Project 1 Report

Xi He xxh2229@rit.edu

January 13, 2010

In project 1, I conduct a statistical test on Advance Encryption Standard (AES) and try to disprove that AES behaves as a random function. I also try to run the test as parallel as I can in order to shorten the running time. This report gives a description of how the program is designed, what is the design consideration, what is the running result of the program and so on.

1 Design

In the project, I encrypt N 128-bit plain text blocks into N 128-bit cypher text blocks, iterate each cypher text block and for each cypher text bit position count the numbers of times the cypher text bit is 1 and the numbers of times the cypher text bit is 0. Then based on the data a chi-square statistic is conducted to determine if I can disprove that AES behaves as a random function.

In my program, I use the byte type arrays to store encryption key, the plain text blocks and cypher text blocks. I also use a long type array to store the expected number of times the cipher text bit is 1.

To parallel the program, I partition N 128-bit plain text blocks into a number of parts evenly in accord with available CPU cores, and let a separate thread encrypt each part of plain text blocks and for each cypher text bit count the numbers of times the cypher text bit is 1. All of the numbers of times counted by each thread should be summed up and then the result is inputed as a parameter to a chi-square statistic. Since the problem is a massive parallel problem, there is no dependance between threads. And I think the computation for each thread is even (computation for the AES encryption for each 128-bit plain text block can be regarded as the same.), thus there is no

need to do load balancing. Since every thread needs to access the long type array storing the numbers of times the cypher text bit is 1, the program needs to synchronize the threads. However, synchronization between threads can greatly lower the program's efficiency, and as a solution we adopt the reduction pattern to reduce the synchronization. I implement the synchronization and reduction pattern using the *edu.rit.pj.reduction.SharedLongArray* class in *PJ* library.

2 Result

Table 1: For The First Input Data Set

NT	**T1**	**T2**	**T3**	**T**	**Spdup**	**Effic**	**EDSF**
seq	94834	92474	93695	92474			
1	88975	89935	89495	88975	1.039	1.039	
2	48195	47673	56654	47673	1.940	0.970	0.072
3	32059	38452	35651	32059	2.884	0.961	0.040
4	25751	23557	25713	23557	3.926	0.981	0.052
8	12789	13533	12847	12789	7.230	0.904	0.022

Table 2: For The Second Input Data Set

NT	**T1**	**T2**	**T3**	**T**	**Spdup**	**Effic**	**EDSF**
seq	189828	183535	185992	183535			
1	179779	177336	180550	177336	1.035	1.035	
2	94551	105079	96122	94551	1.941	0.971	0.066
3	62674	62348	61043	61043	3.007	1.002	0.016
4	51720	58949	48780	48780	3.763	0.941	0.033
8	25222	30717	27185	25222	7.277	0.910	0.020

From the above table, we can see that speedup increases as the number of parallel threads increase. The efficiency is 0.9 or so. Actually, the problem in the project is a massive parallel problem, and it can easily be solved in parallel and is a good candidate for an SMP parallel problem. The sequential part of

Table 3: For The Third Input Data Set

NT	**T1**	**T2**	**T3**	**T**	**Spdup**	**Effic**	**EDSF**
seq	276807	276909	278957	276807			
1	268735	269262	266312	266312	1.039	1.039	
2	177313	152146	146918	146918	1.884	0.942	0.103
3	98381	95699	94426	94426	2.931	0.977	0.032
4	76241	77597	77401	77597	3.567	0.892	0.055
8	40618	36173	36093	36093	7.669	0.959	0.012

the program can not be paralleled, and causes the discrepancy between the ideal and the measured speedup and efficiency in the program.

I think this project is useful and helps me a lot. It solidified my concept of parallel computing and share memory. I learned the performance measurement and tuning technique. Most important, I learned from this project the methodology of writing a parallel program.