Overview and Analysis of UPC as a Tool in Cryptanalysis

Technical Report

by

Burt Gordon and Nam Nguyen

Special Project for the

High-performance Computing and Simulation (HCS) Research Laboratory
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL 32611

October 31, 2003

© 2003, HCS Research Lab, U. of Florida. All Rights Reserved.

1. Introduction

Many scientific as well as commercial endeavors rely on the ability to solve complex problems in an efficient and reliable manner. One of the dominant solutions to this problem has been the emergence of parallel computing. To supplement the architectural improvements in this area, parallel programming models have emerged to give programmers alternate ways to approach solving complex and computationally intensive problems. Such models include message passing, shared memory, and distributed shared memory.

Message passing and shared memory are the two most popular ways to implement parallel programs. However, distributed shared memory is quickly gaining momentum. Unified Parallel C (UPC) has facilitated in the rise in use of the distributed shared memory model. UPC is a parallel extension to the ANSI C standard. It allows programmers the ability to create parallel programs across several parallel architectures while still maintaining a familiar C style structure. This approach allows a smaller learning curve for people with C experience to begin creating parallel programs. An important part to the rise in use of UPC has been the emergence of various compilers that support UPC on several different parallel architectures. One of the most mature UPC compilers is created and supported by Hewlett Packard. The current version is V2.1.

Implementations of message-passing models such as MPI leave the responsibility of handling the low-level communication to the programmer. For complex problems such as cryptanalysis, the burden of dealing with communication details adds much complexity to program development [6]. This extra burden raises the need for a programming language that abstracts the programmer from the underlying communication mechanisms. The primary goal of UPC is to provide an easy programming environment while not sacrificing performance.

In this paper we look at the performance of UPC on the ES80 CC-NUMA architecture from HP/Compaq for a small, simulated cryptanalysis problem called the Differential Attack Simulator.

2. UPC and Testbed Overview

2.1 UPC

Parallel programming is an important part of high-performance computing. Because of its ease of use, shared-memory parallel programming is an attractive model for parallel programmers. In particular, the ability to read and write remote memory with simple assignment statements is considerably more attractive than having to learn all the conventions of a message-passing library, even if the latter is portable. At the same time, the quest for performance often makes it desirable to view program data as distributed among a number of local memories [3]. UPC is an attempt to provide the shared programming model that is attractive while not compromising the performance desired.

UPC is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The language provides a uniform programming model for both shared and distributed memory hardware. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor [4]. The communication model in is based on the idea of a shared, partitioned address space, where variables may be directly read and

© 2003, HCS Research Lab, U. of Florida. All Rights Reserved.

written by multiple processors, but each variable is physically associated with a single processor. Under the UPC programming model, THREADS number of threads will be executing independently. The memory is partitioned into two large spaces, a shared space and a private space. Each of the threads has a sole access to a part of the private space. Additionally, the shared space is partitioned into portions each of which has affinity (or logical association) with a given thread. As most massively parallel computers have physically distributed memory, a UPC implementation typically maps a thread along with its private space and the shared memory portion that has affinity to that thread onto one node. The idea is that programmers can place shared data close to threads that will need them the most, which can reduce remote memory accesses and increase performance.

UPC provides a hybrid, user-controlled consistency model for the interaction of memory accesses in shared memory space. Each memory reference in the program may be annotated (using a variety of approaches) to be either "strict" or "relaxed" [3]. Under the strict behavior, the programmer is guaranteed that the program will be executed in sequential order. This trait means that the program will execute in the exact order in which the program was coded. The compiler is required to analyze memory accesses across all nodes and place synchronization within the compiled code to ensure the progression of each thread through the program follows the strict ordering in the original code. Under this model, performance may be hindered for the sake of memory coherence. With the relaxed model, the compiler only analyzes the memory access patterns of each thread individually. Therefore, the compiler can reorder instructions within threads to increase performance as long as the thread memory pattern integrity is maintained. Under the relaxed model, a programmer is not guaranteed memory coherence unless he specifically inserts synchronization into the code.

UPC is designed so that it can be implemented efficiently on a variety of multiprocessor architectures, including shared memory systems as well as distributed shared memory systems. The details of implementation may vary considerably from system to system. The HP (formerly Compaq) UPC compiler is the first commercially available UPC compiler in the world [5]. In our study we use version 2.1 of the HP UPC compiler. The HP compiler has been under development since the inception of UPC, and has run at least in testing mode since December 2000 [5]. Part of our study includes the implementation of the UPC specification by the HP UPC compiler.

2.2 Testbed: ES80 AlphaServer "Marvel"

The ES80 (or Marvel) system is built around the Compaq Alpha EV7 processor, which acts as a system on a chip. The EV7 incorporates an EV68 core, two levels of cache, memory controllers, an Input/Output (I/O) port, and four inter-processor ports all within a single chip [3]. By having the memory controllers and I/O and inter-processor ports on chip, the EV7 is easily scalable while allowing individual processors quick and easy access to resources.

There are four inter-processor ports on each processor labeled North, South, East and West. All communication in or out of the chip, besides memory, is done through a single router. Memory integrity is maintained by directory-based cache coherency [4]. This approach allows for fewer connections and easier maintenance of chip resources. The basic building block of

© 2003, HCS Research Lab, U. of Florida. All Rights Reserved.

these servers is two EV7s connected by their North and South ports.

The system used in this test is comprised of four EV7 processors connected from their North ports to South ports to form a ring. Each processor has 2GB RAMBUS 800 RAM directly connected to its memory controllers. The system has a total of 8GB of RAMBUS 800 RAM available to each processor through the CC-NUMA architecture. There are a total of 14 34.6GB Ultra Wide SCSI III hard drives available to the system. The connection between processors is proprietary with a reported bandwidth of 6.4 GB/s (3.2 GB/s one way) and latencies as low as 15ns.

For this study we use the ES80 AlphaServer as a CC-NUMA system. The operating system is Tru64 5.1B-6 (Rev. 2635). The UPC compiler is V2.1-003 from HP/Compaq. Unless stated otherwise, all code was compiled with the –O3 optimization and UPC used the –smp flag.

3. Differential Attack Simulator

Differential cryptanalysis is a type of code breaking algorithm that is used to attack DES-like ciphers. Differential cryptanalysis involves the analysis of the effect of the plaintext pair difference on the resulting ciphertext difference. By analyzing the fixed XORed value of the plaintext pairs, the partial subkeys used in DES-like encryption algorithm can be guessed. In this project, a sequential differential cryptanalysis algorithm known as the differential attack simulator is analyzed and experimented with for parallelization. The differential attack simulator is the preparation step in the differential analysis of a DES-like algorithm [1]. It creates basic components used in differential cryptanalysis such as S-Boxes, Difference Pair Table, and Differential Distribution Table. These tables can be used to analyze and extract information about the key and subkeys of the cipher. Since there is a massive amount of data involved in the creation of these components, the differential simulator is parallelized using the UPC programming language and will be executed in a distributed shared memory environment.

3.1 S-Boxes

An S-Box is an array containing arbitrary numbers within a certain range. The S-Box serves as a non-linear relationship between the input and the output, as each input combination has a corresponding (but arbitrary) output combination. The number of input bits (*n*) determines the size of the box (array), and the number of output bits (*m*) determines the range of numbers in the array. Consider the example of an S-Box with 8 input bits and 4 output bits: the array will contain 256 (2^8) elements, which are arbitrary numbers from 0 to 15 which represent 4-bit outputs. An illustration of the operation of such an S-Box can be seen in Figure 1.

3.2 Difference Pair Table

S-Boxes are used to create the Difference Pair Table (DPT). Consider the S-Boxes S_o and S_1. If we denote the input of an S-Box as X and the output as Y, then a difference pair of the S-Box is the pair (ΔX, ΔY) where ΔX = X'^X" and ΔY = Y'^Y" (X' and X" are two specific inputs, Y' and Y" are two corresponding outputs of the S-Box).
It is known that if ΔX = X'^X" then X" = ΔX^X'. As a result, we can consider all values of X' with ΔX as a constraint to the value of X" [1]. The values of ΔY are derived from all possible pairs (X', ΔX). These values of ΔY make up the DPT where the row and column are ΔX and X respectively.

© 2003, HCS Research Lab, U. of Florida. All Rights Reserved.

3.3 Differential Distribution Table

The Differential Distribution Table (DDT) is obtained from analyzing the DPT. Each element of the DDT represents the number of occurrences of a specific value of ΔY (which is used as the row of the DDT) given a value of ΔX (which is used as the column of the DDT). Once the DDT is calculated, it is possible to obtain all possible inputs and outputs given their differences. Moreover, it is possible to obtain the key involved in the creation of an S-Box if we know the input pairs and output differences of the S-Box.

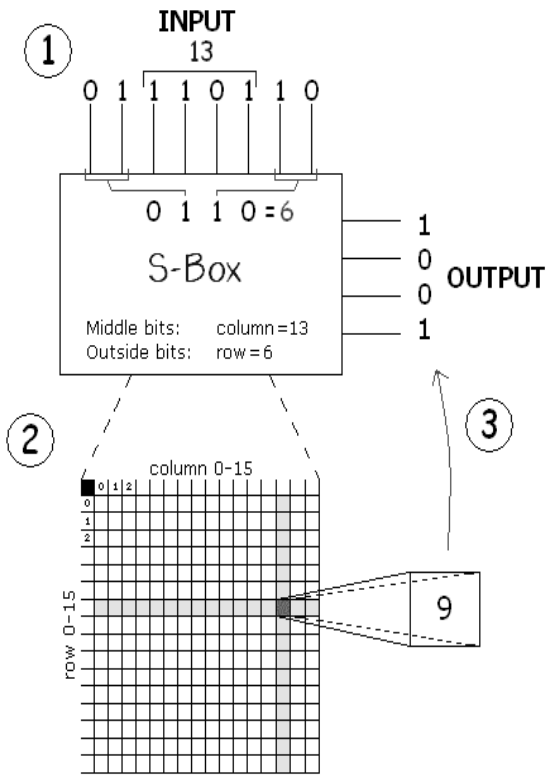

Figure 1: Illustration of Operation of an S-Box.

3.4 Design and Experiment Preparation

The parallel differential simulator used in this project is developed by parallelizing an existing sequential differential simulator algorithm. There are two main parts of the simulator:

a) Create S-Boxes: this part is not parallelized in this experiment since strong dependencies exist across loop iterations and, therefore, it is very difficult to parallelize. Furthermore, this part only makes up a very small fraction of the total execution time and, therefore, with parallelization would not result in a significant improvement in performance.

b) Calculate DPT and DDT tables: this part is parallelized in this experiment since this part consumes most of the execution time. This part lends itself naturally to parallelization since there is no dependency existing across loop iterations.

The parallel version of the differential simulator is written in UPC and the underlying architecture used in this experiment is the Marvel architecture. This experiment is repeated using 1, 2, and 4 processors in Marvel and performance is compare against the sequential version. In this experiment, the effect of block size distribution on the total execution time is also studied and analyzed.

4. Results and Analyses

The following are the results obtained from executing and measuring the parallel differential attack algorithm. The first graph (Figure 2) compares the speedup of the parallel version to the sequential one. This experiment was conducted with 1, 2, and 4 processors using UPC. In each of these three cases, three different methods of accessing shared data are examined: shared pointer, share block pointer, and local pointer. The second graph (Figure 3) contains the results of executing the parallel differential attack algorithm on either 2 or 4 processors with various data block sizes.

© 2003, HCS Research Lab, U. of Florida. All Rights Reserved.

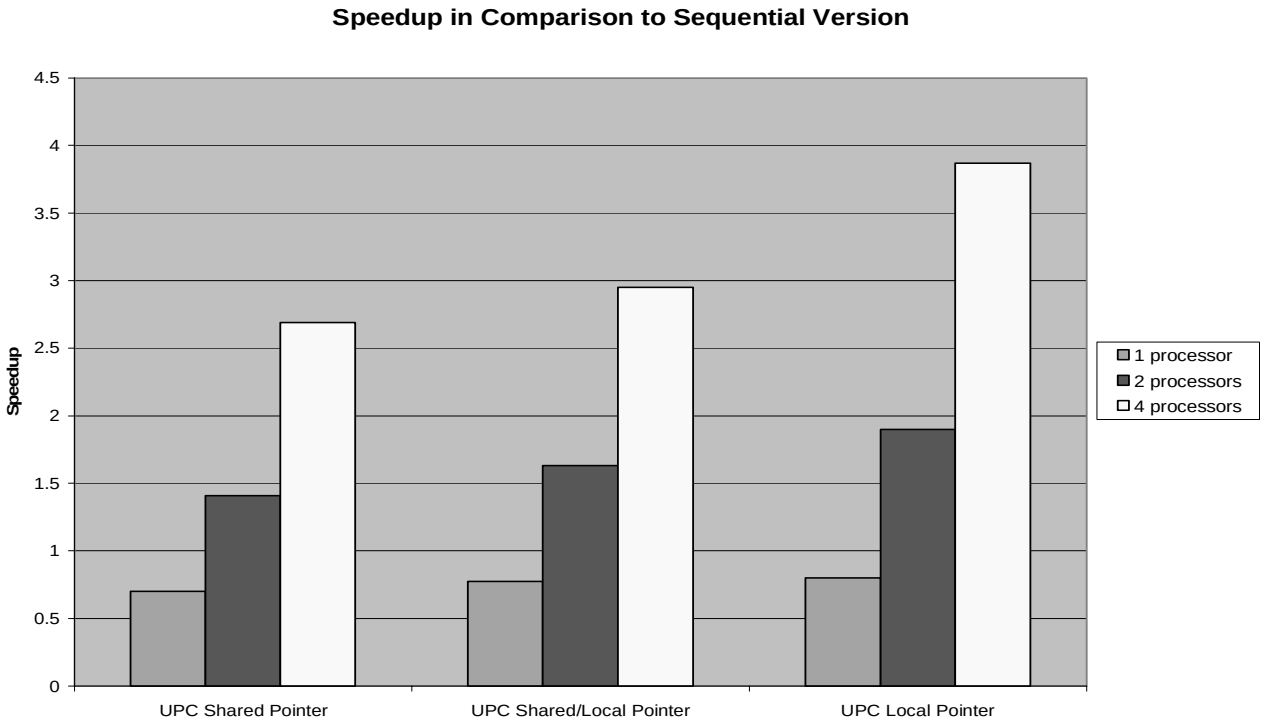

Figure 2: Speedup of Differential Simulator in UPC

From Figure 2, it is observed that the speedup of the parallel algorithm increases almost linearly with the number of processors for each of the three methods of accessing shared data. It is also observed that among these three methods, accessing all shared data using local pointers is the most efficient approach. In UPC, using share pointers generally introduces overhead since address translation is needed. This overhead can be tremendous depending upon the organization of the algorithm and the nature of the shared data. Since the first method introduces overhead each time shared data is accessed, the improvement in performance due to parallelization is offset by the overhead involved. By contrast, if all accesses to shared data are through local pointers, as in the last group of Figure 2, performance is greatly improved. UPC provides an efficient method to cast a shared pointer to a local pointer, and this technique has been used exclusively in this experiment to achieve high speedup. Experiments are also conducted with part of the shared data using local pointers. The performance is higher than using strictly shared pointers but lower than using strictly local pointers. An important fact in this experiment is that, despite the overhead involved in using shared pointers, the parallel algorithm can still obtain reasonable speedup because there is a large amount of data that can be computed in parallel and the data is very evenly distributed across the processors. In cases where there is not much opportunity for parallelization or where data is not evenly distributed, it is quite possible to obtain slowdown (i.e. speedup of less than one) in performance.

© 2003, HCS Research Lab, U. of Florida. All Rights Reserved.

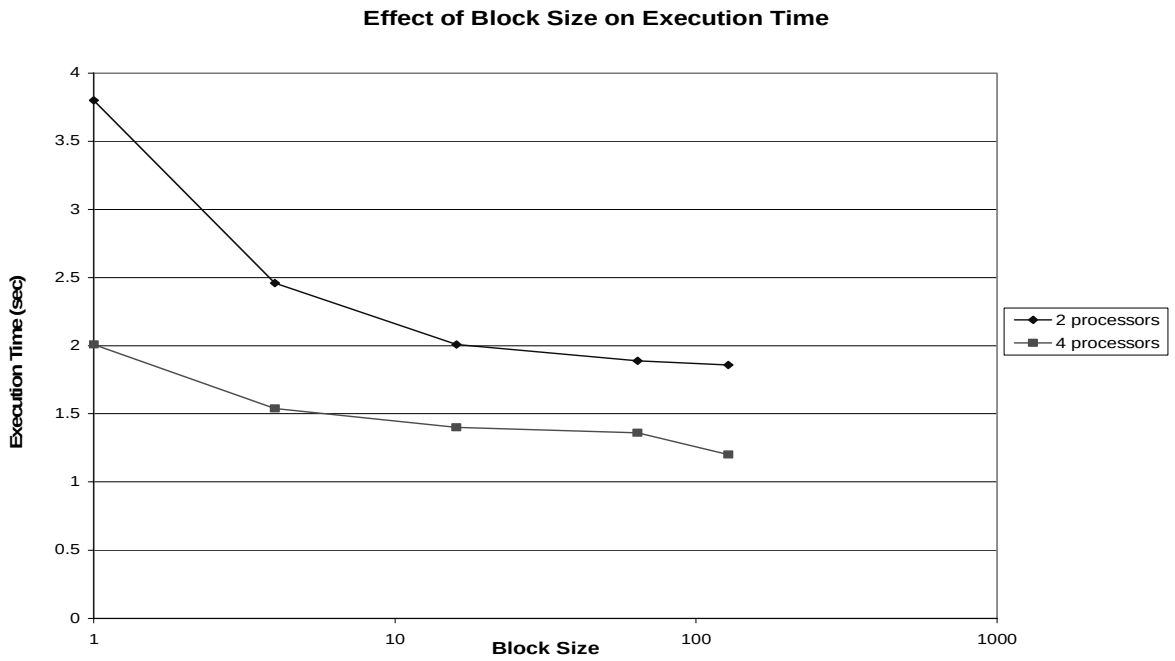

Figure 3: Block Size Results for Differential Simulator

Figure 3 gives the results from executing the parallel algorithm on 2 and 4 processors in Marvel using different data block sizes while using local pointers. In UPC, the block size determines the number of consecutive elements of a shared array to be distributed the processors in a round-robin fashion. The general trend observed in this graph is that the total execution time decreases with an increase in the block size, resulting in improved performance. This behavior can be explained by the fact that a larger block size reduces the number of times needed to cast shared pointers to local pointers. This situation exists because only the pointer of the first element of the shared block needs to be converted to a local pointer. Accessing subsequent elements of the block can be achieved by using the previously obtained local pointer. Since changing the block size does not have any effect when executing on a single processor, it is not included in this experiment.

5. Conclusions

UPC is gaining momentum in the parallel computing world. As an alternative to message passing, UPC is easy to work with and can in some cases provide nearly linear performance gains over sequential programs. Cryptanalysis is one area in which parallel programming can greatly improve performance.

Data distribution and access are two key considerations in the creation of UPC programs. By using large block sizes, performance of an UPC application can be improved. However, increasing the block size alone does not necessarily ensure better performance. Larger block sizes, coupled with local pointers to shared memory, can have the most noticeable performance increase. Casting shared data pointers to local pointers allows faster access to data with affinity to the local node than shared pointers.

© 2003, HCS Research Lab, U. of Florida. All Rights Reserved.

Cryptanalysis is a wide field with many different approaches. Future studies could include other aspects of cryptanalysis. Additionally, several other UPC compilers are now available. A study on performance differences between compilers and various platforms and parameter settings using cryptanalysis code for benchmarking is another avenue of future research.

6. Acknowledgements

The work in this study was made possible through support of the High-Performance Computing and Simulation Laboratory at the University of Florida. Additionally, the members of the HP UPC Compiler team were extremely helpful in troubleshooting problems and giving hints on performance improvements. Finally, special thanks go to Chris Conger and Matt Murphy n the HCS Lab for all their help with cryptology.

7. References

[1] K. Ooi, B.C. Vito, “Cryptanalysis of S-DES,” *International Association for Cryptologoic Research (IACR),* Vol. 2002, pp.44-46, 2002.

[2] “Compaq UPC Unified Parallel C (UPC) Programmer's Guide,” http://www.tru64unix.compaq.com/upc/upcus.htm, May 2002.

[3] W. Carlson, J. Draper, D. Culler, K. Yelik, E. Brooks, K. Warren, “Introduction to UPC and Language Specification”, http://www.gwu.edu/~upc/pubs.html, May 1999.

[4] http://upc.nersc.gov, 2003.

[5] http://www.tru64unix.compaq.com/upc/, 2003.

[6] E. Nahum, S. O’Malley, H. Orman, R. Schroeppel, “Towards High Performance Cryptographic Software,” *Proc. Third IEEE Workshop on the Architecture and Implementation of High Performance Communications Subsystems (HPCS)*, Mystic, Conn, August 1995.