Arquiteturas de Alto Desempenho 2025/2026

Practical class AAD_P04 (2025-10-06)

Tomás Oliveira e Silva

1 Objectives

Learn how to parallelize code across multiple machines the hard way (sockets in a server/clients paradigm). List of things you will learn about:

- how to use sockets to create a server — `bind()`, `listen()`, `accept()`
- how to use sockets to create a client — `connect()`
- how to parallelize a program using a server/client paradigm

2 Sockets

In order to communicate with other computers, or with other processes running on the case computer (in this case there exist other ways to do that), all modern operating systems have a so-called network stack. The network stack manages all communications links (ethernet, wifi) to the outside, and implements all relevant communication protocols. We are only interested here in the interface of the network stack with our programs. That is done using `sockets`. The sockets we are interested in are TCP/IP sockets (stream sockets), because they ensure reliability in the delivery of the data. (At this point, if you are not familiar with sockets, please spend as much time as necessary to at least learn the basic things about them.)

A socket is, roughly speaking, an endpoint of a communication link. Its data structure is created using the `socket()` function. After creation, the socket can be prepared to

- accept connections — using the `bind()`, `listen()`, and `accept()` functions, or to
- connect to another socket — using the `connect()` function.

When a connection between two sockets has been established, it is possible to send and receive data using

- the `send()` or `write()` function to send data, and
- the `recv()` or `read()` function to receive data.

In uncommon scenarios, the endianness (byte order of an integer stored in a memory location) of the two endpoints may be different, and so to transfer information it is necessary to pay attention to that. One possibility is to serialize (transform everything to a sequence of bytes) all data that is send, and to use a matching deserialization on the receiving side.

3 A simple server/client program

The file `server_clients.c` source code file contains the entire code of a simple server and the matching code of a simple client. The entire code was placed in a single file

- because it is not very large and the server and client share some of the code,
- to show that this kind of thing can be done,
- to show how conditional compilation (enabling/disabling parts of the code at compilation time) can be performed,

- the code is meant as an example, and so it was not designed to be reused without change in other projects, and, finally,
- because the teacher likes to do things in this way. (I really dislike keeping one or more `.h` files up to date with the contents of one of more `.c` files).

Things to do:

- Take a good look at the entire source code.
  - In particular, study how messages are exchanged between the server and each client in a way that takes care of the endianness of the processor on each side of the communication link.
  - Also, pay attention to the flow of messages between the server and the client. For each message one side sends, the other must have matching code to receive it! If you are using the `vim` editor, you can examine how this is done using the `:vsplit` command to open two views of the same file (learn how to change the cursor to the other view…).
  - The server code uses an alarm to terminate the server execution after some time. This was done as a safety measure: if you forget to kill the server, it will die by itself after some time.
- Compile the code:

```
$ make server client
```

- Start the server. On one terminal window:

```
$ ./server 50000
```

  `50000` is the port number the server will use. Two or more servers running at the same time on the same machine (same IPv4 address) must use different port numbers.
- On other terminal windows (use as many as you like), run the client:

```
$ ./client 127.0.0.1 50000
```

  `127.0.0.1` is the IPv4 address of the local machine, where the server is running.
- Try to connect a client to a server running in another computer (perhaps your neighbor's computer, or a classroom computer). To do that you will need the IPv4 address of that computer. On GNU/Linux, just run the "`ifconfig`" command on that computer and read the IPv4 address from the `inet` line of the appropriate interface. You may also use the "`ip a`" command. Depending on the firewall configuration, you may, or may not, be able to connect to the server. If not, a mobile hot spot may be your friend…
- Split the `server_clients.c` source code file into three components: the common code (in files `x_common.c` and `x_common.h`), the server code (in files `x_server.c` and `x_server.h`), and the client code (in files `x_client.c` and `x_client.h`). The `makefile` already has rules to compile the split source code. The `x_` prefix in the file names of the split code serves only to distinguish the two code bases; with it we can have different rules in the `makefile` to build the two versions of the programs.

4 Putting it all together: parallelization of `MandelbrotCount.c`

It is now time to put all that you learned in this class in action. Adapt the server/client code to run `MandelbrotCount.c` in parallel. Can you do part of the computation using someone else's computer?