Evaluation of OpenMP for the Cyclops Multithreaded Architecture

George Almasi[2], Eduard Ayguadé[1], Călin Caşcaval[2], José Castaños[2],
Jesús Labarta[1], Francisco Martínez[1], Xavier Martorell[1], José Moreira[2]
[1]CEPBA-IBM Research Institute, UPC - Barcelona, Spain.
{eduard, jesus, fmartin, xavim}@ac.upc.es
IBM Thomas J. Watson Research Center - Yorktown Heights, NY
{gheorghe,cascaval,castanos,moreira}@us.ibm.com

Abstract. *Multithreaded architectures have the potential of tolerating large memory and functional unit latencies and increase resource utilization. The Blue Gene/Cyclops architecture, being developed at the IBM T. J. Watson Research Center, is one such systems that offers massive intra-chip parallelism. Although the BG/C architecture was initially designed to execute specific applications, we believe that it can be effectively used on a broad range of parallel numerical applications. Programming such applications for this unconventional design requires a significant porting effort when using the basic built-in mechanisms for thread management and synchronization. In this paper, we describe the implementation of an OpenMP environment for parallelizing applications, currently under development at the CEPBA-IBM Research Institute, targeting BG/C. The environment is evaluated with a set of simple numerical kernels and a subset of the NAS OpenMP benchmarks. We identify issues that were not initially considered in the design of the BG/C architecture to support a programming model such as OpenMP. We also evaluate features currently offered by the BG/C architecture that should be considered in the implementation of an efficient OpenMP layer for massive intra-chip parallel architectures.*

1 Introduction and motivation

Multithreaded architectures are a promising trend for the design of future high-performance microprocessor cores. Their ability to tolerate large memory and functional unit latencies and to increase resource utilization put them in the right position to achieve a high number of instructions per cycle (IPC). Tera MTA [28] and SMT [34] (an example of which is Intel Hyperthreading technology [6]) have followed this approach. Another trend is the integration of several microprocessor cores in the same chip, such as in the IBM Power4 [31]. Each processor has its own resources and shares the access to higher levels in the memory hierarchy such as off-chip main memory.

Multiprocessors systems-on-a-chip based on the replication of multithreaded cores offer a complexity-conscious alternative to future chip designs. The Blue Gene/Cyclops (BG/C) chip [9], which is the core of a new family of multithreaded architectures developed by IBM Research, consists of a large number

Almasi, G. [et al.]. Evaluation of OpenMP for the Cyclops multithreaded architecture. "Lecture notes in computer science", Juny 2003, vol. 2716, p. 69-83. The final authenticated version is available online at https://doi.org/10.1007/3-540-45009-2_6

of simple thread units simultaneously executing independent streams of instructions. Each thread behaves like a simple, single-issue, in order processor. Groups of threads share floating-point units and caches. All threads share a single address space implemented with an embedded DRAM memory in the same chip, resulting in a flat memory hierarchy with high bandwidth and low latency.

Making these new parallel architectures truly usable requires portable and easy-to-understand programming models that allow the exploitation of parallelism to applications written in standard high–level languages. Pthreads-like approaches are always possible but require a large programming effort. The user has to face the complexity of managing the parallelism at application level, manually handling thread creation, work distribution, allocation of variables and synchronization. The built-in parallel programming model provided by BG/C falls in this category. OpenMP [20] has emerged as the standard for shared-memory parallel programming. OpenMP applications are simple to program, portable across a range of shared-memory parallel platforms, and achieve near optimal parallel performance. The goal of this paper is to prove that OpenMP is a valid programming model for a machine that supports fine–grain multithreading, such as BG/C, and thus provide the user with a simple programming model for a complex machine.

We have ported the NthLib user-level threads library [19] to Cyclops in order to develop an experimental research platform, and used the Linux version of the NanosCompiler[10] to generate code. The current version does not consider some specific hardware features offered by the architecture. We will discuss about this limitation in Section 6, where we propose changes that will allow OpenMP to better exploit processor resources.

This paper is organized as follows: Section 2 describes the main characteristics of the BG/C architecture family and in particular, the configuration used in this paper. Section 3 describes the implementation of the OpenMP layer based on the NanosCompiler and NthLib. Section 4 describes the set of microbenchmarks and a subset of the NAS BT benchmarks used to obtain the experimental results presented in Section 5. The later section also shows the feasibility of programming OpenMP applications for BG/C. Section 6 discusses the explicit support for OpenMP that would be required in the architecture and the issues that should be considered to tune the implementation of the OpenMP layer. Finally, Section 7 outlines related work and Section 8 concludes the paper and outlines major directions in our future work.

2 The Blue Gene/Cyclops architecture

The main characteristic of the BG/C design is the integration of embedded DRAM, processing logic and communications hardware on the same piece of silicon. The proximity of memory and processors results in a flat memory hierarchy which overcomes the von Neumann bottleneck (processor performance improves faster than the capacity of memory to serve it) observed in conven-

tional designs. Instead of hiding latencies through out-of-order or speculative execution, BG/C nodes tolerate latencies through massive parallelism. The solution adopted by BG/C is to use multiple threads in a single node so that, if a thread stalls for a memory reference, other threads can make progress. As a result each thread unit is simpler and expensive resources, such as FPUs and caches, are shared between different threads.

The organization of the BG/C chip is shown in Figure 1. At the base of the BG/C hierarchy are thread units. BG/C is a multithreaded design where thread units are simple computing processors that issue and execute instructions in program order. Each thread can issue an instruction at every cycle if resources are available and there are no dependences with previous instructions. Each thread unit consists of a register file (64 32-bit single precision registers, that can be paired for double precision values), a program counter, a fixed-point ALU, and an instruction sequencer.

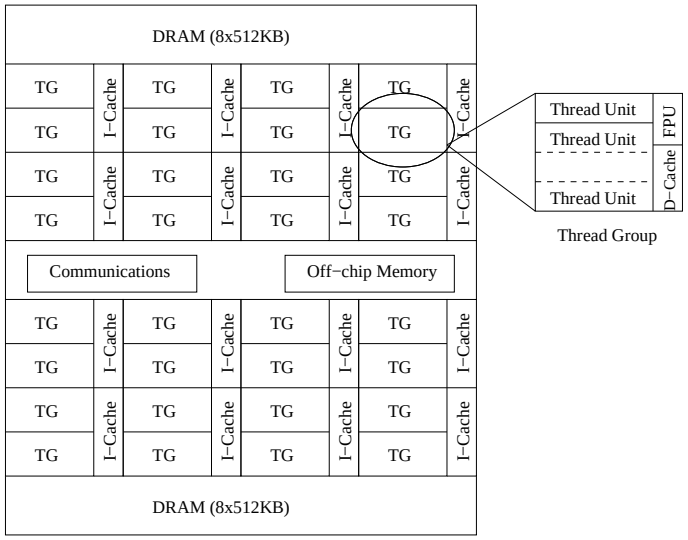

Fig. 1. Block diagram for a prototype of the BG/C architecture.

Groups of threads units share an FPU and a data cache. Threads can dispatch a floating point addition and a floating point multiplication at every cycle. The base architecture in $0.18\mu m$ CMOS technology and 32 FPUs achieves a peak performance of 1 GFlops per FPU at a clock cycle of 500 MHz, for a total chip performance of 32 GFlops.

Each of the 32 16 KB data caches (one per thread group) has 64-byte lines and is 8-way set associative. By default, all data caches behave as a global, coherent cache. The data caches are shared among all threads in the chip. Thus, a thread can access data in the cache of another thread group with lower latency

than going to memory. Instruction caches are 32 KB, 8-way set-associative with 64-byte line size. In the base architecture, one instruction cache is shared by 2 thread groups. Unlike the data caches, the instruction caches are private to the threads in the thread groups. In addition, to improve instruction fetching, each thread unit contains a Prefetch Instruction Buffer (PIB) of 32 instructions.

The reference design considered in this paper has 16 banks of on-chip memory shared between thread units. Each bank is 512 KB for a total of 8 MB of embedded memory. The banks provide a contiguous address space to the threads. The latency to any bank is uniform. Addresses are interleaved to provide higher memory bandwidth. The unit of access is a 32-byte block, and threads accessing two consecutive blocks in the same bank will see a lower latency in burst transfer mode. The peak bandwidth of the embedded memory system is 40 GB/s (64 bytes every 12 cycles in each of the 16 banks).

In addition to the default all-shared cache behaviour, the architecture supports an entire spectrum of access schemes through *interest groups* [5], from no sharing at all to caches shared at different levels. Any memory location can be placed in any cache under software control. The same physical address can be mapped to different caches depending on the logical address. An important use of this flexible cache organization is to exploit locality and shared read-only data. For example, data frequently accessed by a thread, such as stack data or constants, can be cached in the local cache by using the appropriate interest group. The hardware does not implement any coherence mechanism to deal with multiple copies of a memory line in different data caches.

Four global inter-thread hardware barriers are provided through a special purpose register (SPR). These barriers are implemented as a wired OR for all or a user defined subset of the threads on the chip.

The BG/C chip also provides six input and six output links. These links allow a chip to be directly connected in a three dimensional topology (mesh or torus). The links are 16-bit wide and operate at 500 MHz, giving a maximum I/O bandwidth of 12 GB/s. In addition, a seventh link can be used to connect to a host computer. These links can be used to build larger systems without additional hardware. Another port permits the access to external (off-chip) memory. However, these latter characteristics are not the focus of this paper.

BG/C executables (kernel, libraries, applications) are currently being generated with a cross-compiler based on the GNU toolkit, re-targeted for the BG/C instruction set architecture. This cross-compiler supports C, C++, and FORTRAN 77.

The performance results shown in Section 5 are generated by an architecturally accurate simulator which executes instructions from the BG/C instruction set, modeling resource contention between instructions, and thus estimating the number of cycles executed per instruction. The configuration parameters used for the simulations in this paper are listed in Table 1.

In addition, each chip runs a resident system kernel, which executes with supervisor privileges. The kernel supports single user, single program, multithreaded applications within each chip. The kernel exposes a single-address

space shared by all threads. Due to the small address space and large number of hardware threads available, no resource virtualization is performed in software: virtual addresses map directly to physical addresses (no paging) and software threads map directly to hardware threads. The kernel does not support preemption (except in debugging mode), scheduling or thread priorities. Every software thread is preallocated with a fixed size stack per thread (selected at boot time), resulting in fast thread creation and reuse.

Table 1. Design parameters for the reference BG/C architecture. In (a) we show the number of cycles for execution and latency of the main instruction types. Execution is the number of cycles the functional unit is busy; latency is the additional delay until the results of the operation are available.

(a) instructions

Instruction type	Execution	Latency
Branches	2	0
Integer multiplication	1	5
Integer divide	33	0
Floating point add, mult. and conv.	1	5
Floating point divide (double prec.)	30	0
Floating point square root (double prec.)	56	0
Floating point multiply-and-add	1	9
All other operations (except memory ops.)	1	0

(b) components

Component	# of units	Params/unit
Threads	1-256	single issue, in-order, 500 MHz
FPUs	32	1 add, 1 multiply
D-cache	32	16 KB, 8-way assoc., 64-byte lines
I-cache	16	32 KB, 8-way assoc., 64-byte lines
Memory	16	512 KB

3 Towards OpenMP for BG/C

OpenMP for BG/C is based on the NanosCompiler and the NthLib components. The OpenMP NanosCompiler is a source-to-source translator for Fortran77 based on Parafrase-2 [23]. NthLib is a runtime library designed to provide an efficient support to the OpenMP execution model on shared-memory multiprocessors. Fine grain parallel tasks are implemented as efficiently as other thread packages: the application creates work descriptors and supplies them to the participating threads [18]. A mechanism to spawn coarse grained parallel tasks, called *nanothreads* [19] is also available. This mechanism is more expensive but allows the explotation of multiple levels of parallelism.

Figure 2 presents the software architecture used for supporting OpenMP. Kernel-level threads are the processor abstraction in our environment. User-

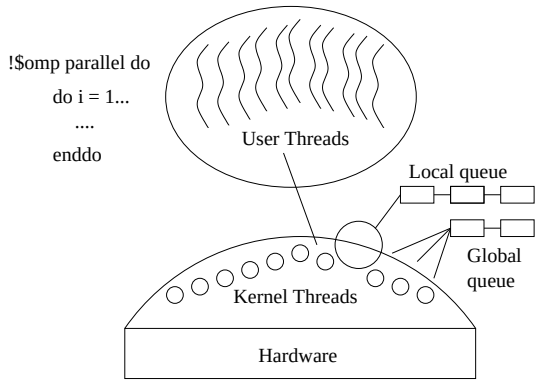

Fig. 2. Software architecture of OpenMP on BG/C.

level threads are supported on top of the kernel-level threads, and they represent the abstraction for work. Kernel-level threads are created when the application starts and are kept alive during the entire execution of the application. User-level threads are spawned as needed to create parallel regions – each thread executing a share of the whole work. Assigning the task to a thread is implemented by queueing the nanothread or work descriptor in one of the per-processor queues or in the global queue. The per-processor queues allows us to exploit locality; the global queue can be used for load balancing reasons, but it is not used for the experiments presented in this paper.

The implementation of NthLib for BG/C currently uses few specific system services and architectural features. Its only requirements are a processor allocation mechanism, control over stack placement and memory management, and a set of atomic memory operations. The thread creation mechanism provided by the BG/C system library is the *bg_svc_thread_create_specific* call. This primitive creates a kernel–level thread to run on a specific hardware thread. This way, the kernel–level threads may be mapped to specific processors. Other service calls similar to pthreads are used for thread management, like *bg_svc_thread_join*. Stack management was implemented giving fixed-size stacks to threads on creation, thanks to the ability of the thread creation service to use a designated stack as the thread stack. The implementation assumes the default all-shared cache organization inside the chip.

4 Benchmark description

In order to evaluate the performance of our OpenMP implementation for BG/C, we have used a set of microbenchmarks and a subset of the NAS benchmarks, version 2.3. The purpose of the microbenchmarks is to compare the performance of the OpenMP parallelization and the performance of the hand-optimized versions to execute on BG/C. The NAS benchmarks show that our OpenMP implementation scales to handle large, realistic applications.

4.1 Micro-benchmarks

Throughout our evaluation we have used microbenchmarks to study specific properties of the architecture. In this paper, we present results for two scientific kernels: dense matrix multiplication and sparse matrix-vector product. Their simplicity allows the validation of results by direct study of the assembly listings, which increases our confidence in the correctness of the experiments.

Three versions for each kernel have been coded: Pthreads, OpenMP and Pthreads without optimizations (*wco*). In the Pthreads versions the program contains code to fork and join parallel threads. In addition, each thread executes code to determine the portion of work that has to execute and synchronizes with other threads by means of barriers. The OpenMP version simply contains the parallel and work-sharing directives necessary to express the same parallelization strategy (or the closest one, if not possible). The compiler takes care of generating the code for distributing the work and synchronizing across the threads.

MM The matrix multiplication kernel, MM, computes $AB = C$ with $A_{m \times p}$, $B_{p \times n}$ and $C_{m \times n}$, where $m = 192$, $n = 192$ and $p = 100$ using the simple three-nested loops algorithm (high school matrix multiply). The data set results in a storage requirement of about 0.59 MB. That size essentially fits in the global D-cache.

The pthread-based implementation of MM distributes the matrix C evenly among $t = r \times s$ threads, resulting in each thread owning a rectangular section of C. Each thread computes only the portion of C it owns. The OpenMP implementation of MM distributes the work evenly among the columns of C: one-level STATIC block distribution of the iterations in the loop that traverse the columns of the result matrix. MM requires no synchronization between threads.

SPARSE, the multiplication of a sparse matrix by a vector, is the main kernel of many iterative linear solvers. Our implementation represents the sparse matrix S using *row-indexed sparse storage* [27,24]. This scheme stores the diagonal elements and the non-zero elements in a vector of values `val`. The columns of the non-zero elements are stored in an integer vector `idx`.

The inner loop of the sparse-matrix vector product $Sx = y$

```
for (k=idx[i]; d < idx[i+1]; k++)
   y[i] += val[k] * x[idx[k]]
```

requires three memory loads for every non-zero element k. The location of the dependent loads for the indirect access to x is particularly difficult to predict and the latency is difficult to hide. For that reason most sparse-matrix vector codes suffer from poor performance.

In both implementations (Pthreads and OpenMP), the rows of the matrix S and the solution vector y are partitioned between threads. This method does not require thread synchronization. A fill parameter f controls the sparsity of

the matrix: one of every f elements in each row i is non zero, starting at column i mod f. Thus, the vector x is traversed in sequential order.

Threads multiply one or two rows of the matrix at the same time, and in the manual implementation the inner loop is unrolled 8 times. The test problem $Sx = y$ with matrix size 1024×1024 and fill factor $f = 4$ requires 3.03 MB of main memory.

4.2 NAS benchmarks

We have also evaluated our BlueGene/Cyclops OpenMP implementation with a subset of the NAS benchmarks which are a representative set of computing intensive applications. We have used the OpenMP Fortran77 benchmarks in NAS PBN [12], version 2.3.

We have simulated fully both CLASS S and CLASS W benchmarks, although the sets are not the same because of memory capacity problems. The programs simulated from CLASS S (the smallest class) are MG, FT, SP, CG, and LU. Although they are small in data sizes and number of iterations, the complexity of the application is the same, enabling a complete evaluation of the OpenMP compilation environment in a reasonable amount of simulation time. The programs simulated in CLASS W are MG, BT, SP, CG, and LU. Their description can be found in [3][7].

5 Experimental Results

In this section, we show the results of our simulations. All the examples are naively implemented with no hand-coded manual optimizations for scalar performance (loop unrolling, blocking, ...). We briefly make some comments about how they impact performance and the reader is referred to [2]. That report evaluates specific features of the BG/C architecture using applications fine-tuned to execute at maximal performance.

For the micro-benchmarks, we plot performance results in MFLOPs when considering the parallelism fork and join overheads and when just considering the useful work executed in parallel (plot labeled *wco*). For the NAS subset we plot the speedup relative to the sequential version.

5.1 Micro-benchmarks

Figures 3 and 4 show that the performance obtained by the OpenMP version of these benchmarks is similar to that of the hand-coded pthreads versions (in some cases, even better). They tend to scale at least as well as the hand-coded Pthreads versions.

There is an anomaly worth highlighting, though, in MM. The *OpenMP* plot shows that the performance improvement stalls at 96 threads. This is due to the different work distribution schemes used in the Pthreads and OpenMP versions. The Pthreads versions simultaneously distributes work from two of the

loops in the nest where the matrix multiply is done. The OpenMP version just distributes work from one of the loops. Since the parallelized loop executes 192 iterations, using more that 96 threads results in a highly unbalanced assignment of iterations to threads (two iterations are assigned to some of them and one to the rest). Linearizing the loop or using two levels of parallelism would produce a balanced distribution of work, thus achieving the same performance.

By contrast, overheads are the problem for the degradation of performance in SPARSE – the code that distributes the work among threads is executed once in the Pthreads version, but multiple times in the OpenMP version.

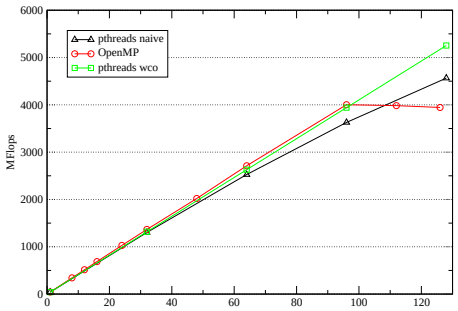

Fig. 3. MM performance

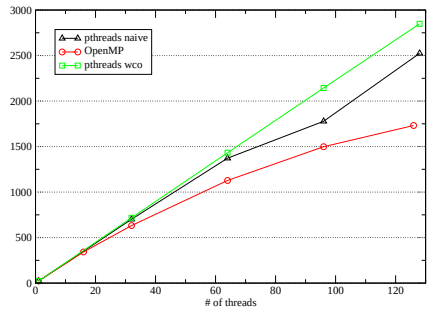

Fig. 4. SPARSE performance

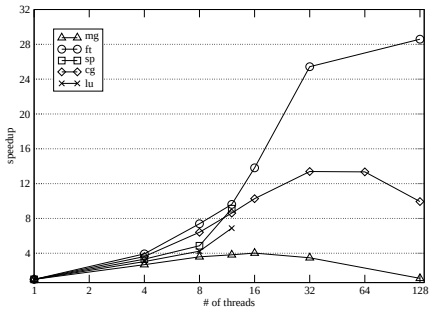

Fig. 5. NAS CLASS S scalability

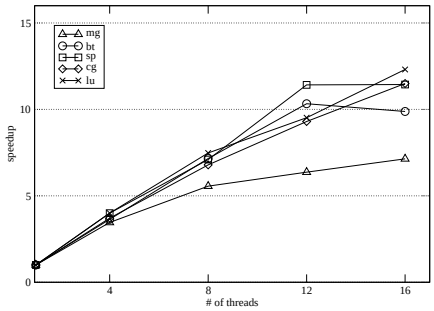

Fig. 6. NAS CLASS W scalability

5.2 NAS benchmarks

Figure 5 shows the scalability of a subset of the CLASS S NAS benchmarks in BG/C. CLASS S has been selected to show how the architecture performs with small data sets. Figure 6 shows the scalability of some CLASS W benchmarks. The reason for missing some of the benchmarks is the incipient state of the OpenMP environment, the limited hardware memory size, and limitations in

the simulator when running applications requiring a large memory size and a large number of threads.

The performance of most of the CLASS S benchmarks improves up to 16 processors. In this case, the scalability shown for CLASS S is better than the scalability obtained in other shared memory architectures, making us believe that further work in this direction will yield the expected results.

The first experiments done with CLASS W reveal that scalability improves as the data set grows. We believe that this is promising and shows that BG/C should be able to support large OpenMP applications.

6 Improving OpenMP Support for BG/C

As mentioned in Section 3, the NthLib library has been ported to BG/C without considering specific architectural features of the target machine. Some proposals to enhance the support to the OpenMP programming model in BG/C are described in this section.

- **Local versus global work descriptors**. In the current implementation of NthLib for BG/C, the master thread generates a work descriptor for each thread participating in the parallel execution, and supplies it to its per-thread local queue. This allows assignment of work in a very flexible way. However, in OpenMP this flexibility is not necessary, so that a single global work descriptor could be created and supplied to all per–thread local queues. This would reduce creation overheads, especially when the number of threads is large.
- **Take advantage of interest groups in data caches**. In the current implementation, all caches are shared. It would be possible to use other cache sharing possibilities in order to privatize variables. For example, when multiple levels of parallelism are exploited, groups of threads can be defined (OpenMP extensions supported by the NanosCompiler). In this case, threads in the same group share data that is privatized among groups.
- **Using hardware barriers**. BG/C offers an efficient implementation for barriers. However, this support is not sufficient in NthLib because a thread needs to look for work on the queues while waiting on a barrier to be open. We plan to solve this problem using split barriers (offering for example *barrier_enter* and *barrier_leave* primitives). In this way, a thread could execute useful work while waiting on a barrier. In addition, we can devise a tree-like scheme, in which different threads that map to the same physical thread use different hardware barriers from the 4 available. This approach trades the number of global barriers available for a larger number of threads being able to synchronize.
- **Fine grain synchronization**. Locks could be implemented using hardware locks. These locks could stop the thread (or make it spin on an specific purpose register instead of in local cache) while waiting for the unlocking thread. In [35], it has been proposed to introduce two new instructions (*acquire* and *release*), as well as a locking hardware structure (*lock_box*). These

locks could have different implementations for the threads inside the same Thread Group and for threads in different Thread Groups. This contrasts with the current BG/C implementation, that is a via the *test_and_set* atomic operation provided by the ISA. This could also help to reduce power consumption because resources used by sleeping threads could be turned off.

- **Eliminating the idle loop**. When a thread is waiting in the idle loop, it is wasting resources without a real need (for instance, cache bandwidth or energy). In a normal multiprocessor system, the idle loop spins in the local cache, thus wasting no resources at all; however, in BG/C the data cache is shared among threads so the spinning consumes a portion of its total bandwidth. A possibility would be to include hardware mechanisms to efficiently implement the idle loop similar to the one mentioned above to implement hardware locks. This would mean to stop the activity of a thread while waiting in the idle loop. The activity would be resumed as soon as work is queued in its per–thread local or global queues.
- **Eliminating user-level threads**. We can eliminate the user-level threads which causes many overheads, and implement thread creation directly via *bg_svc_thread_create_specific* calls. This would solve the barrier problem – we can use hardware barriers. This would also eliminate the idle loop. Although it has been proven that this is not viable in current multiprocessor systems, it should be studied for multithreaded systems as the BG/C with very low thread creation overhead. An hybrid implementation could allow choosing among the flexibility of user-level threads or the better performance of no scheduling levels. In this way, OpenMP applications that use a small number of threads (less than or equal to the number of hardware threads in a chip) could map the software threads to the physical threads, thus avoiding the overhead of context switches.

7 Related Work

Architectures that integrate processors and memories on the same chip are called Processor-In-Memory (PIM) or Intelligent Memory architectures. They have been spurred by technological advances that enable the integration of compute logic and memory on a single chip. These architectures deliver higher performance by reducing the latency and increasing the bandwidth of processor-memory communication. Examples of such architectures are EXECUBE [15], IRAM [22], Shamrock [14], Imagine [25], FlexRAM [13,32], DIVA [11], Active Pages [21], Gilgamesh [38] and MAJC [33]. The PIM chip is used either as a co-processor (Imagine, FlexRAM), or as the main engine in the machine (IRAM, MAJC, Shamrock), or as a *cell* in a larger system (MIT RAW [1,36], EXECUBE and BG/C). Another classification could be based on the number and type of the processors: FlexRAM and Imagine include many (more than 32) relatively simple processors, while EXECUBE, IRAM, MAJC, Piranha [4] and Shamrock include only a few (4-8). BG/C goes beyond what has been proposed, using hundreds of processors.

Simultaneous multithreading exploits both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. It was shown to be a more effective approach to improve resource utilization than superscalar execution. Results presented in [8,34] support our work by showing that there is not enough instruction-level parallelism in a single thread of execution, therefore it is more efficient to execute multiple threads concurrently.

The Tera MTA [28,29] is another example of a modern architecture that tolerates latencies through massive parallelism. In the case of Tera, 128 thread contexts share the execution hardware. This contrasts with BG/C, in which each thread has its own execution hardware. Both architectures can tolerate long latencies.

As far as we know, this is the first attempt to port an OpenMP runtime system to a massive parallel multithreaded system on–chip. The porting is based on the experience gained over the years on implementing such an environment on top of other execution environments, including small SMPs and large cc-NUMA. Vendors also provide fine-tuned implementations for their target machines, such as SGI IRIX MP[30] library or the IBM run-time library for AIX. For example, the SGI MP library provides a complete execution environment for each application, supporting thread creation, management, synchronization and NUMA features, such as memory placement. The library is aware of the machine load, trying to adjust the parallelism which is exploiting to the available resources. A number of projects also try to extend the use of OpenMP to clusters with DSM (Distributed Shared Memory) support. The long latencies experienced when accessing remote data and the memory granularity at the page level impose new constraints in these implementations [17,26].

The Nanos execution environment, which is the source for the two components used to implement OpenMP on top of BG/C, focus on adaptability at different levels, the effective exploitation of nested parallelism and the specification of precedence relations among computations that form pipelines. All these aspects form a set of extensions to OpenMP whose impact must be investigated in BG/C.

A number of studies have been recently published in which different compiler optimizations are evaluated for multithreaded architectures. For example, [16] relaxes and modifies some of the requirements on code scheduling and data access used by current compilers.

As stated before, the BG/C architecture is focused on the execution of a single multithreaded application within each chip. Other architecture proposals such as α–Coral [37] provides for mostly hardware managed simultaneous multiprogramming and multithreading environment. The Nanos environment also offers workload management at the software level with the CPUmanager component, specially designed for malleable OpenMP applications.

8 Conclusions

The Blue Gene/Cyclops architecture provides an excellent platform for studying programming environments for multithreaded architectures. Writing and porting applications for the BG/C architecture is not a simple process [2]. The very large number of threads (one or two orders of magnitude larger than similar architectures), the complexity of the cache organization, and the sharing of caches and floating point units are not yet easily modeled statically by compilers. The Pthreads execution model used until now for BG/C closely matches the hardware but adds another level of complexity to the process of writing software for BG/C.

In order to simplify this task, this paper introduces the implementation of an OpenMP environment for on–chip massive parallel architectures. This OpenMP environment together with the simulation environment for BG/C allows the exploration of a large number of SMT configurations with low programming effort. This permits us to better understand what are the trade-offs between multithreading characteristics and which properties are worth integrating in our implementation of the NthLib library.

This paper also shows that more tuning of our library is still required. With simple hand-optimized kernels, BG/C has demonstrated that its architecture is able to perform a very large percentage of the peak floating point performance [2] offered by the architecture. The results shown in previous sections showed that OpenMP applications can behave on par with Pthreads programs. Most of the optimizations used in [2], such as loop unrolling and register tiling, are orthogonal to the programming environment and will apply well to our OpenMP benchmarks. Nevertheless, there is also room for improvement in the case of large applications, such as the NAS benchmarks, as shown.

References

1. Anant Agarwal. Raw computation. *Scientific American*, August 1999.
2. George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Jr. Henry S. Warren. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. In *MEDEA Workshop on On-Chip Multiprocessor: Processor Architecture and Memory Hierarchy related Issues*, September 2002.
3. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
4. L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.
5. Călin Caşcaval, José Castaños, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, José E. Moreira, Karin Strauss, and Jr Henry S. Warren. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the 8th International Symposium of High Performance Computer Architecture*, February 2002.

6. Intel Corporation. Intel hyperthreading technology. http://www.intel.com/info/hyperthreading. 2003.
7. D.Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, R. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report Technical Report RNR-94-007, NASA Ames Research Center, March 1994.
8. Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, pages 12–18, September/October 1997.
9. Frances Allen et al. Blue gene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2):310–328, 2001.
10. M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting flexible multilevel parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(9), August 2000.
11. M. W. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCross, J. Brockman, W. Athas, A. Srivasava, V. Freech, J. Shin, , and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of SC99*, November 1999.
12. H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of the NAS parallel benchmarks and its performance. Technical Report Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
13. Yi Kang, Michael Huang, Seung-Moon Yoo, Zhenzho Ge, Diana Keen, Vinh Lam, Prattap Pattnaik, and Josep Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*, October 1999.
14. P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Frontiers of Massively Parallel Computation Symposium*, 1996.
15. Peter M. Kogge. The EXECUBE approach to massively parallel processing. In *Intl. Conf. on Parallel Processing*, August 1994.
16. Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *International Symposium on Microarchitecture*, pages 114–124, 1997.
17. H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on network of workstations. In *Proc. of Supercomputing'98*, 1998.
18. X. Martorell, E. Ayguadé, J.I. Navarro, J. Corbalán, M. González, and J. Labarta. Thread fork/join techniques for multi–level parallelism exploitation in NUMA multiprocessors. In *Proceedings of the 13th Int. Conference on Supercomputing ICS'99*, June 1999.
19. X. Martorell, J. Labarta, J.I. Navarro, and E. Ayguadé. A library implementation of the nano-threads programming model. In *Proceedings of Euro-Par'96*, August 1996.
20. OpenMP Organization. OpenMP Fortran application interface, v. 2.0. *www.openmp.org*, June 2000.
21. Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A computation model for intelligent memory. In *International Symposium on Computer Architecture*, pages 192–203, 1998.
22. David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM: IRAM. In *Proceedings of IEEE Micro*, April 1997.

23. Constantine D. Polychronopoulos, Milind B. Girkar, Mohammed Resa Haghighat, Chia Ling Lee, Bruce P. Leung, and Dale A. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *1989 International Conference on Parallel Processing*, volume II, pages 39–48, St. Charles, Ill., 1989.
24. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes in C. In *Cambridge University Press*, 1992.
25. S. Rixner, W.J. Dally, U.J. Kapasi, B. Khailany, A. Lopez-Lagunas, P.R. Mattson, and J.D. Owens. A bandwidth-efficient architecture for media processing. In *31st International Symposium on Microarchitecture*, November 1998.
26. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for an smp cluster, 1999.
27. Scientific Computing Associates, Inc. *PCGPACK user's guide*.
28. A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchel, J. Feo, and B. Koblenz. Multiprocessor performance on the Tera MTA. In *Proceedings Supercomputing '98*, Orlando, Florida, Nov. 7-13 1998.
29. A. Snavely, G. Johnson, and J. Genetti. Data intensive volume visualization on the Tera MTA and Cray T3E. In *Proceedings of the High Performance Computing Symposium - HPC '99*, pages 59–64, 1999.
30. Silicon Graphics Computer Systems. Origin2000 and Onyx2 performance tuning and optimization guide. Technical Report Doc. num. 007-3430-002, 1998.
31. J. M. Tendler, J. S. Dodson, Jr. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.
32. Josep Torrellas, Liuxi Yang, and Anthony-Trung Nguyen. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Sixth International Symposium on High-Performance Computer Architecture*, January 2000.
33. M. Tremblay. MAJC: Microprocessor architecture for Java computing. In *Hot Chips*, August 1999.
34. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.
35. Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *HPCA*, pages 54–58, 1999.
36. Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, September 1997.
37. M. Yankelevsky and C. D. Polychronopoulos. α–Coral: A multigrain, multithreading processor architecture. In *Proceedings of International Conference on Supercomputing'01*, 2001.
38. H. P. Zima and T. Sterling. The Gilgamesh processor-in-memory architecture and its execution model. In *Workshop on Compilers for Parallel Computers*, June 2001.