Factory: An Object-Oriented Parallel Programming Substrate for Deep Multiprocessors

Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos

Department of Computer Science, The College of William and Mary,
Williamsburg, VA 23187–8795
{scotts, cda, dsn}@cs.wm.edu

Abstract. Recent advances in processor technology such as Simultaneous Multithreading (SMT) and Chip Multiprocessing (CMP) enable parallel processing on a single die. These processors are used as building blocks of shared-memory multiprocessor systems, or clusters of multiprocessors. New programming languages and tools are necessary due to the complexities introduced by systems with multigrain, multilevel execution capabilities. This paper introduces Factory, an object-oriented parallel programming substrate which allows programmers to express multigrain parallelism, but alleviates them from having to manage it. Factory is written in C++ without introducing any extensions to the language. Because it leverages existing C++ constructs to express arbitrarily nested parallel computations, it is highly portable and does not require extra compiler support. Moreover, Factory offers programmability and performance comparable to already established multithreading substrates.

Keywords: Multithreading substrate, Object-oriented parallel programming, Deep parallel architectures, Multiparadigm parallelism, Portability, Programmability.

1 Introduction

Conventional processor technologies capitalize on increasing clock frequencies and on using the full transistor budget to exploit ILP. The diminishing returns of such approaches have shifted the focus of computer systems designers to clustering and parallelism. Current mainstream processors such as SMTs, CMPs and hybrid CMP/SMTs exploit coarse-grain thread-level parallelism at the microarchitectural level [1,2]. Thread-level parallelism is pervasive in high-end microprocessor designs as well [3,4].

Disparity in memory access latencies and the multiple levels of parallelism offered by emerging hardware necessitate programming languages, libraries and tools that enable users to express and control such parallelism. Furthermore, programmers need the means to map different granularities of parallelism to the different levels offered by emerging hardware. Unfortunately, current industry standards for expressing parallelism, such as MPI [5] and OpenMP [6], do not rise

L.T. Yang et al. (Eds.): HPCC 2005, LNCS 3726, pp. 233–242, 2005.

to these challenges, because they are designed and implemented with optimized support for a flat parallel execution model and provide little to no additional support for multilevel execution models.

In this paper, we present *Factory*, an object-oriented parallel programming substrate written entirely in C++. Factory was designed as a substrate for implementing next-generation parallel programming models that naturally incorporate multiple levels and types of parallelism controlled by an intelligent runtime system. Factory is functional as a standalone parallel programming library without requiring additional compiler or preprocessor support. The main goals of Factory are to:

- Provide a clean, object-oriented interface for writing parallel programs.
- Provide a type-safe parallel programming environment.
- Define a unified interface to multiple types of parallelism.
- Allow effective exploitation and granularity control for multilevel and multitier parallelism within the same binary.
- Provide a pure C++ runtime library which does not need external interpreter or compiler support.

We outline the design and implementation of Factory and evaluate its performance using a multi-SMT compute node as a target testbed. Our primary contribution is a concrete set of object-oriented capabilities for expressing multiple forms of parallelism in a unified manner, along with generic runtime mechanisms that enable the exploitation of multiple forms of parallelism in a single program. As such, Factory can serve as a runtime library for next-generation, object-oriented parallel programming systems that target deep, multigrain parallel architectures. Factory also makes contributions in the direction of implementing more efficient object-oriented substrates for parallel programming. Its features include lock-free synchronization, flexible scheduling algorithms that are aware of SMT/CMP processors and hierarchical parallel execution, and localized barriers for independent sets of work units.

The rest of this paper is organized as follows: Section 2 briefly discusses prior work which relates to Factory. In Section 3 we present the design of Factory. Section 4 compares Factory's performance with other multithreading programming models and substrates and shows that Factory can exploit the most commonly used forms of parallelism without compromising performance. We discuss future work and conclude the paper in Section 5. An extended version of this paper can be found in [7].

2 Related Work

There is a large body of earlier work in multithreading programming models and object-oriented frameworks for parallel programming. We focus on recent and active projects with strong relation to Factory.

Cilk [8] is an extension to C with explicit support for multithreaded programming. Cilk is designed to execute strict multithreaded computations and

provides firm algorithmic bounds for the execution time and space requirements of these computations. Although Factory shares some functionality with Cilk, it has a different and broader objective, since its focal point is the exploitation of multilevel and multiparadigm parallelism, including task-level, loop-level and divide-and-conquer parallelism.

OpenMP [6] is an industry standard for programming on shared memory multiprocessors. OpenMP is particularly suitable for expressing loop based parallelism in multithreaded C, C++ and Fortran programs. Instead of explicitly extending the language, programmers use compiler directives that adhere to the OpenMP standard to express parallelism. The current OpenMP standard has limitations related to the orchestration and scheduling of multiple levels of parallelism. A limited form of static task-level parallelism is supported in OpenMP via the use of parallel sections. Dynamic task-level parallelism is not currently supported by the OpenMP standard, although some vendors, such as Intel, provide platform-specific implementations [9,10]. Factory differs from OpenMP in that it provides a generic object-oriented programming environment for expressing multiple forms of parallelism explicitly and in a unified manner, while providing the necessary runtime support for effectively scheduling all forms of parallelism.

X10 [11] is an ongoing project at IBM to develop an object-oriented parallel language for emerging architectures. The proposed language has a rich set of features, including C++ extensions to describe clustered data structures, extensions to define activities (threads) for both communication and computation and associate these activities with specific nodes, and other features. We view Factory as a complementary effort to X10, which places more emphasis on the runtime issues that pertain to the management of multigrain parallelism, without compromising expressiveness and functionality. Furthermore, Factory can be used as a supportive runtime library for extended parallel object-oriented languages such as X10.

3 Design

The design of Factory focuses on leveraging existing C++ constructs to express multiple types of parallelism at multiple levels. We find the mechanisms provided by C++ expressive enough that we do not have to resort to defining a new language or language extensions which require a separate interpreter or compiler. The combination of inheritance and a sophisticated type system allows the design of a clean, well defined, high-level interface for the development of efficient parallel code. The implementation of Factory solely in C++ and exclusively at user level makes it a multithreading substrate that is portable across different architectures and operating systems. A more detailed presentation of the design, including a small object allocator optimized for object reuse across multiple threads, can be found in [7].

3.1 Enabling Multiparadigm Parallelism with C++

C++ enables the programmer to define class hierarchies. Factory exploits this feature to define all types of parallel work as classes which inherit from a general

work class. However, deeper in the hierarchy, classes are dissociated according to the type of work they represent. In the context of this paper we focus on task- and loop-parallel codes, however the Factory hierarchy is easily extensible to other forms of parallelism as well.

Inheritance allows the expression of different kinds of parallelism, with different properties, via a common interface. Factory exploits the C++ templates mechanism in order to adapt the functionality and the behavior of the multithreading runtime according to the requirements of the different forms of parallel work. As a result, Factory allows programmers to easily express different kinds of parallel work, with different properties, through a common interface. At the same time, they can efficiently execute the parallel work, transparently using the appropriate algorithms and mechanisms to manage parallelism.

Work as Objects: Objects are the natural way to represent chunks of parallel work in an object-oriented programming paradigm. Parallel work can be abstracted as an implementation of an algorithm and a set of parameters, which in turn can be directly mapped to a C++ object (represented as a `work_unit`). Specific chunks of the computation are consequently represented as objects of a work unit class.

The user-defined member function `work()` implements the computation for the specific work unit, and its member fields serve as the computation's parameters. For each type of computation the programmer defines a new class. Objects instantiated from this class represent different chunks of the computation. At runtime, Factory executes the `work()` member function of each `work_unit` object.

More details on the Factory interface can be found in [7].

Work Inheritance Hierarchy: All different kinds of Factory work units export a common API to the programmer as a way to enhance programmability. However, in order to differentiate internally between different kinds of work units and provide the required functionality in each case, Factory work units are organized in an inheritance hierarchy. The hierarchy structure facilitates the addition of new types of work, or the refinement of existing types, without interfering with unrelated types.

The `work_unit` base class is the root of the work inheritance hierarchy. It defines the minimal interface that a work unit must provide. Programmer defined work units do not inherit directly from `work_unit`, but rather from classes at the leaves of the inheritance tree, which correspond to particular types of work.

The `tree_unit` class derives directly from `work_unit`, and is used to express parallel codes that follow a dependence driven programming model. Work units which derive from `tree_unit` are organized as a dependence tree at runtime, which is used by Factory to enforce the correct order of work unit execution. Both `task_unit` and `loop_unit` derive from `tree_unit` and they are used by programmers to define task- and loop-parallel work chunks respectively. These classes provide the required support and functionality for the efficient execution of each specific type of parallel computation. A `plain_unit` can, in turn, be used for codes that are not dependence-driven and directly manage the execution of work chunks at the application level.

Work Execution: All the interaction of applications with the Factory runtime system occurs through an object of the `factory` class[1]. While `work_unit` classes are used to express the parallel algorithms, the `factory` class provides the necessary functionality for their creation, management and execution. The `factory` class defines member functions for starting and stopping parallel execution, as well as creating, scheduling, and synchronizing work units.

3.2 Scheduling

Factory incorporates a generic, queue-based runtime system based on local, per execution context work queues. The later are implemented using non-blocking, lock-free FIFO and LIFO queue management algorithms [12]. The queue hierarchy can be easily extended in order to map more accurately to the target parallel architecture. We have implemented several kinds of scheduling algorithms based on LIFO and FIFO execution order of work units, but programmers can also define their own, according to the specific needs of their applications. Our performance evaluation section demonstrates that Factory schedulers achieve identical or better performance than both generic and customized, application-embedded user-level schedulers.

Factory uses kernel threads as execution vehicles. Each execution vehicle is bound to a specific execution context and has its own local work queue, from which it receives work through the active scheduling algorithm. Load balancing is achieved via work stealing from remote queues. Factory provides hierarchy-conscious work stealing algorithms, which favor work stealing between execution contexts close in the architectural hierarchy.

3.3 Synchronization

Factory provides support for the efficient execution of dependence-driven parallel codes. Each work unit employs a *children* counter to keep track of the number of in-flight children work units. As a result, a dependence tree is dynamically formed and maintained at runtime. The leaves of the tree are work units without dependencies, which are either currently executing, or are ready to execute in the future. The internal nodes represent work units whose execution is blocked because they have to wait for the termination of their children before they can continue.

Correct order of execution is enforced through Factory barriers, which operate on a particular work unit. The execution is either blocked until all children work units in the dependence subtree of the calling work unit have terminated (`child_barrier()` member function of the `factory` class), or until both the children and the work unit itself have terminated (`root_barrier()` member function).

Whenever a barrier prevents further execution of a work unit, the corresponding execution vehicle is not blocked. The user-level scheduling algorithm is invoked, and the execution vehicle starts executing other work units. When

[1] Throughout the paper we use the notation Factory to refer to the multithreading substrate and `factory` to refer to the class.

the dependencies of the blocked work unit are satisfied, the blocked work unit is allowed to resume.

4 Performance Evaluation

We have experimentally evaluated the performance of Factory on a multilevel parallel architecture, namely an SMT-based multiprocessor. We compare Factory against other popular parallel programming models, namely OpenMP, Cilk and parallelization using POSIX threads.

Our experimental platform is a quad SMP, based on Intel Hyper-Threaded (HT) processors. Intel HT processors share most of the internal processor resources between 2 simultaneously executing threads. The system is equipped with 2 GB of main memory and runs Linux (2.6.8 kernel). We created our binaries using the Intel Compiler suite for 32-bit applications (version 8.1).

We experimented using both microbenchmarks to assess the overheads for managing parallelism and parallel applications to compare Factory against the aforementioned parallel programming models. All experiments throughout our evaluation have been executed 20 times. We report the average timings across all 20 repetitions. The 95% confidence interval for each data point has always been lower than 1.7% of the average.

4.1 Minimum Granularity of Exploitable Parallelism

The minimum granularity of parallelism that can be effectively exploited by any multithreaded substrate is directly related to the degree of overheads—both architecture-specific and software-related—associated with the creation and management of parallel jobs.

The minimum granularity experiment consists of a variable number of `pause` assembly instructions. The number of the instructions is reduced until a break-even point is identified, at which point the sequential execution is as fast as the parallel one. The sequential execution time of the number of instructions corresponding to the break-even point is the minimum granularity. We represent work with `pause` instructions because they incur as minimal interference as possible when executed simultaneously on the different execution contexts of a single HT processor. The minimum granularity is also a factor of the number of threads used for the parallel execution. We thus evaluate the minimum granularity for the parallel execution with 2, 4 and 8 threads which are either packed on as few or spread to as many physical CPUs as possible. The different binding schemes allow the evaluation of both intra- and inter-processor parallelism overheads.

Table 1 summarizes the measured minimum exploitable granularity of Factory and the other multithreading systems. We compare Factory against Cilk, which supports only strict multithreaded computations with recursive task parallelism, and OpenMP. For the latter, we distinguish between the task- and loop-minimum granularities, as the OpenMP runtime uses different mechanisms for each. For task parallelism we use Intel compiler's workqueue extensions to OpenMP [6,10]. Factory uses the same mechanisms for creating parallel work

Table 1. Comparison of the minimum granularity of effectively exploitable parallelism

	2 Threads		4 Threads		8 Threads
	1 CPU	2 CPUs	2 CPUs	4 CPUs	4 CPUs
Factory	6.2μsec	6.2μsec	10μsec	10μsec	26μsec
Cilk	121μsec	81μsec	153μsec	153μsec	222μsec
OpenMP task	20μsec	20μsec	26μsec	24μsec	202μsec
OpenMP loop	10μsec	6.2μsec	6.2μsec	4.2μsec	68μsec

units, regardless of whether these work units are used for task- or loop-parallelism. As a result, it is represented by only one entry in the table.

Factory's minimum task granularity is finer than Intel's task queue implementation in OpenMP and remains competitive with OpenMP's loop granularity even though Intel's implementation of loop- and task-level parallel execution is heavily optimized. At the same time, Factory proves able to exploit significantly finer granularity than Cilk. Although the point where Cilk achieves speedup is relatively high, the break-even point is significantly lower, close to the performance of OpenMP tasks. This behavior can be attributed to the fact that for very fine-grain parallel work, the Cilk runtime actually schedules multiple tasks to the same kernel thread. Hence, Cilk requires a relatively large work load before multiple threads are used to execute it. Both Cilk and OpenMP perform better when threads are spread to as many physical CPUs as possible. Factory overheads, on the other hand, are uncorrelated with thread placement, making it a more predictable multithreading substrate for deep, multilevel parallel systems.

4.2 Factory vs. Other Programming Models

Radiosity is an application from the Splash-2 [13] benchmark suite which computes the equilibrium distribution of light in a scene. It uses several pointer-based data structures and has an irregular memory access pattern. The code uses application-level task queues and applies work stealing for load balancing. Radiosity tests Factory's ability to handle fine grain synchronization, since it is sensitive to the efficiency of synchronization mechanisms [14]. It also allows a direct comparison of Factory with POSIX Threads as underlying substrates for the implementation of hand crafted parallel codes. Porting the original code to Factory required just the conversion of the task concept to a work unit object. Both implementations were executed with the options `-batch -largeroom`. The performance results are depicted in Figure 1.

Factory consistently performs better than the POSIX Threads, mainly due to its efficient, fine-grain synchronization mechanisms. There is a 17% performance improvement from 4 to 8 threads, which is significantly less than the 72% improvement from 1 to 4 threads. This degradation is caused by each Radiosity thread using almost all shared execution resources.

We tested Factory using both LIFO and FIFO lock-free scheduling policies. LIFO execution ordering yielded better performance due to temporal locality, since data shared between the parent and children work units are likely to be found in the processor cache if a LIFO ordering is applied.

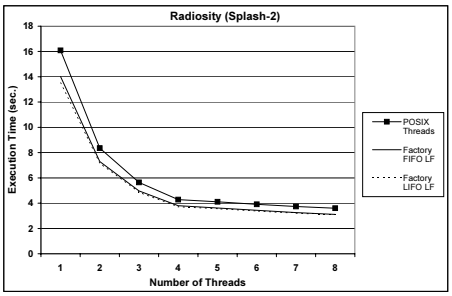

Fig. 1. Performance of Factory and POSIX Threads Radiosity implementations

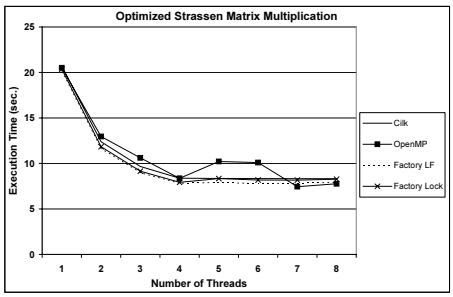

Fig. 2. Performance of Factory, Cilk, and OpenMP taskq for parallel, Strassen matrix multiplication

As a next step, we experimented with an optimized parallel implementation of the Strassen algorithm from the Cilk distribution. The algorithm is applied on 2048x2048 double precision floating point matrices. The OpenMP version of the application is based on Intel's OpenMP extensions for the support of task queues. Once again, the conversion to the Factory programming model was straightforward. We replaced recursive Cilk functions with work unit classes (`task_unit`). The conversion to OpenMP was also simple: recursive calls to Cilk functions have just been preceded by OpenMP `task` directives.

As shown in Figure 2, we also experimented with lock-free and lock-based queue implementations in Factory. All four implementations attain good scalability until 4 threads. After that point, at least one processor is forced to execute threads on both SMT contexts. When more than 4 threads are used, the OpenMP implementation suffers erratic performance. Cilk is not affected by intra-processor parallelism. It should be noted that Cilk's work stealing algorithm avoids locking the queues in the common execution scenario [15]. The Factory implementation that uses a lock-based queue implementation also suffers a performance degradation at 5 and 6 threads. However, the problem is solved if lock-free queues are used.

The performance degradation at 5 and 6 threads is related to synchronization. Previous studies indicate that lock-free algorithms are more efficient than lock-based ones under high contention or multiprogramming [12]. The execution of more than one thread on the execution contexts of SMT processors often has similarities to a multiprogrammed execution on a conventional SMP. As a result of resource sharing, SMT-based multiprocessors may prove more sensitive to the efficiency of synchronization mechanisms than conventional SMPs.

5 Conclusions and Future Work

We have presented Factory, an object-oriented parallel programming substrate, which allows the exploitation of multiple types of parallelism on deep parallel architectures. Factory uses a unified and clean interface to express different, po-

tentially nested forms of parallelism. Its implementation allows its use both as a standalone parallel programming library and as a runtime system for high-level object-oriented languages for parallel programming. The performance optimizations of Factory include lock-free synchronization for internal concurrent data structures and scheduling policies which are aware of the topology of hierarchical parallel systems. We have presented performance results that illustrate the efficiency of the central mechanisms for managing parallelism in Factory and justify our design choices for these mechanisms. We have also presented results obtained from the implementation of parallel applications with Factory which show that Factory performs competitively with other parallel programming models for shared-memory systems.

We regard Factory as a viable means for programming emerging parallel architectures and for preserving both productivity and efficiency. We plan to extend Factory in several directions, including the introduction of hierarchical scheduling algorithms that are aware of the type of parallelism they manage and the integration of precomputation into the runtime.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CAREER:CCF-0346867, and ITR: ACI-0312980. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

1. Tullsen, D.M., Eggers, S., Levy, H.M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: Proceedings of the 22th Annual International Symposium on Computer Architecture. (1995)
2. Hammond, L., Hubbert, B.A., Siu, M., Prabhu, M.K., Chen, M., Olukotun, K.: The Stanford Hydra CMP. IEEE Micro **20** (2000) 71–84
3. Takayanagi, T., Shin, J., Petrick, B., Su, J., Leon, A.: A Dual-Core 64b UltraSPARC Microprocessor for Dense Server Applications. In: Proc. of the 41st Conference on Design Automation (DAC'04), San Diego, CA, U.S.A. (2004) 673–677
4. Cascaval, C., Castanos, J., Ceze, L., Dennea, M., Gupta, M., Lieber, D., Moreira, J., Strauss, K., H. S. Warren, J.: Evaluation of a Multithreaded Architecture for Cellular Computing. In: 8th International Symposium on High-Performance Computer Architecture (HPCA-8), Cambridge, MA, U.S.A. (2002) 311–321
5. Forum, M.P.I.: MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230 (1994)
6. OpenMP Architecture Review Board: OpenMP Application Program Interface. Version 2.5 edn. (2005)
7. Scott Schneider, C.D.A., Nikolopoulos, D.S.: Factory: An Object-Oriented Parallel Programming Substrate for Deep Multiprocessors. Technical Report WM-CS-2005-06, The College of William and Mary (2005) http:// www.cs.wm.edu/ scotts/papers/wm-cs-2005-06.pdf.

8. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. In: Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming. (1995)
9. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible Control Structures for Parallelism in OpenMP. Concurrency: Practice and Experience **12** (2000) 1219–1239
10. Xinmin, T., Girkar, M., Shah, S., Armstrong, D., Su, E., Petersen, P.: Compiler and Runtime Support for Running OpenMP Programs on Pentium and Itanium architectures. In: Proceedings of the Eighth International Workshop on HighLevel Parallel Programming Models and Supportive Environments, Nice, France (2003) 47–55
11. Ebcioglu, K., Saraswat, V., Sarkar, V.: X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access. In: 3rd International Workshop on Language Runtimes. (2004)
12. Michael, M.M., Scott, M.L.: Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In: Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing (PODC'96), Philadelphia, Pennsylvania, U.S.A. (1996) 267–275
13. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proceedings of the 22th International Symposium on Computer Architecture, Santa Margherita Ligure, Italy (1995) 24–36
14. Radović, Z., Hagersten, E.: Efficient Synchronization for Non-Uniform Communication Architectures. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (2002) 1–13
15. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1998) 212–223