Parallelization Algorithm for Android OS using OpenMP

Marin Aranitasi	extsuperscript{1}, Gent Daci	extsuperscript{2}, Erid Bejko	extsuperscript{3} and Igli Tafa	extsuperscript{4}

	extsuperscript{1}Center for R&D on IT, Polytechnic University of Tirana
Tirana, Albania
maranitasi@fti.edu.al

	extsuperscript{2}Department of Computer Engineering, Polytechnic University of Tirana
Tirana, Albania
ggdaci@fti.edu.al

	extsuperscript{3}Department of Computer Engineering, Polytechnic University of Tirana
Tirana, Albania
ebejko@fti.edu.al

	extsuperscript{4}Department of Computer Engineering, Polytechnic University of Tirana
Tirana, Albania
itafa@fti.edu.al

Abstract
Developments in mobile technology have escalated with unprecedented speeds in the last years in all areas of implementation. One of these areas is multicore computing and parallel programming. Though most modern mobile devices are at least multithreaded and commonly multicore, few applications can benefit such technology. In this paper we show the benefits of multithreaded programming using OpenMP, a freeware API which allows users to parallelize specific segments of a program written in C or other languages. The testing has been done in a quad core processor (hence, without using hyper threading technology) and even in this scenario the benefit is evident.

Keywords: Mobile devices, Multicore processing, Multithreading, Parallel algorithms, Parallel programming.

1. Introduction
Computing power along with many other technology elements has had an exponential growth in the latest years, whereas a high-class workstation of the 1980s is a common personal computer nowadays. The introduction of multicore processors has drastically increased processing power, in the sense that a user can run multiple processes theoretically at the same time without causing too much trouble to the CPU. This part is pretty simple because most processes are sequential and every one accesses its own thread. This way the work is divided pretty easily. The tricky part is when we build a CPU-hungry application that has to access all the resources a CPU can offer. In this case parallelization techniques come in handy. The most commonly used technique is through OpenMP, a freeware API [1] which can be setup and utilized through gcc[2] released under the General Public License. Through OpenMP we can parallelize code using the fork-join model. It’s clear that not all parts of code can be parallelized, thus we should rearrange the code in such way that most of it can be parallelized. Another known method for parallelizing programs is MPI (Message Passing Interface), but we chose to stay with the first one due to the fact that MPI is harder to implement and is commonly used in cluster computing [3]. The code we chose to parallelize is related with the calculation of prime numbers from 1 (which is not a prime) to a specific number. The problem though may look pretty simple but in reality it’s not. The fact is we cannot calculate how many prime numbers we have in this segment. We have to check them one by one to give an answer. In this case parallelization is the perfect tool that comes in handy, since each iteration doesn’t depend from the previous step.
2. Related work

Since the idea and the algorithm was very clear to us, the hard part was the OpenMP implementation on a Android OS and due to the lack of experience, the syntax and commands of OpenMP. The implementation part was overcome easily through [4] an open source package installer of gcc and all the necessary components to run a fully functional OpenMP application on a Android based mobile phone. On the other side the literature available about OpenMP is immense and makes it really easy to get your application ideas up and running in no time [1]. Furthermore, even though our idea was the least to say spontaneous, it’s amazing how many similar works you can find online. This shows the many implementations OpenMP has in various programming areas. Many of these implementations were related to algorithm optimization such as sorting algorithms or sequence matching [5] and most importantly with mathematical implementations such as differential equation solving, root finding etc [6]. All these works, even though not specifically related to my work, helped me build a robust and hazard free code, avoiding the commonly known race conditions.

3. Experimental theory

The reason we chose prime number calculation is mainly related to the fact that it requires a lot of time and CPU power to give the exact number of primes i.e. from one to one million. If we raise the stakes even more the Calculation may require days to be completed on a regular sequential way. On the other hand the importance of the problem can be classified scientific for many reasons. The first one being that the number of primes is infinite as Euclid’s theorem suggests. The other practical importance is in the RSA encryption method [7] which is based on prime numbers. The larger the numbers, the more secure the encryption.

The algorithm which we have built works in two main parts or loops being more specific. The first loop goes through numbers 2 and the number specified and checks if the number is a prime or not. This is the loop in which the code gets forked, since each iteration doesn’t depend on the previous one. Each number is checked using a self-built function called is_prime(). The function goes in another loop which does the modulo operation, hence finds the remainder of division of our number with every other smaller number. If this operations gives a remainder of 0, the number in question is not a prime number. The second loop is not forked due to the nature of the problem. In the end the outer loop is joined, and we get the result through a count variable which is shared and used from all the threads. Figure 1 depicts the flowchart of the above explained algorithm [8].

The trickiest part of the code is the parallelization of the for loop, which is easily solved by the possibilities OpenMP gives us. The #pragma omp for directive applied right before the for loop divides each iteration to the cores in automatic saving a lot of programming time. On the other hand, the #pragma omp parallel shared(count) directive makes sure the count variable is initialized only once and utilized by all the threads in the same [9].

The time is calculated from within the program and displayed during the execution.

![Flowchart of prime number calculation](image)

Figure 1: The flowchart algorithm of prime number calculation
4. Experimental phase

As mentioned before, the operating system we used to execute our application is Linux based Android. The mobile device is Samsung Galaxy S5, quad core processor running at an over clocked frequency of 2.5 GHz.

After entering in the root of the Linux kernel of the device and navigating to the correct directory, we give the following command:

```
gcc -fopenmp prime_numbers.c -o prime
```

The fopenmp flag shows that we’re about to compile a multithreaded application from the `prime_numbers.c` code and we want to save the executable file with the name prime. After the code compiles we simply type `prime` in the terminal, where prime is the name of the executable we just created and the program executes as shown in Figure 2.

```
gcc -fopenmp prime_numbers.c -o prime
```

```
prime
The program displays the number of prime numbers from 1 to the number specified
Please specify the last number 1000000
Number of primes calculated in serial is 9592
Executed in 0.000000 seconds
Number of primes calculated in parallel is 9592
Executed in 0.000000 seconds
```

The program first asks us to input the number of elements which will be checked. After that is done the program does all the rest. The code which can be reviewed in the Appendix is built in such way that both serial and parallel calculations are done in a single execution to get a better idea of the advantages and disadvantages in time. Below we have a detailed resume of runtimes in serial and parallel. A fact worth being mentioned is that with just a few lines of code we can built an application which consumes a huge amount of processing time and power.

Table 1 shows the execution times for various cases in serial, whereas Table 2 shows the execution time for the same cases in parallel.

### Table 1: Execution times in serial

<table>
<thead>
<tr>
<th>Case number</th>
<th>Elements</th>
<th>Ex. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3000</td>
<td>0.031</td>
</tr>
<tr>
<td>2</td>
<td>4000</td>
<td>0.047</td>
</tr>
<tr>
<td>3</td>
<td>5000</td>
<td>0.062</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>0.14</td>
</tr>
<tr>
<td>5</td>
<td>20000</td>
<td>0.46</td>
</tr>
<tr>
<td>6</td>
<td>50000</td>
<td>2.436</td>
</tr>
<tr>
<td>7</td>
<td>100000</td>
<td>8.999</td>
</tr>
<tr>
<td>8</td>
<td>200000</td>
<td>34.221</td>
</tr>
</tbody>
</table>

### Table 2: Execution times in parallel

<table>
<thead>
<tr>
<th>Case number</th>
<th>Elements</th>
<th>Ex. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3000</td>
<td>0.016</td>
</tr>
<tr>
<td>2</td>
<td>4000</td>
<td>0.017</td>
</tr>
<tr>
<td>3</td>
<td>5000</td>
<td>0.017</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>0.078</td>
</tr>
<tr>
<td>5</td>
<td>20000</td>
<td>0.308</td>
</tr>
<tr>
<td>6</td>
<td>50000</td>
<td>1.763</td>
</tr>
<tr>
<td>7</td>
<td>100000</td>
<td>6.618</td>
</tr>
<tr>
<td>8</td>
<td>200000</td>
<td>24.881</td>
</tr>
</tbody>
</table>

5. Conclusions

As the above results clearly show, the advantages of the parallel execution are evident and beyond any doubt. The fascinating part is in the first three cases in which the elements were going up by a thousand and yet the execution times in parallel remained the same, while in serial they were growing proportionally. The difference is evident in the case 3 scenario where we get a parallel execution time three times faster than the serial one. From that point forward the ratio begins to drop. One explanation is that the second for loop begins to kick in a lot more than the first times, and since this loop is not parallelized, it diminishes the advantages of multithreading. So the usage of parallel execution increases the performance of the
mobile device, so directly increases the possibility of the detection of possible malware attacks.

6. Future Work

As future works we see the possibility of expanding this specific application by making it even more robust and giving it a better multithreaded approach. On the other hand, the field of parallel programming is immense and there are many algorithms that can leave space for improvement or revision by utilizing this technology. Another thing we intend exploring deeper is the field of mobile security, and how the implementation of such type of algorithms can affect the security performance of the mobile device.[10]

References

[1] Chandra R. (2009), Parallel Programming in OpenMP
The source code of the algorithm written in C:

```c
#include <stdio.h>
#include <omp.h>
#include <time.h>

int is_prime(int x)
{
    int i;
    for (i = 2; i <= x; i++)
    {
        if((x%i == 0) && (i != x))
            return 0;
    }
    return 1;
}

void main()
{
    int i, number;
    int count_s = 0;
    int count_p = 0;
    clock_t begin_s, end_s, begin_p, end_p;
    float seconds_s, seconds_p;
    printf("The program displays the number of prime numbers from 1 to the number specified\n");
    printf("Please specify the last number ");
    scanf("%d", &number);

    //Single-Threaded calculation of prime numbers
    begin_s = clock();
    for(i = 2; i < number; i++)
    {
        if(is_prime(i) == 1)
            count_s++;
    }
    end_s = clock();

    //Multi-Threaded calculation of prime numbers
    begin_p = clock();
    #pragma omp parallel default(none), private(i), shared(count_p, number)
    {
        #pragma omp for
        for(i = 2; i < number; i++)
        {
            if(is_prime(i) == 1)
                count_p++;
        }
    }
    end_p = clock();

    seconds_s = (float)(end_s - begin_s)/CLOCKS_PER_SEC;
    printf("Executed in %f seconds\n", seconds_s);
    printf("Number of primes calculated in parallel is %d\n", count_p);
    seconds_p = (float)(end_p - begin_p)/CLOCKS_PER_SEC;
    printf("Executed in %f seconds\n", seconds_p);
}
```