ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems

Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Håkan Grahn, *Member, IEEE*
School of Computing, Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
mzeeshan01@gmail.com, yuchen9760@gmail.com, hakan.grahn@bth.se

Abstract—Multicore processors are the main computing platform in laptops, desktop, and servers today, and are making their way into the embedded systems market also. Using benchmarks is a common approach to evaluate the performance of a system. However, benchmarks for embedded systems have so far been either targeted for a uni-processor environment, e.g., MiBench, or have been commercial, e.g., MultiBench by EEMBC. In this paper, we propose and implement an open source benchmark, ParMiBench, targeted for multiprocessor-based embedded systems. ParMiBench consists of parallel implementations of seven compute intensive algorithms from the uni-processor benchmark suite MiBench. The applications are selected from four domains: Automation and Industry Control, Network, Office, and Security.

Index Terms—Concurrent Programming, Multiprocessor Systems, Performance Evaluation.

1 Introduction

MULTICORE processors are the main technology in laptops, desktops, and server systems, e.g., AMD Opteron, Intel Core 2 Duo and Quad, and IBM POWER4. Recently, multicore processors are paving their way also into embedded systems for many reasons, e.g., better performance and lower power consumption as compared to uni-core processors.

A number of hardware vendors develop and market system-on-chip (SoC) devices [5], and multicore processors is predicted to be a key technology in future high-performance embedded systems [3]. The small increase in power consumption will likely be justified by the large increase of computational power available to the embedded system applications by including multicore processors.

Benchmarks are used for assessing the (relative) performance among various software and hardware platforms [7], [9]. A benchmark is a set of applications whose execution results are the evidence of the performance of an execution platform. A number of benchmarks has been proposed including Dhystone, LINPACK, SPEC, Whetstone, and MediaBench [9], which focus on specific areas of computation.

The embedded systems domain comprises a wide range of different types of applications. Is has been noted that just one type of application is not enough to characterize the embedded domain, and therefore, a benchmark suite for this domain shall capure this application diversity [4], [6]. Multibench and CoreMark [4] target benchmarking of embedded systems. Multibench evalutes embedded multicore systems, but the drawback is that it is commercial. CoreMark targets several major tasks for embedded applications, it only tests the performance of a single core. MiBench [6], which is distributed as open source, has also been proposed for benchmarking of embedded systems and comprises 35 sequential applications. For general parallel systems, there exists a number of benchmark suites, e.g., SPLASH-2 [10] and PARSEC [2]. However, to the best of our knowledge no open source benchmark suite exists that specifically targets parallel embedded systems.

We propose ParMiBench, an open source parallel benchmark suite targeted for embedded systems. Seven applications from MiBench [6] are included in the proposed benchmark suite and they are selected from four application domains: automotive/industrial control, office, network, and security. The parallel implementations have been done using Pthread and standard C. The performance of ParMiBench is evaluated in terms of speedup of the parallel implementations against the sequential ones. A complete description of ParMiBench along with performance characterizations of the applications are presented in [8].

2 MiBench

MiBench [6], on which ParMiBench is based, has been proposed for benchmarking of uniprocessor-based embedded systems. It tries to capture the application diversity of the embedded system area and consists of 35 embedded applications from six application domains:

Manuscript submitted: 01-May-2010. Manuscript accepted: 09-Jun-2010. Final manuscript received: 16-Jun-2010.

- Automotive and Industrial Control, which demonstrates applications in embedded control systems, deals with performance of basic math, bit manipulation, data input/output, etc.
- Consumer Devices, which covers many consumer devices, e.g., scanners, digital cameras, and PDAs, and focuses on multimedia applications with, e.g., image manipulation.
- Office Automation, which includes text manipulation algorithms to represent office machinery like printers, fax machines, and word processors.
- Networking, which wrap up switches and routers, and performs shortest path calculations, tree and table lookups, and data input/output.
- Security, which includes various algorithms related to data encryption, decryption, and hashing.
- Telecommunications, which accentuate on voice encoding/decoding, check sum calculation, and frequency analysis.

3 PARMIBENCH

ParMiBench is a benchmark that specifically evaluates the performance of multiprocessor-based embedded systems. It mainly assesses and evaluates the performance features in terms of speedup. Its structure refers to EEMBC and MiBench that organize the selected applications according to some categories or application domains. A categorized benchmark enables the users to examine their design more effectively for a particular market segment of embedded devices [6]. ParMiBench provides four categories: Automation and Industry Control, Network, Office, and Security.

MiBench is a widely used benchmark for uniprocessor-based embedded systems, and we have selected seven applications from MiBench for parallelization and inclusion in ParMiBench. The difference between the applications in ParMiBench as compared to those in MiBench is that they run on multiprocessor-based embedded systems, i.e., the ParMiBench applications are parallel versions of the same applications found in MiBench. The parallel implementations of all applications can be run on Unix/Linux platforms supporting Pthreads and C. All ParMiBench applications are compiled with the GNU Compiler Collection (GCC).

The major design decisions in ParMiBench are as follows. The whole input data are read into memory, workers access it and write the result into unique files or buffers in order to reduce I/O waiting time. The work is equally distributed among workers by using static load balancing. Improper load balancing increases the discrepancy of execution time between subtasks. In most cases, coarse-grained task decomposition has been used to reduce synchronization and communication overhead. An input data partition strategy is used where data dependency is a bottleneck and it is difficult to partition the program logic.

ParMiBench exercises the target platform mainly from the scalability perspective. It exercises the CPU and memory performance of a system, while keeping the synchronization and communication overhead low. Further, it does not particularly stress I/O operation. It helps to answer the question how a given platform scales for a certain set of data. In [8], performance metrics such as speedup, overhead, and efficiency are discussed in more detail.

3.1 Bitcount

Bitcount measures the processor's bit manipulation ability by counting the number of bits using different counting strategies. It has nine sub-algorithms whose outputs are the number of bits in the input data which value is 1. Bitcnts is an entry program that can invoke each sub-algorithm to count the number of bits. The other eight sub-algorithms implement different bit counting strategies.

The parallel Bitcnts, Bitcnt_1, and Bitstring algorithms are implemented using a combination of recursive and data decomposition techniques. The division into tasks as well as the partitioning of the input data is done to obtain a good load balance on each processor. The other sub-algorithms are invoked as units by the parallel Bitcnts, in which the data decomposition strategy is preferably used.

3.2 Susan

Susan is an image recognition application, which recognizes corners and edges in Magnetic Resonance Images of the brain. It is used for vision-based quality assurance, and performs adjustments for threshold, brightness, spatial control, and image smoothness. The small input data is a black-white image of a rectangle while the large input is a complex picture.

Susan is composed of different functions, and these functions are all parallelized by using data decomposition. The decomposition is applied on the outermost for loop, which iterations are decomposed according to the number of workers.

3.3 Basicmath

Basicmath is used for benchmarking mathematical calculations in embedded processors, which sometimes do not have dedicated hardware support. It performs simple mathematical calculations, e.g., cubic function solving, angle conversions from degrees to radians, and integer square root. The input data set used for benchmarking is a fixed set of constants.

The parallelization of Basicmath is done by data partitioning, i.e., the number domain for different functions is partitioned into different sets. The tasks are implemented through a master-worker strategy, which comprises different stages: data limits definition, subtasks creation and allocation, worker initialization, workers performing their work, and finally,

the master collects the data returned by the different workers.

3.4 Patricia

A Patricia trie is a sparse leaf-nodes based data structure used instead of a full tree. Branches with only a single leaf node in a Patricia trie are collapsed upwards in the trie to reduce the traversal time as compared to full trees, but at the expense of code complexity. In many network applications are routing tables represented by Patricia tries. The input data set is an IP traffic list from active web servers.

The addressing function has been parallelized by partitioning the IP addresses list into sublists using a master-worker strategy. Each sublist is assigned to the available number of processors.

3.5 Dijkstra

Dijkstra calculates the single-source and all-pairs shortest paths in a graph represented by an adjacency matrix.

Dijkstra single-source shortest path has been parallelized using two strategies: single and multiple queue implementations. In the first one, all processors share a single queue, whereas in the second one, each processor maintains a local queue. In the all-pairs shortest path problem, parallel Dijkstra uses a data decomposition strategy in such a way that one processor handles one vertex to get its single-source shortest paths.

3.6 Stringsearch

The Stringsearch benchmark finds a specific word in a number of given phrases by employing case sensitive or insensitive comparison algorithms.

The partitioning strategy is to partition the entire pattern collection into a number of sub-pattern collections according to the number of workers allocated. The size of each sub-pattern collection contains $\lceil n/p \rceil$ successive patterns. The static master-worker model is composed of three phases. In first phase, a file containing the search string and a file containing the patterns are read. For load balancing, the number of patterns to search for is equally divided to the available workers, i.e., the number of tasks generated is equal to the number of workers. In the second phase, workers are created as work is allocated. The workers search in parallel for the patterns, where each worker uses the sequential implementation of the string search algorithms. Each worker stores the results in memory. Finally, in third phase, the master displays the found strings.

TABLE 1
Input data set sizes.

Algorithm	Summary
Bitcount	An input of long type (31 bits with 1). Bitcnts: 112500 iterations. Bitcnt_1: 200000 iterations. Bitstring: 9200 iterations.
Susan	PGM picture: 2.8 MB
Basicmath	Small data set: 500 Mega numbers Large data set: 1 Giga numbers
Patricia	Text file containing 5000 IP addresses
Dijkstra (shortest path)	All-pairs: 160x160 matrix Single-source: 2000x2000 matrix
Stringsearch	Input data set size: 16 MB, 32 MB 1024 patterns or keys of length (m): 5
SHA	16 text files of sizes 300 to 911 kB

3.7 SHA-1

The Secure Hash Algorithm (SHA-1) is an iterative one-way hash function cryptographic algorithm, which can process a message to produce a message digest. It generates digital signatures used in the secure exchange of cryptographic keys.

SHA-1 is parallelized using a static master-worker strategy composed of four different stages. The first stage starts by deciding the input partition size (the input is a number of files for which a digest is calculated) and task generation. In the second stage, all files' data whose digest that need to calculated are read into memory by the master. The worker creation is done in the third stage. Each worker calculates the digest and writes it to the memory location assigned to them. The master waits for the workers to finish their work. Finally, the master writes the output into separate digest files.

4 Performance Evaluation

4.1 Evaluation Methodology

All performance measurements are done on a server with two 2.0 GHz Intel Xeon E5335 quad-core processors (i.e., 8 cores in total), 16 GB of main memory, 15,000 rpm SAS disks, and running Ubuntu 8.04. A summary of the input data set sizes used for testing of the parallel applications is given in Table 1.

4.2 Performance Results

The performance results in terms of speedup for all applications are shown in Fig. 1.

Linear speedup is achieved in parallel Basicmath and it scales well for large data set sizes. The best speedup of parallel Bitcount is obtained by using 6 processors but with 8 processors, a decreasing trend is observed. The parallel Dijkstra implementation of the All-Pairs shortest path solution gives linear speedup. However, the two implementations for the single-source shortest path solution, i.e., single and multiple queues, give small increments in speedup. However,

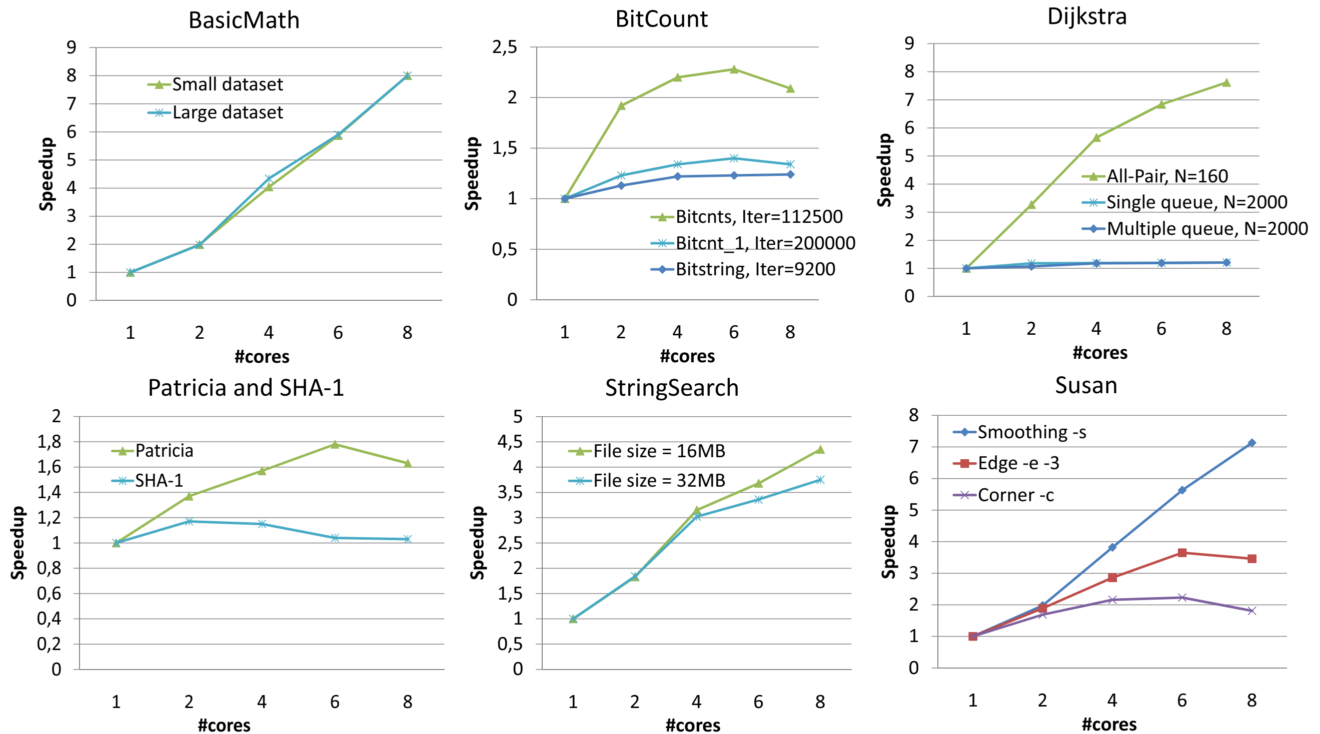

Fig. 1. Speedup measurements for the ParMiBench applications.

we observe that the speedup increases as the number of processors increases.

In parallel Patricia, the maximum speedup is observed for 6 processors. In case of SHA-1, a good speedup for a small number of processors is observed, but when the number of processors increases it goes down. We observe in parallel SHA-1 that the number of files are equally distributed among the workers for load balancing purpose, but different file sizes contribute to load unbalance. That effect is observed as a reduced speedup.

Parallel Stringsearch scales well for large text files; the speedup increases as the number of processors increases. In parallel Stringsearch, we found from the experiment that our approach scales well though we increase our text file size two times. For a majority of the functions in Susan a limited speedup is achieved.

5 Concluding Remarks

In this paper, we have proposed a parallel benchmark for multiprocessor-based embedded systems, and presented the performance results on an eight-processor machine. We have obtained a good speedup for many of the applications as the number of processors increases. We also found that the parallel approaches scale well when we increase the problem size. Therefore, we hope that ParMiBench would help potential users to evaluate the performance of multiprocessor-based embedded systems. The source code for ParMiPBench can be accessed through http://code.google.com/p/multiprocessor-benchmark/.

Acknowledgments

We would like to thank the anonymous reviewers for many suggestions and helpful comments, which have greatly improved the paper presentation.

References

[1] K. Asanovic et al., “The Landscape of Parallel Computing Research: A View from Berkeley”, Tech. report UCB/EECS-2006-183, Dept. Electrical Eng. and Computer Science, Univ. of Calif., Berkeley, 2006.

[2] C. Bienia, S. Kumar, J.P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” *Proc. of the 17th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 72-81, 2008.

[3] K. De Bosschere et al., “High-Performance Embedded Architecture and Compilation Roadmap,” *Transactions on High-Performance Embedded Architectures and Compilers I*, Lecture Notes in Computer Science, Vol. 4050, pp. 5-29, 2007.

[4] EDN Embedded Microprocessor Benchmark Consortium, http://www.eembc.org.

[5] L. Eggermont, Ed., “Embedded Systems Roadmap”, STW Technology Foundation, http://www.stw.nl/Programmas/Progress/ESroadmap.htm. 2002.

[6] M.R. Guthaus et al., “MiBench: A free, commercially representative embedded benchmark suite,” *Proc. of the IEEE Int'l Workshop on Workload Charac-terization (WWC-4)*, Dec. 2001.

[7] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 1996.

[8] Y. Liang and S.M.Z. Iqbal, “OpenMPBench - An Open-Source Benchmark for Multiprocessor Based Embedded Systems,” Master thesis report MCS-2010:02, School of Computing, Blekinge Institute of Technology, Sweden, Jan. 2010.

[9] W.J. Price, “A Benchmark Tutorial,” *IEEE Micro*, 9(5):28-43, Sep. 1989.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: characterization and methodological considerations,” *Proc. of the 22nd Int'l Symp. on Computer Architecture*, pp. 24-36, 1995.