Proceedings of the 4th WSEAS Int. Conf. on Information Security, Communications and Computers, Tenerife, Spain, December 16-18, 2005 (pp224-228)

Parallelization of the AES Algorithm

WŁODZIMIERZ BIELECKI, DARIUSZ BURAK
Faculty of Computer Science and Information Systems
Szczecin University of Technology
49 Zolnierska St., PL-71210 Szczecin
POLAND

Abstract: - In this paper we present the parallelization process of the AES algorithm (Rijndael) along with the description of exploited parallelization tools. The data dependences analysis of loops and appropriate loop transformations were applied in order to parallelize the sequential algorithm. The OpenMP standard was chosen for representing the parallelism of the AES algorithm. Speed-up measurements for a parallel program are presented.

Key-Words: - AES, parallelization, data dependences analysis, loop transformation, OpenMP, shared-memory multiprocessors

1 Introduction

In addition to security level, the cipher speed is the most important feature of cryptographic algorithms. It is well known that by the same security level even a little difference of the cipher speeds may cause the choice of the faster cipher. Therefore, it is so important and useful to enable the use of symmetric multiprocessors (SMP) for running in parallel these algorithms. In this paper we propose a software approach based on transformations of a source code written in C representing the sequential AES algorithm. The design of parallel algorithms is connected with the current world tendency towards the hardware implementation of cryptographic algorithms, because it is often based on parallel algorithms. The main contribution of this paper is to present the parallelization process of the AES algorithm along with the description of exploited parallelization methods and speed-up measurements. The paper is organized as follows. In section 2, we briefly review the AES algorithm. Section 3 contains a brief description of parallelization tools that were applied. In section 4, we describe in detail the parallelization process of the AES algorithm. In section 5, we present experimental results regarding the application efficiency of a parallel algorithm.

2 AES Algorithm

Advanced Encryption Standard (AES) is a NIST cryptographic algorithm standard (FIPS PUB 197). It uses the Rijndael symmetric block cipher algorithm developed by Joan Daemon and Vincent Rijmen. The Rijndael algorithm can operate on block lengths of 128, 192 or 256 bits and key lengths of 128, 192 or 256 bits. It contains a various number of rounds: 10, 12, and 14 depending on key and block lengths. The Rijndael algorithm is a substitution-linear transformation cipher (not requiring a Feistel network) that uses for each round triple discreet invertible uniform transformations, called layers: Key Addition Transform, Non-linear Transform and Linear Mix Transform. Operations in Rijndael are defined at byte level or in terms of four-byte words. Encipherment and decipherment processes require the use of several iterative operations so it underlies parallel processing.

To encrypt or decrypt more than one block, there are five official modes of the AES: ECB, CBC, CFB, OFB and CTR [1]. The detailed description of the algorithm is contained in the official Rijndael paper submitted to the NIST for consideration [2] or in the NIST publication [3].

Nowadays, AES is a standard for most government agencies, and is widely used in many applications due to such features as: high security level, high speed and appreciable facilities of software and hardware implementations.

3 Parallelization Tools

In order to parallelize the AES algorithm we have applied the Petit program in order to find data dependences in loops and the OpenMP API to present parallelized source code.

3.1 OpenMP API

The OpenMP Application Program Interface 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. OpenMP directives extend a sequential programming language with Single Program Multiple

Data constructs, work-sharing constructs, synchronization constructs and make possible to operate with shared data and private data. An OpenMP program begins execution as a single task called master thread. When a parallel region is encountered, the master thread creates a team of threads. The statements within the parallel region are executed in parallel by each thread in the team. At the end of the parallel region, the threads of the team are synchronized. Then again only the master thread continues execution until the next parallel region will be encountered. It is necessary to find remedy for all problems connected with programming restrictions on parallel processing to build a valid parallel code [4], [5].

3.2 Petit

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 array data dependences. Petit operates on programs in a simple Fortran-like language and provides indispensable information about data dependences that occur in the analyzing loop [6], [7].

4 Parallelization Process of AES

In order to parallelize the AES algorithm in the ECB mode, we have used Rafael R. Sevilla's implementation of the Rijndael cipher written in C and presented as a Perl module [8]. This choice makes possible to perform efficient parallelization in view of some advantageous features of that source code (a high clarity, enclosing the most of computations in iterative loops, a little number of used functions). In order to enable enciphering and deciphering the whichever number of data blocks, we have created the new functions, the rijndael_enc() for the encryption process and the rijndael_dec() for the decryption process, by analogy with similar functions included in the C source code of cryptographic algorithms (the des_enc(), the des_dec(), the loki_enc(), the loki_dec, the idea_enc(), the idea_dec(), etc.) presented in [9].

In the next subsection, we introduce data dependences types before we begin discuss the parallelization process of the AES algorithm.

4.1 Data Dependences

There are the following types of data dependences that occur in iterative loops [10], [11]:

- Data Flow Dependence- indicates that write-before-read ordering that must be satisfied for parallel computing. The following loop yields such dependeces:

```
for (int i=0; i<n; i++)
    a[i] = a[i-1];
```

- Data Antidependence indicates that read-before-write ordering should not be violated when performing computations in parallel. The loop bellow produces such dependeces:

```
for (int i=0; i<n; i++)
    a[i] = a[i+1];
```

- Output Dependence indicates a write-before-write ordering. The following loop produces such dependeces:

```
for(int i=0; i<n; i++)
    a[0] = a[i];
```

All of the above loops cannot be executed in parallel in such a form, because results generated by parallel loops would be not the same as those yielded with the sequential loops. Thus, it is necessary to transform these loops so as to eliminate Data Antidependences and Output Dependences (Data Flow Dependences cannot be avoided and limit parallelism).

4.2 Parallelization strategy

The data input of the parallelization process is the well-optimized sequential AES algorithm [8].

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

a) finding the most time-consuming functions of the AES algorithm;
b) making preliminary transformations of the most time-consuming loops;
c) data dependences analysis of the most time-consuming loops;
d) removal of data dependences (when possible);
e) constructing parallel loops in accordance with the OpenMP API;
f) verification of a parallelized source code.

The result of the parallelization process is a parallelized AES algorithm.

4.3 The most time-consuming functions

We have carried out experiments with the sequential AES algorithm that encrypts and then decrypts 10 megabytes plaintext in order to find the most time-consuming functions including no I/O functions. We have discovered that such functions are the rijndael_enc() and the rijndael_dec() functions presented bellow:

4.3.1 The rijndael-enc() function

```
void rijndael_enc(RIJNDAEL_context *ctx,
                  UINT8 *input, int inputlen, UINT8 *output)
{
```

```
 int i, nblocks;
 nblocks = inputlen / RIJNDAEL_BLOCKSIZE;
  for (i = 0; i<nblocks; i++) {
        rijndael_encrypt(ctx, input, output);
        input+= RIJNDAEL_BLOCKSIZE;
        output+= RIJNDAEL_BLOCKSIZE;
  }
}
```

4.3.2 The rijndael-dec() function

```
void rijndael_dec(RIJNDAEL_context *ctx,
            UINT8 *input, int inputlen, UINT8 *output)
{
 int i, nblocks;
 nblocks = inputlen / RIJNDAEL_BLOCKSIZE;
  for (i = 0; i<nblocks; i++) {
        rijndael_decrypt(ctx, input, output);
        input+= RIJNDAEL_BLOCKSIZE;
        output+= RIJNDAEL_BLOCKSIZE;
  }
}
```

4.4 Parallelization process of the most time-consuming loops

The most time-consuming loops are included in the rijndael_enc() and the rijndael_dec() functions, thus their parallelization is critical for reducing the total time of the parallel algorithm execution. Taking into account the strong similarity of these loops (there is the only difference between them: the loop included in the rijndael_enc() function calls out the rijndael_encrypt() function, the second one calls out the rijndael_decrypt() function; the rijndael_encrypt() and the rijndael_decrypt() functions are very similar), we discuss only the parallelization process of the 4.3.1 loop (however, this analysis is valid also in the case of the 4.3.2 loop).

In order to apply the data dependences analysis of the 4.3.1 loop we have to put the body of the rijndael_encrypt() function in this loop.

Next, we have to remove existing data dependences by making the following transformations:

a) insert in the beginning of the loop body the following two statements:
   "plaintext = &input[RIJNDAEL_BLOCKSIZE*i];"
   "ciphertext=&output[RIJNDAEL_BLOCKSIZE*i];"
b) remove from the end of the loop body the following two statements:
   "input+= RIJNDAEL_BLOCKSIZE;"
   "output+= RIJNDAEL_BLOCKSIZE;"
c) make the privatization of the following eight variables:
   "i", "plaintext", "ciphertext", "r", "j", "t", "e", "wtxt".

The source code of the loop transformed in accordance with the above markings is suitable to apply the following OpenMP API constructs: parallel region construct ("parallel" directive) and work-sharing construct ("for" directive). This permits us to represent the parallelization of the analyzed loop.

The rijndael_enc() function with the parallelized most time-consuming loop has the following form (in accordance with the OpenMP API):

```
void
rijndael_enc(RIJNDAEL_context *ctx,
            UINT8 *input, int inputlen, UINT8 *output)
{
 int i, nblocks;
 const UINT8 *plaintext;
 UINT8 *ciphertext;
 int r, j;
 UINT32 wtxt[4], t[4], e;
 nblocks = inputlen / RIJNDAEL_BLOCKSIZE;
#pragma omp parallel private (i, plaintext, ciphertext , r,
                              j, t, e, wtxt)
#pragma omp for
for (i = 0; i<nblocks; i++) {
  plaintext=&input[RIJNDAEL_BLOCKSIZE*i];
  ciphertext = &output[RIJNDAEL_BLOCKSIZE*i];
key_addition_8to32(plaintext, &(ctx->keys[0]), wtxt);
   for (r=1; r<ctx->nrounds; r++) {
      for (j=0; j<4; j++) {
          t[j] = dtbl[wtxt[j] & 0xff] ^
ROTRBYTE(dtbl[(wtxt[idx[1][j]] >> 8) & 0xff]^
ROTRBYTE(dtbl[(wtxt[idx[2][j]] >> 16) & 0xff] ^
ROTRBYTE(dtbl[(wtxt[idx[3][j]] >> 24) & 0xff])));
      }
key_addition32(t, &(ctx->keys[r*4]), wtxt);
  }
for (j=0; j<4; j++) {
          e = wtxt[j] & 0xff;
          e |= (wtxt[idx[1][j]]) & (0xff << 8);
          e |= (wtxt[idx[2][j]]) & (0xff << 16);
          e |= (wtxt[idx[3][j]]) & (0xff << 24);
          t[j] = e;
       }
for (j=0; j<4; j++)
     t[j] = SUBBYTE(t[j], sbox);
key_addition32to8(t,&(ctx->keys[4*ctx->nrounds]),
                        ciphertext);
 }
}
```

where "#pragma omp parallel" defines the beginning of the parallel region, and "#pragma omp for" specifies that all the iterations of the associated loop can be executed in parallel.

5 Speed-up Measurements

In order to study the efficiency of the parallel code, we used the computer with the following features:
- 64 x Itanium2 1.5GHz (SGI Altix 3700) (we use up to sixteen processors for the program execution),
- the Intel® C++ Compiler ver.9.0 (that supports the OpenMP 2.5 API).

The results received for the block length of 128 bits, the key length of 256 bits and the plaintext of the size about 17 megabytes are shown in Table 1.

No. of processors	No. of threads	Speed-up		
		Encryption	Decryption	Total
1	1	1,000	1,000	1,000
2	2	1,957	1,996	1,600
4	4	3,112	3,285	1,941
8	8	5,274	6,829	2,222
16	16	5,554	12,357	2,300

Table 1- Speed-up measurements of the AES algorithm

In order to verify the measurements that were performed, we used several plaintext sizes from 1 kilobytes to 20 megabytes and the speed-ups were very similar for all the cases.

The total running time of the AES algorithm consists of the following time-consuming operations:

a) data reading from an input file;
b) data encryption;
c) data decryption;
d) data writing to an output file (both encrypted and decrypted text).

The total speed-up of the parallelized AES algorithm depends considerably on the two major factors: whether the most time-consuming loops are parallelizable and the method of data reading and data writing.

The results confirm that the parallelized codes of the most time-consuming loops (placed in the rijndael_enc() and the rijndael_dec() functions) have sufficient efficiencies.

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 were applied: the fread() function and the 16-bytes block for data reading and the fwrite() function and the 512-bytes block for data writing. The optimal sizes of the blocks were chosen via the appropriate number of tests with various block sizes.

In accordance with Amdahl's Law the maximum speed-up of the whole AES algorithm is limited to 4.10, because the fraction of the code that cannot be parallelized is 0.244. This fraction is calculated as the quotient of the sum of the execution time of all unparallelizable operations divided by the execution time of the whole algorithm.

The difference between the speed-ups obtained for the encryption and the decryption processes and for eight and sixteen processors is due to the fact that during the decryption process (which is executed after the encryption process) data is stored in local memory.

6 Conclusions

In this paper, we describe the parallelization of the AES algorithm. The AES algorithm was divided into parallelizable and unparallelizable parts. We have shown that the iterative loops included in the most time-consuming functions (responsible for the data blocks encryption and decryption) are fully parallelizable. In order to parallelize these loops it is necessary to make appropriate transformations of the body loops (described in the section 4.4) and to use the variable privatization technique.

The experiments carried out on the SGI computer with one, two, four, eight and sixteen threads show that the application of the parallel AES algorithm considerably boost the time of the data encryption and decryption. We believe that the speed-ups received for the most time-consuming loops are satisfactory. The rest of the time-consuming parts of code, contains I/O functions that are unparallelizable because the access to memory is, by its very nature, sequential. Hence, the total speed-up is less than that for the parallelizable part. The parallel AES algorithm presented in this paper can be also helpful for hardware implementations. The hardware synthesis of the AES algorithm will depend on the appropriate adjustment of the data transmission capacity and the computational power of hardware.

References:

[1] Dworkin, M., Recommendation for Block Cipher Modes of Operation: Methods and Techniques, *NIST Special Publication 800-38A,*December 2001

[2] Daemen, J. and Rijmen, V., *AES* Proposal: Rijndael, *AES submission document on Rijndael, Version 2*, September 1999

[3] *FIPS PUB 197, Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, U.S. Department of Commerce, November 2001

[4] OpenMP Application Program Interface, Version 2.5, May 2005

[5] http://www.openmp.org

[6]Kelly,W.,Maslov,V.,Pugh,W.,Rosser,E.,Speisman,T., Wonnacott,D., *New User Interface for Petit and Others Extensions.User Guide*, December 1996

[7] http://www.cs.umd.edu/projects/omega/

[8] http://www.iaik.tu-graz.ac.at/research/krypto/AES/old/~rijmen/rijndael/

[9] Schneier, B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, John Wiley & Sons, 1995.

[10] Allen,R., Kennedy, K., *Optimizing compilers for modern architectures: A Dependence-based Approach*, Morgan Kaufmann Publishers, Inc., 2001

[11] Moldovan, D.I., *Parallel Processing. From Applications to Systems*, Morgan Kaufmann Publishers, Inc., 1993.