Comparison of downloading files from remote url sequentially and in parallel

[1]Aditya Madhyastha, [1]Akash Jain, [1]Akshat Jaitly
(Usn: 1MS18CS011, 1MS18CS014, 1MS18CS015)

Abstract:

Since we live in a technology era.There are lots of files that are being exchanged every second. So we want to download our file as fast as we can..There are lots of factors that affect speed of downloads such as file size, availability, maximum bandwidth allocated, server load etc. So some files may download faster while others are slower which is unpredictable. So we have implemented a parallel system for downloading files which eliminates this problem and in less time we will have a higher number of downloads completed .OpenMp is being used to implement the same.

Keywords- OpenMP, Parallel Downloading, parallelism.

I. Introduction

There is a high demand for parallel programming since the technology industry began to have a huge user base and we got access to big data and hardware to process it. We need more and more computational power in today's world to solve some of the world's scientific problems. We need to make sure that we are leveraging the existing computational power to its fullest before we look for more , let it be a CPU or GPU. There exist multiple interfaces for parallel programming like MPI (Message Passing Interface), OpenMP, CUDA,etc. If we want to run on multiple CPUs with shared memory, OpenMP can be used. When we want to run on multiple CPUs within multiple nodes, MPI comes in handy. When we need GPU programming, CUDA is required. These frameworks can be used in C/C++ and Fortran languages. The basis for OpenMP is the shared memory model. This means all the threads have access to the same global memory and run in parallel. The data can be either globally shared across the threads or private to the thread.

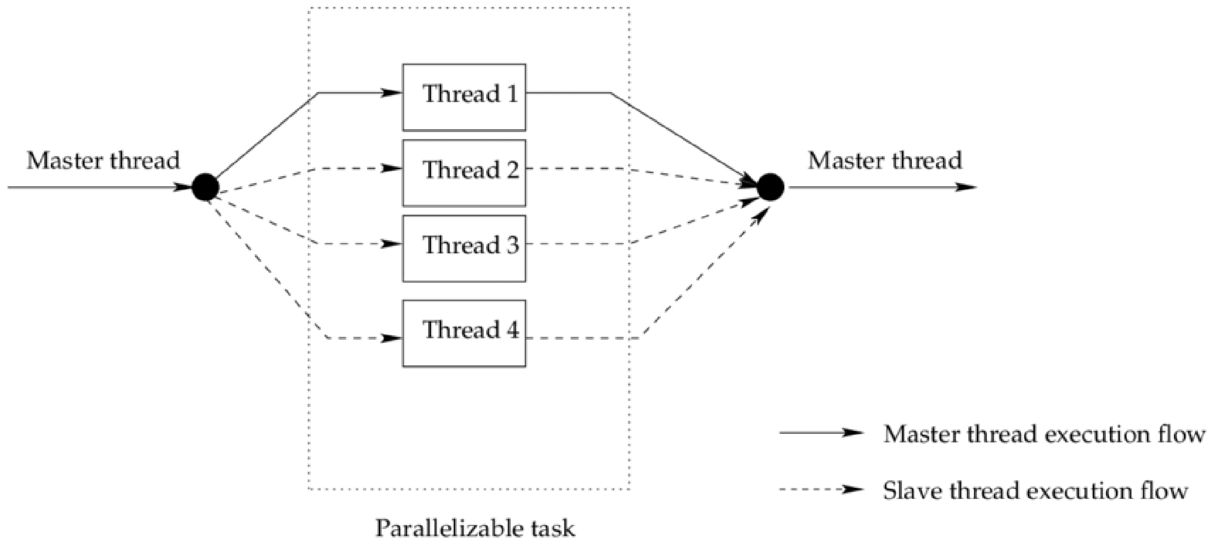


Fig 1. OpenMp threading flow

A fork-and-join model of parallel execution is used by OpenMP. Our program will run the piece of code in that block parallelly if in that particular program, we define “#pragma omp parallel{}” . The program will synchronise between a team of threads and join them to the initial(master) thread when it reaches the end of the block. We can control the flow of applications with this.

II. Literature Survey

OpenMP is a shared-memory multiprocessing Application Program Inference (API) . It provides easy development of shared memory parallel programs . It provides a set of compiler directives for us to create threads, on top of pthreads manage the shared memory and synchronise the operations. The programs are compiled into multithreaded programs and the communications between threads is very systematic as threads share the same memory address space. A set of libraries is provided by the runtime as it maintains a thread pool as well [1]. To switch between sequential and parallel sections, which follow the fork/join model it uses a block-structured approach . A single thread is split into a number of threads at the entry of a parallel block, and a new sequential thread is started when all the split threads have finished . A fine-grained control over the threads is allowed by the directives[2]. Willy Zwaenepoel, Y.Charlie Hu, Honghui Lu, and Alan L Cox (1999), implemented OpenMP on a network of shared memory multiprocessors. The programmer relies on a single, standard shared-memory API for parallelization within a multiprocessor and between multiprocessors. John Bircsak, Zarka Cvetanovic, Peter Craig, Jonathan Harris, RaeLyn Crowell, Carl D. Offner and C. Alexander Nelson (2000), in their paper, describes extensions to OpenMP that implement data placement features needed for Non-Uniform Memory Access (NUMA) architectures. It is required that a programmer controls the placement of data in memory and the placement of computations that operate on that data in order to write efficient parallel programs for NUMA architectures, which have characteristics of both shared-memory and distributed-memory architectures. When computations occur on processors that have fast access to the data optimal performance is obtained . OpenMP does not by itself address these issues. It is supported on various platforms like LINUX, Windows and UNIX and various languages like C, C++, and Fortran [2]. Some of the advantages of OpenMP are1) The compiler takes care of transforming the sequential code into parallel code according to the directives, therefore OpenMP is much easier to use[2]. 2) Without serious understanding of the multithreading mechanism, the programmer can write multithreaded programs [1].

III. Method

parallel.c

```
#include <stdio.h>
#include "utils.h"
#include <omp.h>

int main(int argc, char *argv[])
{
  int nthreads, i, tid;
  init_curl();
  if (argc < 2)
  {
     printf("At Least one url should be passed as arguments.\nUsage:\n\t%s URL1 URL2 ... \n",
*argv);
        exit(1);
    }

    nthreads = argc - 1;
    if (nthreads <= 0)
    {
        printf("Invalid value for NUM_THREADS.\nUsage:\n\t%s URL1 URL2 ... \n", *argv);
        exit(1);
    }

    omp_set_num_threads(nthreads);

    #pragma omp parallel shared(argv, argc) private(i, tid)
    {
        tid = omp_get_thread_num();

        printf("Thread %d starting...\n", tid);

        pull_one_url(*(argv + tid + 1));

    } /* end of parallel section */
    curl_global_cleanup();
    return 0;
}
```

The above file contains the drive code to download the files in parallel using omp threads, it accepts the url from users as cmd line args and creates a thread for each url to download the file.

serial.c:

```
#include <stdio.h>
#include "utils.h"

int main(int argc, char *argv[])
{
    int nfiles, i, tid;
    init_curl();
    if (argc < 2)
    {
        printf("At Least one url should be passed as arguments.\nUsage:\n\t%s URL1 URL2 ... \n",
*argv);
        exit(1);
    }

    nfiles = argc - 1;
```

```
  if (nfiles <= 0)
  {
     printf("Invalid value for NUM_THREADS.\nUsage:\n\t%s URL1 URL2 ... \n", *argv);
     exit(1);
  }

  for(i=0;i<nfiles;i++)
  {

     printf("Download %d starting...\n", i);

     pull_one_url(*(argv + i + 1));

  } /* end of parallel section */
  curl_global_cleanup();
  return 0;
}
```

The above file contains the drive code to download the files in serial using for loop, it accepts the url from users as cmd line args and iterates through each url using for loop to download the file.

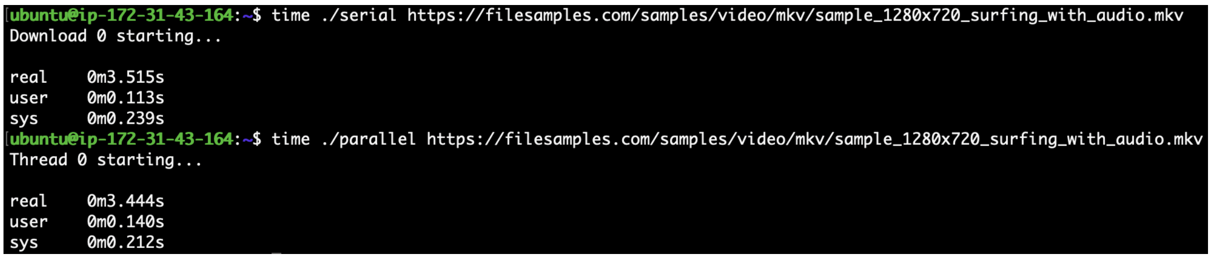
utils.h:

```
#include <curl/curl.h>
#include <stdlib.h>
#include <string.h>

/* Retrieves the file name from a URL */
char * getFileName(char * url)
{
  char *path;
  path = (char *)malloc(strlen(url)*sizeof(char));
  strcpy(path,url);
  char *ssc;
  int l = 0;
  ssc = strstr(path, "/");
  while (ssc)
  {
     l = strlen(ssc) + 1;
     path = &path[strlen(path) - l + 2];
     ssc = strstr(path, "/");
  }
  return path;
}
```

```
/* Saves the downloaded data into a file */
size_t write_data(void *ptr, size_t size, size_t nmemb, FILE *stream)
{
  size_t written = fwrite(ptr, size, nmemb, stream);
  return written;
}

/* Initialises Curl */
void init_curl()
{
  curl_global_init(CURL_GLOBAL_DEFAULT);
}

/* Downloads a file from a URL */
static void *pull_one_url(char *url)
{
  CURL *curl;
  FILE *fp;
  CURLcode res;

  curl = curl_easy_init();

  fp = fopen(getFileName(url), "wb");
  curl_easy_setopt(curl, CURLOPT_URL, url);
  curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
  curl_easy_setopt(curl, CURLOPT_WRITEDATA, fp);
  res = curl_easy_perform(curl);
  if(res != CURLE_OK)
    fprintf(stderr, "curl_easy_perform() failed: %s\n",
          curl_easy_strerror(res));
  /* always cleanup */
  curl_easy_cleanup(curl);
  fclose(fp);

  return NULL;
}
```

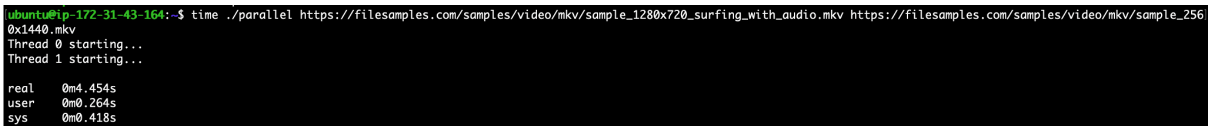
The above file contains all the methods required to download and save a file from the url, **getFileName** method extracts file name from the url, **init_curl** method initialises the global state of curl, **pull_one_url** downloads and saves a file from url.

III. Results



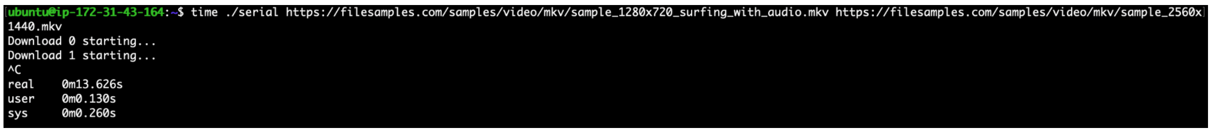
```
ubuntu@ip-172-31-43-164:~$ time ./serial https://filesamples.com/samples/video/mkv/sample_1280x720_surfing_with_audio.mkv
Download 0 starting...

real    0m3.515s
user    0m0.113s
sys     0m0.239s
ubuntu@ip-172-31-43-164:~$ time ./parallel https://filesamples.com/samples/video/mkv/sample_1280x720_surfing_with_audio.mkv
Thread 0 starting...

real    0m3.444s
user    0m0.140s
sys     0m0.212s
```

Fig 1. Result of serial and parallel downloading 1 file



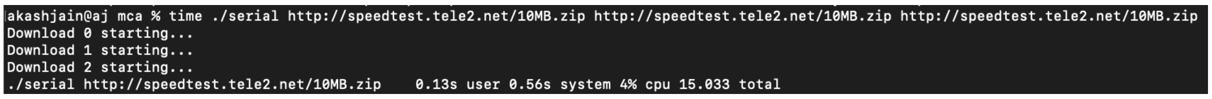
```
ubuntu@ip-172-31-43-164:~$ time ./parallel https://filesamples.com/samples/video/mkv/sample_1280x720_surfing_with_audio.mkv https://filesamples.com/samples/video/mkv/sample_256
0x1440.mkv
Thread 0 starting...
Thread 1 starting...

real    0m4.454s
user    0m0.264s
sys     0m0.418s
```

Fig 2. Result of parallel downloading 2 files



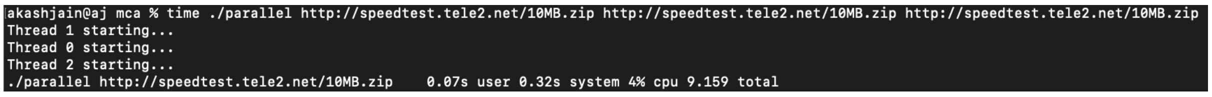
```
ubuntu@ip-172-31-43-164:~$ time ./serial https://filesamples.com/samples/video/mkv/sample_1280x720_surfing_with_audio.mkv https://filesamples.com/samples/video/mkv/sample_2560x
1440.mkv
Download 0 starting...
Download 1 starting...
^C
real    0m13.626s
user    0m0.130s
sys     0m0.260s
```

Fig 3. Result of serial downloading 2 files



```
akashjain@aj mca % time ./serial http://speedtest.tele2.net/10MB.zip http://speedtest.tele2.net/10MB.zip http://speedtest.tele2.net/10MB.zip
Download 0 starting...
Download 1 starting...
Download 2 starting...
./serial http://speedtest.tele2.net/10MB.zip    0.13s user 0.56s system 4% cpu 15.033 total
```

Fig 4. Result of serial downloading 3 files



```
akashjain@aj mca % time ./parallel http://speedtest.tele2.net/10MB.zip http://speedtest.tele2.net/10MB.zip http://speedtest.tele2.net/10MB.zip
Thread 1 starting...
Thread 0 starting...
Thread 2 starting...
./parallel http://speedtest.tele2.net/10MB.zip    0.07s user 0.32s system 4% cpu 9.159 total
```

Fig 5. Result of parallel downloading 4 files

No.of Files Downloaded	**Average File Size**	**Time taken by parallel downloading**	**Time taken by serial downloading**
1	64MB	3.444s	3.515s
2	67.16MB	4.454s	13.626s
3	10MB	9.15s	15.033s

Table 1. Comparison of download time for different no. of files

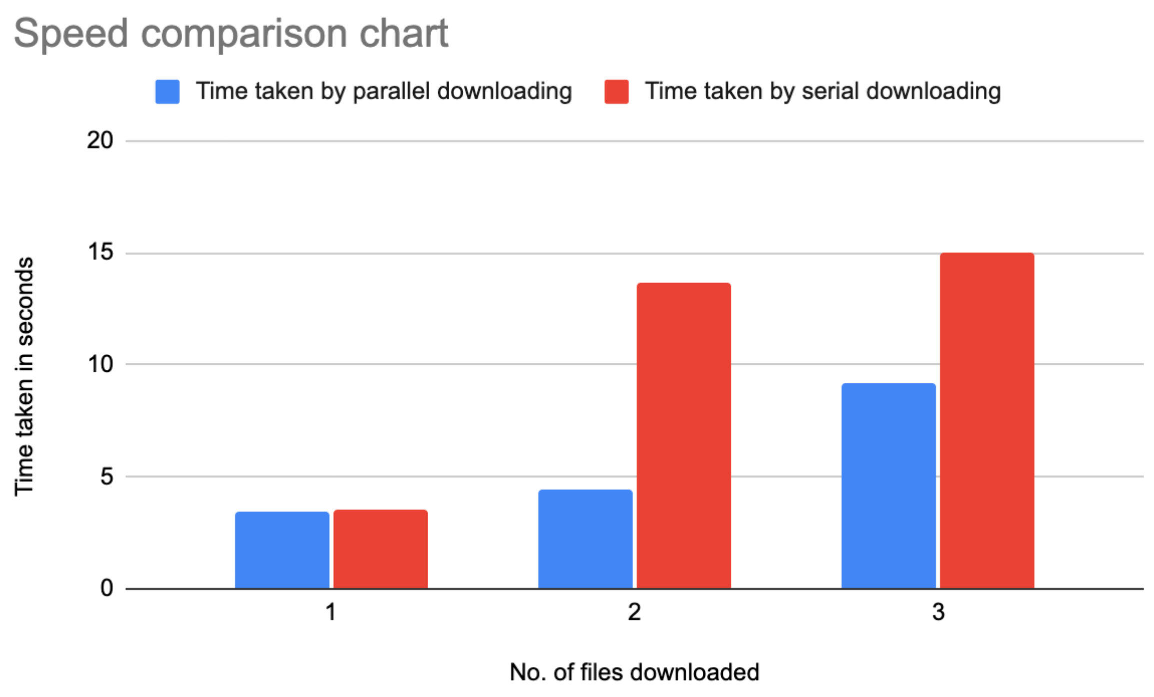


Fig 6. Speed comparison chart for serial and parallel downloading of files

From the above chart we can infer that the download speed of files in parallel is better as compared to the speed when downloaded in serial fashion.

References:

[1] OpenMP Architecture Review Board, ―OpenMP Application Program Interface,‖ 2008, http://www.openmp.org/mp-documents/spec30.pdf.

[2] B. Barney, Introduction to Parallel Computing, Lawrence Livermore National Laboratory, 2007, https://computing.llnl.gov/tutorials/parallel_comp/ .

[3] R. Awari, "Parallelization of shortest path algorithm using OpenMP and MPI," 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2017, pp. 304-309, doi: 10.1109/I-SMAC.2017.8058360.

[4] Parallel Programming with OpenMP in C | by Samir Huseynzade | Medium

[5] Sharma, Sanjay. (2012). Performance Analysis of Parallel Algorithms on Multi-core System using OpenMP. International Journal of Computer Science, Engineering and Information Technology. 2. 55-64. 10.5121/ijcseit.2012.2506.