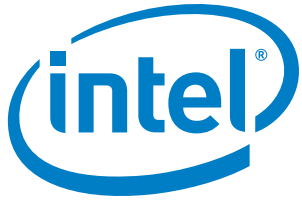

Scalable Parallelism with Intel® Threading Building Blocks

This paper discusses the approach to parallelize the Data Encryption Standard (DES) algorithm with Intel® Threading Building Blocks and how it can scale for future processors.

Shwetha Doss
Software and Solutions Group
Intel Corporation

Introduction

Multicore technology is pervasive. To take advantage of these multicore processors, software developers are required to develop highly parallel applications that can scale up to today's and tomorrow's processors.

Intel® Threading Building Blocks (TBB) is a C++ runtime library that abstracts the low-level threading details and provides a set of generic constructs which helps you write scalable parallel programs.

This paper discusses the approach to parallelize the Data Encryption Standard (DES) algorithm with Intel® Threading Building Blocks and how it can scale for future processors. It is organized as follows. Section 2 contains a description of application scalability. Section 3 briefly describes the DES algorithm. Section 4 describes the TBB implementation of the DES algorithm. Section 5 presents experimental results obtained for a parallel DES algorithm.

Scalability

Scalability is the quality of an application to expand efficiently to accommodate greater computing resources so as to improve application performance. For example, it can refer to the capability of a system to increase total throughput under an increased load when resources are added. Some of the resources might be the increase in number of cores and threads in the new processors or an increase in memory capacity.

Factors inhibiting scalability

There are various factors that inhibit application scaling. Intel® Thread Profiler and VTune™ Performance Analyzer are a couple of the tools which can help you identify possible problems that prevent your application from scaling.

Some of the common factors inhibiting scalability are:

Granularity and parallel overhead

Granularity is defined as the ratio of computation to synchronization of the threads. While dividing the computation into independent tasks, a fine-grained decomposition yields a large number of smaller tasks and a coarse-grained decomposition yields a small number of larger tasks. To benefit from threading, the amount of computation for each thread should be larger than the overhead of thread creation, synchronization, scheduling, and management.

Load imbalance

Load imbalance is a situation where all the threads are not doing equal amounts of computation. Load imbalance occurs when the granularity of tasks is large enough that some threads are left idle for a significant percentage of the computation time, waiting for the busy threads to finish. It represents the force-opposing parallel overhead that determines the ideal granularity. This results in idle processor time which, in turn, impacts performance.

Synchronization overhead

Synchronization constructs are necessary in parallel applications to ensure correct results when dealing with shared data, but they reduce performance because the code inside the synchronization construct is serial. To mitigate this problem, use atomic synchronization constructs like Interlocked operations available in Windows* threads or the #pragma omp atomic available in OpenMP* programming. They are ideal for incrementing a single variable, since these constructs consume fewer cycles than critical sections, mutex, and semaphores.

Amdahl's law

Amdahl's law predicts the estimated speedup that can be achieved by converting a serial application to a parallel application. This first step estimates the benefit you can achieve by threading your application on a particular processor. Also, it gives an estimate of how well the application scales as more processors/cores are added.

$$T_{parallel} = \{(1-P) + P/\text{Number of processors}\} T_{serial} + \text{overhead}$$

Tserial refers to the time taken to run the application before introducing any threads. Let's assume that we decide to thread a particular region of code, so the P in Amdahl's law refers to the fraction of the calculation that can be parallelized. The portion of code that is not threaded remains serial and its time is given by (1-P). A certain amount of overhead is introduced while parallelizing the application due to the synchronization constructs.

$$\text{Scaling} = T_{serial} / T_{parallel}$$

Data Encryption Standard (DES) Algorithm

DES was published in 1977 by the National Bureau of Standards for use in commercial and unclassified US Government applications. DES uses an electronic codebook (ECB) encryption mode, where the 64-bit input plain text is converted to a 64-bit ciphertext using the 56-bit key.

Algorithm overview

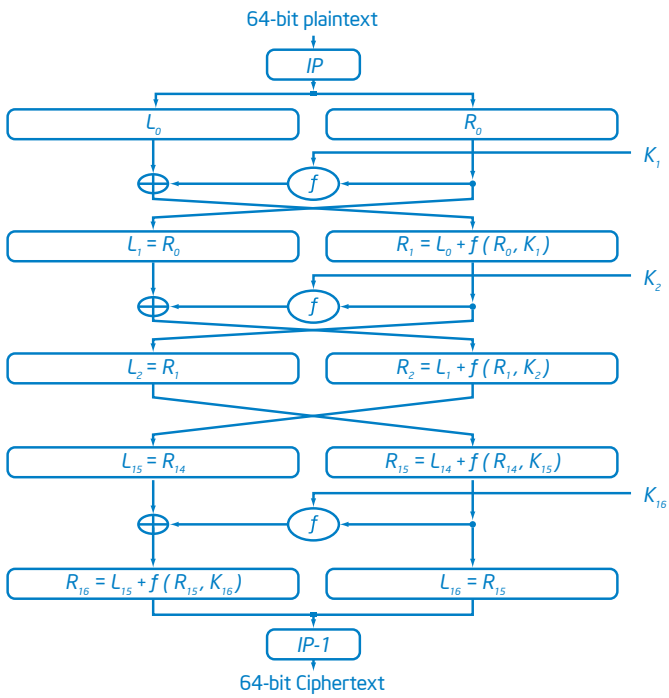

Figure 1: Describes the Data Encryption Standard Algorithm

DES operates on 64-bit blocks of plaintext. After the initial permutation, each block is broken into L_0 and R_0 components, each 32-bits long. sixteen rounds of identical operations follow, where the function, *f*, combines the data with the key. After the sixteenth round, R_{16} and L_{16} are combined and a final permutation (inverse of the initial permutation) finishes off the algorithm.

$$L_i = R_i - 1$$

$$R_i = L_i - 1 \text{ XOR } f(R_i - 1, K_i)$$

In each round, the key bits are shifted, and then 48 bits are selected from the 56 key bits. The right half of the data is expanded from 32 bits to 48 bits via an expansion permutation, combined with 48 bits of shifted and permuted key via an XOR, substituted for 32 new bits using a substitution algorithm and permuted again. These four operations comprise the function *f*. The output of *f* is combined with the left half via XOR. The result of these operations becomes the new left half; the old left half becomes the new right half. These operations are repeated 16 times, making 16 rounds of DES.

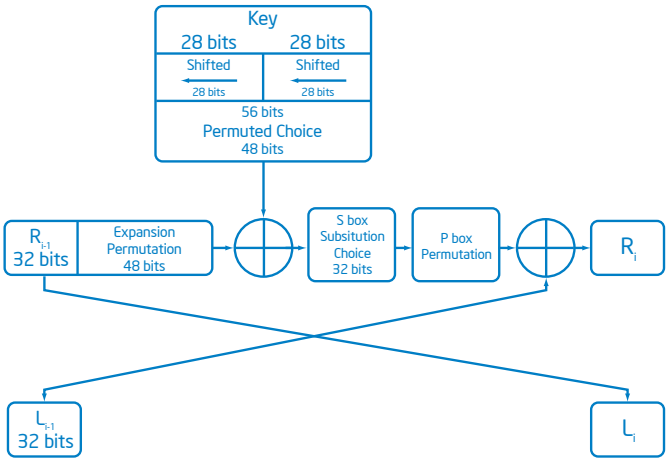

Figure 2: Illustrates the function *f*( ) of DES

Initial permutation

The initial permutation transposes the input block. The 64 bit plain-text is permuted based on the table below:

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Key transformation

The 64-bit DES key is reduced to a 56-bit key by ignoring every eighth bit. After that, the keys are permuted based on the table below::

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

The 56-bit key is divided into two 28 bit halves. Then, the halves are shifted left by either one or two bits depending on the round.

Round	Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Compression permutation

The initial permutation transposes the input block. The 64 bit plain-text is permuted based on the table below:

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Expansion permutation

In this operation the right half of the data R_i is expanded from 32 bits to 48 bits. Because this operation changes the order of the bits as well as repeating certain bits (hence increasing the number of bits); this stage is known as an expansion permutation.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

S-box substitution

After the compressed key is XORed with the expanded block, the 48-bit results moves to a substitution operation. The substitutions are performed by eight substitution boxes or S-boxes.

P-box permutation

The 32-bit output of the S-box substitution is permuted according to a P-box. This permutation maps each input to an output position; no bits are used twice and no bits are ignored.

16	2
7	8
20	24
21	14
29	32
12	27
28	3
17	9
1	19
15	13
23	30
26	6
5	22
18	11
31	4
10	25

The P-box permutation is XORed with the left half of the initial 64-bit block

Final permutation

The final permutation is the inverse of the initial permutation and is based on the table below:

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Scalability analysis of DES

VTune Performance Analyzer provides different collectors like Sampling, Call Graph, and Counter Monitor, which collect information to help you tune your application and system for performance.

In this case we used VTune event-based sampling. Sampling is a non-intrusive technology that periodically interrupts the processor based upon user configured counters, and takes a snapshot of the execution context. The Sampling technology enables you to analyze system-wide software performance. It helps you to identify the performance critical processes, threads, modules, functions, and lines of code running on your system. This can include code, which is consuming most of the CPU time or generating the majority of processor stalls. Event-based sampling collects clock ticks and instructions retired by default.

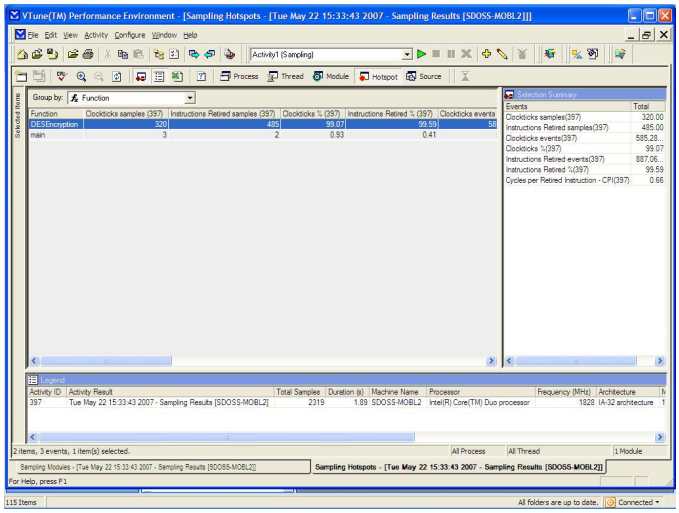

Figure 3: Illustrates the DESEncryption( ) consuming 99% of the clock-ticks of the entire application

For the DES Encryption, algorithm it indicates that the *for* loop in the `DESEncryption( )` function that encrypts blocks in the ECB (Electronic Code Block) mode is the hot function since it consumes 99% of the clock ticks for the entire application. The clock ticks as indicated by VTune Analyzer give the P for Amdahl's law.

The scaling expected on a 4-processor system, assuming there is no overhead in creating and managing the threads, is

Scaling (4 processor) = 100/ {1 + 99/4}

= 100/25.75

= 3.88X

Since we are ignoring the overhead in creating and managing the threads, the scaling we achieve is always less than the expected scaling, 3.88X.

Also, the computations inside the *for* loop in `DESEncryption( )` are data parallel, meaning that each thread can do the same calculations on different data. Since there are no dependencies between each 64-bit block of data, each can be encrypted separately. All of these factors make `DESEncryption( )` a good candidate for threading.

Intel Threading Building Blocks

Intel Threading Building Blocks is a C++ runtime library that provides high-level generic implementations of parallel control structures and concurrent data containers.

Intel Threading Building Blocks represents the culmination of years of threading research. The dilemma in dealing with ever-increasing counts of cores is efficient thread scheduling. Rather than dealing with threads, Intel TBB presents the idea of tasks. These are units of work that can be scheduled on physical threads by the library in an efficient manner, balancing the system overhead and load imbalance while maximizing cache conservation.

Applications written with Intel TBB are portable across Microsoft* Windows*, Linux* and Mac* operating systems.

Intel TBB overview

Intel TBB provides a variety of parallel control structures that are suited for loop parallelizing, concurrent data containers, as well as a scalable memory allocator, task scheduler, and a variety of synchronization primitives.

Intel® Threading Building Blocks Features

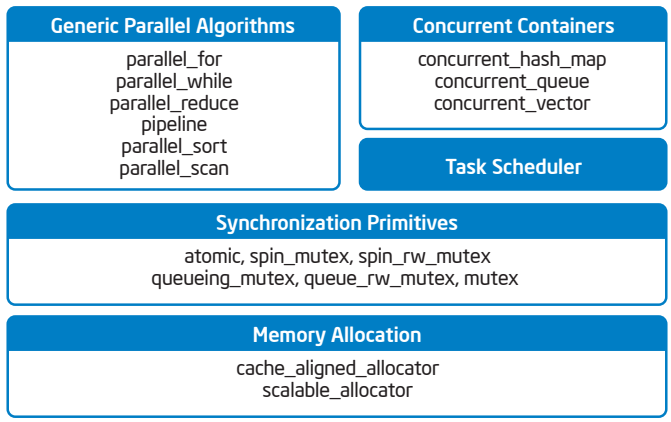

Figure 4: Illustrates the different constructs in the TBB library

Parallel algorithms

The easiest route to parallelism is to parallelize a loop whose iterations can each run independently.

The signature for parallel_for is :

```
parallel_for(blocked_range<size_
t>(0,n,IdealGrainsize), Instance of the
clsss(arguments))
```

The *parallel_for* function divides the iteration space into multiple iteration spaces. Each of these iteration spaces is given to the task scheduler, each of which can be mapped to independently executing threads. *Parallel_for* works in conjunction with the *blocked_range* interval construct to define a range for dividing work among the tasks. The developer can use *blocked_range2d* for two-dimensional space and can define his own iteration space.

Balance comes with the selection of a grain size, a minimum sub-interval to divide the range: too small a number pushes the application into the range of excessive parallel overhead while a range that is too large leads to load balance issues, particularly as the application is moved to machines with more cores.

Grainsize

Grain-size represents the smallest sub-range that is considered profitable for moving to another core. Ideal grain size is the happy balance between parallel overhead and load imbalance.

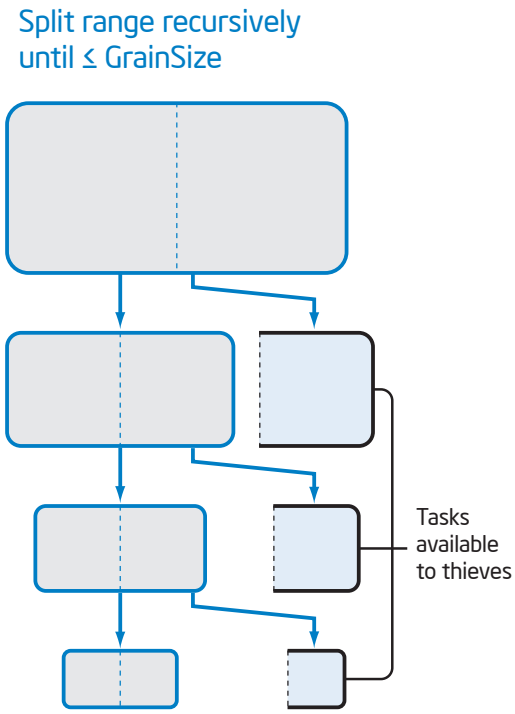

Figure 5: Illustrates the how the parallel constructs in TBB work

Grainsize helps you overcome the parallel overhead cost. If the grain size is too small, the overhead may exceed the useful work. If the grain size too large, the parallelism may be reduced.

Intel Threading Building Blocks version 1.1 supports *auto_partitioner( )* as a tech preview feature. The *auto_partitioner()* heuristically chooses a grain size, which aims to reduce overhead while avoiding load imbalance.

The signature for *parallel_for()* with *auto_partitioner( )* is:

```
parallel_for(blocked_range<size_
t>(0,n), Instance of the class(arguments),
auto_partitioner())
```

Parallelizing using TBB

Initializing the task scheduler

When the Intel Threading Building Blocks task scheduler is initialized with default values, the thread pool that will be used for processing the tasks is created. The developer can defer the creation of this thread pool and also can control how many threads he wants to create by changing the arguments passed to the *task_scheduler_init.*

The constructor for *task_scheduler_init* takes an optional parameter that allows you to specify the number of threads to be created.

```
#include "tbb/task_scheduler_init.h"
using namespace tbb;

int main (void) {

        .......
        tbb::task_scheduler_init init;
        .......
```

The *parallel_for* control structure

In our example below, the *parallel_for( )* function breaks the iteration interval [0, blocks) into chunks of 10,000, each of which are run on a separate thread. Each of these chunks is processed by the *operator()* declared in the class *TBBDESEncryption( )*

```
void tbb_parallel_for_TBBDESEncryption
(DESStruct *dc, unsigned char * cp, int blocks)
{
        parallel_for(blocked_range<int>
(o,blocks,10000), TBBDESEncryption (dc, cp));
{
```

The *parallel_for( )* requires a copy constructor, which creates an new object and initialized it.

The *operator( )* function does the same computation on each chunk of data. A *blocked_range<int>* describes a one-dimensional iteration space over the data type *int.* The *for* loop is embodied in the *operator( )* function and this for loop is parallelized.

```
class TBBDESEncryption{
       ....
              //constructors
public:
       TBBDESEncryption (DESStruct *dc,
unsigned char* cp) :m_dc(dc),m_cp(cp)
       {
       }
       void operator () (const blocked_
range<int> &r)const
       {
              ....
              for(....)
              {
              }

       }

}
```

Scalable programming with Intel TBB

The control structures provided by Intel Threading Building Blocks support data decomposition, which makes them scalable for current and future processor cores.

Data decomposition can be defined as the computation that can be independently applied to each collection of data, permitting a degree of parallelism that can scale with the amount of data. Intel Threading Building Blocks enables multiple threads to operate on different data sets. As new processors are added, the data sets are divided into smaller sets and each processor can independently process them.

Performance Results

System configuration

The testing environment consisted of Intel® Xeon® processor 5160 (4GB RAM, two 3.0 GHz dual core processors). The 64-bit operating system was Microsoft Windows Vista Business*. The 32-bit application was compiled with the Intel® C++ Compiler for Windows* 9.1.034 and Intel Threading Building Blocks for Windows* version 1.1. The input file used for encryption was a 10MB file

Performance analysis

The serial version was compiled with O3 and vectorzing enabled. The optimizing flag O3, which is supported in the Intel® Compiler, enables aggressive loop transformations and perfecting. Intel Compiler is the only one available that supports vectorization. The vectorized serial version ran in 0.59 seconds on a 2-core system and in 0.17 seconds on a 4-core system.

The TBB version with a grain-size of 10,000 or with *auto_partitioner()* being used ran in 0.34 seconds on a 2-core system and in 0.05 seconds on a 4-core system.

Since *auto_partitioner( )* uses heuristics, there may be cases where it may not yield better performance. We recommend tuning the grain-size for the application based on system testing.

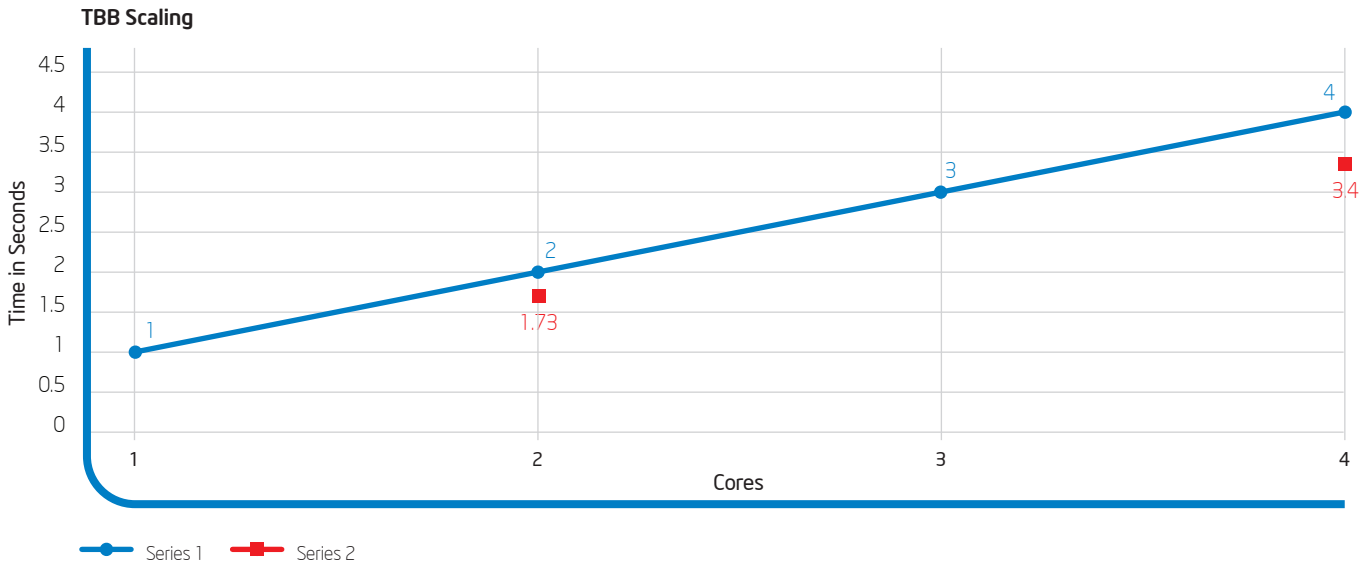

Figure 6: Compares the perfect scaling and the scaling achieved with Intel TBB

Conclusion

Software applications need to be threaded in order to take advantage of new multicore processors. Intel Threading Building Blocks provide a rich set of parallel algorithms which are tested and tuned for current and future Multi-core processors. Developers can achieve scalable parallelism with Intel TBB and can easily adapt to the new growing processor technology.

References

1. Charlie Kaufman, Radia Perlman, Mike Speciner "Network Security PRIVATE Communication in a Public World 2nd edition"

2. Bruce Schneier "Applied Cryptography, Protocols, Algorithms and Souce Code in C"

3. DES in Federal FIPS 46 and 81 standards http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

4. Parallelization of the Data Encryption Standard (DES) algorithm http://www.springerlink.com/content/v6h008374v67048v/

5. Intel® Threading Building Blocks Product Web page http://www3.intel.com/cd/software/products/asmo-na/eng/294797.htm

6. Demystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms http://www.devx.com/cplus/Article/32935