



W&M ScholarWorks

---

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

---

2005

## Factory: A n Object-Oriented Parallel Programming Substrate for Deep Multiprocessors

Scott Arthur Schneider  
*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>

 Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Schneider, Scott Arthur, "Factory: A n Object-Oriented Parallel Programming Substrate for Deep Multiprocessors" (2005). *Dissertations, Theses, and Masters Projects*. Paper 1539626845.  
<https://dx.doi.org/doi:10.21220/s2-2nvh-8p27>

This Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

FACTORY: AN OBJECT-ORIENTED PARALLEL  
PROGRAMMING SUBSTRATE FOR DEEP MULTIPROCESSORS

---

A Thesis

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Master of Science

---

by


Scott Arthur Schneider

2005

## APPROVAL SHEET

This thesis is submitted in partial fulfillment of  
the requirements for the degree of

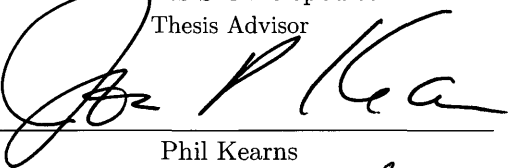
Master of Science

  
\_\_\_\_\_  
Scott Schneider

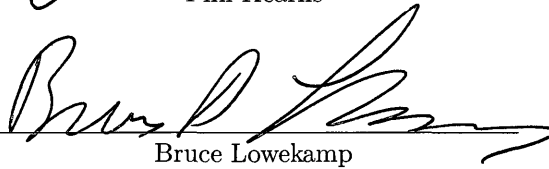
Approved by the Committee, June 2005

  
\_\_\_\_\_

Dimitrios S. Nikolopoulos  
Thesis Advisor

  
\_\_\_\_\_

Phil Kearns

  
\_\_\_\_\_

Bruce Lowekamp

Yes, Dad, it's done.

# Table of Contents

<b>Acknowledgments</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>Abstract</b>	<b>x</b>
<b>1 Introduction</b>	<b>2</b>
<b>2 Related Work</b>	<b>6</b>
<b>3 Design</b>	<b>10</b>
3.1 Enabling Multiparadigm Parallelism with C++ . . .	11
3.1.1 Work as Objects . . . . .	12
3.1.2 Work Inheritance Hierarchy . . .	13
3.1.3 Work Execution . . . . .	14
3.2 Scheduling . . . . .	15
3.3 Memory Management . . . . .	17

3.4	Synchronization . .	20
<b>4</b>	<b>Programming Examples</b>	<b>23</b>
4.1	Task Parallelism: Fibonacci . . . . .	23
4.2	Loop Parallelism: Naive Matrix Multiplication . . . . .	27
4.3	Multiparadigm Parallelism: Strassen Matrix Multiplication .	28
4.4	Programmability Comparisons . .	30
<b>5</b>	<b>Performance Evaluation</b>	<b>33</b>
5.1	Minimum Granularity of Exploitable Parallelism .	35
5.2	Managed vs. Unmanaged Memory Allocation . .	39
5.3	Memory Management . . . . .	41
5.4	Factory vs. POSIX Threads: Splash-2 Radiosity .	43
5.5	Factory vs. OpenMP : NAS IS . . . . .	45
5.6	Factory vs. Cilk and OpenMP: Single-level Parallel Strassen Matrix Multi- plication . . . . .	46
5.7	Factory vs. OpenMP: Multilevel Parallel Strassen Matrix Multiplication .	48
5.8	Thread Binding . .	50
<b>6</b>	<b>Conclusions and Future Work</b>	<b>52</b>
	<b>Bibliography</b>	<b>55</b>
	<b>Vita</b>	<b>59</b>

## ACKNOWLEDGMENTS

This thesis wouldn't be possible without the guidance of my advisor, Dr. Dimitrios S. Nikolopoulos. While Dr. Christos Antonopoulos was not officialy my advisor, in many ways he served as one, and this project would have taken far longer to complete without his assitance.

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.

# List of Tables

3.1	<code>work_unit</code> member functions . .	12
3.2	<code>factory</code> member functions .	15
4.1	Factory lines of code comparison . .	31
5.1	Hardware and software Experimental environment . .	34
5.2	Granularity comparisons . .	37
5.3	Managed and unmanaged execution times . .	40



# List of Figures

3.1	Work inheritance hierarchy .	13
3.2	Slab allocator internal organization . .	18
3.3	Unmanaged work example . .	20
3.4	Work unit dependence tree .	21
4.1	Fibonacci declaration . .	24
4.2	Fibonacci execution . .	25
4.3	Fibonacci definition . .	26
4.4	Naive_Matmul execution . .	27
4.5	Naive_Matmul definition . .	27
4.6	Strassen declaration . .	28
4.7	Strassen definition .	29
4.8	Before_Matrix_Add definition . .	30

5.1	Slab allocator vs. <code>new/delete</code> . .	42
5.2	Radiosity evaluation . .	44
5.3	NAS Integer Sort evaluation . . .	46
5.4	Single-level Strassen evaluation . .	47
5.5	Multiple-level Strassen evaluation . .	49
5.6	Comparison of binding schemes . .	50

## ABSTRACT

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

FACTORY: AN OBJECT-ORIENTED PARALLEL  
PROGRAMMING SUBSTRATE FOR DEEP MULTIPROCESSORS

# Chapter 1

## Introduction

Conventional processor technologies capitalized 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 [23, 37]. Thread-level parallelism is pervasive in high-end microprocessor designs as well. The Cray X1 main processing node allows the simultaneous execution of four streams, each of which can exploit a dedicated vector processing unit [34]. Sun's early efforts in the Hero project resulted in research prototypes of chip multithreading processors which allow simultaneous execution of 32 to 64 threads [28, 35]. IBM's Cyclops processor allows the execution of up to 128 threads over a non-cache-coherent DSM substrate on a single chip [13].

Alongside large degrees of parallelism on a single chip, there is a clear trend towards designing parallel systems with nested clustered organizations, (e.g., a large array of boards,

where a single board may contain tens of compute nodes and each compute node may be able to run tens of threads). Due to the extreme disparity in memory access latencies and the multiple levels of parallelism offered in hardware, such computer organizations necessitate programming languages, libraries and tools that enable users to express both multiple forms and multiple levels of parallelism. Furthermore, programmers need the means to control the granularity of parallelism at different levels and match it to the capabilities of parallel and/or multithreaded execution mechanisms at different layers of the hardware. Current industry standards for expressing parallelism are not suited for these architectures, 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. MPI [19], a message passing standard for parallel programs, is optimized for a single level of parallel execution and incorporates hardware heterogeneity only in its internal communication mechanisms. Although multilevel parallel programs can be constructed using MPI at all levels [16], or MPI plus OpenMP [29], the MPI implementation itself does not include special features to manage multilevel parallelism efficiently. OpenMP, a standard for parallel programming on shared-memory machines, supports loop-level and task-level parallel execution well at a single level, but its support for nested parallel execution is limited, inflexible and largely implementation-dependent.

This thesis presents *Factory*<sup>1</sup>, 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 par-

---

<sup>1</sup>The name *Factory* is inspired by the fact that a factory is the place where workers (threads) perform work.

allelism, while delegating the task of orchestrating parallelism at different levels to an intelligent runtime environment. Factory is functional as a standalone parallel programming library without requiring additional compiler or preprocessor support. However, its design does not prevent its use as the runtime environment of a compiler for explicitly parallel programs. The main goals of Factory are to:

- Provide a clean object-oriented interface for writing parallel programs and preserving the advantages of object-orientation, particularly with respect to programmer productivity.
- 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 multi-tier parallelism within the same binary.
- Provide a pure C++ runtime library which can be easily integrated into existing languages and parallel programming models without the need for extra interpreters or compilers.

We outline the design, implementation and performance evaluation of Factory, using a multi-SMT compute node as a target testbed. Factory is complementary to concurrent efforts for developing object-oriented parallel languages for deep supercomputers [17], the foci of which are to increase expressiveness, enable performance optimizations for data access locality and improve overall productivity via language extensions. Its primary contribution

in this domain 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 such 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, parallel architectures. Factory also makes contributions in the direction of implementing more efficient object-oriented substrates for parallel programming. Its features include an efficient multithreaded memory management mechanism, the means to merge application-embedded memory management with library memory management, 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 thesis is organized as follows: Chapter 2 discusses prior work in the area of object-oriented parallel systems, languages and libraries which relate to Factory. In Chapter 3 we present the design of Factory. Chapter 4 provides detailed programming examples to illustrate its use. Chapter 5 compares Factory's performance with other methods of writing multithreaded programs and shows that Factory can exploit the most commonly used forms of parallelism without compromising performance. We discuss future work and conclude in Chapter 6.



## Chapter 2

# Related Work

C++ libraries for parallel programming are as old as C++ itself; the first library implemented in the language was a means to manage tasks at user-level [33]. Before then, there was already a considerable body of work in the areas of object-oriented frameworks for parallel programming and user-level multithreading languages and libraries. Instead of detailing all such projects, we focus on active work and categorize other related work by their similarities.

Cilk [7] is an extension to C with explicit support for multithreaded programming. A more recent version of Cilk, named Hood [30], is written entirely in C++ and shares similar algorithmic properties with the original version, albeit with a more efficient implementation. 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 (such as the use of work queues as a parallel execution mechanism), 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. Cilk focuses on the optimal execution of specific classes of task-level multithreaded computations on single-level parallel systems. Unlike Cilk, Factory does not require language extensions. Factory can be easily used to implement Cilk's scheduling and memory management algorithms. We evaluate the performance of Factory against Cilk using representative applications in Section 5.

Charm++ [25] is a parallel extension to C++ that uses various kinds of objects to represent computations and communication mechanisms in a distributed system. The focus of the Charm++ runtime system has been on providing dynamic load balancing strategies for clusters and multicomputers. Charm++ does not provide specific functionality for exploiting multigrain parallelism in architectures with nested parallel execution contexts. Factory's current implementation is focused on the improvement of parallel execution capabilities of tightly coupled shared memory multiprocessors. It is however, by design, extensible to distributed memory architectures without changes in its core functionality.

There are many other languages and libraries which use an object-oriented approach to express parallelism. Most are for distributed parallel programming, such as pC++ [8], CC++ [14], Orca [5], Amber [15], and Mentat [22]. PRESTO [6] is a predecessor to Amber which is for shared-memory machines, and  $\mu$ C++ [11] takes a similar approach. Like Charm++, these projects leverage an object-oriented design to express parallelism. Of these projects, most chose to extend C++ to create a new parallel programming language (CC++, pC++, Mentat,  $\mu$ C++). Orca, however, is not an extension of a sequential language, but a new language designed explicitly for parallel programming. Factory differs from these

languages and libraries in that it targets deep multiprocessors and has a unified interface to the two kinds of parallelism most commonly used on shared memory machines.

OpenMP [29] is an industry standard for programming on shared memory multiprocessors. OpenMP is particularly suitable for expressing loop based parallelism in multithreaded programs. Instead of explicitly extending the language, programmers use compiler directives that adhere to the OpenMP standard to express parallelism. The standard currently supports C, C++ and Fortran. Despite the convenience of the programming interface, the OpenMP standard has limitations and inflexibility, particularly with respect to the orchestration and scheduling of multiple levels of parallelism. A limited form of static task-level parallelism can be supported in OpenMP via the use of parallel sections. Dynamic task-level parallelism is not currently supported in a standardized manner in OpenMP, although some vendors, such as Intel, provide platform-specific implementations [31, 41]. 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 [18] is an ongoing project at IBM to develop an object-oriented parallel language for emerging architectures. Among other ongoing projects, X10 is closest to the Factory in terms of design principles and objectives. The proposed language has a very 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 man-

agement 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.

The goal of the STAPL [2] project is to provide a parallel counterpart to the C++ Standard Template Library. Instead of providing explicit support for expressing parallelism, the programmer uses parallel algorithms and data structures. Efforts such as STAPL are also complementary to Factory. Factory could be used as a runtime library to support parallel execution within the algorithms of STAPL.

## Chapter 3

# Design

The design of *Factory* focuses on leveraging existing C++ constructs to express multiple types of parallelism at multiple levels. C++, being an efficient object-oriented programming language with extensive support for generic programming [21], is uniquely qualified for this task. 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. Inheritance facilitates the generalized expression of work. The sophisticated type system allows the library to adapt to different types of work at compile time. The combination of the two provides programmers with a clean, well defined, high-level interface which offers scheduling, synchronization and memory management functionality and can be exploited for the efficient development of parallel code.

The implementation of *Factory* solely in C++ and exclusively at user level makes it a multithreading substrate portable across different architectures and operating systems. *Factory* requires only a limited machine-dependent component for interfacing with the na-

tive kernel threads and implementing synchronization constructs with architecture-specific instructions. Even this component though, can be generalized, at least on UNIX-class systems, via an implementation on top of POSIX threads [24]. Our current prototype utilizes machine dependent synchronization primitives for efficiency reasons. These primitives, however, are implemented on most multiprocessor architectures, and re-targeting them to a different architecture is trivial.

### 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.

### 3.1.1 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 generic C++ object. In Factory, this abstraction is implemented with the `work_unit` class, and specific chunks of a computation are consequently represented as objects of the class. Table 3.1 outlines the user-defined member functions of the `work_unit` class.

Member Function	Parameters	
<code>work_init()</code>	<i>purpose</i>	Initialize a newly created <code>work_unit</code> .
	<i>member function parameters</i>	Variables to initialize all members of the work unit class. The last parameter must be a pointer to the parent <code>work_unit</code> .
<code>work()</code>	<i>purpose</i>	Definition of work that <code>work_unit</code> will perform.
	<i>member function parameters</i>	None.

**Table 3.1:** Member functions defined by the programmer in a `work_unit` class.

The member function `work()` defines 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.

The `work_init()` member function serves as the initializer of a newly created work unit. It can be used by the programmer as a means of providing the parameters required by the computation routine. This approach facilitates implicit type checking of work unit parameters at compile-time.

### 3.1.2 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. This hierarchy is depicted in Figure 3.1.

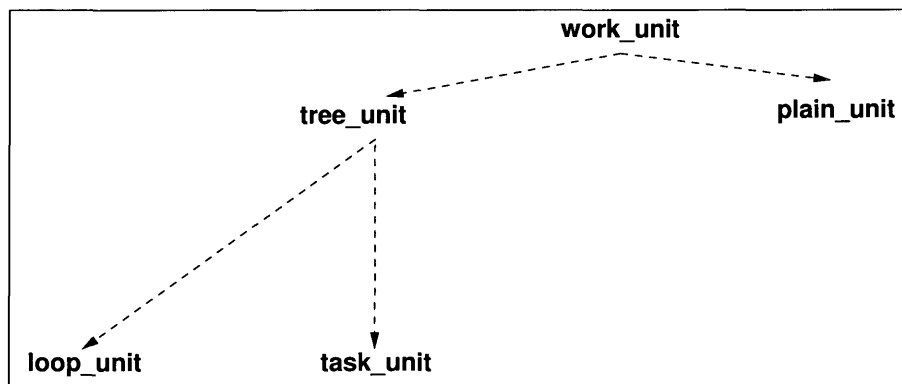


Figure 3.1: The work inheritance hierarchy.

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, which is also not directly available to programmers, 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 run-time, 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 internally the required support and functionality for the efficient execution of the specific type of parallel computation, in a way transparent to the programmer.

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. In this case, the functionality offered by `tree_unit` and its subclasses is not necessary.

The hierarchy structure facilitates the addition of new types of work, or the refinement of existing types, without interfering with unrelated types. Moreover, programmers may use the multiple inheritance features of C++ in order to define classes that combine the characteristics of application-internal classes and classes of the Factory work unit hierarchy.

### 3.1.3 Work Execution

All the interaction of applications with the Factory runtime occurs through an object of the `factory` class<sup>1</sup>. While `work_unit` classes are used to express the parallel algorithms, the `factory` class provides the necessary functionality for their creation, management and execution. Table 3.2 summarizes the member functions of the `factory` class exported to the programmer.

The class defines member functions for starting and stopping kernel threads (which are used as execution vehicles), creating and scheduling work units, and synchronizing work

---

<sup>1</sup>Throughout the paper we use the notation `Factory` to refer to the multithreading substrate and `factory` to refer to the class.

Member Function	Parameters	
object construction	<i>purpose</i>	Construct a new <code>factory</code> object.
	<i>member function parameters</i>	$n_{thr}$ : Number of execution contexts to use. May be omitted. LOGICAL, PHYSICAL: Use one execution context per execution context or per physical processor respectively. LIFO_STEAL, LIFO_LOCAL, FIFO_STEAL, FIFO_LOCAL, LIFO_STEAL_SMT, FIFO_STEAL_SMT: Choose between different scheduling algorithms; execute work units in LIFO/FIFO order; activate work stealing or exclusively check local queue; apply SMT-conscious work stealing.
	<i>template parameter</i>	<code>mixed_work</code> in the case of heterogenous work, or the user-defined name of the work unit class in the case of homogenous work.
<code>spawn()</code>	<i>purpose</i>	Spawn a new <code>task_unit</code> .
	<i>member function parameters</i>	Parameters the task unit expects, as defined in the <code>work_init()</code> member function for the specific task unit class.
	<i>template parameter</i>	The name of the task unit class being spawned if the <code>task_unit</code> is to execute heterogenous work; none for homogenous work.
<code>spawn_for()</code>	<i>purpose</i>	Spawn a new <code>loop_unit</code> .
	<i>member function parameters</i>	The first two parameters specify the bounds of the loop, the rest are the parameters the loop unit expects, as defined in the <code>work_init()</code> member function for the specific loop unit class.
	<i>template parameter</i>	The name of the loop unit class being spawned if the <code>loop_unit</code> is to execute heterogenous work, none for homogenous work
<code>start_working()</code>	<i>purpose</i>	Start the execution vehicles (kernel threads).
	<i>member function parameters</i>	None.
	<i>template parameter</i>	None.
<code>stop_working()</code>	<i>purpose</i>	Stop the execution vehicles (kernel threads).
	<i>member function parameters</i>	None.
	<i>template parameter</i>	None.
<code>root_barrier()</code>	<i>purpose</i>	Wait until the root <code>work_unit</code> and all its children have completed execution.
	<i>member function parameters</i>	The <code>work_unit</code> that is the root of the computation.
	<i>template parameters</i>	None.
<code>child_barrier()</code>	<i>purpose</i>	Wait until all children of this <code>work_unit</code> have completed execution.
	<i>member function parameters</i>	The <code>work_unit</code> to wait upon.
	<i>template parameters</i>	None.

Table 3.2: Member functions of the factory class.

units. In Section 4 we describe the member functions in further detail and we demonstrate their proper use through a programming example.

## 3.2 Scheduling

Factory incorporates a generic, queue-based runtime system which can be used as the basis for the implementation of a multitude of scheduling algorithms. The current implemen-

tation is based on local, per execution context work queues. 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.

The internal queue hierarchy of Factory is implemented using non-blocking, lock-free FIFO and LIFO queue management algorithms [27]. Non-blocking, lock-free algorithms have been shown to outperform lock-based ones whenever there is high contention on a shared resource or the multiprogramming degree is higher than one. Our experimental results, presented in Section 5, indicate that non-blocking, lock-free implementations can also be beneficial under moderate contention, when the contending threads are executed on the execution contexts of the same physical processor.

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. As a measure for the implicit preservation of locality, newly generated work is added to the local queue of the thread that spawned it. 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 architecture hierarchy. For SMT- and CMP-based multiprocessors this translates to favoring work stealing between threads that run on the execution contexts of

the same physical processor, rather than across threads running on different physical CPUs.

### 3.3 Memory Management

The use of objects to represent work units necessitates the frequent creation and destruction of many small objects over the execution of an application. These objects have a relatively short average lifespan and the frequent allocation and deallocation of such objects can become a severe bottleneck. In order to alleviate this problem and enhance its scalability, Factory integrates a customized memory manager. User-defined work unit objects are allocated through a slab allocator [9] which is capable of managing objects of varying sizes. The allocator can satisfy simultaneous requests for multiple types of objects, by multiple threads. The architecture of the slab allocator is depicted in Figure 3.2.

The main focus during the design of the slab allocator has been the support of simultaneous memory allocations and deallocations, by multiple threads, and the elimination of memory management-related contention. For each object type the slab allocator handles, there is a private, per thread list of slabs. Upon an allocation request, the slab allocator identifies the appropriate group of slabs and accesses the slab list associated with the requesting thread. Slabs can be in one of the following three states: *all free*, *partially free* or *full*. Whenever a slab becomes *full*, it is moved to the end of the slab list. This practice results in the first slab in the list having a free slot in the vast majority of the cases. Each slab with free slots maintains a pointer to one. This slot is used to satisfy the memory request. Afterwards, the slab is searched linearly to determine the position of the next free

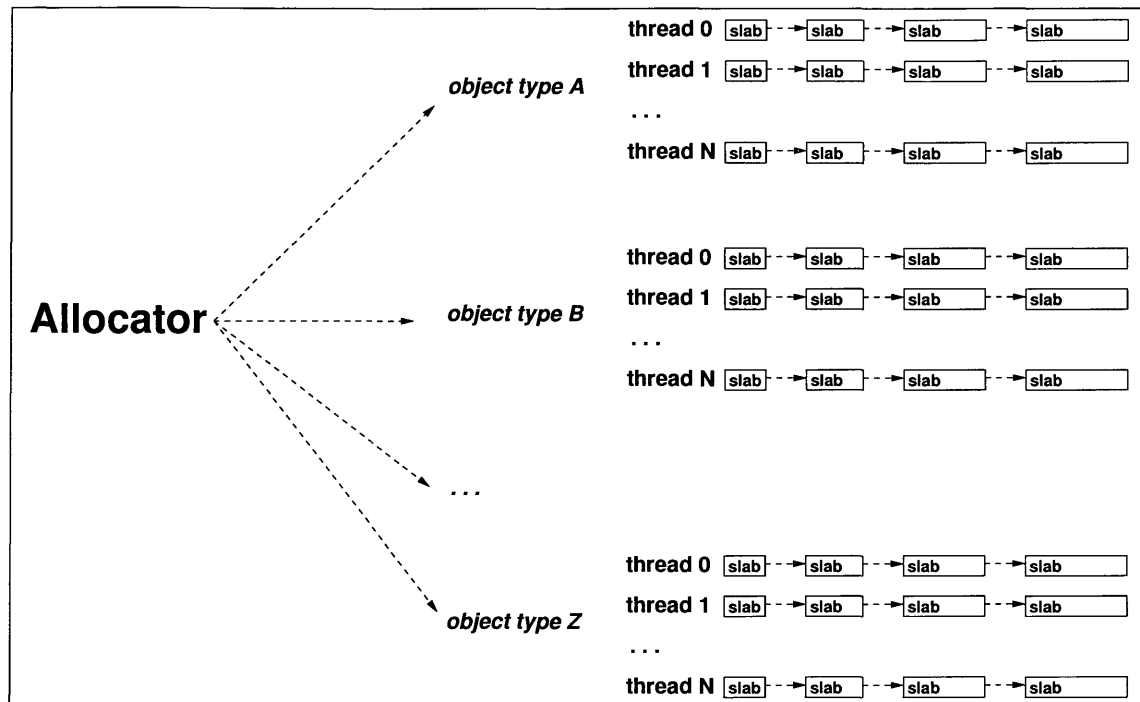


Figure 3.2: Slab allocator internal organization.

slot. If, however, the slab becomes full, its status is set to *full* and, on the next access, it will be moved to the end of the list.

If all slabs corresponding to the specific object type and thread are full, then a new slab is allocated. We progressively grow slab sizes, in order to reduce the number of slab allocations for applications with high memory requirements and low memory recycling. However, a high slab growth rate would soon cause requests of very large memory segments, increasing unnecessarily the virtual memory footprint of the application. On the other hand, a low slab growth rate might result in too many memory requests, which would be satisfied sequentially by the operating system, causing the bottleneck the slab allocator tries to eliminate. We have heuristically identified a growth rate of 1.25 to be a sweet-spot in this

tradeoff.

Currently, the slab allocator has no mechanisms for balancing or migrating slabs across processors because none of the applications we used to test Factory needed such functionality. All of the applications had uniform memory requirements across all threads for the duration of the program.

We refer to the mechanisms that Factory uses internally to handle all work unit allocations and deallocations through the slab allocator as the *managed* approach to memory management.

Although Factory uses an optimized internal memory allocator, it is possible to further enhance the efficiency of memory management by taking into account application-specific characteristics. Some applications have a particular property that can be used to entirely circumvent the need for Factory to allocate and manage work units. In general, a work unit is an abstraction of some sort of computation. The formal representation of this abstraction is the class definition itself. It is often the case that there is a 1-to-1 correspondence between work units and application data structures they operate on. Formally, this property holds if each work unit instance represents a computation applied on one and only one instance of a data structure of the application. The allocation and deallocation of such data structures is already explicitly managed by the application, thus nullifying the memory management overhead in Factory, should data structures be directly associated with work units.

Merging work units and application-specific data structures is possible through multiple inheritance, which is necessary if the target application data structure is already a part of an application-internal inheritance hierarchy. This approach combines the computation with

```

class Triangle: public TriangleQuality, public task_unit {
private:
    // Not shown: Triangle related private data members and functions

public:
    // Not shown: Triangle related public member functions
    void work_init(Cavity* _cavity, double* _xy, double _area, double _angle, Triangle* parent);
    void work();
};

```

**Figure 3.3:** An example of unmanaged work using the Triangle data structure from PCDM.

the data structure itself; there is no longer a separate class that represents just the computation abstraction. An example from the PCDM [3] application evaluated in Section 5.2 is presented in Figure 3.3. We refer to this alternative method of memory management as the *unmanaged* approach and we exploit it in Factory to improve performance in fine-grain parallel codes with very large numbers of work units. In the *unmanaged* approach, the application programmer is responsible for initializing the work unit, managing its allocation and deallocation, and merging work unit code with application data structure code. Although the *unmanaged* approach is certainly more intrusive than the *managed* one, the significant performance benefits it offers in some applications outweigh its complexity.

### 3.4 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 work units, i.e., work units it has spawned and have not yet finished their execution. When the work chunk associated with a work unit is executed, the parent of the work unit is notified, by updating its *children* counter. As a result, a dependence tree is dynamically formed and maintained at run-time. 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 that are currently executing or have executed in the past, but have to wait for the termination of their children before terminating themselves. Task-parallel programs tend to form deep dependence trees, while data-parallel, loop-based codes form shallow trees, as shown in Figure 3.4. In the diagram, work units are represented as nodes in a tree. Work units are dependent on their children and all subtrees. Sibling work units, however, are independent and can execute in parallel. Hence, the parallelism in task based work is limited by how wide the tree is at any given level, which is the same as how many tasks are spawned by each task. Loop based work is as parallelizable as the number of execution vehicles, but the profitability of parallelizing loop based work depends on the size of the loop.

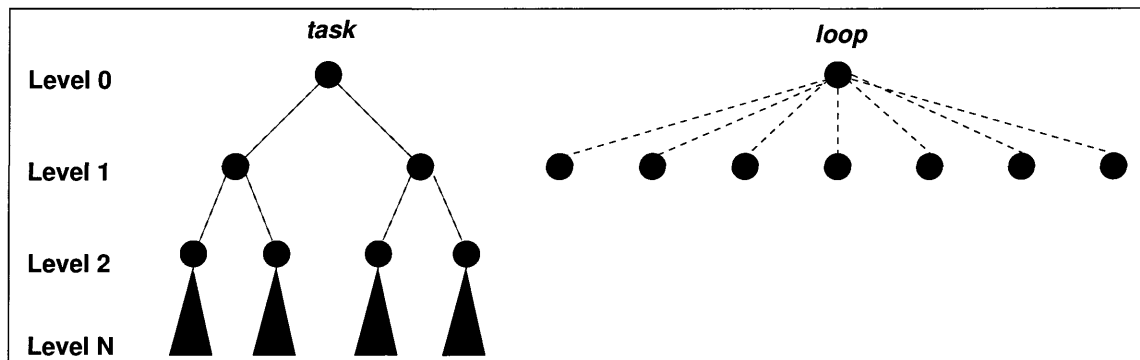


Figure 3.4: Example work unit dependence tree.

Correct order of execution is enforced through Factory barriers, which operate on a particular work unit. Barriers come in two versions: 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 at the root of the subtree have terminated (`root_barrier()` member function).



Whenever a not-yet-achieved barrier prevents further execution of a work unit, the corresponding execution vehicle is not blocked. Factory code implementing the barrier invokes the user-level scheduling algorithm instead, and the execution vehicle starts executing other work units. When the dependencies of the blocked work unit are satisfied, then the barrier is achieved and the work unit resumes execution.

## Chapter 4

# Programming Examples

This chapter illustrates, through detailed programming examples, how to code parallel programs using Factory. The first example uses task-based parallelism to introduce how parallelism is defined and how the programmer interacts with the Factory runtime. The second example introduces how loop parallelism is achieved with Factory. The third and final example uses concepts from the two previous examples to code a multiparadigm and multilevel parallel algorithm. Finally, we compare programming with Factory to other multithreaded programming models.

### 4.1 Task Parallelism: Fibonacci

This section uses the Fibonacci sequence to demonstrate how task parallelism is achieved using Factory.

The `factory` object `fibber` is the programmer's handle to interact with the Factory

```
factory<Fibonacci> fibber(8, LOGICAL, FIFO_STEAL);
```

**Figure 4.1:** Declaration of a Factory object for the Fibonacci sequence.

runtime system, as illustrated in Figure 4.1. The template parameter to the `factory` object is the class `Fibonacci`, which is the name of Fibonacci work unit. The constructor parameters specify how many execution contexts to use, Factory's view of the processors and what scheduling algorithm to apply. If the number of execution contexts is not provided as a parameter, Factory automatically detects the number of execution contexts available in the system and uses all of them. The second parameter controls how Factory views processors. The constant `LOGICAL` means that all hardware execution contexts of the system should be viewed as independent processors. For example, if the system is a 4-way SMP with SMT processors and each SMT processor has two hardware execution contexts, then `LOGICAL` implies that Factory should view the system as 8 identical processors, each with its own scheduling queue. If the constant `PHYSICAL` is used instead, Factory will view the system as 4 physical processors, each with its own local scheduling queue. This distinction is necessary because some codes that were not designed to run on multiple execution contexts per physical CPU may experience slowdown due to shared resource contention. The scheduling algorithm specified, namely, `LIFO_STEAL`, enforces a LIFO execution order. Each processor first queries its local queue for work and idle processors perform work stealing from remote queues. Factory also provides algorithms in which processors execute work in FIFO order, with or without work stealing, as well as SMT/CMP aware work stealing algorithms which were discussed in Section 3.2. The scheduling algorithm can be changed at runtime, however all work units managed by the same Factory object use the same scheduling algorithm, which is a reasonable choice for most practical purposes.

```
fibber.start_working();
Fibonacci* root = fibber.spawn(N, &result (Fibonacci*)NULL);
fibber.root_barrier(root);
fibber.stop_working();
```

**Figure 4.2:** Starting the Fibonacci sequence.

Figure 4.2 shows how the parallel computation is started. The `start_working()` member function forks the kernel threads that will be used as execution vehicles to run work units. The `spawn()` member function initiates the computation by creating the root `task_unit` of the program. Factory handles the creation and allocation of work units, and passes the parameters of `spawn()` on to the newly created work unit objects. The last parameter to `spawn()` is a pointer to the parent work unit, i.e., the work unit that spawned the newly created one. Since the last parameter is `NULL` in this case, this spawned work unit has no parent and becomes the root of a new dependence tree. The `spawn()` member function returns a handle to the newly created work unit which is used as a parameter to the `root_barrier()` member function. Upon the call to `root_barrier()`, the main thread of control for the program is blocked until the spawned computation has completed. The underlying kernel thread, however, is not blocked. It can still participate in the execution of other work units. Finally, `stop_working()` is invoked to join and destroy all kernel threads of the program that Factory spawned and used as execution vehicles, as they are no longer needed.

The Fibonacci work unit class, `Fibonacci`, is defined in Figure 4.3. Since the work unit is task based, it inherits from the `task_unit` class. The algorithm itself is defined in the `work()` function. Inside `work()`, the calls to `spawn()` create new `Fibonacci` work units which complete the recursion. Each of these work units is passed the `this` pointer to indicate that the spawned work units are children of the current work unit. This information is used

```
class Fibonacci: public task_unit {
private:
    int n;
    int* result;

public:
    void work_init(int _n, int* _result, Fibonacci* parent) {
        task_unit::work_init(parent);
        n = _n;
        result = _result;
    }

    void work() {
        if (n < 2) {
            *result = n;
        } else {
            int part1, part2;

            fibber.spawn(n - 1, &part1, this);
            fibber.spawn(n - 2, &part2, this);
            fibber.child_barrier(this);

            *result = part1 + part2;
        }
    }
};
```

Figure 4.3: Definition of a Fibonacci work unit.

internally in Factory for the construction and maintenance of the work unit dependence tree. The call to `child_barrier()` ensures the correct order of execution; when it is passed the `this` pointer, as in this case, it waits until all of this work unit's children have finished execution.

The `work_init()` member function is used by the Factory runtime to initialize each newly constructed work unit. The final parameter must be a pointer to the parent work unit, and a call must be made to the base class' `work_init()` member function. Note that the parameters passed to `spawn()` match the type signature of the `work_init` member function. This signature match allows the compiler to enforce type safety with each call to `spawn()`.

## 4.2 Loop Parallelism: Naive Matrix Multiplication

To demonstrate loop based parallelism, we present a naive matrix multiplication. Instead of presenting a complete example as we did in the previous section, we highlight the differences between loop and task parallelism.

```
naive_matmul root;
multiplier.spawn_for(0, N, STATIC, one, two, result, &root);
multiplier.root_barrier(&root);
```

**Figure 4.4:** Starting the naive matrix multiplication.

Figure 4.4 shows how loop parallel work is started. The `spawn_for()` member function is used for loop work, and it expects three parameters that `spawn()` does not. The first two parameters are the boundaries of the whole iteration space of the loop. The third parameter identifies the algorithm to be used for scheduling loop iterations to execution vehicles. Factory currently offers only a `STATIC` scheduling policy, however the extension with other policies, such as those offered by OpenMP, is straightforward. Factory uses this scheduling algorithm to divide the iteration space in chunks of work and assign chunks to execution vehicles.

```
class Naive_Matmul: public loop_unit {
private:
double* one, two, result;

public:
// Not shown: work unit initialization
void work() {
for (int i = loop_start; i < loop_stop; i += loop_step)
for (int j = 0; j < N; ++j)
for (int k = 0; k < N; ++k)
result[i * N + j] += one[i * N + k] * two[k * N + j];
}
};
```

**Figure 4.5:** Definition of a naive matrix multiplication work unit.

The naive matrix multiplication work unit, `Naive_Matmul`, is defined in Figure 4.5.

Because it is loop based work, it derives from `loop_unit`. This class provides three constants for parallelizing the loop: `loop_start`, `loop_stop`, and `loop_step`. These values are set by the Factory runtime and depend on the loop scheduling policy.

### 4.3 Multiparadigm Parallelism: Strassen Matrix Multiplication

This section presents a multiparadigm and multilevel parallel algorithm. We are specifically using Strassen's matrix multiplication [32] as an example algorithm. Strassen matrix multiplication exposes two levels of parallelism: task-level parallelism, via recursive calls for the calculation of intermediate matrix products, and loop-level parallelism within the calculation of each matrix product.

```
factory<mixed_work> matmul(8, LOGICAL, LIFO_STEAL);
```

**Figure 4.6:** Declaration of a Factory object for heterogenous work.

The declaration of the `factory` object is slightly different in the case of heterogenous work, as shown in Figure 4.6. The template parameter to the `factory` object is now the predefined class `mixed_work`, which indicates that Factory will manage multiple types of work units. Before, the presence of a programmer defined work unit told the Factory runtime system to manage only one kind of work unit.

The definition of the **Strassen** work unit is depicted in Figure 4.7. The algorithm has been parallelized at two levels: each recursive call is executed in parallel, and the matrix additions and subtractions at each level of recursion are also parallelized. The parallel work

```

class Strassen: public task_unit {
private:
    double* a, b, c;
    int matrix_size, a_width, b_width, c_width;

public:
    // Not shown: work unit initialization ...
    void work() {
        if (matrix_size <= BASE_CASE_SIZE)
            sequential_matmul(a, b, c, matrix_size, a_width, b_width, c_width);
        else {
            const int size = matrix_size >> 1;
            // Not shown: quadrant declarations and allocations ...
            matmul.spawn_for<Before_Matrix_Add>( 0, size, STATIC,
                                                a11, a12, a21, a22,
                                                b11, b12, b21, b22,
                                                s1, s2, s3, s4,
                                                t1, t2, t3, t4,
                                                size, a_width, b_width, c_width, this);

            matmul.child_barrier(this);

            matmul.spawn<Strassen>(a11, b11, p1, size, a_width, b_width, size, this);
            matmul.spawn<Strassen>(a12, b21, p2, size, a_width, b_width, size, this);
            matmul.spawn<Strassen>(s1, t1, p3, size, size, size, size, this);
            matmul.spawn<Strassen>(s2, t2, p4, size, size, size, size, this);
            matmul.spawn<Strassen>(s3, t3, p5, size, size, size, size, this);
            matmul.spawn<Strassen>(s4, b22, p6, size, size, b_width, size, this);
            matmul.spawn<Strassen>(a22, t4, p7, size, a_width, size, size, this);
            matmul.child_barrier(this);

            matmul.spawn_for<After_Matrix_Add>( 0, size, STATIC,
                                                c11, c12, c21, c22,
                                                p1, p2, p3, p4, p5, p6, p7,
                                                u2, u3, u6,
                                                size, c_width, this);

            matmul.child_barrier(this);
            // Not shown: quadrant deallocations ...
        }
    }
};

```

Figure 4.7: Definition of a Strassen work unit.

units of the recursive calls inherit from the `task_unit` class. The computation itself is defined in the `work()` function. To synchronize nested and recursive parallel work, the calls to `spawn()` within a work unit are passed the `this` pointer to indicate that spawned work units are children of the current work unit.

The factory member function `spawn_for()` is called to spawn loop-parallel work. The template parameters to `spawn()` and `spawn_for()` specify the exact type of work unit to be spawned (`Strassen`, `Before_Matrix_Add`, `After_Matrix_Add`). To guarantee the correct



execution order, work units are synchronized in three cases with calls to `child_barrier()`. Whenever `child_barrier()` is invoked, the parent work unit waits for the termination of all its children.

```

class Before_Matrix_Add: public loop_unit {
private:
    double* a11, a12, a21, a22,
           b11, b12, b21, b22,
           s1, s2, s3, s4,
           t1, t2, t3, t4;
    int matrix_size, a_width, b_width, c_width;

public:
    // Not shown: work unit initialization...
    void work() {
        for (int i = loop_start; i < loop_stop; i += loop_step)
            for (int j = 0; j < matrix_size; ++j) {
                s1[i * matrix_size + j] = a21[i * a_width + j] + a22[i * a_width + j];
                t1[i * matrix_size + j] = b12[i * b_width + j] - b11[i * b_width + j];
                s3[i * matrix_size + j] = a11[i * a_width + j] - a21[i * a_width + j];
                t3[i * matrix_size + j] = b22[i * b_width + j] - b12[i * b_width + j];
                s2[i * matrix_size + j] = s1[i * matrix_size + j] - a11[i * a_width + j];
                t2[i * matrix_size + j] = b22[i * b_width + j] - t1[i * matrix_size + j];
                s4[i * matrix_size + j] = a12[i * a_width + j] - s2[i * matrix_size + j];
                t4[i * matrix_size + j] = b21[i * b_width + j] - t2[i * matrix_size + j];
            }
    }
};

```

Figure 4.8: Definition of the `Before_Matrix_Add` class.

Figure 4.8 depicts the implementation of the class `Before_Matrix_Add`, which is a work unit that derives from `loop_unit`. The programmer defines the matrix arithmetic that is part of the Strassen algorithm in the `work()` member function. The bounds `loop_start` and `loop_end` of the parallelized loop, i.e. the outermost one, as well as the loop stepping `loop_step`, are transparently set by the effective loop scheduling policy.

## 4.4 Programmability Comparisons

Our programming examples showed how to use Factory to define task-based parallelism, loop-based parallelism, and multiparadigm parallelism which uses both task and loop paral-

lelism. Our examples demonstrate that Factory can express potentially complicated forms of parallelism in a clean and concise manner. In this section we compare the complexity of the Factory implementations of these algorithms with the Cilk and OpenMP versions. For each implementation, porting the code to use Cilk involves the definition of `cilk` procedures which have an 1-to-1 correspondence with work unit classes. However, since Cilk does not offer explicit looping mechanisms, the programmer has to undertake the task of the management and distribution of the loop iteration space to threads. The corresponding OpenMP implementation expresses task work units as a recursive function. Each recursive call is preceded by a work distribution directive for task-parallel work. Loop-parallel work is expressed inline, and is once again preceded by the appropriate work distribution directives.

	Factory	Cilk	OpenMP	
			task	loop
<b>Fibonacci</b>	55 lines	32 lines	49 lines	–
<b>Naive Matrix Multiplication</b>	57 lines	–	–	29 lines
<b>Single-level Strassen</b>	634 lines	601 lines	612 lines	–
<b>Multi-level Strassen</b>	733 lines	–	614 lines	

**Table 4.1:** A comparison of the number of lines of code in the example programs using Factory, Cilk, and OpenMP.

Table 4.1 summarizes the lines of code required for programming four different parallel programs using the same programming style conventions in Factory, Cilk and OpenMP. Fibonacci, Naive Matrix Multiplication and Multi-level Strassen are presented earlier in this chapter. Single-level Strassen is an implementation of the Strassen algorithm that only parallelizes the recursive calls. In general, Cilk has the most concise expression of task parallelism, and OpenMP has the most concise expression of loop parallelism. However, directly using Factory as a multithreading substrate involves, in general, comparable pro-

programming effort as programming with Cilk or OpenMP. At the same time, Factory does not require algorithmic modifications as Cilk often does. Moreover, Factory does not need compiler support and is thus independent of particular compiler implementations. As a result, it is significantly more portable and can be easily integrated into existing projects. Implementing these programs using only POSIX threads requires programmers to design and implement their own system for expressing and executing task- and loop-parallelism. Consequently, such programs would be significantly longer and the expression of parallelism would likely be problem specific.

## Chapter 5

# Performance Evaluation

We have experimentally evaluated the performance of Factory on an SMT-based multi-processor. The use of such a multilevel parallel architecture allows us to experiment with different options for exploiting nested parallelism with Factory. It also facilitates the assessment of the efficiency of alternative scheduling policies offered by Factory, which take into account the characteristics of the two disjoint levels of available parallelism, namely within the execution contexts of each physical processor and within the different physical processors of the same SMP. We compare Factory against other popular parallel programming models, namely OpenMP, Cilk, and manual parallelization using POSIX threads.

Table 5.1 outlines the hardware and software configuration of our experimental platform. The Intel Hyper-Threaded architecture follows the SMT organization [37]: a relatively wide superscalar processor core executes mixes of instructions originating from two threads of control. The Hyper-Threaded architecture shares most of the resources of the processor between the threads. In particular, the caches, the data TLB, and all execution units are

Processor	Intel Xeon with Hyper-Threaded Technology
Frequency	2 GHz
Cache	8KB, 4-way associative DL1, 12KB instruction IL1 512KB, 8-way associative unified L2 1MB, 8-way associative unified L3
TLB	64-entry fully associative DTLB 2x64-entry fully associative ITLB
Memory	2GB DRAM
Operating System	SUSE Linux 9.2, 2.6.8-24.13-smp kernel
Compiler	Intel Compiler for 32-bit applications, Version 8.1

**Table 5.1:** Hardware and software configuration of the experimental platform used to evaluate Factory.

shared and are made available—on demand—to any thread. The processor maintains a private set of per thread architectural registers, as well as a private instruction TLB for each thread.

We experimented using both microbenchmarks and parallel applications. The evaluation with microbenchmarks assesses the overhead for managing parallelism, or equivalently, identifies the minimum granularity of exploitable parallelism by each of the target parallel programming models. We also used microbenchmarks in order to evaluate the performance of the slab allocator integrated with Factory.

Experiments with real applications also compare Factory against the aforementioned parallel programming models. We focus on both task- and loop-parallel applications. Factory and OpenMP natively support task- and loop-level parallelism through their respective APIs<sup>1</sup>. Cilk and POSIX threads do not offer explicit support for both forms of parallelism, however it is always possible to express task- and loop-level parallel algorithms at the cost

---

<sup>1</sup>In fact the support for task-parallel codes by the official OpenMP standard (i.e. through `SECTIONS` directives) is still immature. However, Intel has introduced OpenMP extensions for the support of task queues [41].

of additional—and often significant—overhead for the programmer.

We have also used PCDM [3], a parallel mesh generation application, to assess the effectiveness of the unmanaged approach to memory allocation offered by Factory. PCDM is particularly demanding in terms of efficient memory management.

The final experiment compares the effectiveness of thread binding schemes using one of the Factory implementations of an application. Our results indicate that Factory’s performance does not depend on thread placement.

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, so it is not plotted on the graphs. The only exception is the experiment evaluating the performance of the memory management. In this case, the 95% confidence interval boundaries are reported on the corresponding graph.

## 5.1 Minimum Granularity of Exploitable Parallelism

The minimum granularity of parallelism that can be effectively exploited by any multi-threaded substrate is directly related to the degree of overhead associated with the creation and management of parallel jobs. The minimum exploitable granularity is partially dependent on architecture-specific parameters, such as the context-switch overhead. Multi-threading substrates introduce additional overheads for the creation and destruction of the data structures used to represent chunks of parallel work, for the execution of scheduling algorithms, and for the synchronization between jobs. Thus, it is important to investigate

whether the implementation of a multithreading library maintains such overheads as close as possible to the limits imposed by the architecture.

More formally stated, the parallel execution time ( $T_{//}$ ) of a fully parallelizable job can be expressed as:

$$T_{//} = \frac{T_{seq}}{n_{thr}} + Overhead(n_{thr}), \quad (5.1)$$

where  $T_{seq}$  stands for the sequential execution time of the job,  $n_{thr}$  for the number of threads used for the parallel execution and  $Overhead(n_{thr})$  for the overheads associated with the exploitation of parallelism (as a function of  $n_{thr}$ ). The minimum granularity of exploitable parallelism ( $T_{gran}(n_{thr})$ ) is the  $T_{seq}$  for which:

$$T_{seq} = \frac{T_{seq}}{n_{thr}} + Overhead(n_{thr}). \quad (5.2)$$

Tasks with sequential execution time less than  $T_{gran}(n_{thr})$  can not be executed efficiently in parallel with  $n_{thr}$  threads, since the overheads outweigh the benefits of parallel execution. It is obvious that as the overhead introduced by the multithreading substrate increases, the minimum granularity of exploitable parallelism also increases accordingly.

The experiment for the evaluation of  $T_{gran}(n_{thr})$  is organized as follows. The parallelized job consists of a variable number of `pause` machine 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 with  $n_{thr}$  threads. 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

Hyper-Threaded processor. The minimum granularity may be coarser for realistic codes<sup>2</sup>, however this does not affect the comparison of Factory against the other multithreading substrates. In general, it is reasonable to expect that the minimum exploitable granularity when threads are running on the same processor is different than when threads are running on different processors, because of the implications of resource sharing inside the processor.

As discussed earlier, 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. In the 2 threads experiments, threads are bound to either 2 different physical CPUs, or to the 2 execution contexts of a single CPU. Similarly, 4 threads can be executed on either 2 or 4 physical CPUs. Finally, the experiments using 8 threads are executed on 4 physical processors, with 2 execution contexts active on each processor. The different binding schemes allow the evaluation of both intra- and inter-processor parallelism overheads.

	2 Threads		4 Threads		8 Threads
	1 CPU	2 CPUs	2 CPUs	4 CPUs	4 CPUs
<b>Factory</b>	6.2 $\mu$ sec	6.2 $\mu$ sec	10 $\mu$ sec	10 $\mu$ sec	26 $\mu$ sec
<b>Cilk</b>	121 $\mu$ sec	81 $\mu$ sec	153 $\mu$ sec	153 $\mu$ sec	222 $\mu$ sec
<b>OpenMP task</b>	20 $\mu$ sec	20 $\mu$ sec	26 $\mu$ sec	24 $\mu$ sec	202 $\mu$ sec
<b>OpenMP loop</b>	10 $\mu$ sec	6.2 $\mu$ sec	6.2 $\mu$ sec	4.2 $\mu$ sec	68 $\mu$ sec

**Table 5.2:** Comparison of the minimum granularity of effectively exploitable parallelism.

Table 5.2 summarizes the measured minimum exploitable granularity of Factory and the other multithreading systems. We compare Factory against Cilk, which supports only strict

---

<sup>2</sup>The minimum granularity in this case will also depend on the instruction mix executed by the different threads on the same physical processor. The two execution contexts on a Hyper-Threaded processor share functional units. If the instruction mix between the two contexts causes conflicts in the shared functional units, then thread execution is effectively serialized.



multithreaded computations with recursive task parallelism, and OpenMP. For the latter, we distinguish between the minimum granularity that can be exploited by the loop execution mechanism and the one exploitable by the task execution mechanism. OpenMP runtime libraries use different mechanisms for the two types of parallelism. We have evaluated the minimum granularity of task parallelism using Intel compiler's workqueue extensions to OpenMP [29, 41]. Factory uses the same mechanisms for creating parallel work 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. Table 5.2 does not include experimental results for the minimum exploitable granularity of applications parallelized directly with POSIX threads. POSIX threads are implemented on Linux directly on top of kernel threads, with an 1-to-1 correspondence between each POSIX and kernel thread. Thus, they incur excessive overhead if used directly for the parallelization of fine-grain computations. As a consequence, POSIX threads are typically used only as execution vehicles, combined with a user-level threads package or an application-specific work representation and management mechanism, such as application-level work queues.

Factory's minimum task granularity is finer than Intel's task queue implementation in OpenMP. Factory's granularity remains competitive with OpenMP's loop granularity as well. At the same time, Factory proves able to exploit significantly finer granularity than Cilk. Although the point where Cilk starts achieving 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 run time actually schedules multiple tasks to the same execution vehicle (kernel thread). Hence, Cilk requires a relatively large work load before multiple threads are used to execute it.

It should be pointed out that Intel’s implementation of loop- and task-level parallel execution is heavily optimized. Sophisticated compile-time techniques, such as multi-entry threading [36] are used. Multi-entry threading avoids generating separate modules and functions for loop and task bodies. The benefits of these compile-time optimizations are evident in the minimum granularities measured: the minimum exploitable granularity is actually reduced for the parallel execution with 2 and 4 threads. The fact that Factory performs comparably to this implementation without being supported by compile-time optimizations is indicative of its efficiency.

Both Cilk and OpenMP generally perform better when threads are spread to as many physical CPUs as possible. Factory overheads, on the other hand, are uncorrelated with thread placement. This property makes Factory a much more predictable multithreading substrate for deep, multilevel parallel systems.

## 5.2 Managed vs. Unmanaged Memory Allocation

The distinction between managed and unmanaged work allocation has been discussed in Section 3.3. In this section, we evaluate the performance gains unmanaged work allocation can offer.

PCDM (Parallel Constrained Delaunay Mesh Generation) [3] is a method for creating unstructured meshes in parallel, while guaranteeing the quality of the resulting mesh under geometric, qualitative criteria. The method is based on the Bowyer-Watson kernel [10, 39]. The algorithm first identifies an offending triangle which does not satisfy the qualitative

criteria. The triangle is deleted and a new point is inserted at the circumcenter of the offending triangle. The kernel then performs a cavity expansion; it detects the immediate or higher order neighbors of the offending triangle, whose circumcircles include the newly inserted point (*incircle* test). The triangles in the cavity of the offending triangle are also deleted. Finally, the area is retriangulated by connecting points at the boundary of the cavity with the newly inserted point. The cavity expansion accounts for almost 60% of the total execution time of PCDM and is similar to a breadth-first search of a graph. It can be executed in parallel, however it offers limited concurrency (2 on average). Each cavity expansion has an average duration of 4 to 6  $\mu$ sec on our experimental platform.

The main data structure of the algorithm is a graph of the triangles comprising the mesh. Nodes of this graph, which are triangles, are deleted during cavity expansions and new nodes are inserted during the retriangulations. Each Factory work unit corresponds to an *incircle* test for a specific triangle. Due to its extremely fine granularity, and due to the strict 1-to-1 relation between work units and triangles, PCDM is a good candidate for the evaluation of the benefits of unmanaged work in Factory. In the Factory implementation of PCDM, the triangle data structure inherits directly from a Factory `task_unit` class. Since the allocation and deallocation of triangles is already handled natively by PCDM, work unit creation and memory management inside Factory is no longer necessary.

	1 thread	2 threads
Managed	61.7sec	98.8sec
Unmanaged	57.7sec	89.9sec

**Table 5.3:** Comparing execution times of PCDM with managed and unmanaged work unit allocation.

Table 5.3 outlines the performance gains from merging the management of work units with that of application triangles in PCDM. The reported execution times were obtained by executing PCDM for an output problem size of 10 million triangles. We used either one or two Hyper-Threads on a single processor on our experimental platform. In this context the unmanaged approach provides a measurable performance benefit (a reduction in execution time ranging between 6.4% and 9.0%).

PCDM does not scale because of excessive scheduling and synchronization overheads. These problems become even more pronounced when the threads are executed on different physical processors. Hyper-Threading actually reduces overhead, by allowing synchronization operations to take advantage of the shared cache. The scalability problems of PCDM are endemic and can not be solved without better hardware mechanisms for creating, scheduling and synchronizing threads [3]. The experimental results reported here simply illustrate the potential of the unmanaged approach to work unit allocation in Factory.

### 5.3 Memory Management

The performance of any multithreading library is sensitive to the efficient management of its own data structures. Since work in Factory is represented by small objects and these objects are the dominant unit of memory allocations, we opted to implement an efficient user-level, small object, multithreaded allocator as discussed in Section 3. Each execution vehicle has its own list of slabs from which it allocates objects. Maintaining lists for each thread allows the allocator to satisfy simultaneous memory requests from multiple threads

and also implicitly promotes locality. The performance of our allocator versus the C++ `new / delete` operators is depicted in the diagram of Figure 5.1.

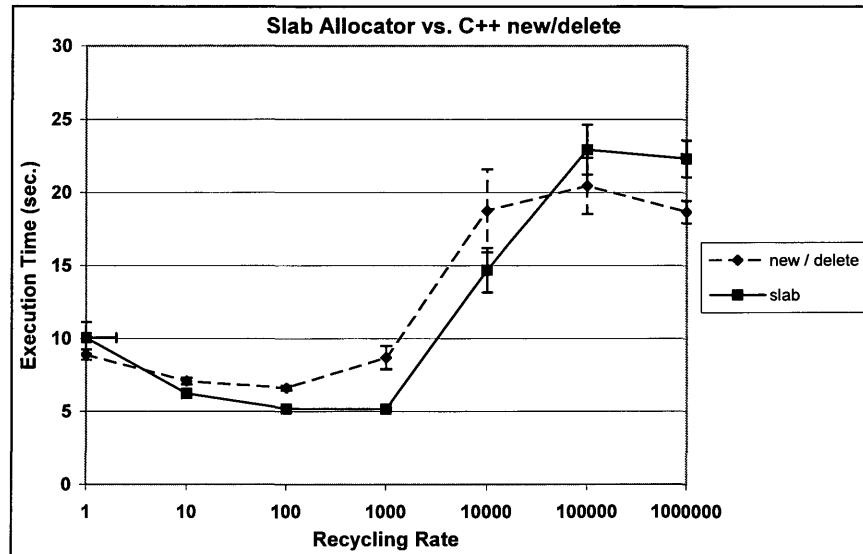


Figure 5.1: Comparison of the slab allocator with `new/delete`.

Each of the 8 threads participating in the experiment allocates  $10^7$  work units. The horizontal axis represents the period of work unit recycling, i.e., the number of consecutive work unit allocations before the first deallocation takes place. For example, an x-axis value of 10 indicates 10 work unit allocations followed by 10 deallocations. By varying this frequency, we can simulate different recycling rates that Factory might encounter in a real application. The results indicate that our allocator is consistently better suited for small object allocations among multiple threads when the recycling rate is between 10 and  $10^4$ . This range corresponds to task based codes with deep levels of recursion.

The improvement, in this range, over native memory allocation can be attributed to the fact that our memory allocator is designed to avoid contention during memory management

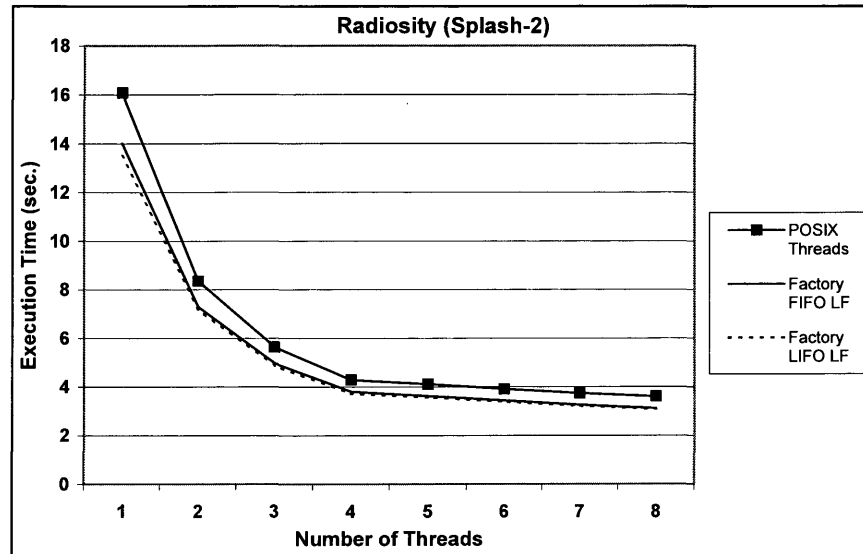
in the common case. Since each thread has access to its private list of slabs, it does not have to compete with other threads to satisfy a memory request. When objects, however, are recycled with a period higher than  $10^4$ , the average slab size tends to become relatively large. As a result, a significant amount of time may be spent identifying free objects inside the slab.

This experiment realistically simulates the pressure experienced by the memory manager during a Factory execution. Work units are, in most cases, deallocated by the same execution vehicle that initially allocated them. The only exception is when work units are migrated to different execution vehicles as a result of work stealing. However, the percentage of migrated work units is typically negligible compared with the total number of work units created by a program.

## 5.4 Factory vs. POSIX Threads: Splash-2 Radiosity

Radiosity is an application from the Splash-2 [40] benchmark suite. It computes the equilibrium distribution of light in a scene. It uses several pointer-based data structures and 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. As Radavić and Hagersten have already demonstrated [12], its performance is sensitive to the efficiency of synchronization mechanisms. Radiosity 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 5.2.



**Figure 5.2:** Comparison of the performance of Factory and POSIX Threads Radiosity implementations.

Factory consistently performs at least 13% faster than the POSIX Threads implementation, mainly due to its efficient, localized, fine-grain synchronization mechanisms. There is almost no performance improvement if more than 4 threads are used. This can be attributed to the fact that one Radiosity thread per physical CPU manages to effectively use almost all shared execution resources. However, the additional SMT contexts provide only marginal performance benefits.

We tested Factory using both LIFO and FIFO scheduling policies. In all cases, the internal queues have been implemented using lock-free algorithms. LIFO execution ordering yielded better performance due to temporal locality. Data shared between the parent and children work units are likely to be found in the processor cache if a LIFO ordering is

applied. The same trend has also been observed for the experiments presented in the following sections. As a result, in these sections we report only experimental results that have been attained using a LIFO execution ordering.

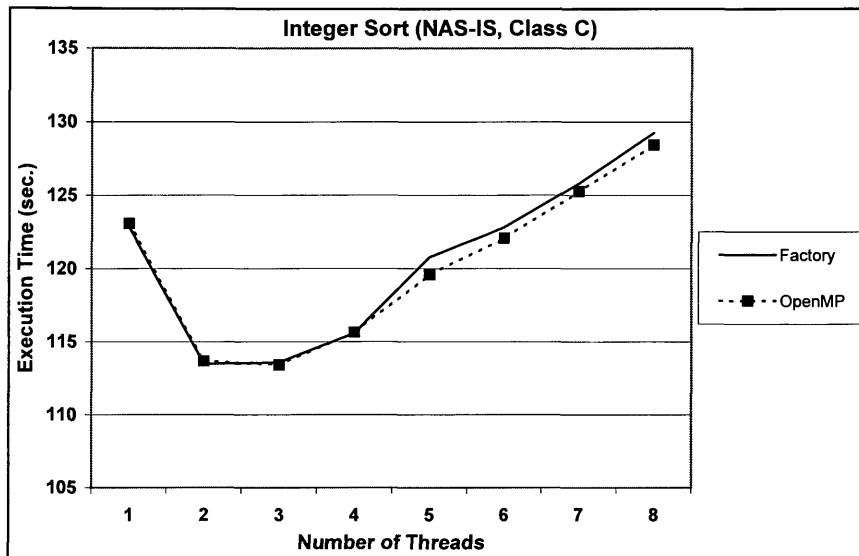
## 5.5 Factory vs. OpenMP : NAS IS

Integer Sort (IS) is part of the NAS [4] benchmark suite. We are using the OpenMP version of the 3.1 release of the benchmarks. The sorting method IS implements is often used in particle simulation codes. The application stresses integer execution units and data communication paths. The conversion of the application to the Factory programming model is straightforward. Each `omp parallel` for OpenMP work sharing construct is substituted by the definition of a `loop_unit` class and called with `spawn_for()`.

All experiments have been performed using the Class C problem size, which sorts  $2^{27}$  keys. The results are depicted in Figure 5.3.

Neither the OpenMP nor the Factory implementation of IS scales well on our platform. In fact, the use of more than three threads results in slowdown. Dell has already identified the performance problem of IS on Xeon-based PowerEdge servers [1, 26]. The source of the problem has been pinpointed to the saturation of the system bus. As mentioned previously, IS has high memory bandwidth requirements. Two IS threads are enough to saturate the bus that connects processors to the main memory. The addition of more threads has adverse effects for two reasons. First, it results to more conflicts on the system bus. Second, more than one thread shares the cache hierarchy on each processor, thus reducing the effective





**Figure 5.3:** Comparison of OpenMP and Factory implementations of the NAS IS (Class C) application.

cache size and resulting in more memory references being satisfied by main memory, through the system bus.

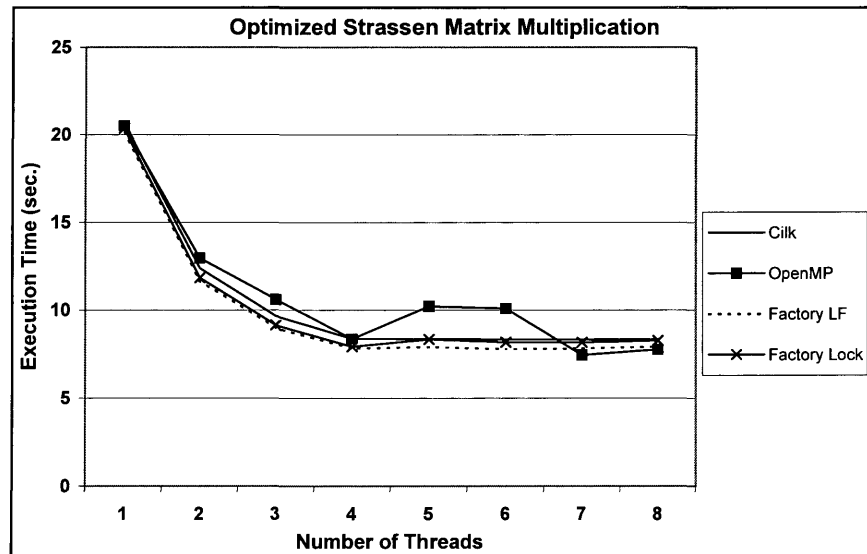
In any case, the Factory implementation always performs within 1% of the OpenMP version, despite the fact that Intel OpenMP compilers take advantage of OpenMP semantics to guide aggressive, compile-time optimizations.

## 5.6 Factory vs. Cilk and OpenMP: Single-level Parallel Strassen Matrix Multiplication

We have used an optimized, single-level 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 ex-

tensions for the support of task queues, which facilitate the implementation of task-parallel codes in OpenMP.

Once again, the conversion to the Factory programming model was straightforward. We replaced recursive Cilk functions by work unit classes (specifically, work units of type `task_unit`). The conversion to OpenMP was also simple: recursive calls to Cilk functions have just been preceded by OpenMP `task` directives.



**Figure 5.4:** Performance of Factory, Cilk, and OpenMP `taskq` for a single-level, parallel, Strassen matrix multiplication.

As shown in Figure 5.4, 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 [20].

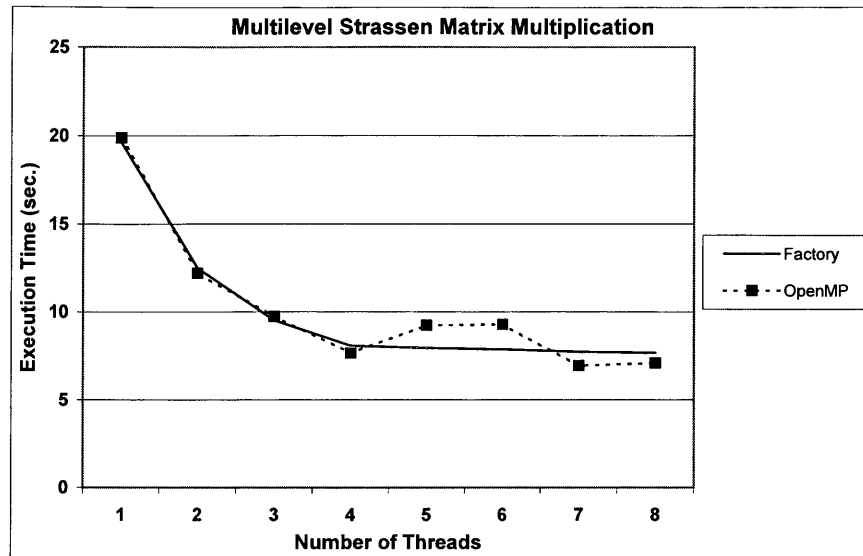
The Factory implementation that uses a lock-based queue implementation also suffers a performance degradation at 5 and 6 threads. However, the problem is solved if lock-free queues are used. In fact, the lock-free Factory implementation outperforms all others in all but 2 cases: OpenMP is more efficient than Factory when 7 or 8 threads are used.

Our experiments suggest that 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, i.e., when the runnable threads are more than the available processors [27]. The execution of more than one thread on the execution contexts of SMT processors often has similarities to multiprogrammed execution on a conventional SMP. If the shared processor resources can not satisfy the simultaneous requirements of all threads, the threads will eventually have to time-share the resources. As a result, SMT-based multiprocessors may prove more sensitive to the efficiency of synchronization mechanisms than conventional SMPs.

## 5.7 Factory vs. OpenMP: Multilevel Parallel Strassen Matrix Multiplication

In Chapter 4 we presented a multilevel parallel implementation of the Strassen algorithm with Factory. In this section we evaluate the performance of that implementation and we compare it to the corresponding OpenMP multilevel code. The experimental results are depicted in Figure 5.5.

The Factory implementation scales consistently up to 4 threads. When 5 or more threads



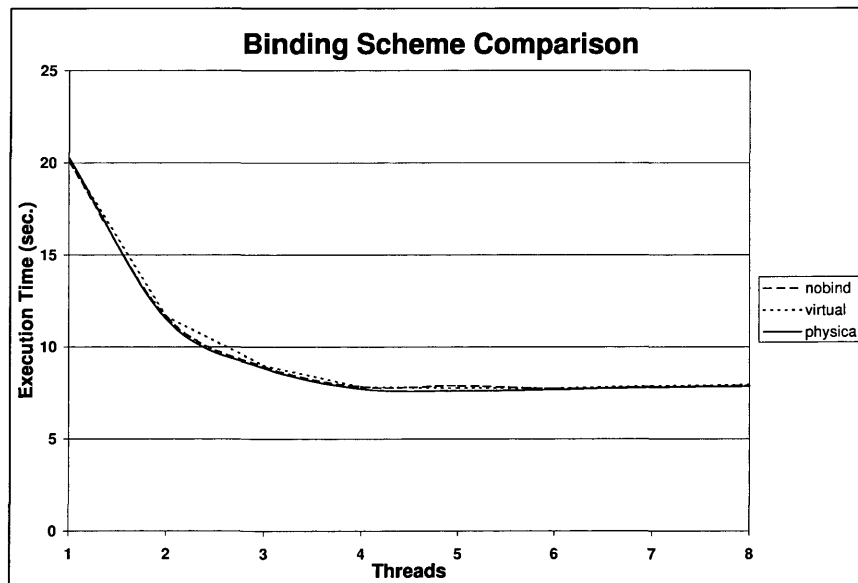
**Figure 5.5:** Performance of a Factory and an OpenMP implementation of multilevel parallel Strassen matrix multiplication.

are used, resource sharing inside each SMT processor limits execution time improvement. Using 8 threads activates all 8 execution context on the 4 SMT processors of the system. However, the exploitation of all execution contexts offers performance improvement of only 0.5 seconds over the execution with 4 threads. The multilevel Factory implementation is slightly slower than the single-level one. This is expected, since the scalability of the single-level code is not limited by the lack of parallelism, but rather by intra-processor resource sharing. As a result, the exploitation of the second level of parallelism in Strassen simply adds additional parallelism management overhead.

The performance of the OpenMP implementation is comparable to that of Factory. It still, however, experiences the same performance degradation as the single-level code when 5 or 6 threads are used, due to the SMT-unfriendly task queue implementation in the OpenMP compiler backend.

## 5.8 Thread Binding

A common optimization for multithreaded programs running on multiprocessors is to bind each thread to run on a particular processor. The rationale behind this optimization is that if a thread has already been running on a particular processor, that processor's cache is warm with that thread's data. Migrating the thread to a different processor will cause many unnecessary cache misses and likely increase the thread's execution time. An optimal binding of threads on a deep multiprocessor requires prior knowledge of how the multiprocessor is structured. We tested the single-level Strassen application from Section 5.6 with different binding schemes, as shown in Figure 5.6.



**Figure 5.6:** A comparison of different binding schemes using the single-level implementation of Strassen. *nobind* represents letting the Linux scheduler decide thread placement, *virtual* represents binding each thread to one execution context (one virtual processor), and *physical* represents binding each thread to two execution contexts (one physical processor).

We evaluated three different binding schemes: *nobind*, which performs no binding and left thread placement up to the Linux 2.6 scheduler; *virtual*, which binds each thread to one virtual processor (or execution context) just as is done on a standard multiprocessor; and *physical*, which binds each thread to a physical processor (each physical processor has two execution contexts). Our results show that the performance improvement with binding threads is negligible when compared to letting the Linux scheduler manage their placement. After four threads, where a second execution context is active on at least one processor, the binding schemes show a marginal improvement. As expected, the physical binding scheme outperforms the virtual binding scheme. This improvement is expected because each thread can run on two execution contexts (as opposed to one), and on both it is guaranteed to have a warm cache. However, the marginal difference between binding and not binding shows that in the case of the Linux 2.6 scheduler, letting the operating system handle thread placement is appropriate.

These results indicate that Factory's performance is independent of thread placement schemes. While binding threads to one physical processor only marginally improved performance, such binding schemes can expose the underlying processor architecture to the scheduling algorithm. When the scheduling algorithm is aware of the parallelism offered by the processor, then it can schedule work in such a manner to fully exploit the processor's capabilities.

## Chapter 6

# Conclusions and Future Work

We have presented Factory, an object-oriented parallel programming framework, which allows the exploitation of multiple types of parallelism on deep parallel architectures. Factory uses a clean, unified interface to express different, and potentially nested, forms of parallelism. Its design preserves the C++ type system and its implementation allows its use both as a standalone parallel programming library and as a runtime system for high-level object-oriented parallel programming languages. Factory includes a number of performance optimizations, all of which make the runtime system aware of the hierarchical structure of execution resources and memories on modern parallel architectures. The performance optimizations of Factory include efficient multithreaded memory allocation mechanisms that minimize contention and exploit locality; lock-free synchronization for internal concurrent data structures; integration of the management of the parallel work units with the memory management of native application data structures; and scheduling policies which are aware of the topology of execution contexts in multi-SMT or multi-CMP 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 several parallel applications with Factory and we have shown that Factory performs competitively and often better than OpenMP and Cilk, two widely used and well optimized parallel programming models for shared-memory systems. Moreover, we have shown that Factory can outperform manually tuned implementations of parallel applications with hand-coded mechanisms for managing parallelism.

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. First, we plan to investigate hierarchical scheduling algorithms, in which the scheduling policies are localized to groups of work units, according to the type of parallel work performed in each group. In the same context, we plan to investigate algorithms for dynamically selecting the scheduling strategy, using both compile-time and runtime information. Second, we plan to investigate dynamic concurrency control using Factory. Concurrency control is important for fine-grain parallel work running within SMTs or CMPs, because the interactions between threads may prevent parallel speedup within the processor, and the additional execution contexts in the processor may be used for purposes other than parallel execution, such as the overlapping of computation with I/O, or for assisted execution via precomputation of long-latency events [38]. Third, we shall consider the implications of hierarchical parallel architectures on the Factory synchronization mechanisms and investigate how the lock-free synchronization mechanisms can exploit resource sharing within SMTs and CMPs. Finally, we plan to extend Factory to incorporate transparent data



distribution and data movement facilities in order to provide runtime support for emerging chip multiprocessors with non-uniform cache architectures.

# Bibliography

- [1] R. ALI, J. HSIEH, AND O. CELEBIOGLU. Performance Characteristics of Intel Architecture-based Servers. *Dell Power Solutions*, November 2003.
- [2] P. AN, A. JULA, S. RUS, S. SAUNDERS, T. SMITH, G. TANASE, N. THOMAS, N. AMATO, AND L. RAUCHWERGER. STAPL: An Adaptive, Generic Parallel C++ Library. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 193–208, Cumberland Falls, Kentucky, USA, August 2001.
- [3] C. D. ANTONOPOULOS, X. DING, A. CHERNIKOV, F. BLAGOJEVIC, D. S. NIKOLOPOULOS, AND N. CHRISOCHOIDES. Multigrain Parallel Delaunay Mesh Generation: Challenges and Opportunities for Multithreaded Architectures. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS05)*, Cambridge, MA, U.S.A., Jun 2005.
- [4] D. H. BAILEY, E. BARSZCZ, J. T. BARTON, D. S. BROWNING, R. L. CARTER, L. DAGUM, R. A. FATOCHI, P. O. FREDERICKSON, T. A. LASINSKI, R. S. SCHREIBER, H. D. SIMON, V. VENKATAKRISHNAN, AND S. K. WEERATUNGA. The NAS Parallel Benchmarks – Summary and Preliminary Results. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM Press.
- [5] H. E. BAL, M. F. KAASHOEK, AND A. S. TANENBAUM. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.
- [6] B. N. BERSHAD, E. D. LAZOWSKA, AND H. M. LEVY. PRESTO: A System for Object-oriented Parallel Programming. *Software: Practice and Experience*, pages 713–732, August 1988.
- [7] R. BLUMOFÉ, C. JOERG, B. KUSZMAUL, C. LEISERSON, K. RANDALL, AND Y. ZHOU. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.
- [8] F. BODIN, P. BECKMAN, D. GANNON, S. NARAYANA, AND S. X. YANG. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 93.
- [9] J. BONWICK. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, pages 87–98, 1994.

- [10] A. BOWYER. Computing Dirichlet Tessellations. *Computer Journal*, 24:162–166, 1981.
- [11] PETER A. BUHR, GLEN DITCHFIELD, RICHARD A. STROOBOSCHER, B. M. YOUNGER, AND C. ROBERT ZARNKE. Concurrency in the object-oriented language c++. *Software - Practice and Experience*, 22(2):137–172, 1992.
- [12] Z. RADOVIĆ; AND E. HAGERSTEN. Efficient Synchronization for Non-Uniform Communication Architectures. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
- [13] C. CASCAVAL, J. CASTANOS, L. CEZE, M. DENNEA, M. GUPTA, D. LIEBER, J. MOREIRA, K. STRAUSS, AND JR. H. S. WARREN. Evaluation of a Multi-threaded Architecture for Cellular Computing. In *8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, pages 311–321, Cambridge, MA, U.S.A., February 2002.
- [14] K. MANI CHANDY AND C. KESSELMAN. CC++: A Declarative Concurrent Object Oriented Programming Notation. Technical report, California Institute of Technology, September 1992.
- [15] J. CHASE, F. AMADOR, E. LAZOWSKA, H. LEVY, AND R. LITTLEFIELD. The amber system: parallel programming on a network of multiprocessors. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 147–158, New York, NY, USA, 1989. ACM Press.
- [16] S. DONG, D. LUCOR, AND G. EM. KARNIADAKIS. Flow Past a Stationary and Moving Cylinder: DNS at Re=10,000. In *Proceedings of the IEEE 2004 Users Group Conference (DOD\_UGC'04)*, pages 88–95, Williamsburg, VA, U.S.A., Jun 2004. IEEE.
- [17] K. EBCIOGLU, V. SARASWAT, AND V. SARKAR. The IBM PERCS Project and New Opportunities for Compiler-Driven Performance via a New Programming Model. Compiler-Driven Performance Workshop (CASCON'2004), October 2004.
- [18] K. EBCIOGLU, V. SARASWAT, AND V. SARKAR. X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access. In *3rd International Workshop on Language Runtimes*, 2004.
- [19] MESSAGE PASSING INTERFACE FORUM. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, 1994.
- [20] M. FRIGO, C. E. LEISERSON, AND K. H. RANDALL. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.
- [21] R. GARCIA, J. JARVI, A. LUMSDAINE, J. SIEK, AND J. WILLCOCK. A Comparative Study of Language Support for Generic Programming. *SIGPLAN Not.*, 38(11):115–134, 2003.

- [22] ANDREW S. GRIMSHAW. Easy-to-use object-oriented parallel processing with mentat. *Computer*, 26(5):39–51, 1993.
- [23] L. HAMMOND, B. A. HUBBERT, M. SIU, M. K. PRABHU, M. CHEN, AND K. OLUKOTUN. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March-April 2000.
- [24] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 2: Thread Extensions (C Language), IEEE Standard 1003.1c. Standards Database, 1995.
- [25] L. V. KALE AND S. KRISHNAN. CHARM++ : A Portable Concurrent Object-Oriented System Based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, A. Paepcke, editor, pages 91–108. ACM Press, September 1993.
- [26] T. LENGI, R. ALI, J. HSIEH, AND C. STANTON. A Study of Hyper-Threading in High-Performance Computing Clusters. *Dell Power Solutions*, November 2002.
- [27] M. M. MICHAEL AND M. L. SCOTT. 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)*, pages 267–275, Philadelphia, Pennsylvania, U.S.A., 1996.
- [28] J. MITCHELL. Sun's Vision for Secure Solutions for the Government. National Laboratories Information Technology Summit, June 2004.
- [29] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, Version 2.5 Public Draft edition, November 2004.
- [30] B. ROBERT AND D. DIONISIOS. Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors. Technical report, University of Texas at Austin, 1999.
- [31] S. SHAH, G. HAAB, P. PETERSEN, AND J. THROOP. Flexible Control Structures for Parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(12):1219–1239, 2000.
- [32] V. STRASSEN. Gaussian Elimination is not Optimal. *Numer. Math.*, 23:354–356, 1969.
- [33] BJARNE STROUSTRUP. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.
- [34] JR. T. H. DUNIGAN, M. R. FAHEY, J. B. WHITE III, AND P. H. WORLEY. Early Evaluation of the Cray X1. In *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, Phoenix, AZ, U.S.A., nov 2003.
- [35] T. TAKAYANAGI, J. SHIN, B. PETRICK, J. SU, AND A. LEON. A Dual-Core 64b UltraSPARC Microprocessor for Dense Server Applications. In *Proc. of the 41st Conference on Design Automation (DAC'04)*, pages 673–677, San Diego, CA, U.S.A., June 2004.

- [36] X. TIAN, A. BIK, M. GIRKAR, P. GRAY, H. SAITO, AND E. SU. Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal*, 6(1), Feb 2002.
- [37] D. M. TULLSEN, S. EGGERS, AND H. M. LEVY. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.
- [38] T. WANG, C. ANTONOPOULOS, AND D. NIKOLOPOULOS. smt-SPRINTS: Software Precomputation with Intelligent Streaming for Resource-Constrained SMTs. In *Proc. of EuroPar 2005*, Lisbon, Portugal, August 2005.
- [39] D. F. WATSON. Computing the n-Dimensional Delaunay Tesselation with Application to Voronoi Polytopes. *Computer Journal*, 24:167–172, 1981.
- [40] S. C. WOO, M. OHARA, E. TORRIE, J. P. SINGH, AND A. GUPTA. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.
- [41] T. XINMIN, M. GIRKAR, S. SHAH, D. ARMSTRONG, E. SU, AND P. PETERSEN. 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*, pages 47–55, Nice, France, Apr 2003.

## VITA

Scott Arthur Schneider

Scott Schneider was born on June 18, 1981 in Fairfax County, Virginia. He graduated from Virginia Tech in 2003 with a Bachelor's degree in Computer Science and minors in Math and Physics. He entered The College of William and Mary as a Computer Science graduate student the same year and is continuing his studies to earn his Ph.D.