Parallelization of the IDEA Algorithm

Vladimir Beletskyy and Dariusz Burak

Faculty of Computer Science & Information Systems, Technical University of Szczecin,
49 Żołnierska St, 71-210 Szczecin, Poland
{vbeletskyy, dburak}@wi.ps.pl

Abstract. In this paper, we present results of parallelizing the International Data Encryption Algorithm (IDEA). The data dependence analysis of loops was applied in order to parallelize this algorithm. The OpenMP standard is chosen for presenting the parallelism of the algorithm. The efficiency measurement for a parallel program is presented.

1 Introduction

Considering the fact that a relatively large part of the sequential C source code implementing the IDEA algorithm is filled in with "for" or "do-while" loops and the most of computation is comprised in these loops, there is an opportunity to parallelize this algorithm. A parallel IDEA algorithm permits us to reduce the time of running cryptographic tasks on multiprocessor computers. This problem is also connected with the current world tendency to hardware implementations of cryptographic algorithms (just because we also need parallel algorithms in this case).

The International Data Encryption Algorithm (IDEA), developed at Swiss Federal Institute of Technology in Zurich by James L. Massey and Xuejia Lai, published in 1990 (the algorithm was called IPES (Improved Proposed Encryption Standard) until 1991), and popularized by commercial versions of the PGP protocol, is used worldwide in various banking and industry applications.

The purpose of this paper is to present the IDEA algorithm parallelization.

2 Algorithm Parallelization

A C source code of the sequential IDEA algorithm in the ECB mode contains eight "for" or "do-while" loops (including no I/O function) [1].

We have used Petit to find dependences in source loops and the OpenMP standard to present parallelized loops. Developed at the University of Maryland under the Omega Project and freely available for both DOS and UNIX systems, Petit is a research tool for analyzing data dependences in sequential programs [2].

The OpenMP Application Program Interface (API) supports multi-platform shared memory parallel programming in C/C++ and Fortran on all architectures including Unix and Windows NT platforms. OpenMP is a collection of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism [3].

To build the valid parallel program, it is necessary to preserve all the dependences of the program [4].

The process of the IDEA algorithm parallelization can be divided into the following stages:

- carrying out the dependence analysis of a sequential source code in order to detect parallelizable loops,
- selecting parallelization and transformation methods,
- constructing sources of parallel loops in accordance with the OpenMP API requirements.

The most time-consuming are the idea_enc() and the idea_dec() functions presented below [1]:

2.1

```
void idea_enc (idea_ctx *c, unsigned char *data, int
blocks) {
int i;
unsigned char *d = data;
for (i=0; i<blocks; i++) {
          ideaCipher (d, d , c->ek);
          d += 8;
          }
}
```

2.2

```
void idea_dec (idea_ctx *c, unsigned char *data, int
blocks) {
int i;
unsigned char *d = data;
for (i=0; i<blocks; i++) {
          ideaCipher (d, d , c->dk);
          d += 8;
          }
}
```

Taking into account the strong similarity of these loops (there is the only difference between them – the first loop operates on variable "ek", the second does on "dk"; variables "ek" and "dk" are of the same type), we examine only the 2.1 "for" loop. However, this analysis is also valid in the case of the 2.2 "for" loop.

The parallelization process of the 2.1 loop consists of the five following steps:

- filling in the 2.1 "for" loop by the body of the function ideaCipher(d,d,c->ek) (otherwise, we cannot apply the data dependence analysis),
- conversion of the nested "do-while" loop [1] to an equivalent nested "for" loop,
- replacement of pointer operations with suitable array indexing for "in" and "out" variables,

- removal of the expression “d += 8;” located in the end of the original loop body and the insertion of the statements assigning values to the variables inbuf and outbuf, “inbuf = &d[8*i];” and “outbuf = &d[8*i];”, respectively, in the beginning of the transformed loop body,
- appropriate variables privatization using OpenMP standard directives and clauses.

The skeleton of the parallel 2.1 "for" loop is the following:

```
#pragma omp parallel private
(i,ii,t16,t32,x1,x2,x3,x4,inbuf,outbuf,key,s2,s3,
in,out)
#pragma omp for
for (i=0;i<blocks;i++) {
          inbuf = &d[8*i];
          outbuf = &d[8*i];
          key = c->ek;
          in = (word16 *)inbuf;
          x1 = in[0];
          ...
          for (ii=0;ii<8;ii++) {
          ...
          }
          ...
          out[0] = x1;
          ...
}
```

The innermost "for" loop, included in the parallel 2.1 loop, is unparallelizable without applying advanced techniques of parallelization due to existing anti and output dependences and pointer operations.

The 2.2 "for" loop was parallelized in the same way as the 2.1 loop.

In the similar way, we have parallelized two "for" loops included in the ideaExpandKey() function and one loop included in the ideaInvertKey() function [1].

The remaining three sequentially iterated loops are unparallelizable. There are two reasons of this:
- occurrence of both data dependences and pointer operations in the loop body,
- occurrence of the instruction "return" in the loop body.

3 Experiments

In order to study the efficiency of the parallelization proposed, the Omni OpenMP Compiler has been used to run the IDEA sequential and parallel algorithms. The results received for a 15 megabytes input file using a PC computer with four processors Xeon, 2 GHz is shown in Table 1.

The total running time of the IDEA algorithm consists of the following operations: data receiving from an input file, data encryption and data decryption, and data writing to an output file (both encrypted and decrypted text).

The speed-up of the IDEA parallel algorithm depends considerably on the two factors: the parallelism degree of the idea_enc() and idea_dec() functions and choosing functions responsible for reading data from an input file and writing data to an output file.

The results confirm that the idea_enc() and idea_dec() functions are parallelizable with high speed-up (see Table 1).

The block method of reading data from an input file and writing data to an output file was used. The following C language functions and block sizes was applied: the fread() function and the 10-bytes block for data reading and the fwrite() function and the 512-bytes block for data writing.

The parallelization of the loops included in the ideaExpandKey() and the ideaInvertKey() functions has only a minimal influence on the speed-up value in the case of the software implementation but can be useful for hardware implementations of the parallel IDEA algorithm.

Table 1. Speed-ups of the sequential and the parallel IDEA algorithms

The number of processors	Total time of the IDEA sequential algorithm (sec)	Total time of the IDEA parallel algorithm 1 (sec)	Total time of the IDEA parallel algorithm 2 (sec)	**Total speed-up** of algorithm 1	Total speed up of algorithm 2	The idea_enc() time of the sequential algorithm (sec)	The idea_enc() time of the parallel 1 algorithm (sec)	**The des_enc() speed-up** of algorithm 1	The idea_dec() time of the sequential algorithm (sec)	The idea_dec() time of the parallel algorithm 1 (sec)	**The des_dec() speed-up** of algorithm 1
1	38,70	38,70	38,70	1	1	17,40	17,40	1	19,35	19,35	1
2	-	20,95	21,00	**1,85**	1,84	-	8,90	**1,96**	-	9,80	**1,97**
3	-	14,65	14,75	**2,64**	2,62	-	6,00	**2,90**	-	6,60	**2,93**
4	-	10,85	10,95	**3,57**	3,53	-	4,35	**4,00**	-	4,85	**3,99**

The idea parallel algorithm 1 contains parallel 2.1 and 2.2 "for" loops.
The idea parallel algorithm 2 contains five parallel "for" loops.

References

1. Bruce Schneier: Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, John Wiley & Sons; 2 edition, 1995.
2. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott: New User Interface for Petit and Other Extensions. User Guide. 1996.
3. OpenMP C and C++ Application Program Interface. Ver.2.0. 2002.
4. R. Allen, K. Kennedy: Optimizing compilers for modern architectures: A Dependence-based Approach, Morgan Kaufmann Publishers, Inc., 2001.