OpenMP Implementation on RSA ENCRYPTION

High Performance Computing Project Report

Problem Statement: Encryption and Decryption of Very Large Numbers using the RSA Algorithm

Paul Babu Kadali - CED19I002

Project Description:

We implemented the RSA algorithm to deal with very big numbers and not to use the ready libraries/. This code is capable of Encrypt using public or private keys which is both the encryption and decryption of data. Can take 512 bits each from P and Q.

OpenMP Parallel Code

The total code is more than 700 lines, So I'm only inserting critical code and the functions which I've parallelized here.

```
BigNum Add(BigNum first, BigNum second)
{
    if(EqualZero(first))
        return second;

    if(EqualZero(second))
        return first;
```

```
    BigNum Result;
    int val = 0, carry = 0;
    bool BothNegative = false;

    if(first.negative && second.negative)
        BothNegative = true;

    else if(first.negative)
    {
        first.negative = false;
        Result = Sub(second, first);
        return Result;
    }
    else if(second.negative)
    {
        second.negative = false;
        Result = Sub(first, second);
        return Result;
    }
    int i = 0;
    #pragma omp parallel
    // {
        for(i ; i < Size2048; i++)
        {
            val = (first.Num[i] + second.Num[i] + carry ) % 100;
            carry = (first.Num[i] + second.Num[i] + carry ) / 100;
            Result.Num[i] = val;
        }
    // }

    if(carry != 0)
        Result.Num[i] = carry;


    Result.negative = BothNegative;
    return Result;

}

BigNum Sub(BigNum firstOriginal, BigNum second)
{
    //Op1 - Op2 .. first - second
```

```
        if(EqualZero(second))
        {
            return firstOriginal;
        }
        if(EqualZero(firstOriginal))
        {
            second.negative = true;
            return second;
        }


        BigNum Result, tempResult, first;
        first = CopyOf(firstOriginal);
        int val = 0, NextToMe = 0;
        bool LastNum = false;


        if(second.negative)
        {
            if(first.negative)
            {
                first.negative = false;
                second.negative = false;
                Result = Sub(second, first);
                return Result;
            }
            else
            {
                second.negative = false;
                Result = Add(first, second);
                return Result;
            }
        }
        else
        {
            if(first.negative)
            {
                first.negative = false;
                second.negative = false;
                Result = Add(first, second);
                Result.negative = true;
                return Result;
            }
```

```
    }


    int i = 0;
    #pragma omp parallel
    // {
        for(i; i < Size2048; i++)
        {
            if(LastNum)
                break;
            if(first.Num[i] >= second.Num[i])
            {
                val = first.Num[i] - second.Num[i];
                Result.Num[i] = val;
            }
            else
            {
                if(i == Size2048)
                    LastNum = true;

                int temp = i;
                while(temp < 308)
                {
                    temp++;
                    NextToMe++;
                    if(first.Num[temp] != 0)
                    {
                        first.Num[temp] -= 1;
                        first.Num[i] = first.Num[i] + 100;

                        NextToMe--;

                        temp = i+1;
                        i--;
                        while(NextToMe != 0)
                        {
                            first.Num[temp] = 99;
                            NextToMe--;
                            temp++;;
                        }
                        break;
                    }
                    else if(first.Num[temp] == 0 && temp == 308)
```

```
                        LastNum = true;
                }
            }
        }
    // }

    if(LastNum == true)
    {
        Result = Sub(second, firstOriginal);
        Result.negative = true;
        return Result;
    }

    return Result;

}

BigNum Mul(BigNum first, BigNum second)
{
    BigNum Result;

    if(EqualZero(first))
        return first;
    if(EqualZero(second))
        return second;


    int val = 0, carry = 0;
    bool neg = false;

    if(first.negative && second.negative)
    {
        first.negative = false;
        second.negative = false;
    }
    else if(first.negative)
    {
        first.negative = false;
        neg = true;
    }
    else if(second.negative)
    {
        second.negative = false;
```

```
        neg = true;
    }

    int i ,j;
    #pragma omp parallel
    // {
        for(i = 0; i < Size2048; i++)
        {
            BigNum temp;
            for(j = 0; j < Size2048; j++)
            {
                val = ((first.Num[i] * second.Num[j]) + carry ) % 100;
                carry = ((first.Num[i] * second.Num[j]) + carry ) / 100;

                temp.Num[j] += val;
            }

            if(i!=0)
            {
                for(int k = 308; k != 0; k--)
                {
                    temp.Num[k] = temp.Num[k-i];
                }
                for(int k = 0; k < i; k++)
                {
                    temp.Num[k] = 0;
                }
            }
            Result = Add(Result, temp);
        }
    // }

    Result.negative = neg;
    return Result;
}

DivResult DivSmall(BigNum first, BigNum second)
{
    DivResult Result;

    if(EqualZero(first))
    {
        Result.Result = first;
```

```
        Result.Remainder = first;
        return Result;
    }

    BigNum countArray, One, tempResult = first;
    One.Num[0] = 1;

    bool CheckNeg = tempResult.negative;
    do
    {
        tempResult = Sub(tempResult, second);
        CheckNeg = tempResult.negative;
        if(countArray.Num[0] != 99)
            countArray.Num[0] = countArray.Num[0]+1;
        else
            countArray = Add(countArray, One);
    }
    while(!CheckNeg);

    if(countArray.Num[0] != 0)
    {
        countArray.Num[0] = countArray.Num[0]-1;
        Result.Result = countArray;
    }
    else
        Result.Result = Sub(countArray, One);

    Result.Remainder = Add(tempResult, second);

    return Result;
}
```

Hardware Profiling with likwid:

```
root@felicity:~# lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
    CPU family:          6
    Model:               142
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            10
    CPU max MHz:         3400.0000
    CPU min MHz:         400.0000
    BogoMIPS:            3600.00
```

Number of NUMA domains: 1
likwid-pin pins threads to cores so applications don’t migrate over the course of the job execution and loose cache locality.

```
root@felicity:/media/root/e4a48a94-0dd2-4202-a0b8-0622fe5b880a/School/Semester-7/Parallel-Computing/RSA_Paralleli
zation# likwid-topology -G
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU type:       Intel Kabylake processor
CPU stepping:   10
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               1               0               0               *
5               1               1               0               *
6               1               2               0               *
7               1               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
```

```
root@felicity:/media/root/e4a48a94-0dd2-4202-a0b8-0622fe5b880a/School/Semester-7/Parallel-Computing/RSA_Parallelization# likwid-pin -c 0-8 ./a.out
[pthread wrapper]
[pthread wrapper] MAIN -> 0
[pthread wrapper] PIN_MASK: 0->1  1->2  2->3  3->4  4->5  5->6  6->7
[pthread wrapper] SKIP MASK: 0x0
        threadid 140228767884864 -> hwthread 1 - OK
        threadid 140228759492160 -> hwthread 2 - OK
        threadid 140228751099456 -> hwthread 3 - OK
        threadid 140228742706752 -> hwthread 4 - OK
        threadid 140228734314048 -> hwthread 5 - OK
        threadid 140228725921344 -> hwthread 6 - OK
        threadid 140228717528640 -> hwthread 7 - OK
P:90659935589672134066418069661244537071942135986254663159165058746421811420293
```

The parallelized code runs on 8 initialized threads. (OMP threads set to 8 in this case)
Likwid-perfctr reports on hardware performance events, such as FLOPS, bandwidth, TLB misses and power.

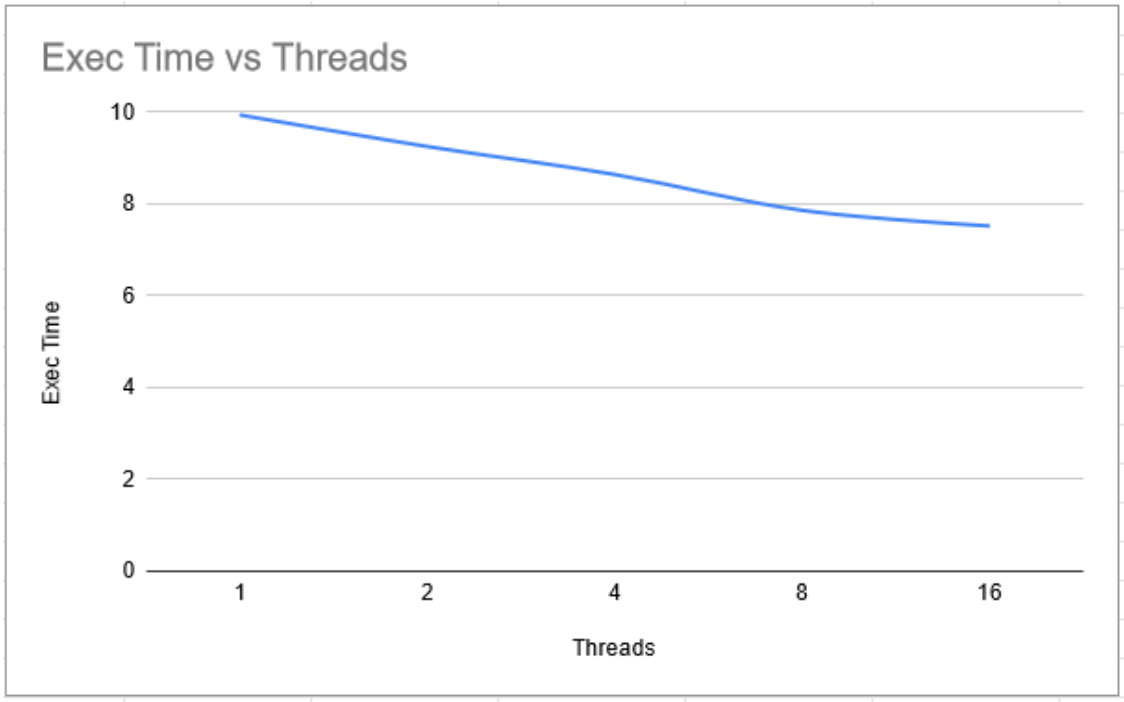
Observations:

Threads	Exec Time	Speedup	Parallelization Factor
1	9.9364	1	0
2	9.2496	0.2180226172	-2.988900085
4	8.64	0.2334053241	-2.189594943
8	7.86	0.2565676845	-0.9658689961
16	7.52	0.2681678191	0.9096694042

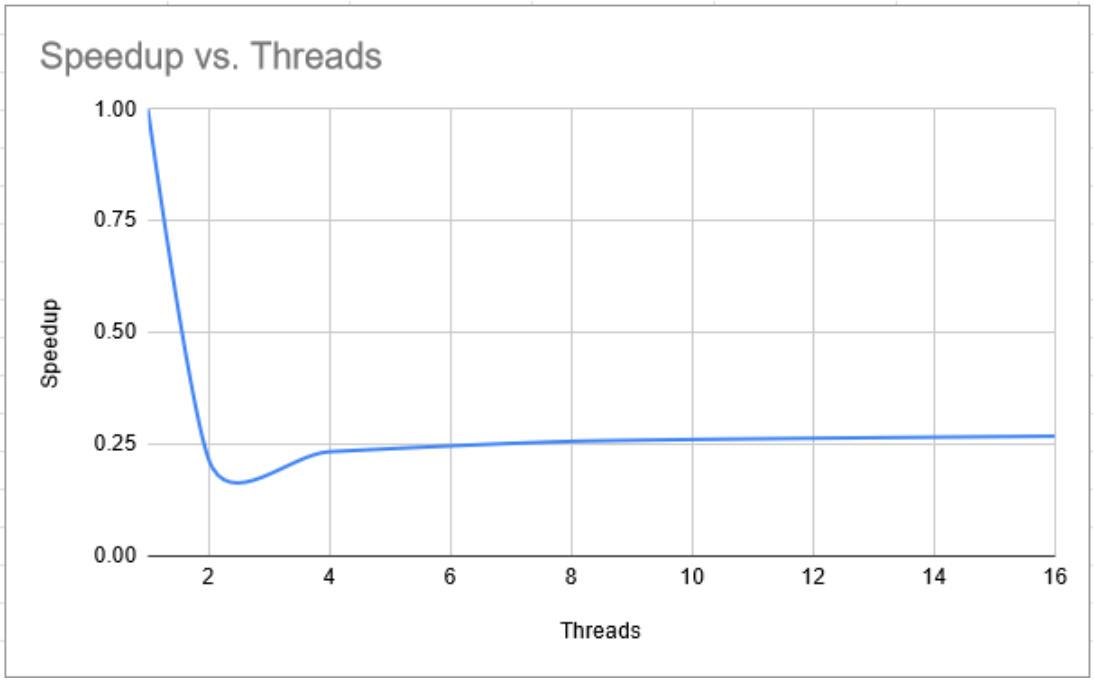
Speed up can be found using the following formula,
$S(n)=T(1)/T(n)$
where, S(n) = Speedup for thread count ‘n’
T(1) = Execution Time for Thread count ‘1’ (serial code)
T(n) = Execution Time for Thread count ‘n’ (serial code)

Parallelization Fraction can be found using the following formula,
$S(n)=1/((1 - p) + p/n)$
where, S(n) = Speedup for thread count ‘n’
n = Number of threads
p = Parallelization fraction

Execution Time vs Threads



Speedup vs Threads



Inference:

From the above observations, we can see that the parallel code scales well with increase in threads. The program sees a reduction in execution time till 16 threads, after which there is negligible increase in speedup. Beyond 32 threads, the cost of context switching takes over and there is an increase in execution time compared to the previous runs.