Sequoia: Programming the Memory Hierarchy

Kayvon Fatahalian Timothy J. Knight Mike Houston Mattan Erez
Daniel Reiter Horn Larkhoon Leem Ji Young Park Manman Ren
Alex Aiken William J. Dally Pat Hanrahan

Stanford University

Abstract

We present Sequoia, a programming language designed to facilitate the development of memory hierarchy aware parallel programs that remain portable across modern machines featuring different memory hierarchy configurations. Sequoia abstractly exposes hierarchical memory in the programming model and provides language mechanisms to describe communication vertically through the machine and to localize computation to particular memory locations within it. We have implemented a complete programming system, including a compiler and runtime systems for Cell processor-based blade systems and distributed memory clusters, and demonstrate efficient performance running Sequoia programs on both of these platforms.

1 Introduction

Writing a high-performance application, whether for a uniprocessor or for a large scale parallel machine, requires the programmer to have non-trivial knowledge of the underlying machine's architecture. On modern systems of all scales, a major aspect of performance optimization involves ensuring that processors do not frequently idle waiting on memory. This requires structuring algorithms and placing data so that data references are serviced by levels of the memory hierarchy as close to the processors as possible, whether it be on-die storage, local DRAM, or remote memory accessed over high-speed interconnect. Writing programs that make efficient use of a machine's memory system is further complicated by desires for program portability. A series of optimizations targeted at the hierarchy of one machine is likely not the correct solution for the particular sizes and physical configurations of other systems.

The need for new programming abstractions for managing memory is becoming acute as the number of parallel processing units on a chip is increasing rapidly and the importance of efficiently utilizing available memory bandwidth is growing. This trend is apparent both in the ubiquity of multi-core microprocessors and in the emergence of stream architectures, such as the Sony/Toshiba/IBM Cell Broadband Engine Processor^{TM} (Cell) [Pham et al. 2005] and Stanford's Imagine [Kapasi et al. 2002] and Merrimac [Dally et al. 2003] processors. In contrast to traditional microprocessors, which provide a single address space and manage the transfer of data between memory and levels of on-chip storage transparently in hardware, these new *exposed-communication* architectures require software to move data in between distinct on- and off-chip address spaces; explicit memory management is necessary for program correctness, not just performance. Thus, the challenges of managing data movement, formerly only a concern when programming large parallel machines, now exist at the node level. These difficulties compound as larger scale systems are considered.

Mechanisms provided to express memory locality in existing parallel languages, such as the designation of local and global arrays in UPC [Carlson et al. 1999], Co-Array Fortran [Numrich and Reid 1998], and Titanium [Yelick et al. 1998], and distributions over locales as in ZPL [Deitz et al. 2004], Chapel [Callahan et al. 2004], and X10 [Charles et al. 2005], do not solve the problem of memory management on exposed-communication architectures. These existing approaches describe the distribution and *horizontal* communication of data among nodes of a parallel machine. They do not address the problem of choreographing data movement *vertically* through the memory hierarchy, a critical aspect of programming modern architectures.

The principal idea of our work is that the movement and placement of data at all levels of the machine memory hierarchy should be under explicit programmer control via first class language mechanisms. This paper presents **Sequoia**, a programming model focused on assisting the programmer in structuring bandwidth-efficient parallel programs that remain easily portable to new machines. The design of Sequoia centers around the following key ideas:

This work is supported in part by the Department of Energy under the ASCI Alliances program (Contract LLNLB523583), in part by National Science Fountation Graduate Research Fellowships, in part by an Intel Foundation Graduate Fellowship, and by donations from the IBM Corporation.

SC2006 November 2006, Tampa, Florida, USA
0-7695-2700-0/06 $20.00 ©2006 IEEE

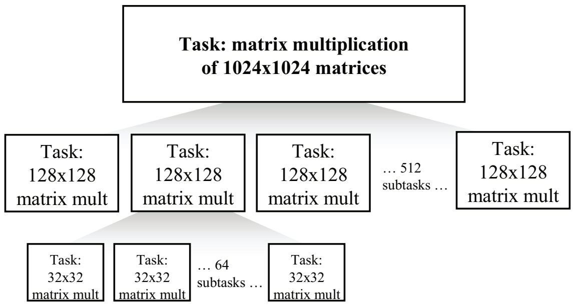
Task: matrix multiplication of 1024x1024 matrices
Task: 128x128 matrix mult
Task: 128x128 matrix mult
Task: 128x128 matrix mult
… 512 subtasks …
Task: 128x128 matrix mult
Task: 32x32 matrix mult
Task: 32x32 matrix mult
… 64 subtasks …
Task: 32x32 matrix mult

Figure 1: Multiplication of 1024x1024 matrices structured as a hierarchy of independent tasks performing smaller multiplications.

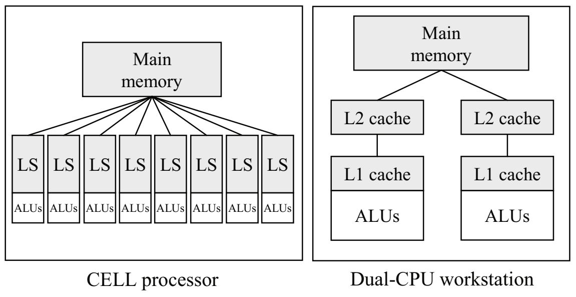

Figure 2: A Cell workstation (left) is modeled as a tree containing nodes corresponding to main system memory and each of the processor's software-managed local stores. A representation of a dual-CPU workstation is shown at right.

- We introduce the notion of hierarchical memory directly into our programming model to gain both portability and performance. Sequoia programs run on machines that are abstracted as trees of distinct memory modules and describe how data is moved and where it resides in a machine's memory hierarchy.

- We use *tasks* as abstractions of self-contained units of computation that include descriptions of key information such as communication and working sets. Tasks isolate each computation in its own local address space and also express parallelism.

- To enable portability, we maintain a strict separation between generic algorithmic expression and machine-specific optimization. To minimize the impact of this separation on performance, details of the machine-specific mapping of an algorithm are exposed to programmer control.

Sequoia takes a pragmatic approach to portable parallel programming by providing a limited set of abstractions that can be implemented efficiently and controlled directly by the programmer. While the compiler implementation described in this paper does not make heavy use of automatic analysis, we have taken care to ensure that Sequoia programs are written within a framework that is amenable to the use of advanced compiler technology.

Details of Sequoia programming are discussed in Sections 2 through 4. Section 5 describes our implementation of a Sequoia language compiler and associated runtime systems for Cell-based workstations and for clusters of PCs. Section 6 discusses the performance obtained by running initial Sequoia applications on both of these parallel platforms.

2 Hierarchical Memory

On modern systems featuring deep memory hierarchies and many parallel processing units, breaking large computations into smaller operations is essential to achieving good performance because it exposes parallelism and results in efficient execution on datasets stored local to processing elements. Common examples of this optimization technique include blocking to increase cache locality and problem decomposition to minimize network communication in MPI programs for clusters. In Figure 1 we illustrate the hierarchical structure of a computation to perform blocked matrix multiplication, an example we revisit throughout much of this paper. In this algorithm, which features nested parallelism and a high degree of hierarchical data locality, parallel evaluation of submatrix multiplications is performed to compute the product of two large matrices.

Sequoia requires such hierarchical organization in programs, borrowing from the idea of space-limited procedures [Alpern et al. 1995], a programming methodology proposed to encourage hierarchy-aware, parallel divide-and-conquer programs. Space-limited procedures require each function in a call chain to accept arguments occupying significantly less storage than those of the calling function. Sequoia tasks (Section 3) generalize and concretize the concept of a space-limited procedure into a central construct used to express communication and parallelism and enhance the portability of algorithms. We have implemented a complete programming system around this abstraction, including a compiler and runtime systems for Cell and distributed memory clusters.

Writing Sequoia programs involves abstractly describing hierarchies of tasks (as in Figure 1) and then mapping these hierarchies to the memory system of a target machine. Sequoia requires the programmer to reason about a parallel machine as a tree of distinct memory modules, a representation that extends the Parallel Memory Hierarchy (PMH) model of Alpern et al. [1993]. Data transfer between memory modules is conducted via (potentially asynchronous) block transfers. Program logic describes the transfers of data at all levels, but computational kernels are constrained to operate upon data located within leaf nodes of the machine tree. The abstract representation of a system containing a Cell processor (at

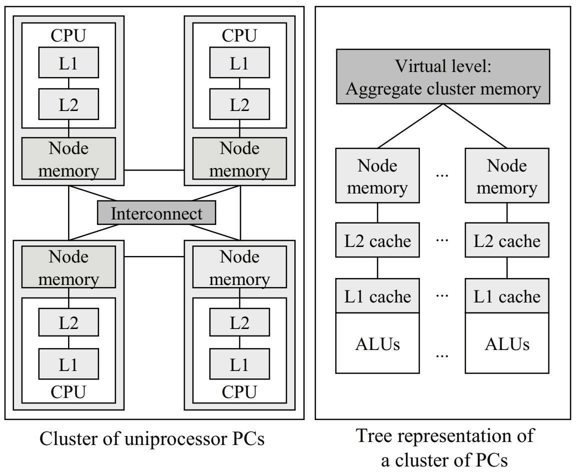
CPU
L1
L2
Node memory
Interconnect
Virtual level: Aggregate cluster memory
L2 cache
L1 cache
ALUs
Cluster of uniprocessor PCs
Tree representation of a cluster of PCs

Figure 3: The point-to-point links connecting PCs in a cluster are modeled as a virtual node in the tree representation of the machine.

left in Figure 2) contains nodes corresponding to main system memory and each of the 256KB software-managed local stores (LSes) located within the chip's synergistic processing units (SPEs). At right in Figure 2, a model of a dual-CPU workstation contains nodes representing the memory shared between the two CPU's as well as the L1 and L2 caches on each processor. Sequoia permits a machine to be modeled with detail commensurate with the programmer's needs. A representation may include modules corresponding to all physical levels of the machine memory hierarchy, or it may omit levels of the physical hierarchy that need not be considered for software correctness or performance optimization.

Establishing an abstract notion of hierarchical memory is central to the Sequoia programming model. Sequoia code does not make explicit reference to particular machine hierarchy levels and it remains oblivious to the mechanisms used to move data between memory modules. For example, communication described in Sequoia may be implemented using a cache prefetch instruction, a DMA transfer, or an MPI message depending on the requirements of the target architecture. Supplying constructs to describe the movement of data throughout a machine while avoiding any reference to the specific mechanisms with which transfers are performed is essential to ensuring the portability of Sequoia programs while retaining the performance benefits of explicit communication.

As with the PMH model, our decision to represent machines as trees is motivated by the desire to maintain portability while minimizing programming complexity. A program that performs direct communication between sibling memories, such as a program written using MPI for a cluster, is not directly portable to a parallel platform where such channels do not exist. Because many machines have complex non-tree topologies we allow our tree abstraction to include *virtual levels* that do not correspond to any single physical machine memory. For example, it is not practical to expect the nodes in a cluster of workstations to communicate only via global storage provided by networked disk. As shown in Figure 3, our model represents a cluster as a tree rooted by a virtual level corresponding to the aggregation of all workstation memories. The virtual level constitutes a unique address space distinct from any node memory. Transferring data from this global address space into the child modules associated with individual cluster workstations results in communication over the cluster interconnect. The virtual level mechanism allows us to generalize the tree abstraction for modeling vertical communication to encapsulate horizontal inter-node communication as well.

3 Sequoia Design

The principal construct of the Sequoia programming model is a *task*: a side-effect free function with call-by-value-result parameter passing semantics. Tasks provide for the expression of:

- **Explicit Communication and Locality**. Communication of data through the memory hierarchy is expressed by passing arguments to tasks. Calling tasks is the only means of describing data movement in Sequoia.
- **Isolation and Parallelism**. Tasks operate entirely within their own private address space and have no mechanism to communicate with other tasks other than by calling subtasks and returning to a parent task. Task isolation facilitates portable concurrent programming.
- **Algorithmic Variants**. Sequoia allows the programmer to provide multiple implementations of a task and to specify which implementation to use based on the context in which the task is called.
- **Parameterization**. Tasks are expressed in a parameterized form to preserve independence from the constraints of any particular machine. Parameter values are chosen to tailor task execution to a specific hierarchy level of a target machine.

This collection of properties allows programs written using tasks to be portable across machines without sacrificing the ability to tune for performance.

3.1 Explicit Communication And Locality

A Sequoia implementation of blocked matrix multiplication is given in Figure 4. The `matmul` task multiplies M x P input matrix `A` by P x N input matrix `B`, accumulating the results into M x N matrix `C` (`C` is a read-modify-write argument to the task). The task partitions the input matrices into blocks

```
1  void task matmul::inner( in    float A[M][P],
2                           in    float B[P][N],
3                           inout float C[M][N] )
4  {
5    // Tunable parameters specify the size
6    // of subblocks of A, B, and C.
7    tunable int U;
8    tunable int X;
9    tunable int V;
10
11   // Partition matrices into sets of blocks
12   // using regular 2D chopping.
13   blkset Ablks = rchop(A, U, X);
14   blkset Bblks = rchop(B, X, V);
15   blkset Cblks = rchop(C, U, V);
16
17   // Compute all blocks of C in parallel.
18   mappar (int i=0 to M/U, int j=0 to N/V) {
19     mapreduce (int k=0 to P/X) {
20       // Invoke the matmul task recursively
21       // on the subblocks of A, B, and C.
22       matmul(Ablks[i][k],Bblks[k][j],Cblks[i][j]);
23     }
24   }
25 }
26
27 void task matmul::leaf( in    float A[M][P],
28                         in    float B[P][N],
29                         inout float C[M][N] )
30 {
31   // Compute matrix product directly
32   for (int i=0; i<M; i++)
33     for (int j=0; j<N; j++)
34       for (int k=0; k<P; k++)
35         C[i][j] += A[i][k] * B[k][j];
36 }
```

Figure 4: Dense matrix multiplication in Sequoia. `matmul::inner` and `matmul::leaf` are variants of the `matmul` task.

(lines 13–15) and iterates over submatrix multiplications performed on these blocks (lines 18–24). An explanation of the Sequoia constructs used to perform these operations is provided in the following subsections.

Defining tasks expresses both locality and communication in a program. While a task executes, its entire *working set* (the collection of all data the task can reference) must remain resident in a single node of the abstract machine tree. As a result, a task is said to run at a specific location in the machine. In Figure 4, the matrices `A`, `B`, and `C` constitute the working set of the `matmul` task. Pointers and references are not permitted within a task and therefore a task's working set is manifest in its definition.

Notice that the implementation of `matmul` makes a recursive call in line 22, providing subblocks of its input matrices as arguments in the call. To encapsulate communication, Sequoia tasks use *call-by-value-result* (CBVR) [Aho et al. 1986] parameter passing semantics. Each task executes in the isolation of its own private address space (see Subsection 3.2) and upon task call, input data from the calling task's address space is copied into that of the callee. Output argument data is copied back into the caller's address space when the call returns. The change in address space induced by the recursive `matmul` call is illustrated in Figure 5. The block of

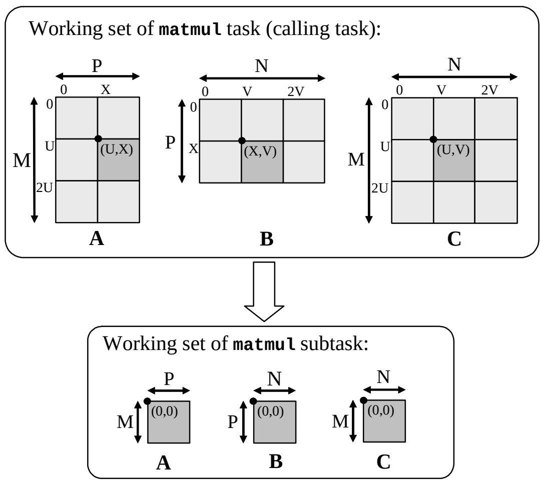
Working set of matmul task (calling task):
P
0
X
N
0
V
2V
N
0
V
2V
M
0
U
2U
(U,X)
P
0
X
(X,V)
M
0
U
2U
(U,V)
A
B
C
Working set of matmul subtask:
P
N
N
M
(0,0)
P
(0,0)
M
(0,0)
A
B
C

Figure 5: The `matmul::inner` variant calls subtasks that perform submatrix multiplications. Blocks of the matrices *A*, *B*, and *C* are passed as arguments to these subtasks and appear as matrices in the address space of a subtask.

size *U* x *X* of matrix `A` from the calling task's address space appears as a similarly sized array in the address space of the called subtask. CBVR is not common in modern languages, but we observe that for execution on machines where data is transferred between distinct physical memories under software control, CBVR is a natural parameter passing semantics.

The mapping of a Sequoia program dictates whether a callee task executes within the same memory module as its calling task or is assigned to a child (often smaller) memory module closer to a compute processor. In the latter case, the subtask's working set must be transferred between the two memory modules upon task call/return. Thus, the call/return of a subtask implies that data movement through the machine hierarchy *might* occur. Explicitly defining working sets and limiting communication to CBVR parameter passing allows for efficient implementation via hardware block-transfer mechanisms and permits early initiation of transfers when arguments are known in advance.

3.2 Isolation and Parallelism

The granularity of parallelism in Sequoia is the task and parallel execution results from calling concurrent tasks. Lines 18–24 of Figure 4 describe iteration over submatrix multiplications that produces a collection of parallel subtasks. (The `i` and `j` dimensions of the iteration space may be executed in parallel while the innermost dimension defines a reduction). In Sequoia, each of these subtasks executes in isolation, a key property introduced to increase code portability and performance.

Sequoia Blocking Primitives	
`blkset`	An opaque Sequoia object representing a collection of array blocks.
`rchop(A, len0, len1, ...)`	Generates a `blkset` containing non-overlapping blocks that tile the multi-dimensional array `A`. Each block is multi-dimensional with size `len0` $\times$ `len1` $\times \ldots$.
`rchop(A, rchop_t(offset0, len0, stride0), ...)`	Generalized form of `rchop` that generates blocksets containing potentially overlapping blocks. The starting array offset, block size, and stride between blocks is specified for every dimension of the source array.
`ichop(A, Starts, Ends, N)`	Generates a set of `N` irregularly-sized blocks from array `A`. Block start and end indices are given by elements in the length-N integer arrays `Starts` and `Ends`.
`gather(A, IdxBlkset)`	Generates a set of blocks by gathering elements from source array `A` using the indices provided in the blocks of `IdxBlkset`. The resulting `blkset` has the same number and size of blocks as `IdxBlkset`.
Sequoia Mapping Primitives	
`mappar(i=i0 to iM, j=j0 to jN ...) {...}`	A multi-dimensional for-all loop containing only a subtask call in the loop body. The task is mapped in parallel onto a collection of blocks.
`mapseq(i=i0 to iM, j=j0 to jN ...) {...}`	A multi-dimensional loop containing only a subtask call in the loop body. The task is mapped in sequentially onto a collection of blocks.
`mapreduce(i=i0 to iM, j=j0 to jN ...) {...}`	Maps a task onto a collection of blocks, performing a reduction on at least one argument to the task. To support parallel tree reductions, an additional *combiner* subtask is required.

Table 1: Sequoia mapping and blocking primitives

Isolation of task address spaces implies that no constraints exist on whether a subtask must execute within the same level of the memory hierarchy as its calling task. Additionally, Sequoia tasks have no means of communicating with other tasks executing concurrently on a machine. Although the implementation of `matmul` results in the execution of many parallel tasks, these concurrent tasks do not function as cooperating threads. The lack of shared state among tasks allows parallel tasks to be executed simultaneously using multiple execution units or sequentially on a single processor. Task isolation simplifies parallel programming by obviating the need for synchronization or locking constructs required by cooperating threads. Sequoia language semantics require that output arguments passed to concurrent subtasks do not alias in the calling task's address space. We currently rely on the programmer to ensure this condition holds.

3.3 Task Decomposition

We now introduce Sequoia's *array blocking* and *task mapping* constructs: first-class primitives available to describe portable task decomposition.

In Sequoia a subset of an array's elements is referred to as an *array block*. For example, `A[0:10]` is the block corresponding to the first 10 elements of the array `A`. The `matmul` task uses the `rchop` (regular chop) blocking function to describe a regular 2D partitioning of its input matrices. In line 13, `rchop` is used to divide the matrix `A` into a set of blocks each U x X in size. This collection is returned in the form of an opaque Sequoia object referred to as a *blockset*. Sequoia provides a family of blocking functions (see Table 1) to facilitate decompositions that range from the simplicity of `rchop` to the irregularity of arbitrary array gathers.

After defining blocksets using `rchop`, `matmul` iterates over the blocks, recursively calling itself on blocks selected from `Ablks`, `Bblks`, and `Cblks` in each iteration. As introduced in Subsection 3.2, the `mappar` construct designates parallel iteration, implying concurrency among subtasks but not asynchronous execution between calling and child tasks. All iterations of a `mappar` or `mapreduce` must complete before control returns to the calling task.

Imperative C-style control-flow is permitted in tasks, but use of blocking and mapping primitives is encouraged to facilitate key optimizations performed by the Sequoia compiler and runtime system. All applications presented in Section 6 use only mapping primitives to describe iteration. A complete listing of Sequoia blocking and mapping constructs is given in Table 1.

3.4 Task Variants

Figure 4 contains two implementations of the `matmul` task, `matmul::inner` and `matmul::leaf`. Each implementation is referred to as a *variant* of the task and is named using the syntax `taskname::variantname`. The variant `matmul::leaf` serves as the base case of the recursive matrix multiplication algorithm. Notice that the Sequoia code to recursively call `matmul` gives no indication of when the base case should be invoked. This decision is made as part of the machine-specific mapping of the algorithm (Section 4).

Inner tasks, such as `matmul::inner`, are tasks that call subtasks. Notice that `matmul::inner` does not access elements of its array arguments directly and only passes blocks of the arrays to subtasks. Since a target architecture may not support direct processor access to data at certain hierarchy levels, to ensure code portability, the Sequoia language does not permit inner tasks to directly perform computation on array elements. Instead, inner tasks use Sequoia's mapping and blocking primitives (Section 3.3) to structure computation into subtasks. Ultimately, this decomposition yields computations whose working sets fit in leaf memories directly accessible by processing units. An inner task definition is not associated with any particular machine memory module; it may execute at any level of the memory hierarchy in which its working set fits.

Leaf tasks, such as `matmul::leaf`, do not call subtasks and operate directly on working sets resident within leaf levels of

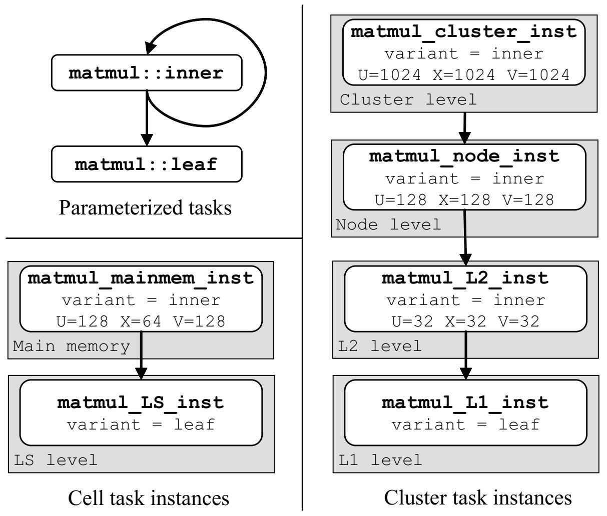

Figure 6: The call graph for the parameterized `matmul` task is shown at top left. Specialization to Cell or to the cluster machine from Figure 3 generates instances of the task shown at bottom left and at right.

```
instance {
 name    = matmul_mainmem_inst
 task    = matmul::inner
 runs_at = main_memory
 calls   = matmul_LS_inst
 tunable   U=128,X=64,V=128
}

instance {
 name    = matmul_LS_inst
 variant = matmul::leaf
 runs_at = LS_level
}
```

```
instance {
 name    = matmul_cluster_inst
 variant = matmul::inner
 runs_at = cluster_level
 calls   = matmul_node_inst
 tunable   U=1024,X=1024,V=1024
}
instance {
 name    = matmul_node_inst
 variant = matmul::inner
 runs_at = node_level
 calls   = matmul_L2_inst
 tunable   U=128,X=128,V=128
}
instance {
 name    = matmul_L2_inst
 task    = matmul::inner
 runs_at = L2_cache_level
 calls   = matmul_L1_inst
 tunable   U=32,X=32,V=32
}
instance {
 name    = matmul_L1_inst
 task    = matmul::leaf
 runs_at = L1_cache_level
}
```

Figure 7: Specification for mapping the `matmul` task to a Cell machine (left) and a cluster machine (right).

the memory hierarchy. Direct multiplication of the input matrices is performed by `matmul::leaf`. In practice, Sequoia leaf tasks often wrap platform specific implementations of computational kernels written in traditional languages, such as C or Fortran.

3.5 Task Parameterization

Tasks are written in parameterized form to allow for specialization to multiple target machines. *Specialization* is the process of creating *instances* of a task that are customized to operate within, and are mapped to, specific levels of a target machine's memory hierarchy. A task instance defines a variant to execute and an assignment of values to all variant parameters. The Sequoia compiler creates instances for each of the various contexts in which a task is used. For example, to run the `matmul` task on Cell, the Sequoia compiler generates an instance employing the `matmul::inner` variant to decompose large matrices resident in main memory into LS-sized submatrices. A second instance uses `matmul::leaf` to perform the matrix multiplication inside each SPE. On a cluster machine, one `matmul` instance partitions matrices distributed across the cluster into submatrices that fit within individual nodes. Additional instances use `matmul::inner` to decompose these datasets further into L2- and then L1-sized submatrices. While parameterized tasks do not name specific variants when calling subtasks, specialized task instances make direct calls to other instances. The static call graph relating `matmul`'s parameterized task variants is shown at top left in Figure 6. Calls among the task instances that result from specialization to Cell and to a cluster are also shown in the figure. Notice that three of the cluster instances, each mapped to a different location of the machine hierarchy, are created from the `matmul::inner` variant (each instance features different argument sizes and parameter values).

Task variants utilize two types of numeric parameters, *array size* parameters and *tunable* parameters. Array size parameters, such as `M`, `N`, and `P` defined in the `matmul` task variants, represent values dependent upon array argument sizes and may take on different values across calls to the same instance. Tunable parameters, such as the integers `U`, `V`, and `X` declared in `matmul::inner` (lines 7-9 of Figure 4), are designated using the `tunable` keyword. Tunable parameters remain unbound in Sequoia source code but are statically assigned values during task specialization. Once assigned, tunable parameters are treated as compile-time constants. The most common use of tunable parameters, as illustrated by the matrix multiplication example, is to specify the size of array blocks passed as arguments to subtasks.

Parameterization allows the decomposition strategy described by a task variant to be applied in a variety of contexts, making the task portable across machines and across levels of the memory hierarchy within a single machine. The use of tunable and array size parameters and the support of multiple task variants is key to decoupling the expression of an algorithm from its mapping to an underlying machine. Tasks provide a framework for defining the application-specific space of decisions that must be made during the process of program tuning. In the following section, we describe the process of tuning and targeting Sequoia applications to a machine.

4 Task Specialization and Tuning

Tasks are generic algorithms that must be specialized before they can be compiled into executable code. Mapping a hierarchy of tasks onto a hierarchical representation of memory requires the creation of task instances for all machine levels. For each instance, a code variant to run must be selected, target instances for each call site must be chosen, and values for tunable parameters must be provided.

One approach to specialization is to rely upon the compiler to automatically generate task instances for a target by means of program analysis or a heuristic search through a pre-defined space of possibilities. In Sequoia, the compiler is not required to perform this transformation. Instead we give the programmer complete control of the mapping and tuning phases of program development. A unique aspect of Sequoia is the *task mapping specification* that is created by the programmer on a per-machine basis and is maintained separately from Sequoia source. The left half of Figure 7 shows the information required to map `matmul` onto a Cell machine. The tunables have been chosen such that submatrices constructed by the instance `matmul_mainmem_inst` can be stored entirely within a single SPE's LS.

In addition to defining the mapping of a task hierarchy to a machine memory hierarchy, the mapping specification also serves as the location where the programmer provides optimization and tuning directives that are particular to the characteristics of the intended target. A performance-tuned mapping specification for `matmul` execution on a cluster is shown in Figure 8. The instance `matmul_cluster_inst` runs at the cluster level of the machine hierarchy, so the distribution of array arguments across the cluster has significant performance implications. The instance definition specifies that task argument matrices be distributed using a 2D block-block decomposition consisting of blocks 1024x1024 in size. The definition also specifies that the transfer of subtask arguments to the individual nodes should be double-buffered across `mappar` iterations to hide the latency of the transfers. As an additional optimization, `matmul_L2_inst` specifies that the system should copy the second and third arguments passed to `matmul::leaf` into contiguous buffers to ensure stride-1 access in the the leaf task.

Mapping specifications are intended to give the programmer precise control over the mapping of a task hierarchy to a machine while isolating machine-specific optimizations in a single location. Performance is improved as details in the mapping specification are refined. While an intelligent compiler may be capable of automating the creation of parts of a new mapping specification, Sequoia's design empowers the performance-oriented programmer to manage the key aspects of this mapping to achieve maximum performance.

5 Implementation

We have experimented with implementing Sequoia on two different platforms, a Cell blade system and a cluster of traditional PCs. When targeting either platform, our source-to-source Sequoia compiler emits C code that interfaces with a platform-specific Sequoia runtime. The input to the compiler is a Sequoia program and a mapping specification for the target machine. The C++ front-end provided by Elsa [McPeak and Wilkerson 2005] was modified to accept Sequoia language syntax and served as the front-end for our compiler.

```
instance {
 name    = matmul_cluster_inst
 task    = matmul
 variant = inner
 runs_at = cluster_level
 calls   = matmul_node_inst
 tunable   U=1024, X=1024, V=1024

 A distribution = 2D block-block (blocksize 1024x1024)
 B distribution = 2D block-block (blocksize 1024x1024)
 C distribution = 2D block-block (blocksize 1024x1024)

 mappar loop-partition  = grid 4x4
 mappar software-pipeline = true

}
instance {
 name    = matmul_node_inst
 task    = matmul
 variant = inner
 runs_at = node_level
 calls   = matmul_L2_inst
 tunable   U=128, X=128, V=128
}
instance {
 name    = matmul_L2_inst
 task    = matmul
 variant = inner
 runs_at = L2_cache_level
 calls   = matmul_L1_inst
 tunable   U=32, X=32, V=32

 subtask arg A = copy
 subtask arg B = copy
}
instance {
 name    = matmul_L1_inst
 task    = matmul
 variant = leaf
 runs_at = L1_cache_level
}
```

Figure 8: A tuned version of the cluster mapping specification from Figure 7. The cluster instance now distributes its working set across the cluster and utilizes software-pipelining to hide communication latency.

The following subsections describe key points of our compiler/runtime implementation and summarize program optimizations conducted by our system to achieve the results discussed in Section 6.

5.1 Cell Compiler and Runtime

The execution of Sequoia programs on Cell utilizes both the chip's PowerPC and SPE cores. Inner task instances assigned to the machine's main memory level are executed by the PowerPC core and instances corresponding to the LS level of the hierarchy are executed by the SPEs. To facilitate this partitioning, the Sequoia compiler emits two sets of C output files. The first, corresponding to main memory tasks, is compiled for PowerPC core execution using GCC 3.4.1. Code generated for LS-mapped tasks is compiled for the SPEs using IBM's XLC. Linking all LS-mapped operations into a single SPE binary often results in a code text region exceeding the SPE's 256KB local store. To minimize code footprint, our compiler separates the binaries from an application's set of leaf tasks into a series of SPE overlays. We do not attempt to analyze the code within leaf tasks to partition large leaf tasks into multiple overlays.

Our Cell runtime system is event driven. A single thread runs on each SPE for the lifetime of the application and is notified of work by the PowerPC core via mailbox messages. For example, calling an LS-mapped task generates a work request sent to a SPE from the PowerPC core. Once this notification is received, a lightweight Sequoia SPE runtime library loads the code overlay corresponding to the requested task and initiates the appropriate transfers of argument data via asynchronous DMA operations.

Our system leverages the structure of Sequoia programs to efficiently utilize the Cell architecture. Limiting communication to task parameter passing enables the use of SPE initiated bulk DMA transfers that make maximal use of memory bus bandwidth and are overlapped with computation when parallel tasks exist. Due to Sequoia task isolation, synchronization between processing cores is only required on task boundaries, therefore events in our system correspond to large operations such as task calls or the execution of entire sequences of tasks within mapping constructs. For example, when a SPE is notified of the start of a `mappar` loop, it independently executes all subtasks from it's assigned section of the iteration space before synchronizing with the other cores. As a result, runtime overhead is minimal when running Sequoia applications on Cell.

5.2 Cluster Runtime

Implementing the Sequoia runtime for a cluster of workstations presented two key challenges: supporting arbitrary mappings of Sequoia's hierarchical task parallelism onto a collection of cluster nodes and implementing the virtual level abstraction required by inner tasks assigned to the cluster level of the machine.

Responsibility for executing inner tasks is not replicated across the cluster. Instead a single node executes a Sequoia program beginning at the root of the task hierarchy until parallelism is encountered. At this point, notification is sent to additional cluster nodes to begin performing subtasks in parallel. In this way control and synchronization interactions between nodes in the cluster runtime, albeit implemented via MPI messages instead of on-chip signaling, are similar to those orchestrated by the Cell runtime between the PowerPC and SPE cores. Since not all algorithms expressible in Sequoia decompose immediately into a large number of parallel operations, multiple levels of parallel task decomposition might occur before all cluster nodes are actively performing computation. To support this level of generality, a node that receives notification to begin work on a task may subsequently notify additional idle nodes to begin executing as the task decomposes into smaller parallel subtasks.

Sequoia's design made the implementation of a cluster-wide virtual address space feasible without significant performance penalties. Since Sequoia inner tasks may not access array elements directly and are only permitted to refer to blocks of arrays when calling subtasks, a fully general implementation of distributed shared memory was not required. Instead, the runtime need only track the distribution of blocks of arrays across the cluster, and move blocks to the appropriate node when they are passed as arguments to node-level subtasks in a Sequoia program. Since these transfers involve large blocks of data and occur with course granularity, the overhead of the virtual level abstraction is small. In our implementation, data communication is implemented via non-blocking MPI messages managed by a separate runtime thread on each node. The program's mapping specification describes how virtual-level arrays are partitioned across the cluster nodes although a more sophisticated runtime might seek to alter this distribution dynamically to improve performance.

Sequoia task isolation and CBVR parameter passing semantics minimize the amount of network communication performed by Sequoia applications. As stated previously, the system transfers all task argument data to a node *en masse* upon task call. When a node-level task begins, all data it manipulates is resident locally; no further communication is necessary for the duration of the task. Only upon task completion are output arguments returned to the appropriate locations in the cluster. As a result, our example programs generate efficient large MPI messages and feature communication patterns similar to those of well-written MPI applications. As in the Cell runtime, the cluster runtime attempts to hide transfer latencies by overlapping communication with the execution of parallel tasks.

5.3 Program Optimizations

As described above, Sequoia programs are constrained to exhibit properties that lend themselves to efficient execution on modern parallel architectures where efficient software management of the bandwidth hierarchy is critical to performance. Even so, our compiler and runtime systems perform additional optimizations that are required for Sequoia applications to perform competitively with platform-specific hand-tuned code on our target machines.

The most important of these optimizations is the efficient implementation of Sequoia's CBVR parameter passing semantics. CBVR semantics are natural when data must be physically copied between memory modules on a task call, such as between main system memory and a SPE LS on Cell, or between nodes of a cluster. However, when parent and child tasks are resident in the same level of the memory hierarchy, copying data in and out of task address spaces is both wasteful of memory and detrimental to performance. We detect

SAXPY	BLAS L1 saxpy performed on 32 million word vectors
SGEMV	BLAS L2 sgemv using a 8192x4096 matrix
SGEMM	BLAS L3 sgemm with matrices of size 4096x4096
ITERCONV2D	15 successive iterations of convolution of a 9x9 filter with a 8192x4096 input signal obeying non-periodic boundary conditions.
FFT3D	Discrete Fourier transform of a complex 256^3 dataset. Complex data is stored in struct-of-arrays format.
GRAVITY	An $O(N^2)$ N-body stellar dynamics simulation on 8192 particles for 100 time steps. We are using Verlet update and the force calculation is acceleration without jerk [Fukushige et al. 2005].
HMMER	Fuzzy protein string matching using Hidden Markov Model evaluation. The Sequoia implementation of this algorithm is derived from the formulation of HMMER-search for graphics processors given in [Horn et al. 2005] and is run on a large fraction of the NCBI non-redundant database.

Table 2: Applications implemented in Sequoia

and remove such copies in Sequoia programs and in these cases rewrite array references in subtasks to be relative to the original data in the parent task's address space. This analysis is performed statically during compilation for Cell-targeted programs and dynamically at runtime when executing on a cluster (higher latencies of cluster operations make this feasible). This optimization can be overridden by the programmer via the mapping specification if copies within an address space are desired, for example, to yield stride-1 access to array data in leaf tasks.

Other optimizations performed by the Sequoia compiler or runtime systems include task-granularity software pipelining to hide memory latency, and loop-invariant code motion to better exploit producer-consumer locality when subsequent tasks make use of the same arguments.

6 Evaluation

In this section we evaluate the performance of Sequoia applications on a prototype Cell blade system and on a cluster of workstations. Our IBM BladeCenter is configured with a prototype blade containing dual-2.4GHz Cell processors sharing bandwidth to 512MB of system memory. On this machine we scaled Sequoia applications from 1 to all 16 SPEs available across both processors. Our cluster setup consists of 16 nodes each with 2.4GHz Intel P4 Xeon processors and 1GB of main memory. Inter-node communication is performed via a Mellanox PCI-X 4X Cougar Infiniband HCA interconnect. The benchmark applications used in our initial experiments are described in Table 2. All programs operate on single-precision values. Notice that this set of benchmarks features widely used algorithms that are highly regular in control flow and data access, making them a good fit for the Sequoia programming model. An evaluation of Sequoia's utility for more complex irregular algorithms is not performed in this paper and is the subject of ongoing work.

	Cell 8 SPE	**Cell 16 SPE**	**Cluster (16 nodes)**	
			Pre-distrib	Overall
SAXPY	3.9 (22GB/s)	4.0 (22.1GB/s)	3.6	0.1
SGEMV	9.8 (18GB/s)	11.0 (20.5GB/s)	11.1	0.2
SGEMM	80.6	160.7	97.9	72.5
ITERCONV2D	62.8	119.4	27.2	19.9
FFT3D	32.3	40.2	6.8	1.98
GRAVITY	73.1	125.2	50.6	50.5
HMMER	9.9	19.1	13.4	12.7

Table 3: Application performance (GFLOP/s) on a single Cell blade (8 SPEs), dual-Cell blade (16 SPEs), and a 16 node cluster. Cluster performance is given with (pre-distrib) and without (overall) datasets pre-distributed across the cluster. Bandwidth is reported for severely memory bound benchmarks.

6.1 Sequoia Performance

Our applications utilize the highest quality leaf task implementations available. If kernel libraries could be obtained, such as FFTW and the Intel MKL for PCs, or the IBM SPE matrix library for Cell, we call these libraries from Sequoia leaf tasks. In all other situations we carefully hand-tuned leaf task implementations using either SSE2 or Cell SPE intrinsics. The ability to leverage existing libraries or highly-tuned platform-specific kernels is a feature of the Sequoia programming model.

Table 3 reports the raw performance (GFLOP/s) of each application running on a single Cell (8 SPEs), on both Cell processors in our blade (16 SPEs), and on a cluster of 16 nodes. Speedup relative to single SPE and single cluster node performance is plotted in Figures 9 and 10. In Table 3, we report performance including and excluding the time to initially distribute datasets across cluster nodes. In our experiments, non-performance-critical application code is written as a sequential C program and Sequoia inner tasks perform each benchmark's core algorithms. Performance measured when including distribution time is indicative of speedup from simply replacing a sequential target machine with a cluster. In a more realistic scenario, data is pre-distributed across nodes as part of a larger Sequoia application and for this reason we also report performance that neglects this cost. The ensuing discussion and the results graphed in Figures 10 and 11 do not include the time to distribute datasets prior to launching the root-level Sequoia task but do incorporate the cost of any data reshuffling that occurs once this task begins.

Our results are competitive with existing implementations of similar algorithms. The greatest raw performance is demonstrated by **SGEMM**, which reaches a rate of over 160 GFLOP/s running on two Cells and nearly 100 GFLOP/s on the cluster. Our implementation of **FFT3D** on the Cell (8 SPE) achieves a rate of 32.3 GFLOP/s, a result comparable to the hand-tuned FFT implementation described by IBM [Chow et al. 2005]. **GRAVITY** on a single Cell performs 3 billion interactions per second, exceeding the 2.3 billion interactions per second realized by the custom hard-

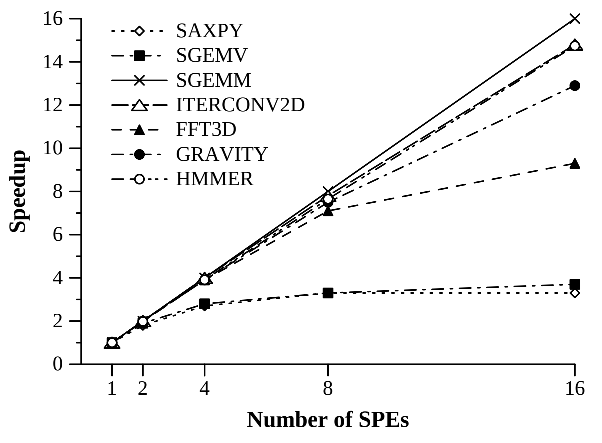

Figure 9: Speedup of Sequoia programs on a dual-Cell blade.

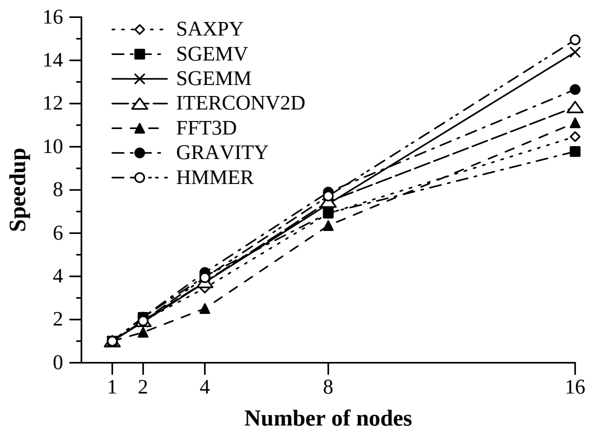

Figure 10: Speedup of Sequoia programs on a cluster of PCs.

ware of GRAPE-6A [Fukushige et al. 2005] although numerical precision differences exist between the implementations (GRAPE-6A uses a combination of fixed and floating point calculations at varying precisions; our computations operate on 32-bit floating point values). Lastly, our Cell implementation of **HMMER** is 5% faster than the implementation presented in [Horn et al. 2005] measured on an ATI 1900XT graphics processor, which was demonstrated to outperform hand-tuned versions of the algorithm on traditional CPUs.

The **SAXPY** and **SGEMV** benchmarks do not scale well to multiple Cell SPEs because they are bandwidth bound. Although the lack of computation and data locality in these benchmarks prevents efficient utilization of available processing units, as shown in Table 3 these Sequoia programs access data at over 20 GB/s, near the peak bandwidth observed on the machine. Since distributing these computations across a cluster yields more aggregate bandwidth than on a single node, **SAXPY** and **SGEMV** scale favorably on the cluster if data is pre-distributed. When the data is not pre-distributed, these benchmarks are limited by cluster interconnect bandwidth.

A key goal of the Sequoia language is to encourage the development of bandwidth efficient programs that maximize processing unit utilization. Figure 11 provides a breakdown of each application's execution into time spent computing in leaf tasks, time waiting on requests from memory, and time performing overhead operations such as waiting at barriers or executing Sequoia runtime logic. On both the Cell blades and on the cluster, Sequoia code utilizes processing units well. Other than the bandwidth-bound **SAXPY** and **SGEMV** tests, and **FFT3D**, which places heavy load on the memory system by conducting transposes of large 3D datasets, the applications rarely require processing units to idle waiting on data from memory.

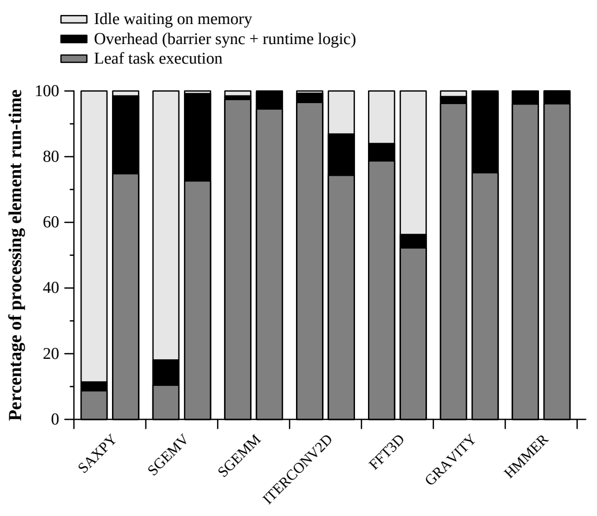

Figure 11: Execution time breakdown for each benchmark when running on a single Cell processor (left bar) and on a 16 node cluster (right bar).

6.2 Sequoia Portability

Sequoia applications are efficient without requiring Sequoia code to be tailored in detail to the requirements of the machines. With the exception of **FFT3D**, no changes were made to Sequoia source to run our benchmarks on either platform. Although the same **FFT3D** Sequoia source ran on both targets, an additional variant was added to minimize internode communication on the cluster. Sequoia variants easily supported this algorithmic change to improve cluster performance without impacting or cluttering parts of the code used on Cell.

Once a Sequoia program was running on one target, porting the application to a second platform required the programmer to perform two changes. First, existing leaf tasks were replaced with implementations specific to the new target machine. Hand-written leaves required a port from SSE2 to Cell SPE intrinsics (or vice-versa), however when libraries were used as leaf implementations, this change was trivial. For example, a one line modification to the **SGEMM** leaf task was required to use the Intel MKL implementation of matrix multiplication instead of the IBM-provided matrix library for Cell. Note that no port is required if leaf tasks are written entirely within Sequoia, such as the implementation of `matmul::leaf` given in Figure 4. The second aspect of porting Sequoia programs involves creating a mapping specification for the new target. The primary difference between the Cell and cluster mapping specifications is the selection of tunable parameter values (recall examples of Cell and cluster mapping files from Figure 7). Using Sequoia, a blocking and decomposition strategy is implemented fully on one platform and is then quickly transferred to the second target machine by changing the scale and parameters of the decomposition. A frequent inclusion in cluster mapping files is information about the distribution of global arrays across the machine's virtual root level. This information is not applicable to application tuning on Cell since a virtual level does not exist.

7 Related Work

Automatic restructuring of programs to improve data locality is possible for affine programs [Lim et al. 2001]. While this type of analysis aims to tackle similar performance goals as our work, it does not currently provide mechanisms for distributed systems and rich memory hierarchies, nor does it work with the many non-affine programs available.

There have been many attempts to incorporate explicit data locality into parallel programming models. Split-C [Culler et al. 1993], Co-Array Fortran [Numrich and Reid 1998], UPC [Carlson et al. 1999], and Titanium [Yelick et al. 1998] present a single program address space, but seek to minimize horizontal communication between processors by designating memory that is local or remote to each program thread. Stream processing languages [Mattson 2002; Buck et al. 2004] also build upon a two-tiered memory model [Labonte et al. 2004], choosing to differentiate between on and off-chip storage. Modern parallel language efforts [Charles et al. 2005; Callahan et al. 2004; Allen et al. 2005] support locality cognizant programming through the concept of distributions (from ZPL [Deitz et al. 2004]). A distribution is a map of array data to a set of machine locations, facilitating a single program namespace despite execution on nodes with physically distinct address spaces. Distributions fail to describe movement of array data up and down the memory hierarchy and are not applicable when arrays are not stored across distributed memories. Our decision to use a hierarchical memory model brings explicit control of communication between nodes and within a node into a common framework.

Previous efforts to model memory hierarchies include the Uniform Memory Hierarchy Model (UMH) [Alpern et al. 1994], which abstracted uniprocessor machines as sequences of memory modules of increasing size. The Parallel Memory Hierarchy Model (PMH) [Alpern et al. 1993] extended this abstraction to parallel architectures by modeling machines as trees of memories. Historically, interest in non-uniform memory access models has been motivated by the analysis of algorithm performance [Jia-Wei and Kung 1981; Vitter 2002]. Instead, we view hierarchical memory as a fundamental aspect of our programming model required to achieve both performance and portability across a wide range of architectures. As stated in Section 2, our design is influenced by the idea of space-limited procedures [Alpern et al. 1995], a methodology for programming machines modeled using the PMH model.

Hierarchically Tiled Arrays (HTA) [Bikshandi et al. 2006] accelerate existing sequential languages with an array data type expressing multiple levels of tiling for locality and parallelism but permit arbitrary element access. As with array distributions, the HTA approach specifies locality by annotating a data type which is less flexible and less portable than Sequoia's approach of using task composition.

The Chameleon [Alverson and Notkin 1993] and CHORES [Eager and Jahorjan 1993] systems also bear similarities to Sequoia in their runtime-based approach to dividing computation into hierarchies of simple operations called chores. These systems were intended to ease parallel programming on shared memory systems, and do not deal with issues such as explicit naming of working sets and address space isolation that are important in the context of our target systems.

Sequoia tasks are a generalization of stream programming kernels [Mattson 2002; Buck et al. 2004]. Tasks and kernels share similarities such as isolation, a local address space, and well specified working sets, but significantly differ in the ability of tasks to arbitrarily nest. Task hierarchies facilitate explicit expression of many levels of locality and parallelism that stream compilers have struggled to find via automatic analysis.

Sequoia's control flow when encountering a parallel mapping of subtasks resembles the thread-less abstraction of concurrency in Cilk [Blumofe et al. 1995], X10 [Charles et al. 2005], Chapel [Callahan et al. 2004], and Fortress [Allen et al. 2005]. Sequoia control flow is constrained in comparison to these languages since the calling task cannot proceed until all subtasks complete (similar to common usage of OpenMP [Dagum and Menon 1998] loops). Cilk introduced a generic concurrency model to facilitate the implementation of sophisticated dynamic scheduling of workloads. We leverage the flexibility of this style of control flow to achieve portability goals.

It is well known that divide-and-conquer strategies lead to algorithms that exhibit high levels of locality [Gustavson 1997]. Cache-oblivious algorithms [Frigo et al. 1999; Frigo and Strumpen 2005] make provably efficient use of a machine's memory hierarchy without regard to the particular size or number of hierarchy levels. The portability of the cache-oblivious approach comes at the cost of lost constant factors in performance and added complexity of expressing algorithms in a cache-oblivious manner. Sequoia algorithms are machine independent and are often written in a divide-and-conquer style, however, they may explicitly refer to the sizes and number of levels in the machine's hierarchy so that decomposition occurs only when crossing memory hierarchy boundaries. Sequoia provides portability across machines featuring widely varying communication mechanisms and rich memory hierarchies by allowing the programmer to describe multiple address spaces, while cache-oblivious algorithms written using traditional languages do not.

A different approach to gain memory hierarchy efficiency is to rely on domain-specific libraries. Meta-compilation of parameterized algorithms automatically tunes such libraries to machines by a heuristic-guided search through a domain-specific parameter space [Whaley et al. 2001; Frigo 1999]. Compiler technology that relies on exposing library semantics [Guyer and Lin 1999; Kennedy et al. 2001] can be used to further enhance performance by optimizing across tuned library calls. Task parameterization via Sequoia tunables and task variants generalizes meta-compilation, as it implicitly defines a parameter space for every Sequoia application. Search techniques can potentially be employed to automatically generate Sequoia mapping files for a new architecture, and a Sequoia compiler could perform global optimizations across tasks.

8 Discussion and Future Work

The Sequoia programming model is a pragmatic approach to reconciling the conflicting goals of performance and portability. Improving the productivity of the performance-conscious programmer requires placing performance critical aspects of the machine under explicit control. At the same time, to ensure portability, the separation of algorithmic expression and machine-specific tuning remains fundamental in Sequoia's design. The explicit expression of communication, the movement of data through the memory hierarchy, parallel computation, and the definition of isolated working sets are achieved through a single abstraction, the task. Sequoia provides first-class language primitives for structuring computations as hierarchies of tasks, a general way of performing the locality improving optimizations already required of programmers tuning for performance. Using Sequoia instead of ad-hoc techniques or platform-specific communication intrinsics, we demonstrated programs running efficiently on two very different exposed-communication architectures, a Cell blade and a cluster of workstations.

Sequoia remains in the early stages of development and although our initial results with important, but highly regular, applications are encouraging, our current designs will certainly need to overcome a number of apparent limitations. Most importantly, it is unclear if the minimal set of primitives Sequoia provides can sufficiently express more dynamic applications where the data computations access cannot be concisely described a priori in the form of a task argument list. Additionally, we have not critically analyzed how Sequoia's task mapping abstractions will generalize to support classes of applications where work is generated dynamically based on the results of previous computations. Similarly, the implications of inner tasks' inability to access array data directly have not been thoroughly explored.

While the existence of Sequoia mapping specifications facilitates the separation of tuning from algorithmic implementation, improvements on our current mechanisms are required to ease the burden of managing both Sequoia code and the separate mapping specification.

Future work on Sequoia will attempt to address these issues without sacrificing the performance of the large class of regular algorithms that benefit from Sequoia's features. In addition, we wish to further investigate the portability of the Sequoia language by adapting the compiler and runtime system to more traditional multicore and multithreaded processors as well as to larger systems comprised of higher numbers of processing elements.

References

AHO, A., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

ALLEN, E., CHASE, D., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE, G., AND TOBIN-HOCHSTADT., S., 2005. The Fortress language specification version 0.707. Technical report. Sun Microsystems.

ALPERN, B., CARTER, L., AND FERRANTE, J. 1993. Modeling parallel computers as memory hierarchies. In *Proc. Programming Models for Massively Parallel Computers*.

ALPERN, B., CARTER, L., FEIG, E., AND SELKER, T. 1994. The uniform memory hierarchy model of computation. *Algorithmica 12*, 2/3, 72–109.

ALPERN, B., CARTER, L., AND FERRANTE, J. 1995. Space-limited procedures: A methodology for portable high performance. In *International Working Conference on Massively Parallel Programming Models*.

ALVERSON, G. A., AND NOTKIN, D. 1993. Program structuring for effective parallel portability. *IEEE Trans. Parallel Distrib. Syst. 4*, 9, 1041–1059.

BIKSHANDI, G., GUO, J., HOEFLINGER, D., ALMASI, G., FRAGUELA, B. B., GARZARN, M. J., PADUA, D., AND VON PRAUN, C. 2006. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 48–57.

BLUMOFE, R., JOERG, C., KUSZMAUL, B., LEISERSON, C., RANDALL, K., AND ZHOU, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3, 777–786.

CALLAHAN, D., CHAMBERLAIN, B. L., AND ZIMA, H. P. 2004. The Cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, IEEE Computer Society, 52–60.

CARLSON, W. W., DRAPER, J. M., CULLER, D. E., YELICK, K., BROOKS, E., AND WARREN, K., 1999. Introduction to UPC and language specification. University of California-Berkeley Technical Report: CCS-TR-99-157.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 519–538.

CHOW, A., FOSSUM, G., AND BROKENSHIRE, D., 2005. A programming example: Large FFT on the Cell Broadband Engine.

CULLER, D. E., ARPACI-DUSSEAU, A. C., GOLDSTEIN, S. C., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., AND YELICK, K. A. 1993. Parallel programming in Split-C. In *Supercomputing*, 262–273.

DAGUM, L., AND MENON, R. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1, 46–55.

DALLY, W. J., HANRAHAN, P., EREZ, M., KNIGHT, T. J., LABONTE, F., AHN, J.-H., JAYASENA, N., KAPASI, U. J., DAS, A., GUMMARAJU, J., AND BUCK, I. 2003. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 35.

DEITZ, S. J., CHAMBERLAIN, B. L., AND SNYDER, L. 2004. Abstractions for dynamic data distribution. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, IEEE Computer Society, 42–51.

EAGER, D. L., AND JAHORJAN, J. 1993. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Trans. Comput. Syst. 11*, 1, 1–32.

FRIGO, M., AND STRUMPEN, V. 2005. Cache oblivious stencil computations. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, 361–366.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, 285.

FRIGO, M. 1999. A fast Fourier transform compiler. In *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, vol. 34, 169–180.

FUKUSHIGE, T., MAKINO, J., AND KAWAI, A. 2005. GRAPE-6A: A Single-Card GRAPE-6 for Parallel PC-GRAPE Cluster Systems. *Publications of the Astronomical Society of Japan 57* (dec), 1009–1021.

GUSTAVSON, F. G. 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev. 41*, 6, 737–756.

GUYER, S. Z., AND LIN, C. 1999. An annotation language for optimizing software libraries. In *Second Conference on Domain-Specific Languages*, 39–52.

HORN, D. R., HOUSTON, M., AND HANRAHAN, P. 2005. ClawHMMER: A streaming HMMer-search implementation. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, 11.

INTEL, 2005. Math kernel library. http://www.intel.com/software/products/mkl.

JIA-WEI, H., AND KUNG, H. T. 1981. I/O complexity: The red-blue pebble game. In *STOC '81: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, 326–333.

KAPASI, U., DALLY, W. J., RIXNER, S., OWENS, J. D., AND KHAILANY, B. 2002. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, 282–288.

KENNEDY, K., BROOM, B., COOPER, K., DONGARRA, J., FOWLER, R., GANNON, D., JOHNSSON, L., MELLOR-CRUMMEY, J., AND TORCZON, L. 2001. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel Distributed Computing 61* (December), 1803–1826.

LABONTE, F., MATTSON, P., BUCK, I., KOZYRAKIS, C., AND HOROWITZ, M. 2004. The stream virtual machine. In *Proceedings of the 2004 International Conference on Parallel Architectures and Compilation Techniques*.

LIM, A. W., LIAO, S.-W., AND LAM, M. S. 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 103–112.

MATTSON, P. 2002. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University.

MCPEAK, S., AND WILKERSON, D., 2005. Elsa: The Elkhound-based C/C++ Parser. http://www.cs.berkeley.edu/˜smcpeak/elkhound.

NUMRICH, R. W., AND REID, J. 1998. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum 17*, 2, 1–31.

PHAM, D., ASANO, S., BOLLIGER, M., DAY, M. N., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., SHIPPY, D., STASIAK, D., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEL, S., WENDEL, D., YAMAZAKI, T., AND YAZAWA, K. 2005. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*.

VITTER, J. S. 2002. External memory algorithms. In *Handbook of Massive Data Sets*, Kluwer Academic Publishers, Norwell, MA, USA, 359–416.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Computing 27*, 1–2, 3–35.

YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., AND AIKEN, A. 1998. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*.