Next Generation System Software for Future High-End Computing Systems

Guang R. Gao, Kevin B. Theobald
Ziang Hu, Haiping Wu, Jizhu Lu
Dept. of Electrical and Computer Engineering
Univ. of Delaware
{ggao, theobald, hu, hwu, jlu}@capsl.udel.edu

Keshav Pingali, Paul Stodghill
Dept. of Computer Science, Cornell Univ.
{stodghil, pingali}@cs.cornell.edu

Thomas L. Sterling
Center for Advanced Computing Research
California Institute of Technology
tron@cacr.caltech.edu

Rick Stevens, Mark Hereld
Argonne National Laboratory, Univ. of Chicago
{stevens, hereld}@mcs.anl.gov

Abstract

Future high-end computers will offer great performance improvements over today’s machines, enabling applications of far greater complexity. However, designers must solve the challenge of exploiting massive parallelism efficiency in the face of very high latencies across the memory hierarchy. We believe the key to meeting this challenge is the design and implementation of new models and languages which address the problems of parallelism and latency on such machines. This paper presents an overview of our ongoing research toward this goal. Specifically, we will develop a suitable program execution model, a high-level programming notation, and a compiler and runtime system based on the underlying models. These are based on our previous work in parallel multithreaded systems, but are suitably enhanced to meet the needs of future high-end computers.

Keywords: EARTH, parallel computing, percolation, programming models, supercomputing

1. Introduction

This position paper discusses novel software technologies which will be needed for efficient implementation of complex applications on next-generation high-performance computers such as IBM Blue Gene, the ASCI teraflops-class systems, and the architecture(s) proposed in the NSF petaflops point-design studies. Such computers will have thousands to millions of processors, and complex multilevel memory hierarchies with memories physically dispersed across the machine. To use such machines effectively, enormous amounts of parallelism must be exposed in programs, and careful attention must be paid to the latency and bandwidth of access to different levels of memory. Applications enabled by such high-end machines are also expected to be significantly more complex, irregular and dynamic than today, presenting a major software challenge.

This challenge can only be met by new software technologies for the design, development, runtime support, and management of applications. We believe the key to meeting this challenge is the design and implementation of (i) a suitable program execution model, (ii) a high-level programming notation which shields the application developer from the complexities of the architecture wherever possible, and (iii) a compiler and runtime system which enable the efficient execution of such high-level programs under this program execution model. The program execution model should be the foundation for an integrated software environment that eliminates unnecessary boundaries between the different components and layers, and enhances the manageability of the entire system. The programming notation should not require radically different programming paradigms in different parts of a given machine; moreover, it should not demand that users significantly rewrite their applications for each new machine. Finally, the compiler and runtime system technology should be powerful enough to deduce information about program behavior which is required for efficient execution but which is not specified explicitly in a high-level application program.

In this paper, we discuss our work toward a novel program execution model and its corresponding programming notation. The model is developed in two phases, producing a base model and an advanced model. The base model is a refinement of the program execution model of the current EARTH multithreaded system [13, 22], modified to support the types of supercomputer architectures envisioned for this study. This refined model features fine-grain multithreading, support for a shared address space with atomic synchronizing memory operations, and dynamic load balancing.
The advanced execution model introduces entirely new concepts, and features percolation with adaptive just-in-time movement and management of thread contexts across the memory hierarchy, for effective latency tolerance. It also features a memory model based on location consistency, which avoids the high overheads of memory coherence.

This paper concludes by discussing the compiler and runtime system support for these models.

2. Background and challenges

In this section, we give a high-level overview of the architectures of future high-end platforms, describe applications that are likely to be run on these computers, and enumerate challenges for next-generation software.

2.1. High-end architectures

A number of high-profile research initiatives, such as NSF’s point design studies [5], the DOE ASCI program [18], and the recent NSF NPACI initiative [19], have triggered renewed interest in high-end computer architectures offering from teraflops to petaflops performance. An emerging consensus is that these systems will have thousands to millions of processors interconnected with multiple levels of networks, and a complex multilevel memory hierarchy with physically dispersed memories. At least three classes of architectures have been studied in some depth:

Revolutionary architectures exploit radically new hardware technologies outside the mainstream of commercial development. One example is the HTMT (Hybrid Technology Multi-Threaded) Architecture [7], in which a (relatively) modest number of 100GHz superconducting processors are employed with a memory hierarchy of at least 4 levels (see Figure 1). HTMT memory latencies increase by orders of magnitude from one level to the next, and these latencies are tolerated without loss of efficiency by multithreaded processor architectures. Less aggressive approaches may still use cutting-edge semiconductor technologies, such as silicon-germanium, delivering clock rate in excess of 20GHz, which will still lead to extremely high memory latencies when measured in CPU cycles.

Evolutionary architectures track current trends in mainstream computer evolution, both in off-the-shelf processors and networking technology, leveraging the huge investments in commercial computing systems. Examples include the ASCI teraflops-scale machines and follow-up architectures, expected to reach several hundred teraflops [18]. Such systems will have several hundred thousand to several million processors. These processors and their memories will be connected by a hierarchical network. The difference between the latency of accessing the memory closest to a processor and the latency to the most remote region of memory is expected to be enormous.

Novel architectures include machines being developed under the IBM Blue Gene and NASA Gilgamesh projects. They use conventional hardware technologies, but use custom processors rather than relying on off-the-shelf components. They are designed to exploit massive fine-grain parallelism for large-scale irregular applications.

2.2. Application development challenges

The availability of such high-end computers will enable the development of applications significantly more complex than today. These new applications will require considerable storage and the exploitation of parallelism at all levels of a program. Moreover, they will need to be far more adaptive than current applications, and need to handle irregular data and control structures. Current methods for building parallel applications (e.g., the SPMD execution model, static scheduling and data partitioning) are usually designed and optimized for a fixed computation platform. New techniques will be needed to map applications to high-end platforms where the computation load, resources, and operation latencies may vary dynamically.

2.3. Challenges to system software

We believe the most important challenge for high-end parallel machines is the management of latency and bandwidth, made difficult by the deep memory hierarchy and the complex applications that need dynamic management. The following is our plan to address these difficulties:

1. First, a program execution model (PXM) is specified, as a solid foundation for the proposed software technologies for high-end architectures with complex latency tolerance and management.
2. Second, a low-level programming model is developed based on the proposed PXM. This model will explicitly expose the memory hierarchy and the performance cost of data or computation movements within the hierarchy. Moreover, it will provide programmable abstractions that can facilitate the latency/bandwidth management to achieve desirable performance.

3. Finally, high-level languages, compilers and run-time systems are required, as discussed in later sections. Together, these components will provide the basis for developing and managing complex applications.

3. Program execution model

The program execution model (PXM) is the basic low-level abstraction of the underlying system architecture upon which our programming model, compilation strategy, run-time system, and other software components are developed. The PXM serves as an interface between the architecture and the software. Unlike an instruction set architecture (ISA) specification, which usually focuses on machine-specific low-level details such as opcodes and registers, the PXM refers to components at a higher level for the whole family of high-end machines (complemented by an ISA specification of each particular machine). The key features necessary to support the PXM are assumed to be implemented directly in hardware or indirectly by a combination of available hardware features and runtime system support.

The PXM is developed in two phases: a base model and an advanced model. The base model refines the program execution model of our EARTH multithreaded system [13, 22] to meet the needs of the supercomputer architectures envisioned for this study. The EARTH model was designed for use on conventional off-the-shelf parallel systems [14, 22], without the deep memory hierarchies expected for the systems discussed in Section 2.1. To exploit such hierarchies effectively, it is necessary to add several major new features to the PXM. These make up the advanced model.

3.1. Base program execution model

In this section, we focus on four key features of the proposed base PXM: threads, memory, synchronization, and dynamic load-balancing. These features provide a basis for latency tolerance and management framework to be discussed under the advanced execution model.

Thread model: Like other multithreaded execution models, our model divides a program into multiple sections, or threads. However, a distinct feature of our thread model is the mechanism used to form, enable, schedule, and synchronize threads. Our thread model comprises two layers. The first, threaded function invocation, forks a new thread to execute another function while the caller continues execution. This is an important source of parallelism. When a threaded function is invoked, a frame of storage is allocated from memory for its local context. Our own experience (with EARTH [11, 21, 23, 26]) and that of others (e.g., Cilk [4]) indicates that lightweight threaded function invocation enhances both the programmability and the efficiency of parallel applications.

The second layer of threads in our base PXM involves fibers — ultra-lightweight threads within the body of a threaded function. A fiber can run as soon as its data and control dependences are satisfied. Compiler and runtime system support will ensure the low overhead of fiber initiation and termination. Fibers provide several benefits. For example, when an operation such as a nonlocal memory operation might incur long or unpredictable latencies, operations that are dependent on the results may be placed in separate fibers. The corresponding runtime system can hide the latency by executing other fibers as long as the program has enough parallelism. This also ensures that a fiber can be executed in a nonpreemptive fashion, avoiding the waste of processor resources (due to idle fibers) and the overheads of saving the state of a preempted fiber.

Memory model: Our PXM will support global addressing through a combination of architecture features and compiler/runtime support. For a multithreaded execution model, globally addressable memory provides a seamless extension of the memory model viewed by threads assigned to the same processing node. Global addressing makes naming logically shared data much easier for the programmer because any thread can directly reference any data in the shared space and the naming model is similar to that on a uniprocessor. Additionally, global naming can improve the programmability and efficiency of applications with irregular and unpredictable data needs [20, 27] and can facilitate support of dynamic load balancing of threads.

One refinement in our base model is a frame-based addressing scheme, first proposed for EARTH [22]. Global addresses are formed as combinations of global frame pointers and local offsets. This scheme yields many benefits, such as simplifying memory analysis for the compiler and permitting more ambitious dynamic load-balancing.

With regard to memory consistency, issues arise as to how replication of nonlocal data should be managed. Our current belief is that the PXM should give the user a choice: either to allocate certain data and their replication explicitly or to adopt an automatic replication mechanism provided by the PXM. Automatic data replication and movement become desirable when the critical path of an application involves frequent irregular and dynamic data accesses at the fine-grained level. Nevertheless, implementing such
an automatic mechanism will be challenging. Our memory model therefore will be developed and implemented in two stages. The first stage, to be realized in the base PXM, will support only atomic synchronizing memory operations (see Section 3.1). The second stage, part of the advanced PXM, will feature the direct use of load/store operations implemented in a new memory model (see Section 3.2).

**Synchronization and scheduling:** Under our PXM, the data and control dependencies among fibers (within the same function) and among threaded function invocations (via their respective fibers) will be made explicit. A synchronization signal will be posted from one fiber to another, either in the same or in another threaded function instance, to inform the recipient that a specific dependence has been satisfied. Our PXM provides atomic operations for sending data (possibly to a global memory location) and posting an event, to guarantee that the data has been properly transferred before fibers that use that data are run.

**Dynamic load balancing:** Under the base PXM, a threaded function can be declared “movable” by the programmer or compiler, allowing the runtime system to run that function on any node. This permits the use of dynamic, low-overhead load-balancing algorithms [2, 13, 15].

**Open research issues in the base PXM:** Research issues we expect to address in the development of the base PXM include the following: (1) Should preemption and non-strict synchronization/evaluation be considered in the fiber model? If so, where, when, and how it should be supported? (2) How would SMP configurations (clusters of processors sharing local memory) best support the base model? (3) How would the upcoming generation of processors which support simultaneous execution of several concurrent threads best support the base model? If so, at what level should it be used: function level, fiber level, or lenient data structures (e.g., I-structures)? (4) Should speculative execution [25] be explored in the thread model? If so, at what level and how? (5) Should the binding of data objects and code be considered for automatic migration?

### 3.2. Advanced program execution model

The goal of the advanced execution model is to enhance the capacity and efficiency of the latency management of the memory hierarchy and its consistency. For this purpose, we propose to study extensions focusing on thread percolation, location consistency, and adaptive load migration.

**Thread percolation:** A key feature of our proposed model is the explicit exposition of the complex memory hierarchy and of the computation and data movement costs to the programming model of high-end machines. This transparency will be achieved by providing programmable abstractions that enable efficient management of the latency/bandwidth across the system. To this end, we plan to develop a new strategy, called thread percolation. Unlike prefetching, which involves moving blocks of data within the memory hierarchy, thread percolation manages contexts, which include data, program instructions, and control state. Moreover, whereas in most prefetching mechanisms a prefetch starts only when a processor issues a request, in thread percolation the main processors might be unaware of the preparation of many contexts throughout the system.

The percolation model also departs from the rigid constraints of classical memory hierarchies and related software layers based on demand paging/caching. Specifically, under the percolation model, the application can steer both the program flow and the data movement across the memory hierarchy to cause the data to “meet” the corresponding threads just in time at the vicinity of the processors where the computation is to be carried out. In other words, the percolation dictates an “active caching” architecture in order to optimally match the behavior of the memory system to the application needs, and to rapidly adapt to changes in the application’s data locality characteristics. The enabling of a thread may involve either gathering the data toward the processor(s) where the thread is enabled, called **inward percolation**, or sending/migrating the thread toward the vicinity of the data, called **outward percolation**.

This percolation model builds on our experience with an earlier percolation model developed for the Hybrid Technology Multi-Threaded (HTMT) architecture in Figure 1 [6, 10]. HTMT introduced a percolation program and execution model that (1) is explicitly multi-threaded; (2) incorporates global memory address space; and (3) explicitly exposes the HTMT memory hierarchy to the programmer. The percolation model extends dynamic prefetching to allow the management of contexts that include data, program instructions, and control states.

In one case study involving dense matrix multiply [6], an analytical study of our algorithm and the percolation process was used to determine the number of operations performed in each memory region and the amount of data exchanged between regions. Estimates for the processing power, network performance and storage capacity were inserted into the analytic model to predict the performance of this algorithm on HTMT. The resulting calculations indicated that with the expected HTMT design parameters, it will be possible to multiply square matrices of size 208,000 in 16.2 seconds, a sustained rate of 1.1 petaflops.

**Extension of the base memory model:** Sequential consistency (SC) and SC-derived models assume memory coherence. We maintain, however, that the role of a memory
consistency model is not to enforce memory coherence but to ensure correct execution of parallel programs. To this end, we propose to study alternative models that do not assume memory coherence. Our intention is to avoid the high communication cost for maintaining coherence across all memory regions in a complex memory hierarchy and among hundreds and thousands of threads.

One promising model is location consistency (LC) [8, 9]. This model represents the state of a memory location as a partially ordered multiset of write operations and synchronization operations. If we use the LC model, it may need to be extended [9], e.g., by redefining the unit of coherence from the word/line level to an object, region, or page.

Whether the LC model or some other approach is selected, the simplicity and familiarity of the SC model should be maintained. Moreover, if SC consistency proves desirable, the model must be extensible to accommodate such a requirement. Furthermore, the model must provide a software caching mechanism that can effectively manage replication and locality. An open question is where and how this mechanism should be introduced, and whether such a caching mechanism can work smoothly and profitably in a machine that implements the thread percolation model.

**Extension of dynamic load-balancing support:** The dynamic load-balancing mechanism proposed for the base model will require further investigation to ensure that interactions with the thread percolation model and the memory consistency model do not impact performance. Specifically, the cost of moving data and/or threads to maintain the physical locality constraint of the percolation model must be considered to improve load balance. Data reuse and replication managed under the LC model should also be part of the consideration. New and adaptive load-balancing algorithms may be needed and if so, we will develop them.

**4. Programming model and Threaded-C**

Once the proposed program execution model has been developed and specified, the next step is to develop a low-level programming model that can serve as a foundation for system software. The simplest approach is to present the PXM operations directly in the language. A program in this language might be produced by a compiler or directly by a programmer. We propose to extend EARTH Threaded-C [22, 24] (a standard ANSI-C with threaded operations) to accommodate our proposed PXM.

**4.1. Basic features of Threaded-C**

The initial specification of our new Threaded-C will begin with modification of the following features from EARTH Threaded-C: the threaded function specification, the fiber specification (represented as threads within a function body in EARTH Threaded-C), the `sync`, `slot` mechanism that enforces fiber and thread synchronization, the set of split-phase operations such as `block_move_sync` and `data_sync`, and the `token` and `invoke` mechanism to control static and dynamic load balancing [22, 24]. We will then add new mechanisms to (1) start the preparation of data for threads that will be enabled in the near future; (2) move data and code among memory regions; (3) pack data and code together to enable their joint percolation; and (4) enable the management of memory space to store data waiting to be percolated, processed, or disposed.

**4.2. Advanced features of Threaded-C**

The next task in our Threaded-C research is to extend the language to reflect the advanced features in the PXM as outlined in Section 3.2.

**Programming model for thread percolation:** Percolation presents two challenges for development of the programming model. The first is to provide user-friendly constructs in the language to express the physical locality requirements of a threaded function. For this we introduce parcels. A parcel encapsulates both the threaded code and the data required to ensure the desired layout for execution.

The second challenge is to organize the complex memory hierarchy so that the data movement and reorganization dictated by thread percolation can be programmed effectively. High-end architectures, such as those proposed for HTMT and Blue Gene, do not have conventional data caches. Rather, each level of memory is considered a “buffer” of the next level; these buffers are directly addressable. Buffer storage allocation and data movement at each level can be fully or partially under the programmer’s control. Moreover, since the penalty of a memory access miss that is in the critical path of a program execution may be severe, we will need to develop schemes that do not rely on a “hidden” automatic coherence management mechanism.

We propose to extend the EARTH Threaded-C language with a set of primitives to implement percolation. Potential candidates include primitives to copy and move blocks of data and dispatch parcels from one level to another in the memory hierarchy and to allocate and release space for a block within the current memory level. Such primitives form the skeleton of a new language that was developed for the HTMT project [6]. This language should be easily extensible to high-end machines in other classes, such as the ASCI architectures and other high end architectures of interest to this project (see Section 2.1). Complete implementation of a language will require additional programming mechanisms to support data reorganization (e.g., gather/scatter, pointer swizzling, generalized corner turn),
The proposed programming and program execution model will provide the foundation for our system software. Included are a number of closely integrated software components: compiler, runtime system (RTS), resource and application management tools, performance debugging, profiling, evaluation tools guided by suitable performance models, and a simulation/emulation platform. Together they provide a basis for developing and managing complex applications.

The proposed software framework is illustrated in Figure 2. One unusual feature is that there is no fixed boundary between different software layers: the shaded regions represent close interactions between the Threaded-C compiler and RTS, and between the RTS and the architecture layer. Moreover, the performance models and tools are integrated across all layers.

Our research will focus on the runtime percolation model and memory management. The project will not cover the development of general OS support beyond this limit.

5.1. Runtime system

The essential features of the RTS are to implement the base and advanced PXM. Both functions will interact with the architecture layer and the Threaded-C compiler. The RTS will also provide resource and application management with the OS layer.

RTS support for the base PXM: The implementation of the base PXM will leverage our experience with the design of the EARTH runtime system. Nevertheless, new challenges must be addressed. For example, current implementations of EARTH are designed for simple distributed-memory systems, whereas future supercomputer memories are likely to be shared, and may be explicitly divided into multiple levels, where we cannot assume a uniform address model at each level. We propose to study two options for addressing memory across remote nodes: using a global address directly to name a location, or using a frame-based address, since most data are encapsulated with frames associated with corresponding threaded function invocation.

RTS support for the advanced PXM features: For the advanced PXM, we will address two main areas: percolation and the memory model.

We envision that runtime system for percolation will comprise three major components: the parcel invocation and termination module (PIT), the parcel assembly and disassembly module (PAD), and the parcel dispatcher and dispenser module (PDD). The PIT invocation manager will invoke enabled parcel functions and send them to the PAD for processing. The PAD assembly manager will move the required code and data into local memory locations. The parcel will then be passed to the PDD, which will reserve space in the fast (local) memory region and then move the data and code associated with the parcel into the reserved

---

**Figure 2. Software architecture**

---

Some of which might be best implemented as library functions.

**Programming under our memory model:** To permit programming under the memory model of the advanced PXM, we propose to extend Threaded-C with a set of simple programming primitives, initially selected from those in the LC model [8, 9] as well as other commonly used shared-memory programming primitives. In addition, we will include operations for undirected control synchronization and acquire-release data synchronization.

Several open issues should have major ramifications on the Threaded-C extensions. One issue is the level of programmability available to the user. Another open issue is the extension of the unit of consistency to a memory object or memory region. We will investigate how this can be reflected in the programming model and Threaded-C; one possibility is to define a Threaded-C++.

Currently, under EARTH Threaded-C, a threaded function can be invoked as a token, which lets the runtime system decide where to run it to improve load balancing. We may wish to increase the flexibility to allow a programmer or compiler to provide more hints, such as the region within which the invocation site should be considered.

Finally, we need to extend Threaded-C and its programming model so the users can express their computation to take advantage of both thread percolation and dynamic load balancing together. The concrete programming notations required, and their semantics and implementation issues, are a part of the proposed research.

5. Integrated software support

The proposed programming and program execution model will provide the foundation for our system software. Included are a number of closely integrated software components: compiler, runtime system (RTS), resource and application management tools, performance debugging, profiling, evaluation tools guided by suitable performance models, and a simulation/emulation platform. Together they provide a basis for developing and managing complex applications.
5.2. Compilers

The compiler must work closely with the runtime system and must interact with the system resource management and the performance models of the related software and hardware components.

Threaded-C compilation: We have identified four key issues regarding Threaded-C compilation:

1. **Code partitioning into fibers for optimal performance.** The compiler must have access to cost models of the underlying architecture and RTS.

2. **Register allocation and synchronization resources.** This issue, which is critical to code efficiency, must be investigated in conjunction with development of the dynamic RTS fiber scheduling policy.

3. **Compilation for percolation.** Techniques are needed for analyzing the data and code movement required by a parcel, minimizing unnecessary copying and maximizing reuse. Optimization requires close interaction between compiler, RTS, and the performance models at both the architecture and the software levels.

4. **Automatic relocation and automatic caching mechanisms.** Efficient algorithms must be devised for these automatic mechanisms.

High-level language compilation: We will use state-of-the-art restructuring compiler technology to provide programmers with a high-level abstraction of the architecture. The language will support constructs for specifying control parallelism using `doall` loops and parallel statement sequences. The programmer will use these constructs to specify coarse-grained parallelism, leaving it to the compiler to detect fine-grained parallelism.

Our compiler must also address the dual problem of latency tolerance (multithreading) and latency avoidance (memory hierarchies). Compiler techniques for machines with memory hierarchies, such as linear loop transformations followed by tiling or shackling [16, 28, 29], reschedule operations within loop nests to promote data reuse but are not concerned with multithreading. On the other hand, compiler techniques for latency-tolerant multithreaded architectures have focused mainly on thread generation issues such as producing code for nonpreemptive, nonblocking threads, but are not concerned with locality enhancement [3, 12]. Both kinds of techniques will be required for the architectural models discussed in this proposal.

Our proposed compilation strategy is based on the data-centric compilation technology developed by Pingali and co-workers [16]. Conventional restructuring compilation technology is control-centric: it reasons directly about the control flow of the program and reorders this control flow to accomplish goals like parallelization or locality enhancement. This technology works well for perfectly nested loops, but extending it beyond perfectly nested loops appears problematic. Even for relatively simple programs like Cholesky factorization, code generated by conventional compilers performs 5 to 10 times worse than hand-coded libraries such as LAPACK [17]. Data-centric code generation addresses some of these problems. Rather than manipulating the control structure of the program directly, the compiler determines the order in which data elements should be fetched into the highest level of the memory hierarchy, and then schedules close together statement instances that touch a given data item. In principle, this scheduling can be done across imperfectly nested loops and across procedure boundaries. Experiments at SGI indicate that, for dense numerical linear algebra applications, data-centric technology improves performance by factors of 2 to 5.
stood, we will address the problem of exploiting the full thread percolation model. Since this model requires that all data touched by a thread be local, data-centric code generation schemes appear particularly well suited here.

References


