JAVA CONCURENT PROGRAM FOR THE SAMARANDACHE FUNCTION

David Power* Sabin Tabirca* Tatiana Tabirca**

*University College Cork, Computer Science Department

*University of Manchester, Computer Science Department

Abstract: The aim of this article is to propose a Java concurrent program for the Smarandache function based on the equation $S(p_1^{k_1}\cdot\ldots\cdot p_s^{k_s})=\max\{S(p_1^{k_1}),\ldots,S(p_s^{k_s})\}$. Some results concerning the theoretical complexity of this program are proposed. Finally, the experimental results of the sequential and Java programs are given in order to demonstrate the efficiency of the concurrent implementation.

1. INTRODUCTION

In this section the results used in this article are presented briefly. These concern the Smarandache and the main methods of its computation. The Smarandache function [Smarandache, 1980] is $S: N^* \to N$ defined by

$$S(n)=\min\{k\in N | k! \vdots n\}\ (\forall n\in N^*). \tag{1}$$

The main properties of this function are presented in the following

$$(\forall a,b\in N^*)\,(a,b)=1 \Rightarrow S(a\cdot b)=\max\{S(a),S(b)\} \tag{2}$$

that gives us

$$S(p_1^{k_1}\cdot\ldots\cdot p_s^{k_s})=\max\{S(p_1^{k_1}),\ldots,S(p_s^{k_s})\}. \tag{3}$$

An important inequality satisfied by the function S is

$$(\forall a\in N^*)\ S(a)\le a\text{, the equality occurring } \textit{iff } a \text{ is prime.} \tag{4}$$

When the number a is not prime this inequality can be improved by $(\forall a\in N^*: \text{a not prime})\ S(a)\le \frac{a}{2}$.

During the last few years, several implementation of The Smarandache function have been proposed. Ibstedt [1997, 1999] developed an algorithm based on Equation (3). The

implementation in U Basic provided a efficient and useful program for computing the values of S for large numbers. Based on it Ibstedt [1997, 1999] studied several conjectures on the Smarandache function. No study of the theoretical complexity has provided for this algorithm so far.

The second attempt to develop a program for the Smarandache function was made by Tabirca [1997]. Tabirca started from Equation (1) and considered the sequence $x_k = k! \bmod n$. The first term equal to 0 provides the value $S(n)$. Unfortunately, the C++ implementation of this algorithm has been proved not to be useful because it cannot be applied for large value of n. Furthermore, this is not an efficient computation because the value $S(n)$ is computed in $O(S(n))$. A study of the average complexity [Tabirca, 1997a, 1998], [Luca, 1999] gave that the average complexity of this algorithm is $O\left(\frac{n}{\log n}\right)$.

2. AN EFFICIENT ALGORITHM FOR THE SMARANDACHE FUNCTION

In this section we develop an efficient version of the algorithm proposed by Ibsedt. A theoretical study of this algorithm is also presented. Equation (3) reduces the computation of $S(n)$ to the computation of the values $S(p_i^{k_i}), i = 1,...,s$. The equation [Smarandache, 1980] that gives the value $S(p^k)$ is given by

$$k = \sum_{i=1}^{l} d_i \cdot \frac{p^i - 1}{p-1} \Rightarrow S(p^k) = \sum_{i=1}^{l} d_i \cdot p^i . \tag{5}$$

This means that if $(d_l, d_{l-1},...,d_1)$ is the representation of k in the generalized base $1, \frac{p^2-1}{p-1},...,\frac{p^l-1}{p-1}$, then $(d_l, d_{l-1},...,d_1)$ is the representation of $S(p^k)$ k in the generalized base $p, p^2,...,p^l$. Denote $b1[i] = \frac{p^i - 1}{p-1}$ and $b2[i] = p^i$ the general terms of these two bases. We remark that the terms of the above generalized bases satisfied:

$$b1[1] = 1, b1[i+1] = 1 + p \cdot b1[i] \tag{6}$$

$$b2[1] = p, b2[i+1] = p \cdot b2[i]. \tag{7}$$

```
public static long Value (final long p, final long k) {
        long l, j, value=0;
        long b1[] = new long [1000]; long b2[] = new long [1000];
        b1[0]=1;b2[0]=p;
        for(int l=0;b1[l]<=k;l++){b1[l+1]=1+p*b1[l]; b2[l+1]=p*b2[l];}
        for(l--,j=l;j>=0;j--){d=p/b1[j];p=p%b1[j];value+=d*b2[j]; }
        return value;
}
```

Figure 1. Java function for $S(p^k)$.

Equation (5) provides an algorithm that is presented in Figure 1. At the first stage this algorithm finds the largest l such that $b1[l] \le k < b1[l+1]$ and computes the generalized bases $b1$ and $b2$. At the second stage the algorithm determines the representation of k in the base $b1$ and the value of this representation in the base $b2$.

Theorem 1. *The complexity of the computation* $S(p^k)$ *is* $O(\log_p p \cdot k)$.

Proof. Let us remark that the operation number of the function Value is $5 \cdot l$, where l is the largest value such that $b1[l] \le k < b1[l+1]$. This gives the following equivalences

$$\frac{p^l - 1}{p-1} \le k < \frac{p^{l+1} - 1}{p-1} \Leftrightarrow p^l - 1 \le k \cdot (p-1) < p^{l+1} - 1 \Leftrightarrow$$

$$\Leftrightarrow p^l \le k \cdot (p-1) + 1 < p^{l+1} \Leftrightarrow l \le \log_p [k \cdot (p-1) + 1] < l + 1 \Leftrightarrow$$

$$\Leftrightarrow l = \lfloor \log_p (k \cdot (p-1) + 1) \rfloor.$$

Therefore, the number of operations is $5 \cdot \lfloor \log_p (k \cdot (p-1) + 1) \rfloor = O(\log_p (k \cdot p))$. ♦

The computation of $S(n)$ is obtained in two steps. Firstly, the prime number decomposition $n = p_1^{k_1} \cdot \ldots \cdot p_s^{k_s}$ is determined and all the values $S(p_i^{k_i}), i = 1,\ldots,s$ are found by using a calling of the function Value. Secondly, the maximum computation is used to find $\max\{S(p_1^{k_1}),\ldots,S(p_s^{k_s})\}$. A complete description of this algorithm is presented in Figure 2.

```
public static long S (final long n) {
        long d, valueMax=0, s=-1;
        if (n= =1) return 0;
        long p[] = new long [1000]; long k[] = new long [1000]; long value[] = new long [1000];
        for(d=2;d<n;d++) if (n % d == 0){
                s++;p[s]=d;for(k[s]=0;n%d==0;k[s]++,n/=d);
                value[s]=Value(p[s],k[s]);
                }
        for(j=0;j<=s;j++) if (valueMax<value[j])valueMax=value[j];
        return valueMax;
}
```

Figure 2. Java function for $S(n)$.

Theorem 2. *The complexity of the function S is* $O(\frac{n}{\log n})$.

Proof. In order to find the prime number decomposition, all the prime numbers less than n should be checked. Thus, at most $\pi(n) = O(\frac{n}{\log n})$ checking operations are performed [Bach & Shallit, 1996] to find the prime divisors $p_1,...,p_s$ of n. The exponents $k_1,...,k_s$ of these prime numbers are found by $k_1 + ... + k_s$ divisions. An upper bound for this sum is obtained as follows

$$n = p_1^{k_1} \cdot ... \cdot p_s^{k_s} \Rightarrow \log n = \log p_1^{k_1} \cdot ... \cdot p_s^{k_s} = \log p_1^{k_1} + ... + \log p_s^{k_s} =$$

$$= k_1 \cdot \log p_1 + ... + k_s \cdot \log p_s \geq k_1 + ... + k_s,$$

because each logarithm is greater than 1. Thus, we have $k_1 + ... + k_s \leq \log n = O(\log n)$.

The computation of all the values $S(p_i^{k_i}), i = 1,...,s$ gives a complexity equal to $\sum_{i=1}^{s} \log_{p_i} p_i \cdot k_i$. An upper bound for this sum is provided by the following inequality

$\log_{p_i} p_i \cdot k_i \le k_i$ that is true because of $p_i \cdot k_i \le p_i^{k_i}$. Taking the sum we find $\sum_{i=1}^{s} \log_{p_i} p_i \cdot k_i \le \sum_{i=1}^{s} k_i = O(\log n)$, therefore the complexity of this computation is $O(\log n)$.

Finally, observe that the maximum $\max\{S(p_1^{k_1}),...,S(p_s^{k_s})\}$ is determined in $s \le \log n$ operations.

In conclusion, the complexity of the Smarandache function computation is $O(\frac{n}{\log n})$. ♦

	n=10000	n=20000	n=30000	n=40000	n=50000	n=60000	n=70000	n=80000
A 1	2804	10075	21411	36803	56271	79304	105922	136567
A 2	2925	10755	23284	39967	61188	86555	115837	149666

Table 1. Running times for the efficient and Tabirca's algorithms.

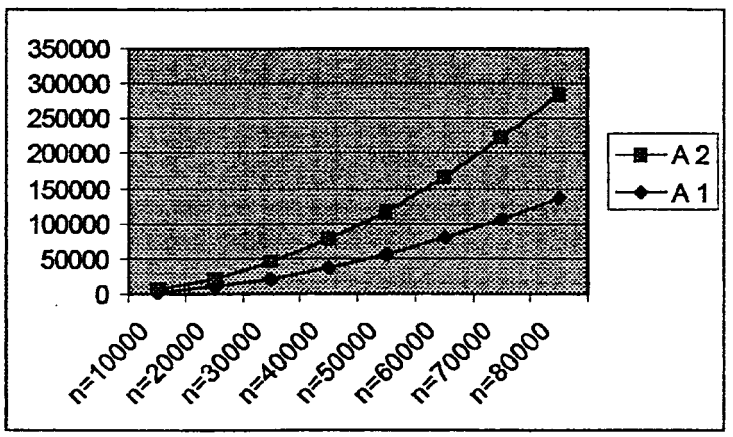

Figure 3. Graphics of the Running Times.

Several remarks can be made after this theorem. Firstly, we have found that finding the prime divisors of n represents the most expensive operation and this gives the complexity of the function computation. Secondly, we have obtained an algorithm with the complexity $O\left(\frac{n}{\log n}\right)$. Therefore, this is better than the algorithm proposed by Tabirca [1988] that has the average complexity $O\left(\frac{n}{\log n}\right)$. Table 1 shows that this algorithm also offers better running times than the

algorithm proposed in [Tabirca, 1997]. These two algorithms were implemented in Java and executed on PENTIUM II machine. The times [milliseconds] of the computation for all the values $S(i)$, i=1,...,n were found, where n=10000,....,80000. Row A 1 gives the times for this efficient algorithm and row A 2 gives the times for the algorithm proposed in [Tabirca, 1999]. Another important remark drawn from Table 1 is that the difference between the times of each column does not increase faster [see Figure 3]. This is happen because the complexity of the algorithm proposed by Tabirca [1997] is $O\left(\frac{n}{\log n}\right)$.

3. JAVA CONCURRENT ALGORITHM FOR THE SMARANDACHE FUNCTION

In this section we present a Java concurrent program for the computation described in Section 2. Firstly, remark that many operations of this algorithm can be performed in parallel. Consider that we know all the prime numbers less than n. Usually, this can be done by using special libraries. Let $p_1,...,p_s$ be these numbers. Therefore, we can concurrently execute the computation of the exponent of p_i and the computation of the value $S(p_i^{k_i})$.

A Java program may contain sections of code that are executed simultaneously. An independent section of code is known as a thread or lightweight process [Smith, 1999]. The implementation presented here is based on equation (3): $S(p_1^{k_1}\cdot...\cdot p_s^{k_s})=\max\{S(p_1^{k_1}),...,S(p_s^{k_s})\}$. Each $S(p_i^{k_i})$ is calculated concurrently in a thread. On single processor systems, the use of threads simulates the concurrent execution of some piece of sequential code. The worst case execution time can be taken as the longest execution time for a single thread. On a multi processor system, given enough processors, each thread should ideally be allocated to a processor. If there are not enough processors available, threads will be allocated to processors in groups. Unlike pure concurrent processes, threads are used to simulate concurrency within a single program. Most current everyday programs use threads to handle different tasks. When we click a save icon on a word processing document typically a thread is created to handle the actual saving action. This allows the user to continue working on the document while another process (thread in this case) is writing the file to disk.

For the concurrent algorithm consider the Java function for $S(n)$ in Figure 2. Typical areas that can be executed concurrently can be found in many loops, where successive iterations of the loop

do not depend on results of previous iterations. In Figure 4, we adapt the for loop (Figure 2) to execute the Value function (Figure 1), responsible for calculating $S(p^k)$, concurrently by creating and executing a ValueThread object. When all the required threads have begun execution, the value of max will not be known until they have completed. To detect this, a simple counter mechanism is employed. As threads are created the counter is incremented and as threads complete their tasks the counter is decremented. All threads are completed when this counter reaches 0.

```
public long S(long n)
{
        if (n==1) return (long);

        Prime decom = new Prime(n);
        noPrimes=decom.noPrime();
        if (noPrimes == 0)
                value = null;
        value = new long[noPrimes];

        for (int k=0;k<noPrimes;k++)
        {
                started++;
                new ValueThread(decom.getPrime(k), decom.getPow(k), this, k);
        }

        while (started > 0)
        {
                try
                {
                        Thread.yield();
                }catch (Exception e)
                {
                }
        }
        return max;
}
```

Figure 4. Modified Java function for $S(n)$, used to concurrently execute the Value function

As each thread completes its task it executes a callback method, addValue (Figure 5). This method is declared as synchronized to prevent multiple threads calling the addValue method at the same time. Should this be allowed to occur, an incorrect value of the number of threads executing would be created. Execution of this method causes the value array declared in method

S (Figure 3) to be filled. This value array will only be completely filled after the last thread makes a call to the addValue method. At this point, the value of max can be determined.

```
public synchronized void addValue(int k, long val)
{
        value[k] = val;
        max = value[0];
        started--;
        if (started == 0)
                for (int i=1; i<=k; i++)
                        if (value[i] > max)
                                max = value[i];
  }
```

Figure 5. The addValue method called by a Thread when its task is completed.

This algorithm illustrates how concurrency can be employed to improve execution time. It is also possible to parallelise the algorithm at a higher level, by executing the function responsible for calculating each $S(n)$ in an independent thread also. Tests of this mechanism however show that it is more efficient to only parallelise the execution of $S(p^t)$.

The concurrent Java program has been run on a SGI Origin 2000 parallel machine with 16 processors. The execution was done with 1, 2, 4 processors only and the execution times are shown in Table 1. The first line of Table 1 shows the running times for Algorithm A1 on this machine. The next three lines present the running times for the concurrent Java program when p=1, p=3 and p=4 processors are used.

	n=20000	n=30000	n=40000	n=50000	n=60000	n=70000	n=80000
A 1	9832	19703	31237	49774	68414	96242	115679
CA (p=1)	9721	19474	30195	49412	68072	95727	115161
CA (p=2)	5786	11238	22872	31928	42825	60326	75659
CA (p=4)	3863	7881	14017	19150	30731	42508	53817

Table 2. Running Times for the Concurrent Program.

4. CONCLUSSIONS

Several remarks can be drawn after this study. Firstly, Equation (3) represents the source of any efficient implementation of the Smarandache function. In Section 2 we have proposed a sequential algorithm with the complexity $O\left(\frac{n}{\log n}\right)$. We have also proved both theoretically and practically that this algorithm is better that the algorithm developed in [Tabirca, 1997].

Secondly, we have developed a Java concurrent program in order to decrease the computation time. Based on the thread technique we have performed concurrently the computation of the values $S(p_i^{k_i})$. This concurrent implementation has proved to be better than the sequential one. Even running with one single processor the times of the concurrent Java program were found better than the times of the sequential program.

References

Bach, E. and Shallit, J. (1996) *Algorithmic Number Theory*, MIT Press, Cambridge, Massachusetts, USA.

Ibstedt, H. (1997) *Surfing on the Ocean of Numbers - a few Smarandache Notions and Similar Topics*, Erhus University Press, New Mexico, USA.

Ibstedt, H. (1999) *Computational Aspects of Number Sequences*, American Research Press, Lupton, USA.

Luca, F. (1999) The average Smarandache function, Personal communication to S. Tabirca. [will appear in *Smarandache Notion Journal*].

Tabirca, S. and Tabirca, T. (1997) Some Computational Remarks on the Smarandache Function, *Proceedings of the First International Conference on the Smarandache Type Notions*, Craiova, Romania.

Tabirca, S. and Tabirca, T. (1997a) Some upper bounds for Smarandache's function, *Smarandache Notions Journal*, **8**, 205-211.

Tabirca, S. and Tabirca, T. (1998) Two new functions in number theory and some upper bounds for Smarandache's function, *Smarandache Notions Journal*, **9**, No. 1-2, 82-91.

Smarandache, F. (1980) A Function in number theory, *Analele Univ. Timisoara*, **XVIII**.

Smith, M. (1999) *Java an Object – Oriented Language*, McGraw Hill, London, UK.

Appendix A

The full code for the concurrent implementation presented in Section 3.

```
// Smarandache.java

import java.io.*;
import java.util.*;

public class Smarandache
{
        public Smarandache()
        {
                long n=0, i, j;
                long val;
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                try
                {
                        System.out.print ("n = ");
                        n = Integer.parseInt(br.readLine());
                }catch (IOException e)
                {
                        System.out.println ("IOException : "+e.getMessage());
                        System.exit(1);
                }

                Smar sm = new Smar();

                Date begin = new Date();
                for (i=1; i<= n; i++)
                {
                        val = sm.S(i);
                }
                Date end = new Date();
                System.out.println ("Time good is "+ (end.getTime() - begin.getTime()));
        }

        public static void main (String args[])
        {
                new Smarandache();
        }
}
```

```
// Smar.java

public class Smar
{
        private long value[];
        private long max = Long.MIN_VALUE;
        private int noPrimes=0;
        private int started = 0;

        public Smar()
        {
        }

        public long S(long n)
        {
                if (n==1)
                        return (long) 0;

                Prime decom = new Prime(n);
                noPrimes=decom.noPrime();
                if (noPrimes == 0)
                        value = null;
                value = new long[noPrimes];

                for (int k=0;k<noPrimes;k++)
                {
                        started++;
                        new ValueThread(decom.getPrime(k), decom.getPow(k), this, k);
                }

                while (started > 0)
                {
                        try
                        {
                                Thread.yield();
                        }catch (Exception e)
                        {
                        }
                }
                return max;
        }
```

```
        public synchronized void addValue(int k, long val)
        {
                value[k] = val;
                started--;
                if (started == 0)
                {
                        max = value[0];
                        for (int i=1; i<=k; i++)
                                if (value[i] > max)
                                        max = value[i];
                }
        }
}
```

```
//Prime.java

public class Prime
{
        private int s;
        private long p[]=new long [1000];
        private int ord[]=new int [1000];

        public Prime()
        {
                s=0;
        }

        public Prime(long n)
        {
                long d;
                for(d=2,s=0;d<=n;d++)
                if(n%d == 0)
                {
                        p[s]=d;
                        for(ord[s]=0;;ord[s]++,n=n/d){if(n%d!=0)break;};
                        s++;
                }
        }

        public int noPrime()
        {
                return s;
        }

        public long getPrime(int i)
        {
                return p[i];
        }
```

```
	public int getPow(int i)
	{
		return ord[i];
	}
}
```

```
// ValueThread.java

public class ValueThread
{
	private long p=0, a=0;
	private Smar owner;
	private int index = 0;

	public ValueThread (long p, long a, Smar owner, int index)
	{
		this.p = p;
		this.a = a;
		this.owner = owner;
		this.index = index;
		run();
	}

	public long pseuPow(long p, long a)
	{
		if (a == 1)
			return (long) 1;
		return 1+p*pseuPow(p,a-1);
	}

	public long Pow(long p, long a)
	{
		if (a == 1)
			return (long) p;
		return p*Pow(p,a-1);
	}

	public void run()
	{
		long rest=a, val=0;
		int k, i;
		for(k=1;pseuPow(p,k)<=a;k++);k--;
		for(i=k;i>0;i--)
		{
			val += Pow(p,i)* (long)(rest / pseuPow(p,i));
			rest %= pseuPow(p,i);
		}
		owner.addValue(index, val);
	}
}
```