will implement in C. The client requests an encrypted file from the server. The files are of fixed size, given as an argument to the server. In practice, you will pre-generate 1000 files with the sizes of the file taken in the argument. They will be stored in volatile memory (RAM). The size of the file will always be the square of a power of 2, enforce this in the parameter parsing. The client gives the index of a file (0 to 999 included) and the key. The key will be used to encode the file with block-by-block matrix multiplication. For instance, with a key of size 2x2 and a file of size 4x4:

Key:

A	B
C	D

File:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Encrypted File:

A*1+ B*5	A*2+ B*6	A*3+ B*7	A*4+ B*8
C*1+ D*5	C*2+ D*6	C*3+ D*7	C*4+ D*8
…			

Figure 2 : Example of encryption. All the boxes are bytes (in phase 1). In phase 2, each box is an uint32_t.

An example of matrix multiplication in python is given at https://colab.research.google.com/drive/18HhfRwPVcdBxbwc25oEg0vuqLuMy6Et5?usp=sharing

You will multiply the key with the 4 colored boxes using matrix multiplication. The key size must divide the file size, up to the file size itself. Enforce the rule, returning error codes if it is not the case. Therefore, you don't have to worry about corner cases like when the key does not perfectly fit into the file.

You will use TCP sockets. The protocol works as follows. The client sends a request with the following format. The first 4 bytes are an unsigned integer which is the index of the file to receive (0 to 999). The next 4 bytes sent by the client compose a single unsigned integer, specifying the size of the square matrix to come (2 in the example). Then will follow N*N bytes (4 in the example) describing the key. Line by line and left to right.

The server then returns an error code in the form of a byte, normally 0 when there's no error. Then the file size (as a total byte number, so 16 in the example) as an unsigned 4-byte integer, and **then the server closes the connection after transmitting those encrypted bytes**. This means a client needs to re-create a new TCP connection to obtain another file. The server must be multithreaded to handle multiple connections in parallel, **but not to accelerate a single request**. If the server is run with -j 4 (see below), it means at maximum 4 files are being encrypted at the same time. Therefore, you can't create as many threads as there are connections, use a pool of threads instead, and/or a FIFO queue.

Pay attention to the endianness. The unsigned integer fields must follow the network byte order (big-endian).

The client's request

4 bytes	4 bytes	N*N bytes
Index of the file (0 to 999)	Size of the key, N (the dimension, not the N*N total)	The key

The server's response

1 byte	4 bytes	F bytes
Error Code	Size of the file F (total byte number)	The encrypted file

Protocol for the request and responses.

The project will be split into 4 phases.

- The goal of the first part of the work (now) is to measure the performance of the server in a systematic way. Do not beat yourself about the encryption in the first phase, as the second phase will be about the optimization of the processing of the first one. Follow the mathematical matrix multiplication block by block.
- In the second phase, you will optimize the processing in a cache-aware fashion, and show how the performance of multiple versions of the algorithm can be affected by the placement of data in the cache and the instructions. You will use profiling techniques to explain these improvements.
- In the third phase, you will optimize the processing using SIMD and other instruction-level parallelism techniques. This part will be done individually.
- In the final phase, you will use queuing theory to model your system.

Evaluation environment

The FINAL graphs must be generated on the intelXX machines of the Intel room (https://wiki.student.info.ucl.ac.be/Mat%C3%A9riel/SalleIntel).

This is important so we can discuss results together in the next phases, with the same hardware. However you must develop and test the system on your own local machine to avoid overwhelming UCLouvain's resources.

Phase 1

Measure the average response time of your system and other relevant metrics you see fit. Obviously, the response time will depend on how many requests the server receives and how "difficult" requests are.
To do measurements, you must not send requests to the server by hand. Instead, you should implement a small client application that sends requests at random times. For the inter-request time, choose some distribution you see fit. To send multiple requests to the server at the same time, you can either write a client application that opens multiple TCP connections, or the client only opens one TCP connection and you start the client application several times in parallel. If using the latest option, be sure to comply with the command line interface below using some wrapper.

The client should not wait for a response before sending the next request. It's perfectly possible that a request is sent while another request is still being processed. Show measurement results for combinations of

- different request rate
- different matrix sizes

… many other parameters as we learned in the first 2 lectures.

Think about different difficulties and how they would affect the performance of the server in different ways (e.g. more CPU intensive vs more network intensive). Plan carefully your experimental design following what we saw in the course.

Choose reasonable values for the parameters: obviously, a request rate of 1 request/day does not produce interesting response time results. Show plots and discuss the results. What is the most important factor (CPU, network, ...) in the response time, depending on the chosen parameters? Here are some tools to measure the network and CPU load on Linux:

- • http://www.binarytides.com/linux-commands-monitor-network/
- • https://www.tecmint.com/command-line-tools-to-monitor-linux-performance/

Expected Output

You should hand in the source code of your implementation (no binaries) on INGInious (through moodle, there is a link) and a written report on Moodle. Both submissions site will open 1~2 weeks before the deadline, as groups cannot be changed after the submission sites are open. The deadline for both ~~is October 16th, 23:59.~~ is October 21st , 23:59.

The report should contain:

- • a description of the implementation, including the client
- • a description of your measurement setup, even if it is the Intel machines for everyone
- • a description of the workload you used in your experiments, i.e. how the clients generate the requests. Include a description of the experimental design and all the variables that you thought of. The factors you selected. The one you left out and their possible impact.

Don't forget to include your names, email addresses, NOMA and group number. Keep your report concise. Don't waste space on long introductions, a table of contents, etc.

Code archive

You must provide a Makefile that will compile the client and server with "make client" and make server", the generated binaries will have the same names as the target. You must put all files in a tar.gz archive that has no root folder.

- The command line of the client should be "./client -k 128 -r 1000 -t 10 127.0.0.1:2241" to generate requests with keys of size 128 at a mean rate of 1000 requests/seconds for 10 seconds to the server 127.0.0.1 on TCP port 2241.
- The command line of the server should be "./server -j 4 -s 1024 -p 2241" to open the server with 4 threads and pre-generate files of 1024*1024 bytes, listening on the TCP port 2241.

If you think something is missing, discuss it on the forum.

Evaluation Criteria for the first part

- Quality of code (comments, correctness, etc.)
- Quality of the report (reasoning, language, readability, clearness of presentation)
- Quality of the experimental design. Workload description. Choice of variables, choice of factors. Explanation of the limits of the measurements. (I.e. : What did you not evaluate? What design choices affect the performance?)
- Quality of the measurements. Choice of visualization, choice and correctness of summarization

Bonus

One obvious important parameter in this system is the network latency. Of course, the request will have a higher completion time if the server is far away from the client.

Due to the complexity of imposing network delays, the study of its impact is kept as a bonus. If you want to follow the bonus, you must do as follows. The bonus is of 2 points/20 if successfully completed.

As you are not root on the Intel machines, you cannot directly create special network devices and induce delays. The solution is to create 3 virtual machines on a single intelXX machine, a client, a server and a “router” machine connecting the first 2 machines that will simulate varying network conditions such as network delay.

All the necessary software is already installed on the Intel machines. If you want to try at home, you need the `qemu` hypervisor and `qemu-kvm` accelerator.

Download the Debian base image with:

```
wget https://cloud.debian.org/images/cloud/buster/latest/debian-10-nocloud-amd64.qcow2
```

Create 3 overlay images for the 3 machines with:

```
qemu-img create -b debian-10-nocloud-amd64.qcow2 -F qcow2 -f qcow2
client.qcow2 20G
```

Launch the server with accelerated QEMU (with KVM, a must!) :

```
qemu-system-x86_64 -m 4096 -drive file=client.qcow2 -nographic -
enable-kvm
```

The user of the machine is root, there is no password.

Exit the VM with Ctrl + A – x

The VM server can be launched with multiple CPUs with : --smp cpus=4.

Please be carefull to killall your VMs when leaving with `killall qemu-system-x86_64` !

VM networking

Add the following parameter to each machine to create a network interface that will provide Internet access for your convenience : `-netdev user,id=mynet0 -device virtio-net-pci,netdev=mynet0`

Then, you will need to create an “internal” network. To the router machine, add 2 interfaces :

```
-netdev socket,id=left,listen=:1234 -device virtio-net-
pci,netdev=left
```

```
-netdev socket,id=right,listen=:1235 -device virtio-net-
pci,netdev=right
```

To the client machines, add :

```
-netdev socket,id=left,listen=:1234 -device virtio-net-
pci,netdev=left
```

To the server machine, add :

```
-netdev socket,id=right,listen=:1235 -device virtio-net-
pci,netdev=right
```

You now have a network with a router in between. You must set the “internal” IP addresses by yourself with “ip addr …”. You can add some delay with netem. See https://wiki.linuxfoundation.org/networking/netem .

Configure the router and gateways in a way that allows the client machine to see the server and vice versa using two different subnets.

Phase 1 correction

A bit of help for the part that needs to work to complete the phase 2.

Server

Here's an example of parsing.

```
    int nthreads;

    while ((opt = getopt(argc, argv, "j:s:p:v")) != -1) {
        switch (opt) {
        case 'j': nthreads = atoi(optarg);break;
        case 's': nbytes = atoi(optarg); break;
        case 'p': port = atoi(optarg); break;
        case 'v': verbose = 1; break;
        default:
            fprintf(stderr, "Usage: %s [-j threads] [-s bytes] [-p port]\n", 
argv[0]);
            exit(EXIT_FAILURE);
        }
    }
```

In phase 2, "nthreads" is not needed, so it eases the programming… a lot.

Creating the listening socket :

```
    // Creating socket file descriptor
    sockfd = socket(AF_INET, SOCK_STREAM, 0));
```

Binding :

```
    memset(&servaddr, 0, sizeof(servaddr));
    // Filling server information
    servaddr.sin_family    = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(port);

    // Bind the socket with the server address
    bind(sockfd, (const struct sockaddr *)&servaddr,
            sizeof(servaddr);
```

For phase 2, the array type changes :

```
#define ARRAY_TYPE uint32_t
```

Allocating files.

```
    pages = malloc(sizeof(void*) * npages);
    for (int i = 0 ; i < npages; i++)
        pages[i] = malloc(nbytes*nbytes *sizeof(ARRAY_TYPE));
    //New requirement for file 0 !
    for (unsigned i = 0; i < nbytes * nbytes; i++)
        pages[0][i] = i;
```

In phase 2, you must not randomize the content of malloc. Just keep whatever random memory is there, except for file 0 that must be initialized as above. This is to avoid a long initialization phase.

Running the server with the while(accept):

```
    printf("Listening...\n");
    if ((listen(sockfd, 128)) != 0) {
        printf("Listen failed...\n");
        exit(1);
    }
    else
        printf("Server listening..\n");

    while( (client_sock = accept(sockfd, (struct sockaddr *)&servaddr,
(socklen_t*)&c)) )
    {
        connection_handler((void*)(intptr_t)client_sock);
    }
```

The function is a reminiscence of the call to pthread_create() that is not needed in phase 2. You could, of course, pass client_sock as an int here.

The connection handler interface is :

```
int connection_handler(void *socket_desc)
{
    //Get the socket descriptor
    int sockfd = (int)(intptr_t)socket_desc;
```

Receiving data:

```
    int tread = recv(sockfd, &fileid, 4, 0);

    tread = recv(sockfd, &keysz, 4, 0);
    //Network byte order
    keysz = ntohl(keysz);



    ARRAY_TYPE key[keysz*keysz];

    unsigned tot = keysz*keysz * sizeof(ARRAY_TYPE);
    unsigned done = 0;
    while (done < tot) {
        tread = recv(sockfd, key, tot- done, 0);
        done += tread;
    }
```

You will note recv() might not give all the data in one go. So you have to loop until you got everything.

Of course, good code should check every function calls returned a correct value 😉

Now the processing itself :

```
    int nr = nbytes / keysz;

    ARRAY_TYPE* file = pages[fileid % npages];
    ARRAY_TYPE* crypted = malloc(nbytes*nbytes * sizeof(ARRAY_TYPE));

    //Compute sub-matrices
    for (int i = 0; i < nr ; i ++) {
        int vstart = i * keysz;
        for (int j = 0; j < nr; j++) {
            int hstart = j * keysz;

             //Do the sub-matrix multiplication
            for (int ln = 0; ln < keysz; ln++) {

                int aline = (vstart + ln) * nbytes + hstart;
                for (int col = 0; col < keysz; col++) {

                    int tot = 0;
                    for (int k = 0; k < keysz; k++) {
                        int vline = (vstart + k) * nbytes + hstart;
                        tot += key[ln * keysz + k] * file[vline + col];
                    }
                    crypted[aline + col] = tot;


                }
            }
        }
    }

    send(sockfd, &err, 1,MSG_NOSIGNAL );
    unsigned sz = htonl(nbytes*nbytes * sizeof(ARRAY_TYPE));
    send(sockfd, &sz, 4, MSG_NOSIGNAL);
    send(sockfd, crypted, nbytes*nbytes * sizeof(ARRAY_TYPE),MSG_NOSIGNAL );
```

Then free, close, etc.

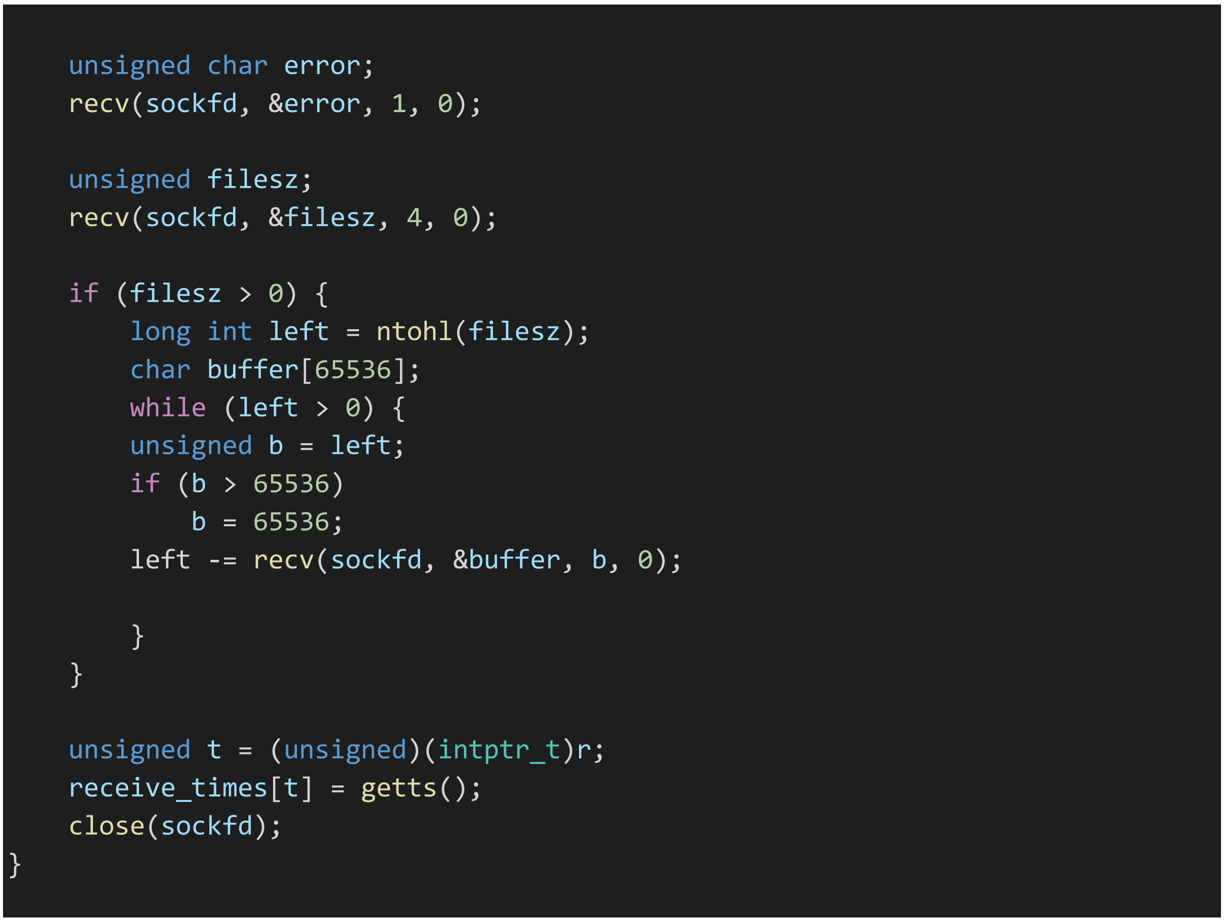
The client

First a main loop just spawns threads at the given rate

```
int diffrate = 1000000000 / rate;
while (getts() - start < (long unsigned)1000000000 * time) {
            next += diffrate
            while (getts() < next) {
                usleep((next - getts()) / 1000);
            }
            sent_times[i] = getts();
            pthread_t thread;
            pthread_create( &thread, NULL, rcv, (void*)(intptr_t)i);
            i++;
}
```

Then the rcv loop of the thread that will create one connection, send the request, and wait for the answer. Then it writes down the time taken to answer.

```
void* rcv(void* r) {
    int ret;
    int sockfd;
    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(port);
    servaddr.sin_addr.s_addr = INADDR_ANY;


    connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));

    //Send file id
    unsigned fileindex = htonl(rand() % npages);

    ret = send(sockfd, &fileindex, 4, 0);

    //Send key size
    int revkey = htonl(keysz);
    ret = send(sockfd, &revkey, 4, 0);

    //Send key
    ret = send(sockfd, key, sizeof(ARRAY_TYPE) * keysz*keysz, 0);
```



```
    unsigned char error;
    recv(sockfd, &error, 1, 0);

    unsigned filesz;
    recv(sockfd, &filesz, 4, 0);

    if (filesz > 0) {
        long int left = ntohl(filesz);
        char buffer[65536];
        while (left > 0) {
        unsigned b = left;
        if (b > 65536)
            b = 65536;
        left -= recv(sockfd, &buffer, b, 0);

        }
    }

    unsigned t = (unsigned)(intptr_t)r;
    receive_times[t] = getts();
    close(sockfd);
}
```

Phase 2

In this phase you will optimize the server code to process the encryption faster. You will apply the techniques seen in the course.

Mandatory compiler options

You cannot use vector instructions in this phase. Compile with -mno-sse2 -mno-avx -mno-avx2 -mno-avx512f -fno-unroll-loops -fno-tree-vectorize -O2

No-unroll-loops will prevent automatic loop unrolling, because this is something you have to study and do yourself.

You must use the -O2 optimization level.

Type and project changes

1) **Change the client and the server so the files and keys contain elements of int32_t types instead of a single byte.**

Therefore:

./server -s 1024

Will create files of 1024*1024*4 bytes.

./client -k 128 127.0.0.1:2241

Will send a key of 128*128*4 bytes.

You will notice the code of your matrix computation will not change.

2) **Do not pre-initialize files. Just do a malloc(sizeof(uint32_t)*nbytes*nbytes), and leave whatever is in RAM as the “random” content.**
3) **Only the file 0 must be initialized, with each element being its own index, eg :**

```
for (unsigned i = 0; i < nbytes * nbytes; i++)
  files[0][i] = i;
```

Mandatory optimizations

You must apply the two following techniques:

- Accessing memory in the order of bytes to enable streamline prefetching
  - ➔ At least, follow the matrix multiplication technique seen in the course, or an equivalent technique to compare line*column and line*line multiplication.
- Loop unrolling (given the requirements there is no need for strip mining)

The 2 optimizations above must be correctly studied, including their interaction using the appropriate techniques as seen in the first two lectures.

Constant parameters in the study:

Consider the study of files of 1024*1024 int32_t entries (so the total size of memory is 4M. **Always use -j1 (one processing thread).** No network latency (if you did the bonus in phase 1).

Dynamic factors in the study:

Consider a key of size 8 and a key of size 128.

The graph must run on the machines from the Intel room, still. **You do not need to study other factors than the key sizes, and the two optimizations (with and without each). The only metric to study is the request latency.** All this phase has to run on the machines from the Intel room. **Do not use VMs for this phase.** Performance counters are not accessible from virtual machines. Just launch client and server on bare metal.

Profiling

The most precise way to measure cache statistics (such as hit rate) is to directly query the CPU. Modern CPUs contain so-called performance counters. For example, Intel CPUs have an internal counter that counts the number of cache misses. The easiest way to access those performance counters (if you don't want to write machine code) is the command-line tool perf. It collects and processes performance events from the Linux kernel. Many parts of the Linux kernel produce such performance events, for example the network stack, the device drivers, and the virtual memory management.

The "perf" tool is is part of the Linux Kernel. It is generally not installed by default. On Debian systems, use "sudo apt install linux-perf" and on Ubuntu "sudo apt install linux-tools"

You can see the list of performance events supported on your computer with

```
perf list
```

Here, we are only interested in the performance events related to the performance counters of the CPU. You can see the CPU counter statistics for a program with

```
perf stat mytestprogram
```

There are many ways to call the perf tool. For example, you can specify the list of events you are interested, how long the measurement should run, etc. You can find some examples on

http://www.brendangregg.com/perf.html

For this part, the most interesting counters are:

- cache-references: number of accesses to all caches (L1+L2+L3).
- cache-misses: number of accesses that could not be served by any cache.
- dTLB-* and iTLB-*: hits and misses for the data TLB and the instruction TLB. We have seen in the course that some CPUs have separate TLBs and caches for accesses to data and code.
- L1-dcache-* and L1-icache-*: statistics for the L1 data cache and L1 instruction cache.
- LLC-*: statistics for the last CPU cache.

Please note that you should take the results only as estimates. CPU models have different performance counters and "perf" sometimes combines several performance counters to calculate a result. That means if you really want to know what the output exactly means you must read the perf source code… In addition, the CPU manufacturers do not always precisely define in their manuals what a counter exactly counts. For example, imagine the following situation: the L1 cache wants to stream-prefetch the next memory block and it sees that the block is already in the cache. Does this count as a cache hit?

In this part, you must explain for each of the optimization why the performance increases, and show it through the profiling numbers. For instance, the first technique enables less cache misses. In which caches? Show it with the number of hits/misses for L1, L2 and LLC.

Contest & free optimizations

You will submit a tar archive as in the first phase to inginious, through moodle. The performance of your code will be rated automatically, and you will be ranked on an online scoreboard. The top students will receive badges. We will not use your client, so your server must comply to the protocol.

A few optimizations we saw in the lectures/ideas :

- Arrange addresses so they don't conflict due to set associativity
- Optimize the working set to have a better cache locality, i.e. increase temporal locality
- Arrange the data so it fits in a multiple of cachelines
- Avoid loops and jumps knowing we will use keys of dimension 8 or 128 with files of 1024.
- If using a function for the sub-matrix multiplication. Consider an inlined version.
- …

Not all techniques will work as well. Show the process, explain why your techniques made sense, how and why you think it failed. In the report, explain the techniques but you don't need to show the interactions of techniques as in the above or do the profiling with perf stat. Simply give a sense of how much improvement all of those did on top of the techniques in the mandatory optimizations part.

Do not re-explain this exercise in the report introduction. Keep it short. But do provide a conclusion.

For the online submission, you can send your whole code but only the server will be used and built through "**make server-optim**". We will launch **./server-optim** with the aforementioned parameters.

Useful tips:

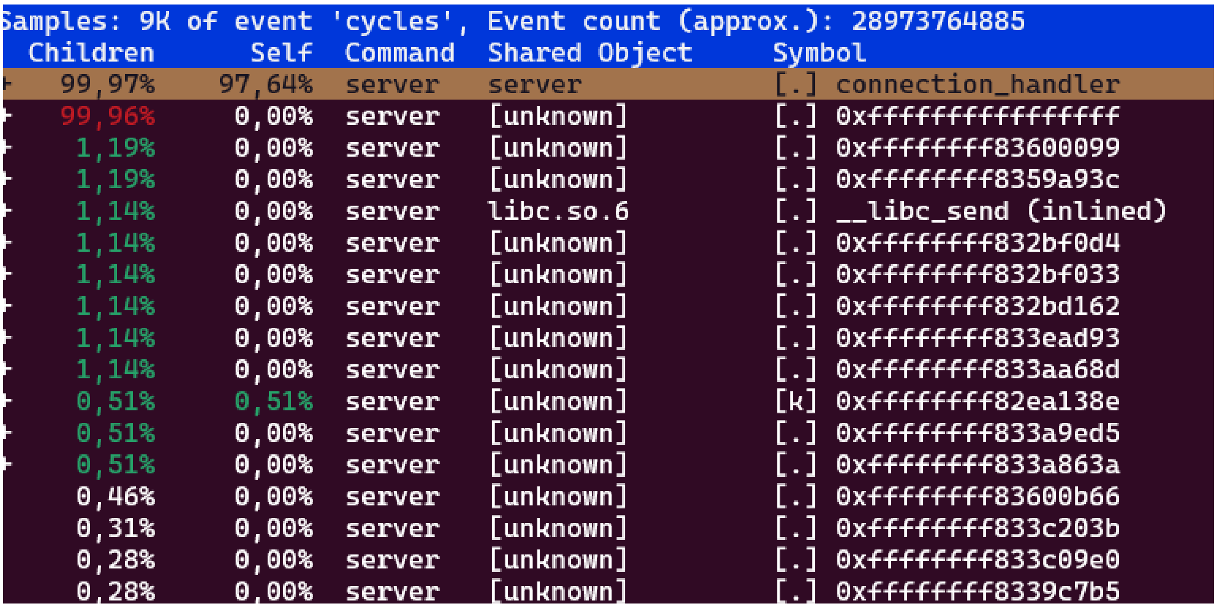
- aligned_alloc allows to allocate memory with a certain memory alignment
- `struct S { short f[3]; } __attribute__ ((aligned (8)));`
  The attribute will force the struct S to be statically allocated to an alignment of 8. Of course if you cast a random address (ie allocated with malloc) to (struct S*) it won't move the memory and won't be aligned. But a global of local variable will be.
- In your makefile you can create a version of the "server" target that compiles with -DOPTIM=0 and the "server-optim" version with -DOPTIM=1. Then in your code use #if OPTIM pragmas to keep both version and easily compare between the optimized version and the normal one. As always, copy-pasting the same code just to change a line will be heavily penalized.

To find what you should optimize, you can compile with "-g" and then run the server with :

```
perf record -e cycles -F 999 --call-graph dwarf ./server -j 1 -s 1024 -p 2241
```

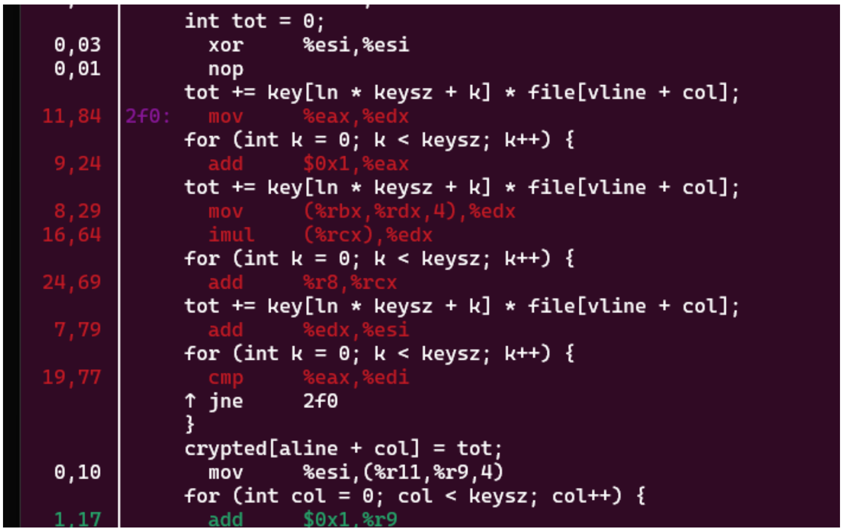
Then launch the client in a way that creates quite a lot of CPU load (but not just explodes by creating too many threads :) . Let it run for like 10 seconds. Then stop both apps.

Then launch "perf report" that will give you an interactive list of functions :



```
Samples: 9K of event 'cycles', Event count (approx.): 28973764885
  Children      Self  Command  Shared Object      Symbol
+   99,97%    97,64%  server   server             [.] connection_handler
+   99,96%     0,00%  server   [unknown]          [.] 0xffffffffffffffff
+    1,19%     0,00%  server   [unknown]          [.] 0xffffffff83600099
+    1,19%     0,00%  server   [unknown]          [.] 0xffffffff8359a93c
+    1,14%     0,00%  server   libc.so.6          [.] __libc_send (inlined)
+    1,14%     0,00%  server   [unknown]          [.] 0xffffffff832bf0d4
+    1,14%     0,00%  server   [unknown]          [.] 0xffffffff832bf033
+    1,14%     0,00%  server   [unknown]          [.] 0xffffffff832bd162
+    1,14%     0,00%  server   [unknown]          [.] 0xffffffff833ead93
+    1,14%     0,00%  server   [unknown]          [.] 0xffffffff833aa68d
+    0,51%     0,51%  server   [unknown]          [k] 0xffffffff82ea138e
+    0,51%     0,00%  server   [unknown]          [.] 0xffffffff833a9ed5
+    0,51%     0,00%  server   [unknown]          [.] 0xffffffff833a863a
     0,46%     0,00%  server   [unknown]          [.] 0xffffffff83600b66
     0,31%     0,00%  server   [unknown]          [.] 0xffffffff833c203b
     0,28%     0,00%  server   [unknown]          [.] 0xffffffff833c09e0
     0,28%     0,00%  server   [unknown]          [.] 0xffffffff8339c7b5
```

We see in this screenshot that the connection_handler function took 99.97% of the CPU time. Press "a" to get the details.

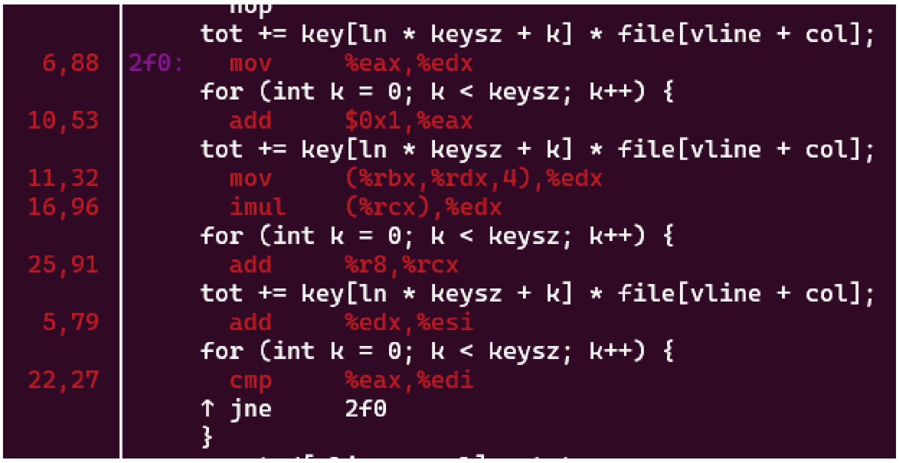


```
               int tot = 0;
   0,03          xor     %esi,%esi
   0,01          nop
               tot += key[ln * keysz + k] * file[vline + col];
  11,84 2f0:     mov     %eax,%edx
               for (int k = 0; k < keysz; k++) {
   9,24          add     $0x1,%eax
               tot += key[ln * keysz + k] * file[vline + col];
   8,29          mov     (%rbx,%rdx,4),%edx
  16,64          imul    (%rcx),%edx
               for (int k = 0; k < keysz; k++) {
  24,69          add     %r8,%rcx
               tot += key[ln * keysz + k] * file[vline + col];
   7,79          add     %edx,%esi
               for (int k = 0; k < keysz; k++) {
  19,77          cmp     %eax,%edi
               ↑ jne     2f0
               }
               crypted[aline + col] = tot;
   0,10          mov     %esi,(%r11,%r9,4)
               for (int col = 0; col < keysz; col++) {
   1,17          add     $0x1,%r9
```

We see which instructions (corresponding more or less to which C lines) take time. Perf is a bit imprecise, the time taken for an instruction might be something happening on the previous instructions. In this last screenshot, cmp does not really take 20% of the time to compare i/j/k, it's the "tot += key[ln * keysz + k] * file[vline + col];" spread in many instructions that slows down the pipeline.

Explaining your improvements with perf report in your report will lead to a ~2/20 bonus.

Note that you can change the "-e cycles" by any perf event. Like "-e L1-dcache-misses".



```
                nop
              tot += key[ln * keysz + k] * file[vline + col];
  6,88 | 2f0:  mov      %eax,%edx
              for (int k = 0; k < keysz; k++) {
 10,53 |        add      $0x1,%eax
              tot += key[ln * keysz + k] * file[vline + col];
 11,32 |        mov      (%rbx,%rdx,4),%edx
 16,96 |        imul     (%rcx),%edx
              for (int k = 0; k < keysz; k++) {
 25,91 |        add      %r8,%rcx
              tot += key[ln * keysz + k] * file[vline + col];
  5,79 |        add      %edx,%esi
              for (int k = 0; k < keysz; k++) {
 22,27 |        cmp      %eax,%edi
              ↑ jne      2f0
              }
```

Here it’s the amount of total L1 data misses which is given in percentages. It’s relatively the same as cycles because the code is slowed down by cache misses.

Deadline

The deadline is the 15th of November at 23:59. You will submit a report with the content mentioned above, and the archive on inginious through Moodle as in the first phase.