Performance Characteristics of OpenMP Language Constructs on a Many-core-on-a-chip Architecture

Weirong Zhu, Juan del Cuvillo, Guang R. Gao

Department of Electrical and Computer Engineering
University of Delaware, Newark, Delaware 19716, U.S.A
{weirong,jcuvillo,ggao}@capsl.udel.edu

Abstract. Recent emerging many-core-on-a-chip architectures present massive on-chip parallelism through hardware support for multithreading. In order to achieve fast development of parallel applications that exploit this massive intra-chip parallelism to achieve highly sustainable performance, suitable programming models are needed. OpenMP, the industry de facto standard for writing parallel programs on shared memory systems, could become a reasonable candidate. To increase our understanding of the behavior and performance characteristics of OpenMP programs on many-core-on-a-chip architectures, this paper presents a performance study of basic OpenMP language constructs on the IBM Cyclops-64 architecture, which consists of 160 hardware thread units in a single chip. Compared with previous work on conventional SMP systems [1], the overhead of OpenMP language constructs on C64 many-core architecture is at least one order of magnitude lower.

1 Introduction

Although advances in IC processing technology have led to hundreds of millions (now reaching 1 billion) of transistors to be fabricated on a single silicon die, the delivered performance versus number of transistors integrated in a chip for conventional single-thread wide-issue superscalar architectures keep declining over time. In order to utilize the transistor budget and mitigate the effects of high interconnect delay, multi-core or many-core-on-a-chip architectures are emerging. Instead of devoting the entire die to a single and complex processor, this new generation of architectural technology proposes to integrate a large number of tightly-coupled simple processor cores on a single chip. The many-core-on-a-chip architecture naturally exploits the thread-level and process-level parallelism, which are expected to be widespread in future applications and multiprocessor-aware operating system and environments [2].

Cyclops-64 (C64) [3, 4] is a petaflop supercomputer project under development at IBM T.J. Watson Research Laboratory. The C64 chip architecture employs the many-core-on-a-chip approach by integrating 160 processing cores on a single chip. To the best of our knowledge, the C64 project is one of the most ambitious projects currently under development. Unlike other academia projects, a Cyclops-64 system is planned to be delivered in 2007.

Given the intra-chip parallelism presented by a many-core-on-a-chip architecture, such as C64, it is important and challenging to provide high level parallel programming

models for application developers to efficiently map the inherent parallelism in applications to a large number of on-chip processing cores. As a de facto industry standard for writing parallel programs on shared memory systems, OpenMP [5] is considered as one of the possible candidates. Parallel application developers express parallelism, work sharing, and synchronization through the OpenMP language constructs. For the purpose of understanding the behavior and performance characteristics of OpenMP-based parallel programs on many-core architectures, it is important to evaluate the performance of OpenMP basic language constructs, whose overhead accounts for up to 12% of the total execution time in some instances [1].

To conduct a prototype study on high level parallel programming models, we ported the Omni-1.6 OpenMP compiler [6] to C64, and optimized the Omni OpenMP runtime system to adapt to the C64 hardware features [7]. In this paper, based on the number reported by the EPCC microbenchmarks [8], we measure and evaluate the performance characteristics of major OpenMP language constructs on a C64 many-core-on-a-chip architecture with up to 160 cores. In addition, we compare our results to previous work on conventional SMP systems and find remarkable differences. In some instances, the overhead on C64 is one order of magnitude lower.

With our study we provide insight regarding the following aspects of software development on many-core architectures: (1) we provide application developers a better understanding of the behavior of OpenMP programs on a many-core architecture; (2) we give library and compiler developers hints regarding possible optimizations and/or language extensions specific to many-core architectures, specifically, to efficiently exploit multi-level memory hierarchies and fast intra-chip synchronization mechanisms; (3) using the OpenMP runtime library optimization as an example to understand the pros and cons of the C64 architecture, we provide software developers hints on how to write and optimize programs for this type of architecture. To the best of our knowledge, this paper is the first attempt that measures and evaluates the performance characteristics of OpenMP language constructs on many-core-on-a-chip architecture with up to 160 cores.

2 Cyclops-64 Architecture

The Cyclops-64 (C64) [3, 4] is designed to serve as a dedicated petaflop compute engine for running high performance applications. A C64 system is built out of tens of thousands of C64 chips connected through a 3D-mesh network. The C64 chip employs the many-core-on-a-chip technology by integrating 160 hardware thread units, half as many floating point units, the same number of embedded SRAM memory banks, and the communication hardware in the same piece of silicon (see Figure 1).

A thread unit, the C64 computation cell, is a simple 64-bit, single issue, in-order RISC processor operating at a moderate clock rate (500MHz). Efficient support for thread level execution, such as thread sleep/wakeup, is incorporated in the thread unit. Resource virtualization mechanisms are not provided by the hardware. For instance, thread execution is non-preemptive, and there is no virtual memory manager.

The three-level (SP, on-chip SRAM, off-chip DRAM) memory hierarchy of the C64 chip is visible to the programmer. C64 does not employ data cache. Instead, a portion

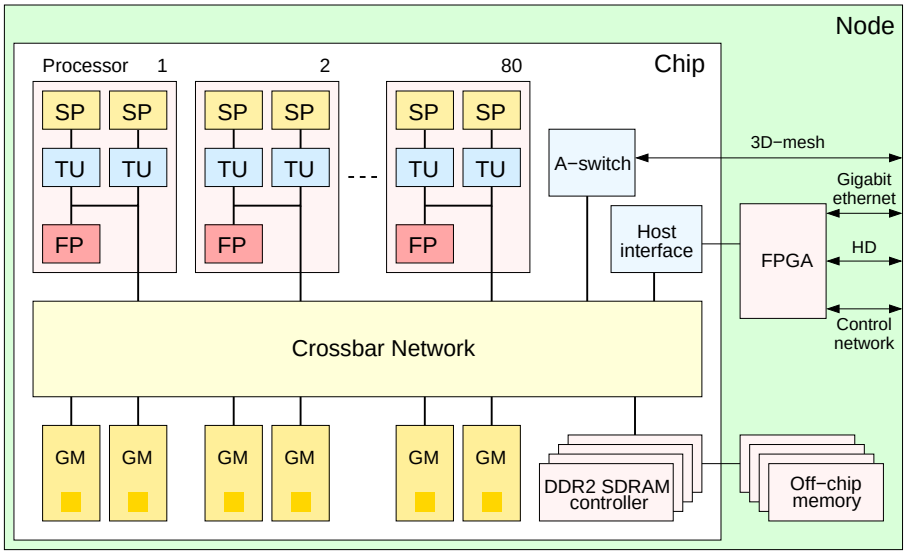


Fig. 1. Cyclops-64 node

of each SRAM bank can be configured as scratch-pad memory, which provides a fast temporary storage to exploit locality under software control. The integration of thread units and memory banks on a single chip is further leveraged with a rich set of hardware supported in-memory atomic instructions. Atomic instructions in C64 only block the memory bank where they operate upon, while the remaining banks proceed servicing other requests. This functionality facilitates the scalability of multithreading programs with intensive synchronization operations.

C64 also employs the Network-on-Chip (NoC) concept, all on-chip resources are connected to an on-chip crossbar network, which sustains a 4GB/s bandwidth per port per direction, 384 GB/s per direction in total. Besides the crossbar network, all the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers.

3 Experimental Infrastructure

As shown in Figure 2, the C64 system software toolchain [9] is the infrastructure for software and application prototype development on the incoming C64 system. The toolchain provides binary utilities (assembler, linker, etc.), GNU CC compilers (3.2.3 and 4.0.2), standard C and math libraries that are derived from those in newlib-1.10.0. A microkernel and the TiNy Threads^{TM}(TNT) runtime system are customized for the unique features of the C64 architecture [10]. The TNT library provides user and library developers an efficient Pthread-like API for thread level parallel programming purpose. The OpenMP compiler and runtime environment is ported from Omni-1.6 [6].

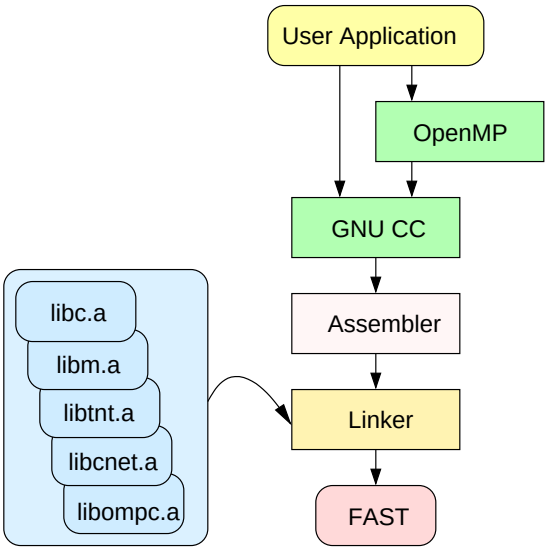


Fig. 2. Cyclops-64 software toolchain

We investigated and optimized the Omni OpenMP runtime library by exploring C64 hardware features, such as the explicitly visible and programmable memory hierarchy, the efficient in-memory atomic instructions, the thread level execution support, and the fast barrier synchronization through the on-chip signal bus [7].

All the experiments are conducted on a functionally accurate simulator (FAST) [11]. FAST is an execution-driven, binary-compatible simulator of a multi-chip C64 system. It accurately reproduces the functional behavior and count of hardware components of a C64 system. In addition, it generates timing information that accounts for the main sources of pipeline delays and stalls such as contention in memory, the crossbar, and/or other functional units. Although not cycle accurate, this information has proven to be useful for performance estimation, characterization and application tuning as well [11]. FAST has been extensively used by the C64 architecture design team at IBM for the purpose of chip design verification, and dozens of system software developer and application scientists for early application development.

4 EPCC Microbenchmarks

In order to understand the performance behavior of an OpenMP application, we use EPCC microbenchmarks [8] to measure the overheads of OpenMP language constructs. The basic methodology employed by EPCC is as follows. First, a reference time is obtained by executing a loop (or loop nests) sequentially without using any OpenMP directive. Then, the overhead is calculated by comparing this reference time with the execution time of the same code extended with OpenMP constructs.

There are three components of the EPCC microbenchmark. The *synchronization benchmark* measures the overhead of OpenMP work-sharing and mutual exclusion directives, such as PARALLEL, PARALLEL FOR, BARRIER, CRITICAL, ATOMIC, and REDUCTION etc.. The *scheduling benchmark* compares different scheduling policies – STATIC, DYNAMIC, and GUIDED. The *array benchmark* measures the overhead of the PARALLEL directives with the PRIVATE, FIRSTPRIVATE, and COPYIN clauses. We execute all three benchmarks on a single C64 chip with up to 128 threads and report the experiment results in the next section.

5 Experimental Results

5.1 Synchronization Microbenchmark

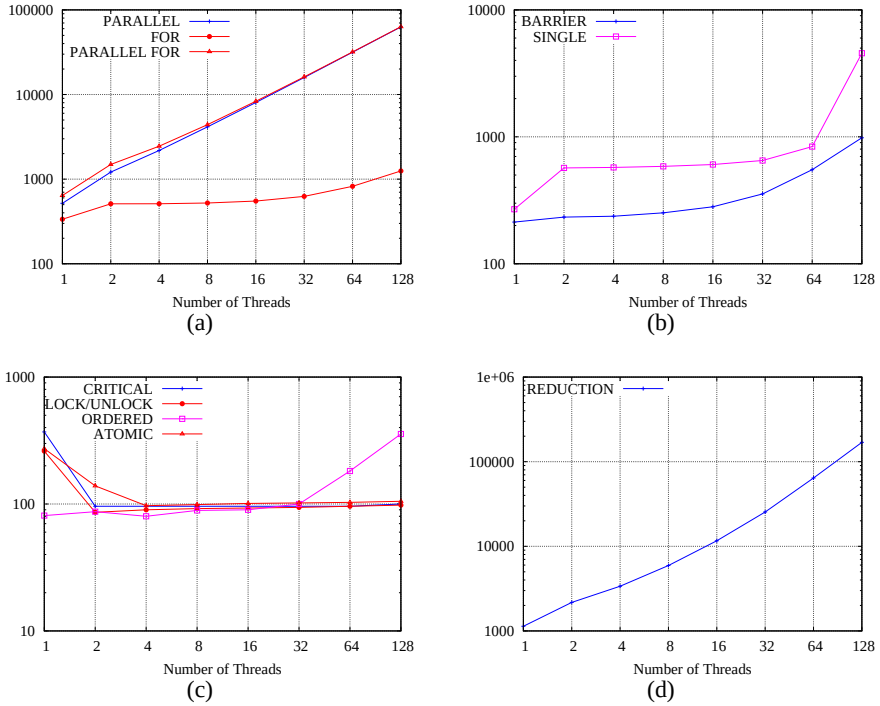


Fig. 3. Overhead (cycles) of Synchronization Directives (a) PARALLEL, FOR, PARALLEL FOR (b) BARRIER, SINGLE (c) Mutual Exclusion (d) Reduction

Figure 3(a) compares the overhead of the PARALLEL , the loop, and the combined parallel work-sharing PARALLEL FOR constructs. It shows that the PARALLEL FOR

construct has overhead similar to that of PARALLEL. This is because the overhead of the FOR construct is much smaller than PARALLEL and remains almost constant. From Figure 3(a) and (b), we can also see that the overhead of FOR is only slightly higher than the overhead of BARRIER, which implies that the cost of FOR is mainly due to the implicit BARRIER at the end of the loop.

Note the high overhead of the SINGLE directive, especially when the number of threads increases to 128. This is because the implementation of SINGLE is very expensive in order to guarantee the semantics of SINGLE. The memory contention incurred to complete the SINGLE operation rises dramatically when the number of threads increases. SINGLE also suggests an implicit barrier.

Because the OpenMP runtime library is carefully designed and tuned to map to the C64 hardware features, and the hardware components of C64 are tightly coupled in a single chip, the PARALLEL and BARRIER constructs incur much lower overhead than on conventional SMP systems. For example, a previous study [1] shows that the overhead of the PARALLEL construct reaches 120 microseconds (108,000 cycles) when running with 70 threads on a 72-node Sun Fire 15K system. Even while running with 128 threads, the same construct only presents a 63,020 cycles overhead. This observation implies that the thread management on a C64 like many-core architecture is much more efficient than common SMP environments.

We customized the well-known linked-list-based MCS spin-lock algorithm [12] to implement the low level lock acquisition and release primitives in the OpenMP runtime library [7]. Unlike common SMP systems where the overhead of lock increases with the number of threads, Figure 3(c) shows that the overhead of mutual exclusion constructs in OpenMP remain within the same range without increasing dramatically. Even for 128 threads, the CRITICAL directive costs only 154 cycles.

The of overhead of the REDUCTION construct increases exponentially, as shown in Figure 3(d). As future work, the reduction operation can be optimized in the runtime library by taking advantage of the C64's rich set of in-memory atomic instructions, which can perform certain operations, such as addition, subtraction, and various logical operations, atomically in memory. From our previous experiences with other benchmarks, such as Table Toy [11], we expect to improve the performance of REDUCTION dramatically.

5.2 Scheduling Microbenchmark

In OpenMP, there are three means for scheduling loop iterations among threads: STATIC, DYNAMIC, and GUIDED [5]. Please note that EPCC only reports the overhead of the GUIDED(n) scheduling policy for small values of n. Figure 4 compares different loop scheduling policies when running on 1 to 128 threads. It is apparent that STATIC and STATIC(128) always incur the lowest overhead in all cases. For the STATIC(n) policy, STATIC(1) causes the largest overhead, and the overhead decreases to the overhead of STATIC with increasing chunk size. Actually, the overhead of STATIC and STATIC(n) increases slowly for runs from 2 threads to 64 threads. When 128 threads are executed concurrently, the overhead is much larger than running with 64 threads because of the high memory contention.

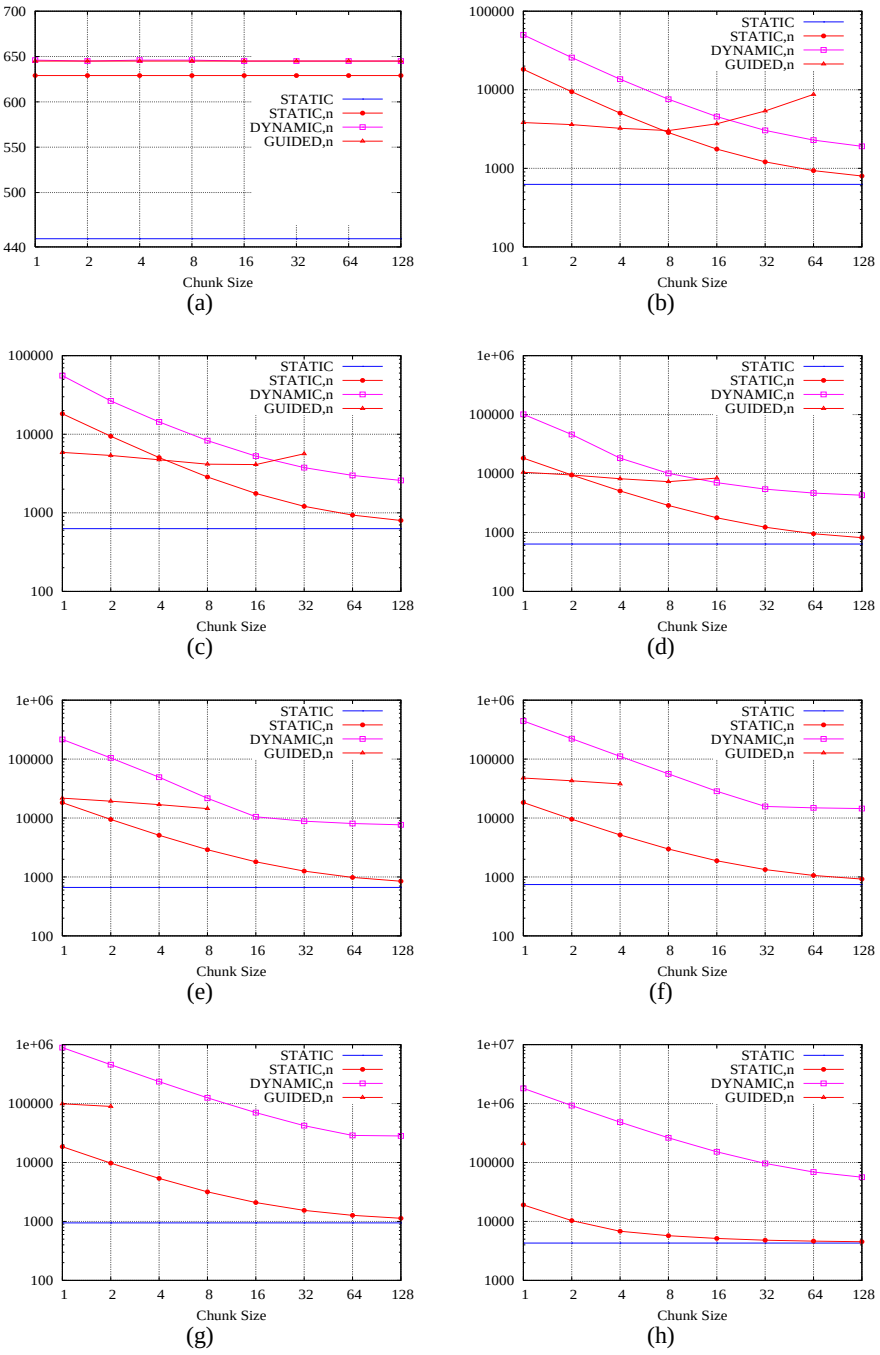


Fig. 4. Overhead (cycles) of Scheduling Policies with (a) 1 Thread (b) 2 Threads (c) 4 Threads (d) 8 Threads (e) 16 Threads (f) 32 Threads (g) 64 Threads (h) 128 Threads

DYNAMIC(1), which is the most fine-grained scheduling policy, generates huge overheads (3,621 microseconds) when running on 128 threads. This is because the small chunk size causes frequent dynamic scheduling function calls, whose execution time is counted as the overhead. As a result, the overhead of static scheduling is multiple orders of magnitude smaller than dynamic scheduling.

The overhead of the GUIDED(n) scheduling is always better than the DYNAMIC(n). The GUIDED(n) policy starts with a large chunk size, then gradually decreases it to n. Figure 4 also demonstrates that the STATIC policy always incurs lower overhead than the GUIDED policy. The overheads measurement suggests that on C64 OpenMP programmer should consider the STATIC scheduling policy as the first option for loop scheduling, given the tasks can be statically balanced. Only if the benefit of dynamic load balancing surpasses the scheduling overhead, the dynamic and guided scheduling policy are worth being chosen.

In the OpenMP runtime library, the dynamic and guided scheduling functions are implemented to frequently access the thread descriptor, and sometimes access the master thread's descriptor by acquiring a lock first. By taking advantage of the explicit programmable multi-level memory hierarchy of C64, we place the thread descriptor of each work thread into its own scratchpad memory, which guarantees very fast accesses, i.e., 1 cycle for a store, 2 cycles for a load. The master thread's descriptor is placed in on-chip global memory, whose access latency is longer than scratchpad but smaller than off-chip memory. By leveraging the C64's in-memory atomic instruction and thread level execution support, the lock/unlock primitives used to guarantee the mutual exclusion for accessing the master thread's descriptor are efficiently implemented as demonstrated in Figure 3(c) [7]. Therefore, compared with common SMP systems, the overhead of loop scheduling is at least an order of magnitude lower on a C64-like many-core-on-a-chip architecture. For example, as reported in [1], when running on a 72-node Sun Fire 15K, the DYNAMIC(1) incurs an overhead of around 27M cycles (30,000 microseconds) with 24 threads, while on C64 it costs 0.44M cycles with 32 threads, and 1.8M cycles with 128 threads. The overhead of STATIC scheduling is 9,000 cycles with 24 threads on a Sun Fire 15K [1], but only 743 cycles with 32 threads, and 4,298 cycles with 128 threads on C64.

5.3 Array Microbenchmark

The *array microbenchmark* measures the overhead of the PARALLEL directive with the PRIVATE, FIRSTPRIVATE, and COPYIN clauses. In the current design of C64 system software, the stack of a thread is placed in its own scratchpad memory and the size of the stack is limited. As a result, in our experiments, we can only run the benchmark with an array size smaller than or equal to 729. As a work in progress, the C64 toolchain will provide support for automatic stack extension, a feature that allows applications that require more stack than available to continue running at the expense of performance. When the stack area is exhausted, the runtime system automatically relocates the stack into off-chip memory. Notice the relocation is performed very quickly, as it requires setting a few registers and copying a few locations from the stack (but not all). If at a later point, the stack shrinks, the runtime system undoes the changes and sets the program stack back to scratchpad memory. However, in order to achieve good

performance, it is not recommended to declare large arrays on the stack (as automatic variables), or make deep recursive function invocations in the program.

As shown in Figure 5, the PRIVATE and FIRSTPRIVATE clauses have similar overheads (the overhead of FIRSTPRIVATE is slightly higher). Compared with the PARALLEL constructs without any data-sharing attribute and data copying clauses, it is also clear that the curves of PRIVATE and FIRSTPRIVATE almost match the curve of PARALLEL constructs. This means attaching the PRIVATE or FIRSTPRIVATE clause to the PARALLEL construct incurs negligible costs. In both cases, the compiler directly allocates the private array in the stack of each thread, which incurs no overhead at runtime.

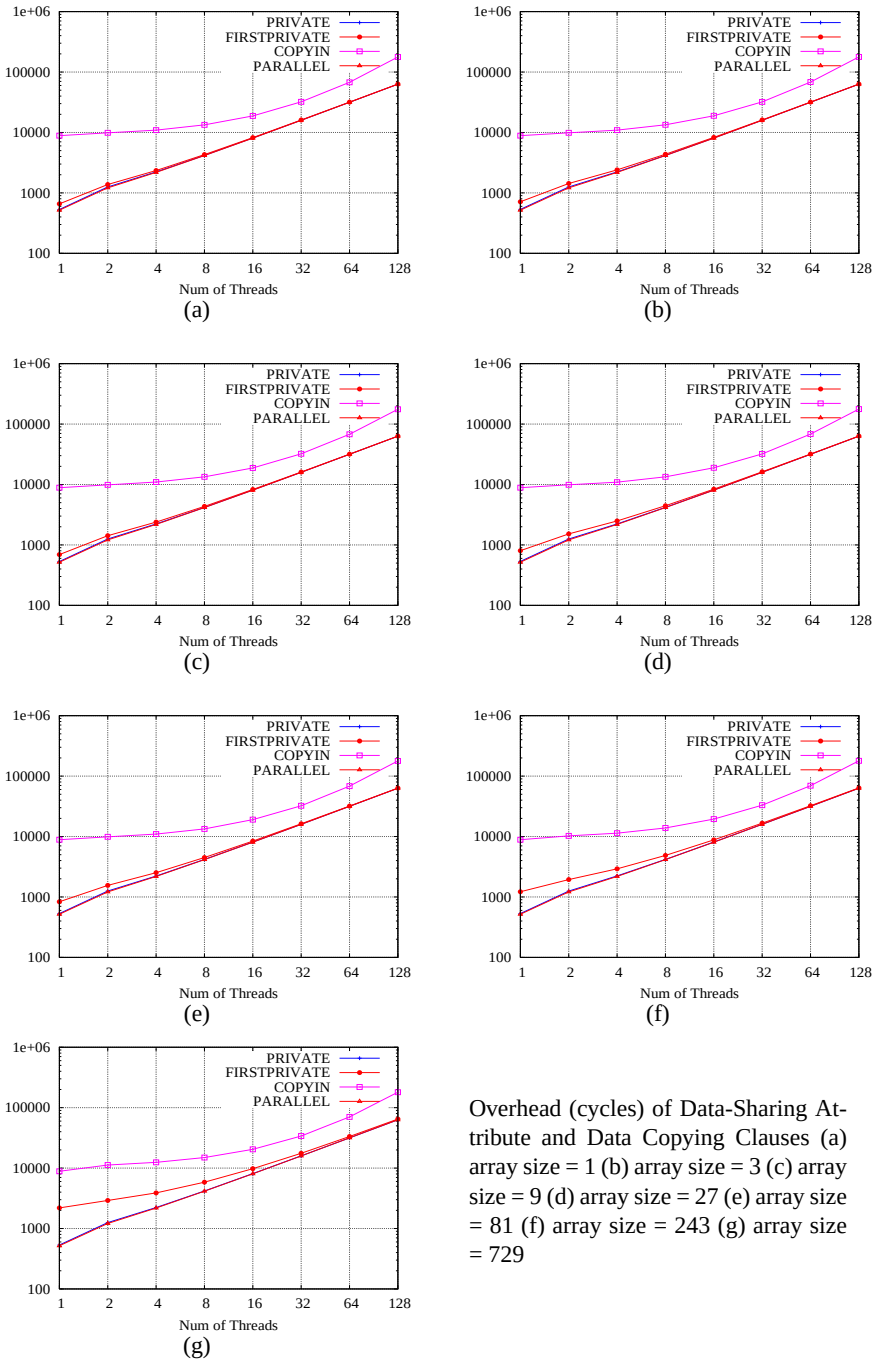
For FIRSTPRIVATE, the C library function *bcopy* is used to initialize the private array by copying the contents of a global array. In the standard C library of C64, routines like *memcpy*, and *bcopy*, are optimized and fine tuned. They are aware of the explicit memory hierarchy. The C64 load and store multiple instructions are used to exploit the memory bandwidth and save cycles from not issuing multiple instructions. In addition, the instruction sequences are manually scheduled to hide memory accessing latencies. Since the array size used in our experiments is small, the copying is performed very efficiently. Therefore, no significant overhead is observed for FIRSTPRIVATE.

From Figure 5, the COPYIN clause generates one order of magnitude larger overhead than the other two clauses. By attaching the COPYIN clause to the PARALLEL directive, the Omni OpenMP compiler generates codes that dynamically allocate the storage for thread private data. The heap manager allocates the thread private data in the on-chip global memory. There are also overheads from lock/unlock operations for using the memory allocator. Moreover, since the data is allocated in the global memory at runtime, the latency of memory accesses in the loop body is much higher than accessing scratchpad memory. This is the reason why COPYIN has much larger overhead than PRIVATE and FIRSTPRIVATE. This suggests a scope for possible optimizations either in the compiler or the runtime system.

6 Related Work

Previous work [7] demonstrated a set of optimizations on the Omni OpenMP runtime library by exploiting C64 hardware features. We introduced the optimization techniques and demonstrated the effectiveness by showing the performance improvement of OpenMP synchronization constructs compared to the unoptimized OpenMP runtime library. This paper presents the measurement and evaluation of all major OpenMP language constructs, including synchronization directives, scheduling policies, and array clauses, with the optimized runtime library on C64. We also compare our results to those previously reported on conventional SMPs. The purpose of this work is to provide the application programmers, compiler and library developers a better understanding of the behavior of OpenMP programs on a many-core architecture.

Most of the previous work on performance characterization of OpenMP were conducted on the general purpose commercial shared memory SMP systems [8, 6, 13–15, 1]. Liao et. al. [16] evaluated OpenMP on chip multithreading platforms. However, the chip multiprocessor (UltraSPARC III) evaluated in the paper only has two cores. To



Overhead (cycles) of Data-Sharing Attribute and Data Copying Clauses (a) array size = 1 (b) array size = 3 (c) array size = 9 (d) array size = 27 (e) array size = 81 (f) array size = 243 (g) array size = 729

Fig. 5.

the best of our knowledge, this paper is the first attempt to measure and evaluate the performance characteristics of OpenMP language constructs on a C64-like (160 cores) like many-core-on-a-chip architecture.

In [17, 18], the authors presented the experiment results of OpenMP NAS benchmarks on an experimental Cyclops architecture. It is worth noting that this experimental architecture was a preliminary design of the Cyclops architecture and it is never to be built, while the first C64 system is planned to be delivered in 2007. Also, this experimental Cyclops architecture included data caches in the design, and the C64 system employs scratchpad memory technology instead of data cache. Neither [17] nor [18] conducted performance characterization of the OpenMP language constructs, since that was not the purpose of those two papers.

7 Conclusion and Future Work

Multi-core or many-core-on-a-chip architecture tends to be widely accepted in the near future. Given the massive intra-chip parallelism, a high level parallel programming model is needed for fast and efficient application development. OpenMP is considered as one reasonable candidate. In order to help the application developer and system software designer to increase the understanding of the performance behavior of OpenMP programs on many-core-on-a-chip architecture, this paper reports the performance characteristics of OpenMP language constructs on the Cyclops-64 chip architecture, which integrates 160 cores in a single chip. As for the future work, we would like to evaluate the performance of OpenMP on C64 with application kernels and benchmarks, such as NAS parallel benchmarks, and the SPEC OMP Benchmark suite.

Acknowledgments

We would like to acknowledge the support from IBM, in particular, Monty Denneau, who is the architect of the IBM Cyclops-64 architecture, ETI, the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), and other government sponsors. We would also like to acknowledge other members of the CAPSL group at University of Delaware, in particular Ziang Hu, Yuan Zhang, Geoff Gerfin, and Brice Dobry.

References

1. Fredrickson, N.R., Afsahi, A., Qian, Y.: Performance characteristics of OpenMP constructs, and application benchmarks on a large symmetric multiprocessor. In: Proceedings of the 17th annual international conference on Supercomputing (ICS'03), New York, NY, USA, ACM Press (2003) 140–149
2. Hammond, L., Nayfeh, B.A., Olukotun, K.: A single-chip multiprocessor. Computer **30**(9) (1997) 79–85
3. Denneau, M., Warren, Jr., H.S.: 64-bit Cyclops principles of operation part I. Technical report, IBM Watson Research Center, Yorktown Heights, NY (2005)

4. Denneau, M., Warren, Jr., H.S.: 64-bit Cyclops principles of operation part II: Memory organization, the A-switch, and SPRs. Technical report, IBM Watson Research Center, Yorktown Heights, NY (2005)
5. OpenMP Architecture Review Board: OpenMP C and C++ application program interface. Technical Report 2.5, OpenMP Architecture Review Board (2005) In http://www.openmp.org/specs.
6. Kusano, K., Satoh, S., Sato, M.: Performance evaluation of the Omni OpenMP compiler. In: the Third International Symposium on High Performance Computing, Tokyo, Japan (2000) 403–414
7. del Cuvillo, J., Zhu, W., Gao, G.R.: Landing OpenMP on Cyclops-64: An efficient mapping of OpenMP to a many-core system-on-a-chip. In: Proceedings of the 3rd ACM International Conference on Computing Frontiers, Ischia, Italy (2006)
8. Bull, J.M.: Measuring synchronization and scheduling overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, Lund, Sweden (1999)
9. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Toward a software infrastructure for the Cyclops-64 cellular architecture. In: Proceedings of 20th International Symposium on High Performance Computing Systems and Applications, St. John's, Newfoundland and Labrador, Canada (2006)
10. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture. In: Fifth Workshop on Massively Parallel Processing, in conjuction with 19th International Parallel and Distributed Processing Symposium (IPDPS 2005), Denver, Colorado, USA (2005) 265
11. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In: Workshop on Modeling, Benchmarking, and Simulation (MoBS2005), in conjuction with the 32nd Annual International Symposium on Computer Architecture (ISCA2005), Madison, Wisconsin (2005)
12. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9**(1) (1991) 21–65
13. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. SIGARCH Comput. Archit. News **29**(5) (2001) 41–48
14. Berrendorf, R., Nieken, G.: Performance characteristics for OpenMP constructs on different parallel computer architectures. Concurrency - Practice and Experience **12**(12) (2000) 1261–1273
15. Prabhakar, A., Getov, V., Chapman, B.: Performance comparisons of basic OpenMP constructs. In: Proceedings of the 4th International Symposium on High Performance Computing. Number 2327, Kansai Science City, Japan (2002) 413–424
16. Liao, C., Liu, Z., Huang, L., Chapman, B.: Evaluating OpenMP on chip multithreading platform. In: First International Workshop on OpenMP, Eugene, Oregon USA (2005)
17. Almasi, G., Ayguadé, E., Cascaval, C., José Castanos, J.L., Martínez, F., Martorell, X., Moreira, J.: Evaluation of OpenMP for the Cyclops multithreaded architecture. Lecture Notes in Computer Science **2716** (2003) 69–83
18. Ródenas, D., Martorell, X., Ayguadé, E., Labarta, J., Almási, G., Caşcaval, C., Castaños, J., Moreira, J.: Optimizing NANOS openMP for the IBM Cyclops multithreaded architecture. In: 19th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2005), Denver, Colorado, USA (2005)