Programming the Memory Hierarchy Revisited: Supporting Irregular Parallelism in Sequoia *

Michael Bauer
Computer Science Department, Stanford University
mebauer@cs.stanford.edu

John Clark
Computer Science Department, Stanford University
jpclark@stanford.edu

Eric Schkufza
Computer Science Department, Stanford University
eschkufz@cs.stanford.edu

Alex Aiken
Computer Science Department, Stanford University
aiken@cs.stanford.edu

Abstract

We describe two novel constructs for programming parallel machines with multi-level memory hierarchies: *call-up*, which allows a child task to invoke computation on its parent, and *spawn*, which spawns a dynamically determined number of parallel children until some termination condition in the parent is met. Together we show that these constructs allow applications with irregular parallelism to be programmed in a straightforward manner, and furthermore these constructs complement and can be combined with constructs for expressing regular parallelism. We have implemented spawn and call-up in Sequoia and we present an experimental evaluation on a number of irregular applications.

1. Introduction

For most of the past two decades clusters of single processor machines have been a very popular high-performance computing platform. These machines are typically programmed using a message passing library such as MPI [1] or a partitioned global address space (PGAS) language such as UPC or Titanium [2, 3]. Characteristic of both programming models is that the programmer is presented with a two-level memory hierarchy: memory is divided into a processor's local memory, where accesses are guaranteed to be relatively fast, and the global or remote memory of all the other processors in the cluster, accesses to which are likely to be slow. With the transition to multicore, however, clusters of multicore machines are becoming much more common, and these machines present at least three interesting levels of memory: the individual core, on-chip, and global memory. The Roadrunner petaflop supercomputer has four or five memory levels, depending on how one counts [4]. These recent, and likely lasting, changes in machine organization have led to interest in programming models for multi-level memory hierarchies that go beyond the two-level view of memory as simply local or global.

Sequoia is a programming system designed for machines with multi-level memory hierarchies. In previous work, the focus of Sequoia was on regular applications, and the issue of how to program irregular applications was explicitly left open [5]. In this paper we present programming constructs designed to support irregular computations on machines with multi-level memory hierarchies, and we describe an implementation of these ideas as extensions to Sequoia. The extensions are conservative in the sense that Sequoia

```
void task<inner> matmul( in    float A[M][P],
                         in    float B[P][N],
                         inout float C[M][N] )
{
  // Code to name submatrices of A, B, and C
  // called Ablks, Bblks, and Cblks, respectively
  // block sizes are given by U, V, and X

  // Compute all blocks of C in parallel.
  mappar (int i=0 to M/U, int j=0 to N/V) {
    mapseq (int k=0 to P/X) {
      // Invoke the matmul task recursively
      // on the subblocks of A, B, and C.
      matmul(Ablks[i][k],Bblks[k][j],Cblks[i][j]);
    }
  }
}
void task<leaf> matmul( in    float A[M][P],
                        in    float B[P][N],
                        inout float C[M][N] )
{
  // Compute matrix product directly
  for (int i=0; i<M; i++)
    for (int j=0; j<N; j++)
      for (int k=0; k<P; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

Figure 1. Dense matrix multiplication in Sequoia.

* This work was supported in part by grants from the Department of Energy's Predictive Science Academic Alliance Program (PSAAP), the Army High Performance Research Center (AHPCRC), and a generous grant of time on the Cerillos supercomputer through the Los Alamos National Lab's Institutional Computing Program.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

programs that do not use the new features perform exactly as before. Furthermore, the extensions complement Sequoia's existing support for regular parallel computations, allowing straightforward implementations of programs that require a mix of regular and irregular parallelism.

To set the stage, we briefly describe how a canonical regular application, dense matrix multiply, is implemented in Sequoia. An analysis of the assumptions underlying the constructs used in this example illustrates the limitations of Sequoia for irregular computations.

The key parts of a Sequoia program for matrix multiplication are given in Figure 1. This program defines two *tasks*, `task<leaf> matmul` and `task<inner> matmul`; `task` is a keyword and `matmul` is the task name, which has two implementations called `inner` and `leaf`. A task is the basic unit of computation and locality in Sequoia. Tasks are *isolated*: tasks cannot refer to global variables or take pointer arguments, and so they cannot directly refer to data in use by other tasks. Tasks communicate via parameters passed to and results returned from calls to other tasks. We refer to the caller as the *parent* task and the callee as the *child* (or *sub-*) task. Task parameter passing is copy-in, copy-out (i.e., call-by-value-result [6]).

During compilation, each task in the source program is assigned to a specific memory (and a specific processor) in the target machine. At runtime, all of a task's inputs and storage for its outputs are resident in the assigned memory. When a parent task and its children are assigned to different memories by the compiler, the child task call at runtime results in communication as the task call's arguments are copied to the child task's memory at the start of the task, and the results are copied back to the parent task's memory when the child terminates. Thus, task calls express the movement of data through the memory hierarchy.

Parallelism is expressed via `mappar`, a looping construct that declares each loop iteration can be executed independently of any other. In Figure 1, `task<inner> matmul` uses a `mappar` to compute each submatrix of the output array `C` in parallel. Inside the `mappar` the analogous, but sequential, `mapseq` construct accumulates the partial results for each subblock of `C` using a sequence of recursive calls to `matmul` on subblocks of `A` and `B`. The recursive call to `matmul` invokes either `task<inner> matmul` or, if we have proceeded far enough in the divide and conquer computation, `task<leaf> matmul`, which is represented here by a naive triply-nested loop. (Which variant is called depends on the number of memory levels of the target machine and is decided in a separate *mapping* phase by either the programmer or the compiler.) A snapshot of the running program for a three-level machine looks like the tree of task invocations shown in Figure 2. The program defines a tree-shaped task hierarchy, with large problems near the root and progressively smaller problems at lower levels of the tree until `task<leaf> matmul` is executed at the leaves of the computation. More background details of Sequoia are discussed in Section 2.

While Sequoia has other important parallel control constructs (e.g., reductions), for the purposes of discussing regular vs. irregular computation, we need only focus on the `mappar` statement in the `inner` variant of `matmul`. An important detail is an implied barrier at the end of the `mappar`: All of the parallel instances of the statement inside the `mappar` must complete before execution continues to the statement after the `mappar`. The example in Figure 1 illustrates two assumptions underlying `mappar`:

1. *The working set of of each subtask is computed in advance.* Task isolation implies that all of the inputs to a task must be known at the time a task is invoked. Furthermore, to guarantee that the child task has sufficient space to complete its computation, at least an upper bound on the size of a task's result must also be known before task invocation.

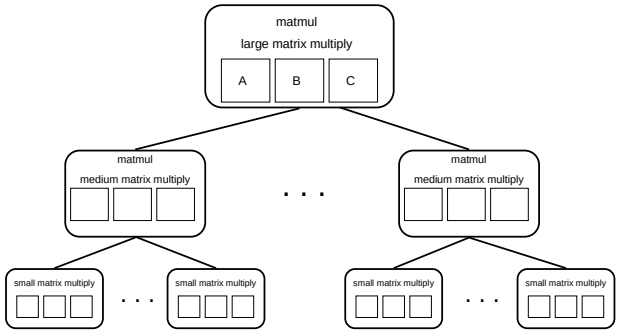


Figure 2. An example task hierarchy for a machine with three levels of memory.

2. *The number of subproblems is computed in advance.* The number of parallel subtasks is fixed on entry to the `mappar`. If the subtasks return results to the parent that represent new parallel work, that work can only be done after the `mappar` is complete.

For our purposes, these two properties can be taken as the definition of a regular parallel computation; we consider a problem where either the number of subproblems is unknown, or the input/output size of the subproblems is unknown in advance, to be irregular. For assumption (1), there are two common situations in which the working set of a task is not known in advance:

- There is a large input data set, but the task only uses a small portion of it, and it is necessary for performance reasons to send the task only that small subset. Unfortunately, the task computes the subset it needs, so the input working set is unknown before task invocation. Algorithms that benefit from caching often have this structure. Effectively, we would like the parent to function as a cache for the child, allowing the child to compute what it needs and then pull additional data from the parent, but isolation prevents this pattern from being expressed directly.
- The output is potentially large relative to the input, so much so that the problem size a task can handle is limited by the size of the output (which must fit in the memory level allocated to the task), not the input. Some search problems where one or a few subtasks may return the entire answer are in this category, as is any problem that has the character of decompressing the input. We would like child tasks to be able to off-load partial results to the parent and then continue executing, thereby enabling children to work on (usually more efficient) larger problem sizes, but again task isolation requires that the entire output is kept at the child and returned on task completion.

The dual problems of unknown input and unknown output size affect not just performance but also how programs are written. Consider a problem in which the input for the natural task one would like to write is unknown. The only way to express this in Sequoia is to write two tasks. The first task A computes the data that is needed and returns an output describing that data request to the parent. The parent then launches a second task B with all of the input data that finishes the job. Essentially, one ends up writing an event-driven system, where events are requests to the parent for more data and the tasks are stages of computation between events. Unfortunately, the problem is even somewhat worse: not only must we program in an event-driven style, but each stage is synchronized by the barrier at the end of a `mappar`, meaning that no type B task can begin until all type A tasks have completed. Of course, if there is more than one unknown input we may need to have more

than two stages in our pipeline of events, further compounding the programming problem.

One way to solve this problem is to adopt programming constructs such as threads or the processes in PGAS languages that are not isolated—for example, threads can share arbitrary state with one another. Isolation, however, is a great property to have, as it dramatically simplifies both program reasoning and the compiler scheduling problem in multi-level machines. What we desire is an explicit, but limited, way to "break out" of isolation.

To relax assumption (1) when needed, we propose the ability of child tasks to *call-up* to the parent. That is, just as parents can invoke tasks on the children (a *call-down*), we add the symmetric ability for children to invoke tasks on parents. Call-up violates complete isolation, but it does so explicitly and, as we shall see, has a natural semantics. Note that with call-up child tasks are still isolated from each other; the change is in the relationship of a child task to its parent task.

Turning to assumption (2), consider a situation where the number of parallel subtasks is initially small, but each subtask generates more jobs of the same kind. This is a common pattern; every worklist algorithm has this flavor, where there is a set of jobs to do, and each job may generate new jobs. Using only `mappar`, a worklist algorithm can only be executed in phases, where all the jobs on the worklist at the beginning of phase i must complete before we can begin executing any of the jobs generated during phase i. If a particular phase has fewer jobs than processors, or about the same number of jobs as processors but the jobs have high variance in execution time, utilization will be unnecessarily low.

To mitigate this problem we would like a parallel control construct that does not fix in advance the number of parallel subtasks it can invoke. We propose *spawn*, a construct that launches an unspecified number of subtasks until some termination condition is met in the parent. Where `mappar` is analogous to a parallel bounded `for` loop, `spawn` is analogous to a parallel `while` loop.

We stress that for regular or nearly regular computations the constructs provided by the existing Sequoia language are expressive and programs are both very portable and efficient [5, 7, 8]. However, previous work on programming multi-level machines and on Sequoia in particular does not address the problem of programming irregular applications, which is the focus of this paper. Our main contributions are:

- we propose two constructs, *call-up* and *spawn*, for programming irregular applications on hierarchical memory machines (Section 3);
- we give a formal operational semantics for a core calculus including call-up and spawn, showing how these constructs fit into an overall language design (Section 4);
- we evaluate an implementation (Section 5) of our proposal on several irregular applications on a cluster, an SMP, and a cluster of SMPs (Section 6).

Section 7 elaborates on related work, and Section 8 summarizes with a discussion of design alternatives and future work.

2. Hierarchical Memory

Before presenting our proposal, this section gives some additional background on Sequoia. Sequoia requires the programmer to target an abstract parallel machine that is a tree of distinct memory modules, a representation that extends the Parallel Memory Hierarchy [9]. Data transfer between memory modules is conducted via (potentially asynchronous) block transfers. Data transfer occurs at all levels of the hierarchy through task arguments, which may be declared `in` (read only), `out` (write only), or `inout` (a task argument that may be both read and written; see Figure 1 for examples). Each task defines a namespace that exists entirely within one memory in the memory hierarchy. Unlike the original Sequoia proposal [5], in our approach there are no restrictions on the computation a task may perform within a level (see Section 3). There may be multiple versions, called *variants* of the same task, allowing, for example, different implementations of the base and inductive cases in divide-and-conquer algorithms (c.f., the `inner` and `leaf` versions of `matmul` in Figure 1).

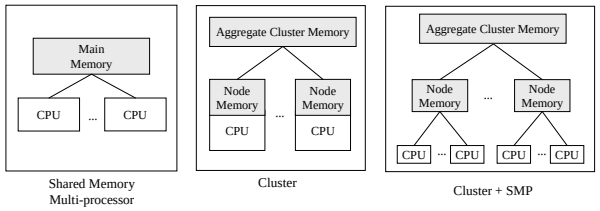


Figure 3. Hierarchical memory machines.

The computation tree described by a Sequoia program is abstract. Neither the width (the number of parallel subtasks) nor the depth of the tree is specified in the program. The communication protocols used to move data through the machine are also abstracted through parameter passing.

Parallel machines are also modeled as trees. A *machine description* defines for a target machine the number of levels and the number and size of memories at each level, among other details. The tree model provides a simple abstraction for programmers to reason about, but there are important non-tree topologies used in practice, particularly a cluster of nodes where peers in the cluster communicate directly with each other rather than through a parent. Following [5], we model clusters using *virtual levels* that do not correspond to any single physical memory. A cluster virtual level is the sum of all the node memories in the cluster; this forms a distinct address space which is separate from the individual node memories, which are the children of the virtual level. Moving data from the virtual level to a particular child node corresponds to communication across the cluster, as data stored somewhere in the virtual level's physically distributed address space is moved to be local to one node. Figure 3 illustrates three typical hierarchical machines: a shared memory multiprocessor (SMP), a cluster with a virtual level that aggregates all of the local node memories, and a common three-level machine, a cluster of SMPs.

The compilation problem for Sequoia is to map the unbounded, abstract task tree on to the fixed, definite machine tree. A *mapping* assigns tasks in the program to specific levels of the target machine's hierarchy. Mappings are either written by hand in a separate mapping language or are generated automatically by an auto-tuner. Previous work suggests auto-tuning is always preferable [10], but the mappings used in this paper are hand-written. Beyond the mapping, the compiler performs important optimizations such as coalescing data transfers, copy elimination across levels of the memory hierarchy, and software pipelining of compute and communication between adjacent memory levels [7]. The compiler also manages the tedious task of generating and compiling code for each level of the hierarchy; as the machines can be heterogeneous multiple distinct platform compilers may be involved. A simple, portable run-time interface abstracts away the actual communication mechanism between different levels of the memory hierarchy (e.g., MPI calls, DMAs, simple loads and stores to RAM, etc.) [8].

3. Supporting Irregular Applications

This section gives an informal overview of our constructs for irregular parallelism in programs for hierarchical memory machines;

Section 4 presents a more formal treatment. We illustrate our ideas using a simple work-list algorithm, shown in Figure 4. We postpone an explanation of what this example actually does until after we have presented the programming constructs.

Unlike the original Sequoia design, we allow general object-oriented (C++) code in any task at any level of the memory hierarchy. Thus, tasks may create and use objects and build pointer data structures. These are confined to within the task however; arguments to subtasks, with one exception, cannot be pointers or references, or structures (e.g., arrays) that contain pointers or references.

The one exception is that a task may take a *parent object*, indicated using the `parent` type qualifier, as an argument. In Figure 4, the `doWork` method (both `inner` and `leaf` variants) takes a pointer to a worklist object `wl` passed from the calling task (not shown). Parent objects may have no public fields. The only operation permitted on a parent object is to invoke its public methods. A parent object method invocation is a *call-up*: the method executes in the *parent*'s address space, not the address space of the task invoking the parent object method. In Figure 4, in the leaf variant of `doWork` the method invocation `wl->addWork(newWork)` adds `newWork` to the worklist `wl` maintained in the parent task's memory level. Similarly, earlier in the same task the parent method call `wl->getWork(work)` pulls work to do off of the work list stored in the parent. A call-up is synchronous: like a regular function call in a standard language, the child task is suspended until the call-up returns its result from the parent.

Call-up introduces concurrency into the Sequoia programming model, because multiple children may attempt to execute a call-up in the parent's address space simultaneously. We enforce the following simple semantics. During subtask execution the parent blocks, meaning it does not perform any other computation until the subtask returns (or, in the case of a `mappar`, until all of the parallel subtasks return). Thus, while call-ups can modify data structures in the parent's heap, there are no races with the parent's execution. We also require that all call-ups execute atomically in some unspecified order in the parent. That is, concurrent call-ups from multiple children are always serializable in the parent's address space. Call-ups may actually be executed in parallel if there are sufficient resources and it is safe to do so, although our current implementation does not include any such optimization.

Consider the methods `getWork` and `addWork` in Figure 4. These methods are invoked only in a call-up and so always execute in the address space of the parent. (Note that these are `leaf` tasks, which with call-ups no longer means that they execute at the leaves of the machine hierarchy; it only means that these tasks have no subtasks.) Thus, while `getWork` and `addWork` modify the worklist data structure, there is no correctness issue because call-ups are atomic.

The parallel control construct *spawn* takes two arguments: a task call and a termination test. A `spawn` may launch any number of instances of the task call, and it may continue to launch new ones at any time during execution of the `spawn`. Note that every spawned subtask is identical, so `spawn` assumes the use of a call-up to retrieve different data for each subtask. A `spawn` terminates when (1) its termination test (evaluated in the address space of the parent) is true, and (2) all subtasks have terminated. Condition (2) is necessary. Consider the method `doWork` in Figure 4, which spawns worker tasks that add and remove jobs from a worklist, and which terminates when the worklist is empty and there are no worker tasks still executing. The worklist may be empty but if there is a subtask running it may insert one or more new tasks into the worklist; thus, we need to know that subtasks cannot invalidate the termination test, which is done by requiring that all subtasks have completed.

We now briefly explain the worklist example. Initially, `doWork` spawns some number of worker tasks which all receive a pointer to the worklist in the parent's memory through a parent object.

```
void task<inner> Worklist::doWork(parent Worklist* wl) {
  spawn(doWork(wl), wl->isDone());
}
void task<leaf> Worklist::doWork(parent Worklist* wl) {
  // Get work[] (an array of size 1):
  int* work;
  wl->getWork(work);
  int unit = work[0];
  delete [] work;

  // Add work (work[0] new elements, each work[0]-1):
  if ( unit > 1 ) {
    int* newWork = new int[unit];
    for ( unsigned int i = 0; i < unit; i++ )
        newWork[i] = unit-1;
    wl->addWork(newWork);
    delete [] newWork;
  }
}
void task<leaf> Worklist::getWork(out int work[]) {
  work = new int[1];
  if ( list_.empty() ) // If the worklist is empty
      work[0] = 0;     //   send no work.
  else
      work[0] = list_.pop();
}
void task<leaf> Worklist::addWork(in int work[]) {
  for ( unsigned int i = 0; i <= work[0]; i++ )
      list_.push(work[i]);
}
bool task<leaf> Worklist::isDone() {
  return list_.isEmpty();
}
```

Figure 4. A paradigmatic worklist implementation.

Each task `doWork` first gets some work using a call-up of the worklist's `getWork` method, and then adds some number of jobs to the worklist using the worklist's `addWork` method. This example illustrates all of the irregular features discussed in Section 1: the parent acts as a cache for the children, holding the current worklist; the children produce varying amounts of output in the form of new jobs to be placed on the worklist; the `spawn` construct allows new work added to the worklist to be allocated to some child task without the need to first synchronize with all of the children.

There is considerable flexibility in the implementation of `spawn`. First, the runtime system is free to launch as many subtasks as necessary to keep the machine busy. Second, the runtime system can evaluate the termination predicate at any time, including while there are still child tasks running, to gain information about whether it is worthwhile to respawn terminated subtasks or not (the number of times the termination test is evaluated is unspecified, and so the test should be side-effect free). Our current runtime implementation of `spawn` prematurely tests the termination condition as part of a heuristic for determining when to respawn children (see Section 5). In addition, this runtime heuristic could be customized easily by the programmer to match specific applications.

We also enforce two restrictions on call-ups to avoid situations that are undefined or very difficult to compile well. First, a parent object only makes sense so long as the parent task instance that created it is executing; thus, a parent object may be used in any child (and more generally, any descendant) of the creating parent task, but may not escape (i.e., outlive) that parent task. The second restriction is that no call-down may occur within a call-up. That is, a method that is used as a call-up (i.e., invoked by a parent object) may have call-ups in its body but not ordinary task calls (call-downs). Allowing call-downs within call-ups would result in a difficult scheduling problem, as it would no longer be easy to

statically determine which tasks might run in parallel. Furthermore, despite considerable experience with `spawn`, we have yet to find an example where allowing a call-down within a call-up would be useful. Both restrictions are easily enforced statically by the type system.

4. Semantics

This section gives a formal treatment of call-up and `spawn`. There is a previously published Sequoia semantics [7] which, unfortunately, is not expressive enough to describe our extensions; the semantics presented here is very different. Like [7], however, our program executions work on *trees of memories*. Also following [7], we model a memory M as a function from names to values, so rather than manipulating addresses we use mnemonic variable names and $M(x)$ looks up the value of variable x in memory M. We use the standard notation $M[x \leftarrow a]$ to denote the memory that is identical to M except that the value a is stored at name x.

A given level of the memory hierarchy has a memory M, zero or more sub-machines $[T_1, \ldots, T_n]$, and two programs P_1 and P_2:

$$
\begin{array}{lcll}
T & := & \langle M, C, P_1, P_2 \rangle & \\
C & := & [T_1, \ldots, T_n] \quad n \geq 0 & \\
P & := & \texttt{Op}_M(A = f(B)) & \mid \ \texttt{If}_M(pred, P_1, P_2) \\
 & \mid & P_1; P_2 & \mid \ \texttt{Copy}_{M_i, M_j}(A, B) \\
 & \mid & \texttt{Mappar}_M(k = start : end, P) & \mid \ \texttt{Spawn}_M(P, pred) \\
 & \mid & \texttt{Up}_M(P) & \mid \ \texttt{wait} \\
 & \mid & \texttt{resume} & \mid \ -
\end{array}
$$

The program constructs are purposely limited to a core calculus to keep the semantics small and tractable: standard sequential constructs (primitive operations, if statements, statement sequencing), the Sequoia constructs (copying data between memory levels, `Mappar`, `Spawn`, and a call-up construct `Up`), and three operations needed by the semantics that do not appear in source programs (`wait`, `resume`, and `-`). Note that every operation that uses memory is subscripted with the memory level it accesses; a copy operation is subscripted with two levels, the source and destination memories, which are always adjacent levels in the hierarchy (i.e., parent and child memories). This semantics is at the level of our implementation's intermediate language, after source programs have been desugared and the number of memory levels (depth of the memory tree) and the number of child memories at each level have been made explicit in the program.

For a given *configuration* at a memory level $\langle M, C, P_1, P_2 \rangle$ there may be two programs executing: one *main task* and one call-up from the child memory level. This closely reflects our implementation, which on most platforms implements a memory level using one thread for the main task and another thread to service call-ups. The program "$-$" represents no program—i.e., an idle resource. Two special cases are the configurations $\langle M, C, P, - \rangle$ (or equivalently $\langle M, C, -, P \rangle$) and $\langle M, C, -, - \rangle$. The former represents a memory level with a main task but no active call-ups, the latter represents a memory with no scheduled computation at all; the memory is *idle*. No configuration has a call-up without also having a non-idle main thread.

Notably missing from the core calculus are task calls, which can be emulated by the other constructs. Given a task definition `task f(in a, out b) { P }`, a task call of `f` can be implemented by using copy operations to copy the `in` parameter to the corresponding formal `a` in the child memory, executing the body `P` of `f`, and then copying the `out` parameter back to `b` in the parent memory. For example,

$$
\begin{array}{l}
\texttt{Mappar}_{\texttt{M}_\texttt{i}}(\texttt{k} = 1 : \texttt{n}, \texttt{f}(\texttt{x}[\texttt{k}], \texttt{y}[\texttt{k}])) \equiv \\
\texttt{Mappar}_{\texttt{M}_\texttt{i}}(\texttt{k} = 1 : \texttt{n}, \texttt{copy}_{\texttt{M}_{\texttt{i}-1},\texttt{M}_\texttt{i}}(\texttt{a}, \texttt{x}[\texttt{k}]); \texttt{P}; \texttt{copy}_{\texttt{M}_\texttt{i},\texttt{M}_{\texttt{i}-1}}(\texttt{y}[\texttt{k}], \texttt{b})
\end{array}
$$

assuming operations in `P` are suitably annotated to read and write data in memory level $\texttt{M}_{\texttt{i}-1}$. Call-ups invoked on parent objects can similarly be expanded into a sequence of primitive operations that copy arguments from the child to the parent memory, execute the body of the call-up, and copy the result back to the child.

Table 1 gives a small step operational semantics for the core calculus. Each rule describes one step of execution: $\langle M, C, P_1, P_2 \rangle \rightarrow \langle M', C', P_1', P_2' \rangle$. The first three rules are for familiar statements: primitive operations $A = f(B)$, if statements (only the rule for a predicate that evaluates to *true* is shown; the symmetric rule for *false* is also standard), and statement sequencing. The interesting thing to note about these representative sequential statements is that they take place in one memory level, having no effect on their child memories. Notice that most of the rules work by executing the first statement P_1 in a sequence $P_1; P_2$ and transitioning to a configuration where P_2 remains to be executed. Thus, the rule for statement sequences simply rearranges statement sequences using the associativity of ";" to ensure the first statement is primitive and not itself a statement sequence. To guarantee statements are always part of a sequence (so that some rule will match) we assume programs are initially of the form $P; -$.

The copy operation comes in two flavors: copying data from parent to child and from child to parent. Note that copies in either direction are initiated by the children; on most architectures this is the more efficient arrangement.

A `Mappar` has two cases. If *start* $\leq$ *end* and there is an idle child, a fresh version of the `Mappar` computation can be launched in that child's memory. If *start* $>$ *end*, then the parent implements a barrier: the parent's main task blocks until all children are idle and the parent has no call-up to service, at which point the main task continues to the next statement. A `Spawn` is similar: a `Spawn` can launch a fresh copy of the parallel computation on an idle child, and if all children are idle, the parent has no call-up to service, and the termination predicate is true, the `Spawn` can terminate and the parent's main task moves on to the next statement. The semantics allows a choice when the termination condition evaluates to *true*: the `Spawn` may terminate (assuming the other conditions for termination are also met) or some children may be respawned instead. This semantics allows implementations maximum flexibility, though we expect that implementations will generally not respawn child tasks when the termination condition is *true*.

In summary, the main differences between a `Mappar` and `Spawn` are that the `Mappar` has a fixed number of instances to execute and each child is given a distinct portion of the work at invocation (represented by the value of k in the child memory in the [`Mappar`] rule), whereas the `Spawn` launches instances until the termination predicate is true. Thus, `Mappar` is like a `for` loop and `Spawn` is akin to a `while` loop.

A call-up $\texttt{Up}(\ldots)$ launches a computation on the parent if the parent is not currently executing another call-up (i.e., the parent's configuration is of the form $\langle M, C, P, - \rangle$). The program invoking the call-up (which may be the child's main task or another call-up that the child is handling) must block until the call-up completes, which is the purpose of inserting a `wait` in the child program and a `resume` at the end of the call-up program in the parent. The [`Resume`] rule restarts the child computation by removing the `wait` when the parent reaches the `resume`. Since only one child can execute at a time, there is always only one `wait` that a `resume` can match.

The [`Swap`] rule switches the order of the two programs in a configuration. All of the rules execute using only the third component of a configuration, so this rule has the effect of switching the active program between the main task and any call-up awaiting service. It is easy to prove (by induction on the length of an execution) that if a memory level has two programs neither of which is $-$, then the main task is always either at a `Spawn` or a `Mappar`; i.e., call-ups can only happen inside of `Spawn` or `Mappar`. The [`Spawn`]

$$\langle M_i, C, \texttt{Op}_{M_i}(A = f(B)); P, U\rangle \rightarrow \langle M_i[A \leftarrow f(B)], C, P, U\rangle \qquad \texttt{[Primitive Op]}$$

$$\frac{M_i(pred) = \mathit{true}}{\langle M_i, C, \texttt{If}_{M_i}(pred, P_1, P_2); P, U\rangle \rightarrow \langle M_i, C, P_1; P, U\rangle} \qquad \texttt{[If]}$$

$$\langle M_i, C, (P_1; P_2); P_3, U\rangle \rightarrow \langle M_i, C, P_1; (P_2; P_3), U\rangle \qquad \texttt{[Sequence]}$$

$$\begin{array}{l}\langle M_i, [\ldots, \langle M_{i-1}^j, C^j, \texttt{Copy}_{M_i, M_{i-1}}(A, B); P^j, U^j\rangle, \ldots], P, U\rangle \quad \rightarrow \\ \langle M_i[A \leftarrow M_{i-1}^j(B)], [\ldots, \langle M_{i-1}^j, C^j, P^j, U^j\rangle, \ldots], P, U\rangle\end{array} \qquad \texttt{[Copy Up]}$$

$$\begin{array}{l}\langle M_i, [\ldots, \langle M_{i-1}^j, C^j, \texttt{Copy}_{M_{i-1}, M_i}(A, B); P^j, U^j\rangle, \ldots], P, U\rangle \quad \rightarrow \\ \langle M_i, [\ldots, \langle M_{i-1}^j[A \leftarrow M_i(B)], C^j, P^j, U^j\rangle, \ldots], P, U\rangle\end{array} \qquad \texttt{[Copy Down]}$$

$$\frac{start \leq end}{\begin{array}{l}\langle M_i, [\ldots, \langle M_{i-1}^j, C^j, -, -\rangle, \ldots], \texttt{Mappar}(k = start : end, P_0); P_1, U\rangle \quad \rightarrow \\ \langle M_i, [\ldots, \langle M_{i-1}^j[k \leftarrow start], C^j, P_0; -, -\rangle, \ldots], \texttt{Mappar}(k = start + 1 : end, P_0); P_1, U\rangle\end{array}} \qquad \texttt{[Mappar]}$$

$$\frac{\begin{array}{l}start > end \\ C = [\langle M_{i-1}^1, C^1, -, -\rangle, \ldots, \langle M_{i-1}^n, C^n, -, -\rangle]\end{array}}{\langle M_i, C, \texttt{Mappar}(k = start : end, P_0); P_1, -\rangle \rightarrow \langle M_i, C, P_1, -\rangle} \qquad \texttt{[Barrier]}$$

$$\begin{array}{l}\langle M_i, [\ldots, \langle M_{i-1}^j, C^j, -, -\rangle, \ldots], \texttt{Spawn}(P_0, pred); P_1, U\rangle \quad \rightarrow \\ \langle M_i, [\ldots, \langle M_{i-1}^j, C^j, P_0; -, -\rangle, \ldots], \texttt{Spawn}(P_0, pred); P_1, U\rangle\end{array} \qquad \texttt{[Spawn]}$$

$$\frac{\begin{array}{l}M_i(pred) = \mathit{true} \\ C = [\langle M_{i-1}^1, C^1, -, -\rangle, \ldots, \langle M_{i-1}^n, C^n, -, -\rangle]\end{array}}{\langle M_i, C, \texttt{Spawn}(P_0, pred); P_1, -\rangle \rightarrow \langle M_i, C, P_1, -\rangle} \qquad \texttt{[Spawn End]}$$

$$\langle M_i, [\ldots, \langle M_{i-1}^j, C^j, \texttt{Up}(P_0^j); P_1^j, U^j\rangle, \ldots], P, -\rangle \rightarrow \langle M_i, [\ldots, \langle M_{i-1}^j, C^j, \texttt{wait}; P_1^j, U^j\rangle, \ldots], P, P_0^j; \texttt{resume}\rangle \qquad \texttt{[CallUp]}$$

$$\langle M_i, [\ldots, \langle M_{i-1}^j, C^j, \texttt{wait}; P^j, U^j\rangle, \ldots], \texttt{resume}, U\rangle \rightarrow \langle M_i, [\ldots, \langle M_{i-1}^j, C^j, P^j, U^j\rangle, \ldots], -, U\rangle \qquad \texttt{[Resume]}$$

$$\langle M_i, C, P, U\rangle \rightarrow \langle M_i, C, U, P\rangle \qquad \texttt{[Swap]}$$

$$\frac{\langle M_{i-1}^j, C^j, P^j, U^j\rangle \rightarrow \langle M_{i-1}^{'j}, C^{'j}, P^{'j}, U^{'j}\rangle}{\langle M_i, [\ldots, \langle M_{i-1}^j, C^j, P^j, U^j\rangle, \ldots], P, U\rangle \rightarrow \langle M_i, [\ldots, \langle M_{i-1}^{'j}, C^{'j}, P^{'j}, U^{'j}\rangle, \ldots], P, U\rangle} \qquad \texttt{[Parallel]}$$

Table 1. Operational semantics.

and [Mappar] rules do not modify the parent memory; thus, there can never be races between a call-up and the parent task. However, it is possible for a copy operation in a child task and a call-up from a different child task to race on an access to the parent's address space. A form of this problem already exists in Sequoia, in that parallel subtasks are forbidden from aliasing `out` parameters; i.e., two parallel child task calls may not write the same output location [5]. We extend this restriction to cover call-ups as well: no child task may overwrite a parallel child task's input or output arguments, either through `out` or `inout` parameters or call-ups. While a suitable static analysis could conservatively check this restriction, our current implementation assumes, but does not enforce, this rule.

Finally, the [Parallel] rule expresses that computation steps can take place in child memories, not just at the parent; this is the rule that models parallel execution at each level of the memory hierarchy.

5. Implementation

A Sequoia runtime sits between two adjacent levels of the memory hierarchy and provides a separate interface for both the parent and the children [8]. We have added two new calls: `SpawnChild` to the parent interface and `CallParent` to the child interface. The declarations of these new calls are shown below.

```
SpawnChild(TaskID taskid, ChildID start, ChildID end,
           TerminationID_t termid);
CallParent(ChildID myid, void *parent_ptr,
           CallupID callid);
```

`SpawnChild` enables a parent task to spawn tasks onto child nodes; `CallParent` enables a child node to invoke a call-up on a parent object. We briefly discuss the implementation of both methods.

`SpawnChild` takes a task to spawn, a range of child nodes on which to spawn tasks, and a termination test. The goals of a good implementation are in tension: to both keep the children busy but also to terminate as soon as possible. We employ a simple heuristic to determine whether a task should be respawned on a given child following the completion of the child's task. We say a child has *finished* when its currently assigned task is finished but it has not been evaluated for respawn. A child has *completed* when the runtime system has evaluated the child for respawn and decided not to respawn the child. When a child has finished it is enqueued for possible respawn. The runtime continually dequeues finished children and evaluates whether to respawn them or not. If more than half the of the children have completed, then the runtime does not respawn the child and adds it to the list of completed children. If fewer than half the children have completed, the runtime checks the termination test. If the test is *false* the child and all completed children are respawned (to maintain high utilization); if the test is *true* the child is added to the list of completed children. When all children have completed, the termination test is checked again; if *true* the `SpawnChild` call terminates; if *false* all children are respawned. We note that other reasonable respawn heuristics exist, and nothing prevents a user from modifying a runtime to include an application-specific heuristic.

When a child invokes `CallParent`, a task is enqueued at the parent's level. A dedicated thread in the parent pulls the call-up off the queue, executes it, and then sends the results back down to the child. Since call-ups are handled sequentially by a single thread of control at the parent this trivially maintains the atomicity property of call-ups. We see two potential future optimizations for executing call-ups. The first is to leverage the isolation property of tasks to allow the compiler to prove statically when it is safe for call-ups to execute in parallel. The second possibility is to use transactional memory to optimistically execute call-ups in parallel and detect conflicts dynamically.

If a call to `CallParent` discovers that the parent pointer passed is not local to the current runtime's address space, the runtime will recursively call `CallParent` on its parent runtime. This will continue passing the call-up up the memory hierarchy until it reaches the runtime containing the object pointed to by the parent pointer. Children can thereby perform call-ups to any of their ancestor levels in the memory hierarchy. By using call-ups to parent pointers at different levels, the programmer has the capability to create hierarchical data caching schemes for deep memory hierarchies.

5.1 Supporting Virtual Levels

Virtual parent levels must implement a distributed shared memory on top of the physically disjoint child memories, usually using MPI. Generally these are the most involved Sequoia runtimes, and present additional challenges for implementing `SpawnChild` and `CallParent`. By default, MPI process zero is designated to execute the parent's program and to hold the parent's data (with the exception of distributed arrays). We make two exceptions to this rule to achieve better performance in virtual levels.

The first exception is respawning tasks in a virtual level. If a task is going to be respawned, it should be done so as quickly as possible to keep processors busy, but in a distributed setting the latency of communicating with the parent is significant. We therefore modify the respawn heuristic described above for virtual levels. The runtime operating at each child node keeps track of the last respawn decision made by the parent. When a child finishes, it locally decides whether to respawn or not using the previous instruction it received. It then communicates to the parent (node 0) that it has finished its task and asks whether to respawn. When the decision from the parent comes back it is cached for determining whether to respawn the next time. This simple form of software pipelining hides the latency of respawn in a distributed environment where the common case is that a task is respawned many times.

The other exception to having node 0 perform all the parent's work is for some call-ups. In the case where a call-up touches only part of a of distributed array that resides on a single node, the runtime passes the call-up to that node for execution. This "owner-computes" optimization is safe because any other call-ups touching the same data will also be sent to the same node and serialized locally. The performance gain that we see from this approach in Section 6.3 is motivation for enhancing our ability to determine when it is safe for call-ups to be performed in parallel. All other call-ups are still handled exclusively by node 0; our runtime assigns a different and lighter load of tasks from a `spawn` statement to node 0 to allow it devote more resources to servicing call-ups.

6. Applications and Evaluation

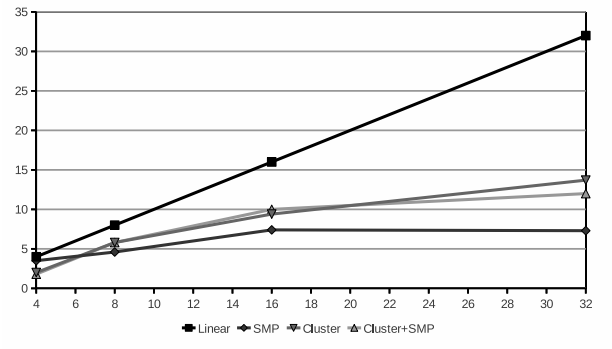
We have implemented call-up and spawn in Sequoia++, an extension of Sequoia. In this section, we evaluate the performance of three representative irregular applications written in Sequoia++: a boolean satisfiability solver (SAT), a sparse matrix-vector multiply (SMVM), and a parallel sample sort. We benchmark these applications on a multi-core SMP, a cluster of Opterons, and a cluster of SMPs. The SMP is an 8-node machine, with 128 GB of main memory. Each node in the SMP is a 4-core AMD Opteron, clocked at 2.3 GHz, with a shared 512 Kb L2 cache. The Cerillos cluster at Los Alamos National Labs consists of 360 nodes, connected by InfiniBand. Each node in the cluster consists of two dual-core AMD Opterons, clocked at 1.8 Ghz, with a shared 1024 Kb L2 cache and 8 GB of main memory. For the cluster experiments we use only 1 core of the dual-core chips and a Sequoia cluster runtime; for our cluster of SMP experiments we treat each dual-core chip as a 2 core SMP (running a Sequoia SMP runtime) and the rest of the machine as a cluster of these small SMPs (running a Sequoia cluster runtime).

In our experiments we use the same number of cores (4, 8, 16, or 32) across all three platforms. Figure 5 plots performance of the three applications across the three architectures, and Figure 6 details profiling information displaying the percentage of time spent in each phase of the computation on different memory levels.

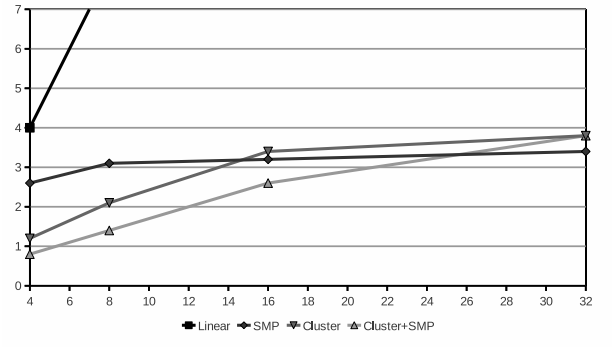
6.1 SAT

The satisfiability problem (SAT) is to determine if there is an assignment of true/false to the boolean variables of a propositional formula that makes the formula true. Most parallel SAT solvers decompose the search space by generating partial assignments and delegating the resulting sub-problems to a sequential solver [11]. While the data sizes are small, the solution time of the subproblems is extremely variable, making good load balancing critical.

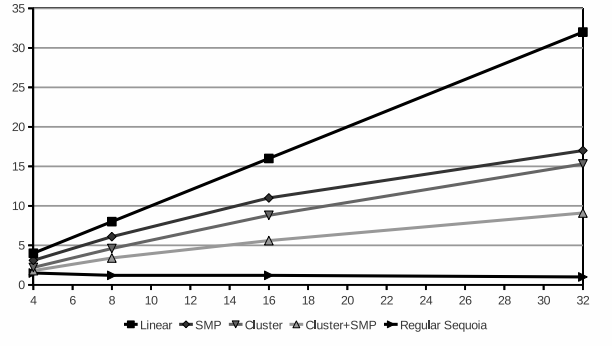
Our SAT implementation is a worklist algorithm similar to Figure 4. The worklist is initialized with n partial solutions, the complete assignments of the $\log_2 n$ most common variables. We then perform a `spawn` over the elements in the worklist. Children remove a partial solution from the worklist, and attempt to complete the solution using MiniSat [12]. A child may discover its problem is satisfiable (in which case it uses a call-up to notify the parent) or unsatisfiable (in which case the child simply returns). Another possibility is that the child exceeds a preset time bound, in which



(a) Speedup for SAT.



(b) Speedup for SMVM.



(c) Speedup for sample sort.

Figure 5. Speedup for each application over an optimized, purely sequential algorithm on three different platforms. Figure (c) also contains a line illustrating the speedup achieved by a best-effort sorting implementation using only the regular language features of Sequoia.

case the child splits the problem in two, continuing to work on one of the subproblems and pushing the other one to the parent's worklist using a call-up.

Recently the annual SAT competitions have introduced a track for parallel solvers; the SAT speedups in Figure 6 are averages over runs on all the 2007 contest problems. To date the parallel SAT contest has been held on 4-way SMPs, and our implementation is competitive, achieving a 3.5X average speedup on a 4-core SMP. (Our leaf sequential solver, MiniSat, is regarded as one of the best open source solver, though there are faster closed source and proprietary solvers.) Performance tails off at larger degrees of parallelism (because more of the subproblems represent speculative work that would not be done by the sequential solver), but continues to improve on the cluster and cluster of SMPs, reaching a maximum of about 14X speedup on 32 processors on the cluster. Interestingly, the SMP does not do so well, topping out at 7.4X speedup on 16 processors. MiniSat caches a significant amount of state and the 4:1 processor-to-L2 cache ratio on the SMP results in more L2 misses than on the cluster and cluster of SMPs where the processor-to-L2 cache ratios are 1:1 and 2:1 respectively. Child tasks spend almost their entire execution performing useful work and less than 1% of their time performing call-ups, indicating that call-ups do not represent a bottleneck to performance for SAT. For brevity, this data is omitted from Figure 6.

6.2 Sparse Matrix-Vector Multiply

Sparse Matrix-Vector Multiplication (SMVM) is a standard kernel used in many scientific applications. Sparse matrices are commonly used to represent large data sets where many of the entries are zero. The distribution of non-zeroes is usually non-uniform, resulting in some parts of the matrix having higher densities of nonzero elements than others. Irregularity in the data representation, and a generally low compute-to-communication ratio, makes SMVM challenging to parallelize. There has been extensive work in optimizing SMVM computations for both sequential [13] and parallel machines with shared [14] and distributed address spaces [15].

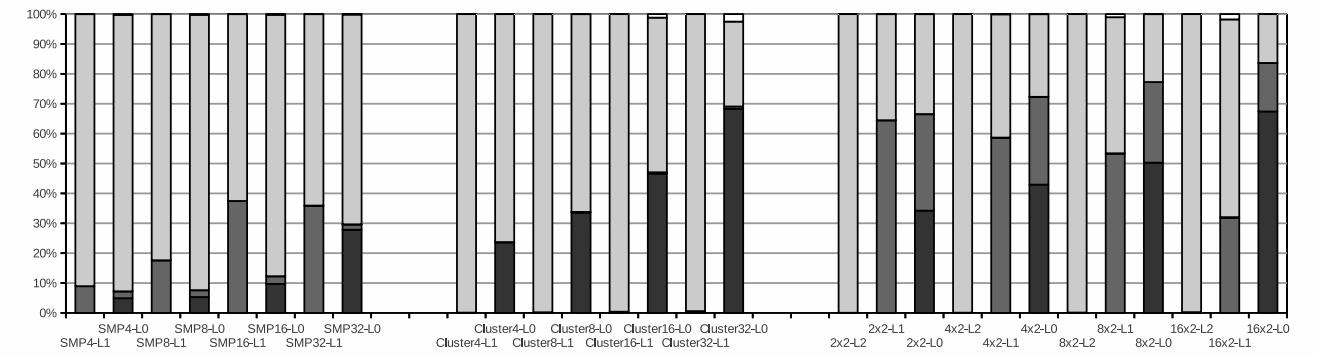
In our implementation of SMVM, sparse matrices are represented in the standard compressed-sparse-row format. While other implementations of SMVM attempt to modify the data representation depending on the matrix [15] we do not customize our code for the input matrix. We achieve parallelism in SMVM by dividing the set of dot products that must be computed into chunks. Since each dot product has a variable number of nonzero elements to be multiplied, load balancing is performed in a manner similar to our SAT implementation. We use a `spawn` statement to launch tasks onto the child processors. Children then call-up and retrieve a set of rows on which to operate. Children that complete their rows continue to call-up to get additional dot products to perform. Note that unlike a `mappar`, where iterations are assigned to processors statically by the compiler, the use of `spawn` decides dynamically which dot products will be evaluated on which processors based on load.

As our benchmarks we chose five matrices from the University of Florida Sparse Matrix Library [16]: `atmosmodd`, `nlpkkt80`, `Freescale1`, `ldoor`, and `nlpkkt120`. These matrices come from real-world applications and range in size from 8.8 to 50 million nonzero elements with varying sparsity patterns.

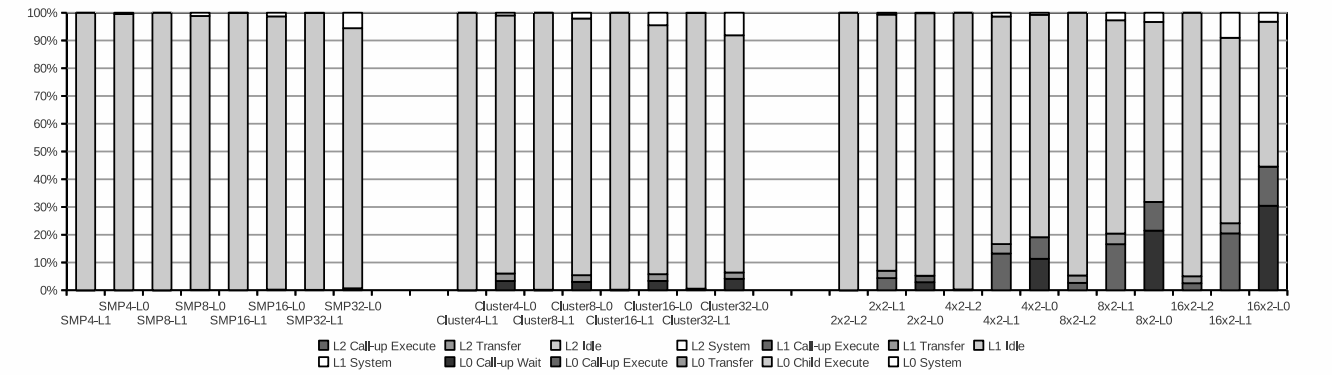
Our reference implementation makes use of the OSKI library for sparse matrix-vector multiplication [13]. OSKI is a purely sequential sparse matrix library capable of dynamically tuning itself for a given matrix at runtime. We set the OSKI library to `ALWAYS_TUNE_AGGRESSIVELY` but we do not include OSKI's tuning time in the reference execution time. This can only make our implementation appear worse, and thus our speedups with respect to OSKI are likely a lower bound on what a user would experience in practice.

In Figure 6, the tightly coupled SMP is able to overcome the parallel overhead at low numbers of processors, but the other two platforms catch up at large machine sizes as memory bandwidth becomes a factor. At 32 processors, all three platforms have approximately the same speedup (a mean of 3.4X-3.8X), which is comparable to the performance of other recent efforts [14, 15]. For example, in [15], speedups (including the time for tuning) range from 2X to 11X with a mean of 4X on 32 nodes. We achieve our best raw performance on the Cluster of SMPs configuration running the `ldoor` matrix multiplication at a sustained rate of 1.57 GFLOPS with 32 child tasks.

The cluster and cluster of SMPs perform better at 32 nodes than the SMP due to additional threads on the SMP being scheduled on



(a) Profile for SMVM.



(b) Profile for sample sort.

Figure 6. Percentage of time spent in different phases of each computation. Each group of columns corresponds to a different platform and each column to a different memory level within that platform. The `Call-up Execute` component indicates the amount of time spent in useful work performing call-ups while the `Call-up Wait` component indicates the amount of time call-ups from a level spent waiting in a queue before being executed. The `Idle` field indicates time that a parent level spent waiting to handle call-ups while child tasks were executing. The remaining fields are identical to [5].

the same socket and causing higher contention for memory bandwidth. Knowing that memory bandwidth is often the bottleneck for SMVM, we can clearly see in Figure 6 that call-ups are not the performance bottleneck for the SMP configuration as the child tasks spend significant portions of their time performing useful work and no more than 30% of their execution performing call-ups. The same cannot be said for the two cluster configurations as we can see the percentage of time children spend waiting for call-ups to execute increases progressively with the number of leaf-level tasks. On the Cluster of SMPs with 32 leaves, children spend in excess of 67% of their time simply waiting for their call-up to execute. The reason for the increased waiting time is the extra latency to communicate call-ups and their arguments over the network. This additional latency decreases the parent's call-up bandwidth as each call-up now requires additional time to execute. As future research we plan to investigate methods of performing call-ups in parallel.

6.3 Sample Sort

Sample sort is considered to be one of the most efficient comparison-based sorting algorithms for distributed memory architectures [17]. It is a generalization of Quicksort, which recursively decomposes its input into $n > 2$ partitions, and sorts each independently. Because partitions are generated based on pivots randomly selected at runtime, there is no guarantee that partitions will be the same size or require the same time to sort. Sample sort is the most complex of our three applications and consists of a mix of sequential, regular, and irregular parallel phases:

- **Phase 1** Sequentially select a random subset of the input array as splitters.
- **Phase 2** A `mappar` over the the input array gives subtasks equal-size subsets of input elements; subtasks compute the partition for each element based on the splitters selected in phase 1.
- **Phase 3** In a `spawn` over the input array, subtasks compute the size of the output partitions: they request a subset of the elements, perform a prefix sum over their offsets in the partitions calculated in phase 2, and reduce their results using a call-up. In a second `spawn` children again request a subset of the elements and write them to the appropriate partition using the previously calculated offsets.
- **Phase 4** In a `spawn` over the partitions generated in phase 3, subtasks request a partition from the parent, sort the elements using C++'s STL sort, and write the results back using a call-up.

Our sample sort achieves good absolute performance on all three platforms; at 32 processors performance ranges from 9X speedup on the cluster of SMPs to 17X on the SMP over the sequential C++ STL sorting algorithm; across all platforms leaf task (level 0) utilization is never less than 51% (for the 32 node Cluster of SMP's experiment) indicating that the majority of the execution time is spent performing useful parallel work. We achieve our maximum

sorting performance on the SMP machine at a rate of 126 million keys per second with 32 leaf tasks.

As an interesting experiment we also wrote a best-effort sort using only the regular features of Sequoia. The results in Figure 5 show that obtaining good sorting performance is difficult when using only the regular features of Sequoia, indicating the need for additional language features to parallelize irregular code.

7. Related Work

Sequoia is designed to give the programmer explicit control over data locality and communication for programming machines with multi-level memory hierarchies. The language accomplishes this goal through isolated tasks that encapsulate data and control in one level of the hierarchy. Complete isolation is problematic for problems where task working sets are most naturally computed by the tasks themselves, and we have proposed extensions to Sequoia that allow selective exceptions to pure isolated tasks.

The PGAS family of languages, such as Split-C [18], Co-Array Fortran [19], UPC [20], and Titanium [3] present a single program address space with SPMD semantics with one thread per processor. Thus, the threads are not isolated from one another; any thread may reference any accessible data in the global address space, and there is no special problem in expressing irregular algorithms. Currently these languages provide only a two-level memory hierarchy.

More recent parallel language efforts [21–23] support locality cognizant programming through the concept of distributions (from ZPL [24]). While also PGAS languages, these designs also provide more abstract and dynamic notions of *place* (X10) or *locale* (Chapel) than the more static SPMD languages discussed above, and while we are unaware of any studies to confirm it, our intuition is that irregular algorithms should be easier to express in these languages. These are still two-level languages, however.

A recent effort proposes Hierarchical Place Trees (HPT) as a unification of the Sequoia and X10 programming models [25]. Like Sequoia, HPT models machines as a tree of memories. Instead of call-up, however, HPT presents a form of global address space as in X10. At any level of the memory hierarchy, data can be referenced at any ancestor level—while not truly global across the machine, this model allows for tasks to read or write extra data outside of their own locale/place if necessary. This model can be simulated using call-up by writing tasks that are remote *read* and *write* methods for parent levels of the memory hierarchy. We considered extending Sequoia with a model similar to HPT, but ultimately decided that call-up was both more flexible and in many cases more efficient: once we have paid the cost to move to another memory location within the machine, it will often be cheaper to perform a computation locally on the data, rather than simply return the data and perform the computation somewhere else. (For example, in the worklist algorithm adding or removing elements from the worklist involves more than memory references.) The difference in design stems in part from a difference in philosophy about the underlying architectures: if processing elements are only or primarily at the leaves of the memory hierarchy, then HPT is a close match to the machine. However, if interior nodes of the machine tree have their own processors then call-up allows us to take advantage of these to carry out computation at those levels.

Hierarchically Tiled Arrays (HTA) [26] accelerate existing sequential languages with an array data type expressing multiple levels of tiling for locality and parallelism, but also permit arbitrary element access and therefore can directly express at least some irregular algorithms. 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.

Stream processing languages [27, 28] also build upon a two-tiered memory model [29], choosing to differentiate between on and off-chip storage. Sequoia tasks are a generalization of stream programming kernels. Tasks and kernels share similarities such as isolation, a local address space, and well-specified working sets, but differ in the ability of tasks to arbitrarily nest. Because these languages enforce strong isolation, they have difficulties similar to Sequoia in expressing highly irregular computations.

Sequoia's control flow, when encountering a parallel mapping of subtasks, resembles the thread-less abstraction of concurrency in Cilk [30], X10 [21], Chapel [22], and Fortress [23]. Sequoia's control flow is constrained in comparison to most of these languages since, for example, the calling task cannot proceed until all subtasks complete (similar to the common usage of OpenMP [31] loops). The addition of `spawn` covers many (perhaps most) of the parallel loop patterns that could not be expressed in Sequoia. Cilk inlets provide a restricted form of call-up, allowing an atomic computation to be performed on the final result of a task (e.g., to perform a reduction across task results).

Previous efforts to model memory hierarchies include the Uniform Memory Hierarchy Model (UMH) [32], which abstracted uniprocessor machines as sequences of memory modules of increasing size. The Parallel Memory Hierarchy Model (PMH) [9] 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 [33, 34]. In Sequoia, hierarchical memory is a fundamental aspect of the programming model, required to achieve both performance and portability across a wide range of architectures. Sequoia has also been influenced by the idea of space-limited procedures [35], a methodology for programming machines modeled using the PMH model.

8. Discussion and Future Work

Sequoia is an attempt to strike a practical balance between performance and portability. In future machines locality is likely to become ever more important, and memory hierarchies are likely to become more complex and diverse—we will have everything from relatively simple two-level multi-core machines to supercomputers with many more levels of memory. The central tenet of Sequoia is that the programmer should have control over and be able to reason about locality and communication. By encapsulating both within a task, and by carefully avoiding any explicit machine dependencies in source programs, Sequoia allows programmers to express locality- and communication-aware algorithms that nevertheless map well to a wide variety of machines.

For regular problems this design works extremely well, but in some sense it cannot work for irregular problems. The fact that the working sets of tasks must be known before task execution is exactly the property that many irregular applications violate, at least if we do not want to write tasks in a low-level event-driven style. Adding parent objects and call-up allow tasks to escape their isolated context and communicate with their parent (and, recursively, any ancestor in the computation tree). The escape from isolation is explicit and tightly constrained, and the semantics are apparently as simple as possible: the only source of concurrency is within the parent's address space, and even then call-ups must execute atomically. Furthermore, the parts of programs that do not use call-up behave exactly as in the original Sequoia design. Thus, call-up and `spawn` can be seen as providing the missing duals of call-down and `mappar`, increasing the expressiveness of the language without changing its character or imposing costs when the features are not used.

One alternative to call-ups is to allow tasks access to data in the address space of any ancestor task. As discussed in Section 7 this is the approach adopted by HPT. Call-ups can also only access ancestor memories, but there are some differences between using di-

rect memory references and call-up. First, call-ups provide a concurrency semantics, guaranteeing atomicity of the invoked tasks, while direct memory references have the usual memory model issues around concurrent reads/writes in parallel machines. Second, call-ups can do more than read or write a single piece of data; once we have shifted to another location in the memory hierarchy, we can also perform an arbitrary computation on that data or amortize communication overhead by performing a bulk-transfer of data. A disadvantage of call-up is that access to the memory of remote ancestors must go through multiple recursive call-ups, while a HPT-based system can presumably more directly avoid any overhead in bypassing intermediate levels to go directly to some data more than one level removed in the memory hierarchy.

9. Conclusion

We have introduced *spawn* and *call-up* as new features for expressing irregular parallelism within the Sequoia programming language. We have described the operational semantics for both spawn and call-up within the Sequoia programming model. Our implementations of a series of irregular applications using spawn and call-up have illustrated competitive performance with other parallel codes. We also have demonstrated that spawn and call-up are the dual to the regular constructs already present in Sequoia, giving programmers the tools necessary to parallelize all phases of their code when programming deep memory hierarchies.

Acknowledgments

The authors would like to thank Evan Cox for his work on the implementation of Sequoia++, and the Department of Energy for access to the Cerillos supercomputer at Los Alamos National Labs.

References

[1] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference*. MIT Press, 1998.

[2] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," Center for Computing Sciences, IDA, Technical Report CCS-TR-99-157, 1999.

[3] K. Yelick *et al.*, "Titanium: A high-performance Java dialect," in *Workshop on Java for High-Performance Network Computing*, 1998.

[4] K. Barker *et al.*, "Entering the PetaFLOP era: The architecture and performance of Roadrunner," in *Supercomputing*, 2008.

[5] K. Fatahalian *et al.*, "Sequoia: Programming the Memory Hierarchy," in *Supercomputing*, November 2006.

[6] A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[7] T. Knight *et al.*, "Compilation for explicitly managed memory hierarchies," in *Symposium on Principles and Practice of Parallel Programming*, 2007, pp. 226–236.

[8] M. Houston *et al.*, "A portable runtime interface for multi-level memory hierarchies," in *Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 143–152.

[9] B. Alpern, L. Carter, and J. Ferrante, "Modeling parallel computers as memory hierarchies," in *Programming Models for Massively Parallel Computers*, 1993.

[10] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. Dally, "A tuning framework for software-managed memory hierarchies," in *Int'l Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 280–291.

[11] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: a parallel SAT solver," vol. 6, pp. 245–262, 2008.

[12] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, 2004, pp. 333–336.

[13] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Inst. of Physics Publishing*, 2005.

[14] A. Buluç *et al.*, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Symposium on Parallelism in Algorithms and Architectures*, 2009, pp. 233–244.

[15] S. Lee and R. Eigenmann, "Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems," in *Supercomputing*, 2008, pp. 195–204.

[16] T. A. Davis, "University of florida sparse matrix collection," *NA Digest*, vol. 92, 1994.

[17] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," *CoRR*, vol. abs/0909.5649, 2009.

[18] D. Culler *et al.*, "Parallel programming in Split-C," in *Supercomputing*, 1993, pp. 262–273.

[19] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[20] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," UC Berkeley Technical Report: CCS-TR-99-157, 1999.

[21] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *Conference on Object Oriented Programming Systems Languages and Applications*, 2005, pp. 519–538.

[22] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade high productivity language," in *Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004, pp. 52–60.

[23] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt., "The Fortress language specification version 0.707. Technical report," Sun Microsystems, 2005.

[24] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Abstractions for dynamic data distribution," in *Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004, pp. 42–51.

[25] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Workshop on Languages and Compilers for Parallel Computing*, 2009.

[26] G. Bikshandi *et al.*, "Programming for parallelism and locality with hierarchically tiled arrays," in *Symposium on Principles and Practice of Parallel Programming*, 2006, pp. 48–57.

[27] P. Mattson, "A programming system for the Imagine Media Processor," Ph.D. dissertation, Stanford University, 2002.

[28] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[29] F. Labonte, P. Mattson, I. Buck, C. Kozyrakis, and M. Horowitz, "The stream virtual machine," in *Int'l Conference on Parallel Architectures and Compilation Techniques*, September 2004.

[30] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Symposium on Principles and Practice of Parallel Programming*, 1995.

[31] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[32] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2/3, pp. 72–109, 1994.

[33] H. Jia-Wei and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Symposium on Theory of Computing*, 1981, pp. 326–333.

[34] J. S. Vitter, "External memory algorithms," in *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2002, pp. 359–416.

[35] B. Alpern, L. Carter, and J. Ferrante, "Space-limited procedures: A methodology for portable high performance," in *Int'l Working Conference on Massively Parallel Programming Models*, 1995.