Yada: Straightforward Parallel Programming

David Gay[b,1], Joel Galenson[a], Mayur Naik[b], Kathy Yelick[a]

[a] *University of California, Berkeley, CA 94720*
[b] *Intel Labs Berkeley, Berkeley, CA 94704*

Abstract

Now that multicore chips are common, providing an approach to parallel programming that is usable by regular programmers has become even more important. This cloud has one silver lining: providing useful speedup on a program is useful in and of itself, even if the resulting performance is lower than the best possible parallel performance on the same program. To help achieve this goal, Yada is an explicitly parallel programming language with sequential semantics. Explicitly parallel, because we believe that programmers need to identify how and where to exploit potential parallelism, but sequential semantics so that programmers can understand and debug their parallel programs in the way that they already know, i.e. as if they were sequential.

The key new idea in Yada is the provision of a set of types that support parallel operations while still preserving sequential semantics. Beyond the natural read-sharing found in most previous sequential-like languages, Yada supports three other kinds of sharing. *Writeonce* locations support a single write and multiple reads, and two kinds of sharing for locations updated with an associative operator generalise the reduction and parallel-prefix operations found in many data-parallel languages. We expect to support other kinds of sharing in the future.

We have evaluated our Yada prototype on 8 algorithms and 4 applications, and found that programs require only a few changes to get useful speedups ranging from 2.2 to 6.3 on an 8-core machine. Yada performance is mostly comparable to parallel implementations of the same programs using OpenMP or explicit threads.

Key words: parallelism, parallel language, language design, determinism, parallel-reduction, parallel-prefix

1. Introduction

Parallel programming is difficult: parallel programs are often hard to write, hard to understand and, especially, hard to debug. And then, after all this effort,

Email addresses: dgay@acm.org (David Gay), joel@cs.berkeley.edu (Joel Galenson), mayur.naik@intel.com (Mayur Naik), yelick@cs.berkeley.edu (Kathy Yelick)
[1] Current address: Google Inc., 345 Spear St, San Francisco, CA.

it is not uncommon to find that the parallel program or algorithm does not scale well, and may not even beat the performance of a sequential implementation. In the past, this has not been such a big problem as parallel machines were expensive and parallel programs relatively rare, so significant programmer effort could be expended in implementation, testing and debugging. However, the world has changed: most of today's computers have multicore processors, and sequential performance improvements are slowing down.

This is both a challenge and an opportunity: a challenge, because it is desirable for many more parallel programs to be written to exploit the now-omnipresent parallel machines; an opportunity, because the metric of success has changed. In the past, the goal was to maximise parallel performance to justify use of an expensive machine. Today any reasonable and easily obtainable parallel speedup (which will hopefully scale as chips with more cores appear) is worth pursuing as it will be applicable to all users. In a way, this situation mirrors the situation in sequential programming. Obtaining maximum performance from sequential code is also hard, involving steps like detailed performance analysis, blocking for caches, rewriting critical loops in assembly language, etc. Conversely, reasonable sequential performance (which improves with new, faster sequential processors) can be obtained simply by compiling programs written in statically-typed languages.

The goal of the Yada project is thus to achieve useful speedups in a parallel language that is easy to write, understand and debug. The approach we have chosen to achieve this goal is to make Yada an explicitly parallel, shared-memory language that "feels" sequential: Yada has nestable parallel control structures – forall-loops over integer ranges and parallel composition ($|;|$) of statements or blocks – that mirror sequential control structures – for-loops and sequential composition ($;$) – and has sequential semantics. Yada programs are understandable as sequential programs, execute deterministically,[2]and can optionally be compiled, and therefore debugged, as sequential programs.

There have been a number of previous parallel languages designed to "feel" sequential, including data-parallel languages like High Performance Fortran (HPF) [16], the OpenMP extensions to C, C++ and Fortran [1], Cilk [6], Jade [31], Deterministic Parallel Java (DPJ) [7] and the Galois system [20]. Most of these languages concentrate on preserving determinism by only allowing read-sharing between parallel threads of execution. In data-parallel languages, other kinds of sharing are impossible. Cilk and OpenMP support one other kind of deterministic sharing (corresponding to parallel reductions), but also allow arbitrary data sharing using locks, at the cost of losing determinism. Jade and DPJ require fairly extensive annotations: programs must identify all shared locations accessed by parallel blocks, either by explicit code (Jade) or complex region annotations (DPJ).

The key idea behind Yada is the belief that providing a sufficient general

[2]With the usual caveats about changes in the environment, files, etc, which also apply to sequential programs.

set of *sharing types* whose operations (reads, writes, accumulations, etc) are restricted to guarantee deterministic operation will allow the efficient expression of a large class of parallel algorithms, avoiding the need to escape into non-deterministic data-sharing (e.g. using locks). The current Yada prototype has four kinds of sharing types,[3] specified via new type qualifiers. These are simple read-sharing, `writeonce` locations, and two sharing types for describing updates to locations with an associative operator like `+` that allow a natural and general way of expressing the classical data-parallel reduction (`reduce(`*`op`*`)`) and parallel-prefix [5] operations (`scan(`*`op`*`)`). This Yada prototype is at least as expressive as traditional parallel SPMD-style programming models using only barrier synchronization and reduction and parallel-prefix library routines (the translation of such programs to Yada is typically straightforward).

Sharing types can be used like any existing type, e.g. in arrays, in structures, as pointer targets, etc. The conditions the programmer must follow to maintain determinism while using these types are expressed as simple restrictions on the parallel *and* sequential composition of operations on a particular location: a `reduce(+)` reduction type allows parallel reads *or* parallel updates with `+`, and places no restrictions on sequential composition; a `writeonce` location allows a read in parallel with a write, forbids a write in parallel or in sequence with another write, and allows a read in parallel or in sequence with another read; etc.

These facilities are significantly more general and convenient than previous languages: OpenMP and Cilk have neither writeonce nor sharing types for parallel prefix; OpenMP imposes significant restrictions on where and how reductions can be used [1, 2.9.3.6]; Jade, DPJ and Galois allow programmers to specify (unchecked) commutativity annotations on methods, to allow, e.g. adding elements to a set in parallel, but this approach does not capture the restrictions necessary on parallel operations for some kinds of shared objects (Section 3). Finally, the Peril-L pseudo-language used to express parallel algorithms in Lin and Snyder's Principles of Parallel Programming [22] contains sharing types similar to Yada, but provides no determinism guarantees.

The Yada prototype is built as an extension to C and currently has two backends: a serial, debugging backend that checks at runtime that programs correctly follow the shared type rules, and an unchecked parallel work-stealing backend built over Intel's Threaded Building Block library. We have evaluated this prototype on eight small algorithms and four larger applications, with few parallelism and sharing annotations (our benchmarks require at most 19 such annotations) required to achieve speedups ranging from 2.2 to 6.3 on an 8-core machine. Section 5.1 describes in detail the annotations and changes required. Furthermore, performance on all applications and all but two algorithms is competitive with other parallel implementations written with explicit threads

[3] We know of at least one other kind of deterministic sharing, broadcast in Kahn networks [18], that we have not implemented yet as it has not been required in any of our benchmarks.

or OpenMP.

In summary, Yada makes the following contributions:

- Deterministic languages can be made much more flexible by providing multiple different abstractions for shared locations that support parallel operations while preserving determinism (Section 2).
- Our evaluation shows that a deterministic language can express many parallel algorithms and obtain useful speedups on real applications (Section 5).
- Our shared location type that allows reads in parallel with updates with an associative operator provides a clear mental model for when a parallel-prefix operation can be used to parallelise sequential code (Section 2). Furthermore, the transformation to use the parallel-prefix operation is performed mostly-automatically by the Yada compiler (Section 4.3.3).

2. Overview

Yada is a shared-memory, explicitly parallel language with deterministic semantics. But beyond this theoretical property, Yada is designed to "feel" sequential: a `forall` loop executes (in parallel) as if the iterations are executed in the usual sequential order, and `s1` appears to execute before `s2` in the parallel composition `s1 |;| s2`. This property is observable (see the `scan(+)` examples below) through Yada's flexible *sharing types*.

Sharing types allow concurrent updates on shared state while preserving Yada's sequential semantics, and are declared via new type qualifiers. For instance, a `reduce(+)` variable allows concurrent updates using `+=` as in the following loop:

```
int reduce(+) sum = 0;
forall (i = 0; i < n; i++) sum += A[i];
```

which expresses a classic parallel reduction in a sequential style.

Sharing types can do much more than just conveniently express well-known data-parallel operators, as we show through the example of the parallelisation of a radix sort (Figure 1). As a brief reminder, radix sorts an input array `A` by processing individual groups of bits within each number. Each pass runs a stable sort on the numbers with respect to different bit groups, and when they have been sorted once for each bit group they are fully sorted. The stable sort for each bit group (lines 8-17 of `radix_C`) operates by building a histogram of the bit group's values in array `x` (lines 8-11), using that histogram to compute a starting offset in a destination array `y` (lines 12-15), then copying the elements of `x` to `y` to their bit group's positions (lines 16-17). Finally, the roles of the `x` and `y` arrays are swapped and processing passes to the next bit group (lines 18, 7). The `RADIX` constant defines the number of bits to sort at once, and the

```
1 void radix_C(int *A, int n) {
2   int B[n], *x = A, *y = B, *tmp, i, pass;
3   int buckets[RADIX];
4   int offset;
5   int offsets[RADIX];
6
7   for (pass = 0; pass < PASSES; pass++) {
8     for (i = 0; i < RADIX; i++)
9       buckets[i] = 0;
10    for (i = 0; i < n; i++)
11      buckets[key(x[i], pass)]++;
12    for (offset = 0, i = 0; i < RADIX; i++) {
13      offsets[i] = offset;
14      offset += buckets[i];
15    }
16    for (i = 0; i < n; i++)
17      y[offsets[key(x[i], pass)]++] = x[i];
18    tmp = x; x = y; y = tmp;
19  }
20 }
```

```
1 void radix_yada(int *A, int n) {
2   int B[n], *x = A, *y = B, *tmp, i, pass;
3   int reduce(+) buckets[RADIX];
4   int scan(+) offset;
5   int scan(+) offsets[RADIX];
6
7   for (pass = 0; pass < PASSES; pass++) {
8     forall (i = 0; i < RADIX; i++)
9       buckets[i] = 0;
10    forall (i = 0; i < n; i++)
11      buckets[key(x[i], pass)]++;
12    forall (offset = 0, i = 0; i < RADIX; i++) {
13      int b = buckets[i];
14      offset += b;
15    andthen:
16      offsets[i] = offset - b;
17    }
18    forall (i = 0; i < n; i++) {
19      int k = key(x[i], pass);
20      offsets[k]++;
21    andthen:
22      y[offsets[k] - 1] = x[i];
23    }
24    tmp = x; x = y; y = tmp;
25  }
26 }
```

Figure 1: Radix sort written in C and Yada. Yada keywords in bold.

`key` function retrieves the specified bits of the number. We elide the last loop to copy results to `A` that is required only if `PASSES` is odd.

If the Yada keywords are ignored in the Yada version (`radix_yada`), it is essentially identical to the C code, though a little more verbose: Yada's features express only *restrictions* on the usual sequential semantics. These restrictions allow the code to be executed in parallel but with the same behaviour as if it were run sequentially.

To preserve familiarity, parallel loops in Yada are written using the `forall` keyword following standard C for-loop syntax, but must express iteration over a simple integer range or they will be rejected at compile-time. We now describe how Yada's sharing types allow us to parallelise the four inner loops of radix sort.

Plain sharing. Locations that are either not shared across loop iterations or are only read can be treated normally in parallel loops. Accesses to these locations will always be deterministic, regardless of the implementation strategy. In `radix_C`, the loop at line 8 accesses independent data and so is easily parallelised.

Reduce. As we saw above, a `reduce(`*`op`*`)` sharing type allows concurrent updates using *`op`*`=` for associative and commutative operators or functions *op*. Commutativity is not strictly required, but gives the implementation more flexibility and hence increases performance. Like all our sharing types, `reduce(+)` is not restricted to scalars: the C loop at line 10 that constructs a histogram performs only `++` updates on the `buckets` array. This loop can be run in parallel in Yada (line 10) by declaring that `buckets` is an *array* of `int reduce(+)` (line 3).

This last example brings up an essential difference between reduce types and a parallel-reduction function. A reduce type expresses the allowable operations on a location; these operations may or may not be implemented using an underlying parallel reduction function. For instance, if there are few concurrent updates to a location it makes sense to implement `reduce(+)` using an atomic-add instruction; if there are many, it makes sense to replicate the location across processors (allowing independent updates) and compute its value using a reduction when it is read, possibly even a sequential reduction if multiple reads happen in parallel.

Finally, it is extremely desirable to allow use of floating-point `reduce(+)` types even though floating-point addition is not actually associative, and the resulting programs are thus not strictly deterministic. Discussions with a parallel numerical analysis expert [12] suggested that repeatability on a given machine for a given parallelism level would be a desirable feature for such programs; however we have not yet investigated how to provide this in our Yada implementation.

Scan. The most complicated of our sharing types is scan. It generalises `reduce(`*`op`*`)` by allowing reads concurrently with *`op`*`=` updates, while preserving the loop's

sequential semantics. Examining `radix_C`, we can see that the loop at line 12 contains both updates (with `+=`) and reads of the shared variable `offset` and the loop at line 16 contains both updates (with `++`) and reads of the array `offsets`. These loops could not be parallelized with `reduce(+)` annotations, as `reduce(+)` does not allow reads in parallel with `+=` updates.

To parallize these loops with `scan(+)` we must obey `scan()`'s one significant restriction (required to allow parallel execution): within a parallel loop, it must be possible to perform all the updates (with *op*`=`) before any of the reads. To ensure that this restriction is followed, Yada requires that the parallel loop be divided into two parts separated by an `andthen` pseudo-label[4] separating the half of the loop that performs updates from the half that performs reads. This restriction forces a fairly straightforward transformation of the original C code to Yada. The Yada version of the first loop (line 12) must update `scan(+)` variable `offset` with `b = buckets[i]` before assigning `offsets`, and therefore must subtract `b` from `offset` in the assignment. Similarly, in the second loop (line 18) the effect of moving the update of `offsets[k]` to before the read of `offsets[k]` is addressed by subtracting one from `offsets[k]` when indexing `y`.

These two examples also exhibit Yada's (hopefully unsurprising) scoping rules for local variables: variables declared outside a `forall` (such as `offset`) are shared by all iterations, while those declared within the `forall` (`b` and `k`) are private to each iteration.

Scan sharing types are implemented using a *parallel-prefix* operation (sometimes also called a parallel scan). A parallel-prefix of an array A with an associative operator *op* computes an array defined element by element as

$$\text{prefix}(A,\ op\ )[i] = A[0]\ op\ \ldots\ op\ A[i]$$

This apparently sequential computation can actually be performed in parallel [5]. As a consequence, it follows that any loop parallelised using `scan()` could have been written using one or more explicit parallel-prefix operations. We have found that the fairly straightforward restrictions on scan sharing types make it very easy to understand when loops can be parallelised using parallel prefix, and helps figure out the necessary loop transformations. Furthermore, we believe that the code written using `scan()` is much clearer than the alternate version using explicit parallel-prefix calls.

This completes the parallelisation of radix sort. Yada has one more sharing type, `writeonce`, that is not used in radix sort.

Writeonce. The writeonce sharing type is for variables that are written and read in parallel: the reads simply block until the write occurs. For example:

[4]These are not real C labels as they may be multiply defined. They are ignored when executing sequentially.

```
int writeonce value;
forall (int i = 0; i < n; i++)
  if (i == 0) value = 42; // write in parallel ...
  else f(value);          // ...  with this read
```

To preserve the property that Yada programs can also be executed sequentially, Yada requires that the write occurs before the read in the program's underlying sequential order. Thus the program above would be illegal if `i == 0` were replaced by `i == 1`. This restriction has the further advantage of preventing deadlock: without it, it is easy to express circular dependencies between writeonce values.

2.1. Language Summary

We present here in one place all of Yada's extensions to C and their associated rules and restrictions.

2.1.1. Expressing Parallelism

The $s_1 \mid;\mid s_2$ construction allows any two statements (including blocks and nested parallel constructs) to be executed in parallel. The `forall` parallel loops follow traditional C syntax, but must have the form:

$$\texttt{forall}\ (\mathit{local} = s; \mathit{local} < e; \mathit{local}{+}{+})\ \mathit{body}$$

where *local* is a local variable whose address is not taken, s and e are identical expressions and *body* is any statement or block, and may include nested parallel statements or calls to functions with parallel statements.[5] The s and e expressions are evaluated once, before the loop starts.

2.1.2. Sharing Types

Yada sharing types are simply type annotations that can be placed anywhere where C allows its existing `const` and `volatile` annotations (with obvious restrictions, e.g. `reduce(*)` has no meaning on a pointer). Two sharing types cannot modify the same type. For instance, the following are all allowed:

```
static int reduce(+) x; // a global allowing parallel accumulation
struct force_vector_2d { // a type allowing parallel force accumulation
  float double(+) forcex, forcey;
};
int scan(+) * writeonce p; // a writeonce-pointer to a scan(+) array
```

while the following are illegal:

```
int reduce(+) writeonce x; // two sharing types
int *reduce(*) x; // multiplication is not defined on pointers
```

[5] Extensions to allow strides and downwards parallel iterations would be straightforward.

The following paragraphs summarize the rules governing the use of Yada's sharing types, and give short examples of valid and invalid use:

plain: A location l of type `t *` has two operations: $*l$ (read) and $*l = v$ (write). The only legal parallel combination is read||read (two reads in parallel). This is the default sharing type when no sharing qualifiers are used, allowing parallel reads of shared and unshared data, but only parallel writes to unshared data.

```
Valid:
int x = f(...);
forall (i = 0; i < 10; i++)
  a[i] = x * i;
```

```
Invalid (parallel writes):
int x = 3;
forall (i = 0; i < 10; i++)
  x = x * 2
```

writeonce: A location l of type `t writeonce *` has two operations: $*l$ (read) and $*l = v$ (write). The only legal parallel combinations are read||read and write||read, with the write occurring before the read in Yada's implicit sequential execution order. This last restriction is sufficient to guarantee that Yada programs that respect the sharing rules cannot deadlock.

```
Valid:
int writeonce pipe1, pipe2;
pipe1 = slow(1) |;|
pipe2 = slow(2) |;|
return pipe1 + pipe2;
```

```
Invalid (read before write):
int writeonce x[11];
forall (i = 0; i < 10; i++)
  x[i] = x[(i + 1) % 10] + 1;
```

reduce: A location l of type `t reduce(op) *` has three operations: $*l$ (read), $*l = v$ (write) and $*l$ *op*$= v$ (accumulate). The only legal parallel combinations are read||read and accumulate||accumulate. The operation is an associative and commutative[6] operator or user-defined function with signature t *op*`(`t`, `t`)`.

```
Valid:
float step(float, float);
float reduce(step) a = 0;
forall (i = 0; i < N; i++)
  a = step(a, v[i]);
```

```
Invalid (invalid parallel write):
int reduce(+) x;
forall (i = 0; i < 10; i++) {
  x = x + f(i);
  if (g(i)) x = 0;
}
```

scan: `scan(`*op*`)` behaves likes `reduce(`*op*`)`, except that scan locations allow both accumulates and reads in the same parallel loop, under the condition that the accumulate happens in the loop's first part (before `andthen`), and the read happens in the loop's second part. A parallel loop can also perform just reads or just accumulates, i.e. `scan` is a strict superset of `reduce` (but `reduce` can be

[6] As mentioned earlier, commutativity increases implementation flexibility and hence performance.

implemented more efficiently).

```
Valid:
float step(float, float);
float scan(step) a = 0;
int reduce(+) large = 0;
forall (i = 0; i < N; i++) {
  a = step(a, v[i]);
andthen:
  if (a > 100) large++;
}
```

```
Invalid (reads before writes):
int scan(+) x;
forall (i = 0; i < 10; i++) {
  for (k = 0; k < 5; k++) {
    x = x + f(i);
andthen:
    a[i] = x;
  }
}
```

It is easy to imagine other useful types that can be restricted to support determinism: sets that support adding elements in parallel and later iterating over them in parallel, message queues restricted to a single-sender and single-receiver, etc. We fully expect that a released version of Yada will include more such types based on our experience parallelizing more programs. However, even as is Yada is able to express a significant variety of parallel algorithms, as we discuss in Sections 5 and 7. In Section 3 we formalize the rules presented above as restrictions on the the parallel **and** sequential composition of operations on shared types. This formalism is able to capture the restrictions necessary for all our current and envisioned sharing types.

An implementation of Yada is free to exploit all or none of the parallelism expressed with `forall` and $|;|$. However, irrespective of the amount of parallelism in use, the sharing rules described above are checked in terms of the program's logical parallelism. Thus the Yada compiler and runtime can decide on a parallel execution strategy independently of the data sharing present in the program (though they may still want to consider sharing for performance reasons).

In the current Yada prototype, these sharing checks are performed at runtime in a special sequential backend that tracks a program's logical parallelism and checks all reads and writes (Section 4.2). We chose runtime rather than static checking for two main reasons. First, our major goal in Yada was to see whether sharing types are a good and effective approach for expressing parallelism, independent of whether they are practically statically checkable or not. Second, DPJ [7] suggests that statically checking sharing is complicated, imposes a significant additional annotation burden, and is not able to express some operations (e.g. permuting an array of bodies in a barnes-hut N-body simulation to improve locality).

The current parallel backend performs no checks and so will only produce correct results if the input does not cause it to violate any of the sharing type rules, i.e. the same input would pass the sequential backend without any runtime errors. It is however clear that a future Yada implementation should perform a static analysis to remove unnecessary checks and report any detected sharing errors, and that the remaining runtime checks should be implemented in a parallel backend, to be able to check determinism on large inputs.

$$
\begin{array}{rcll}
\text{(locations)} & l & \in & L \\
\text{(type)} & \tau & \in & T \\
\text{(type decls.)} & \rho & : & L \to T \\
\text{(operator)} & o & \in & O \\
\text{(operation)} & e & ::= & l.o() \\
\text{(statement)} & s & ::= & \epsilon \mid e \mid \textbf{seq}\ s_1, ..., s_n \mid \\
 & & & \textbf{par}\ (s_1, s_1'), ..., (s_n, s_n') \\
\text{(trace)} & t & ::= & \langle \rho, s \rangle
\end{array}
$$

Figure 2: Execution traces.

2.2. Prototype

We implemented Yada as an extension to C, but the Yada concepts are readily applicable to any other imperative language. We view this implementation as a prototype designed to test the expressiveness of Yada and to investigate the achievable parallel speedups. The prototype is not intended to be a final, complete implementation of a deterministic parallel language and thus takes a number of shortcuts. First, the programmer must do some amount of manual tuning to get good performance: choose under which conditions `forall` loops should be run sequentially rather than in parallel, and choose between multiple implementations of the same sharing type with different performance tradeoffs. Second, the prototype does not guarantee complete determinism, e.g. the addresses returned by `malloc` may be different from run to run, no attempt is made to control non-determinism due to system calls, etc.

3. Determinism Framework

We formally define our determinism-checking rules as an operational semantics[7] over execution traces (Figure 2), which capture a Yada program's nested parallelism structure and the operations performed on a collection of locations of known types (the arguments to the operations are unimportant). The **par** construct captures the two parts (s, s') of a parallel loop containing an `andthen` pseudo-label ($s' = \epsilon$ if it is absent).

For example, the program

```
int writeonce x;
forall (i = 0; i < 2; i++)
  x = i;
```

would generate the trace

$$\textbf{par}(x.wr(), \epsilon), (x.wr(), \epsilon)$$

with $\rho(x) =$ writeonce.

[7]Interested readers not familiar with operational semantics may wish to refer to Plotkin's structured operational semantics paper [28].

$$
\begin{array}{rrcl}
\text{(state)} & \sigma & \in & S = \{\bot, \top, ...\} \\
\text{(environment)} & \Sigma & : & L \to S \\
\text{(state of operator)} & \mathit{ops} & : & (T \times O) \to S \\
\text{(sequential state composition)} & \triangleright & : & S \times S \to S \\
\text{(andthen state composition)} & \triangleright_b & : & S \times S \to S \\
\text{(parallel state composition)} & || & : & S \times S \to S
\end{array}
$$

$$\rho \vdash \epsilon : \lambda l.\bot$$

$$\rho \vdash l.o() : (\lambda l'.\bot)[l \mapsto (\Sigma(l) \triangleright \mathit{ops}(\rho(l), o))]$$

$$\frac{\rho \vdash s_1 : \Sigma_1 \quad ... \quad \rho \vdash s_n : \Sigma_n}{\rho \vdash \textbf{seq } s_1, ..., s_n : (\Sigma_1 \triangleright ... \triangleright \Sigma_n)}$$

$$\frac{\begin{array}{ccc}\rho \vdash s_1 : \Sigma_1 & ... & \rho \vdash s_n : \Sigma_n \\ \rho \vdash s'_1 : \Sigma'_1 & ... & \rho \vdash s'_n : \Sigma'_n\end{array}}{\rho \vdash \textbf{par } (s_1, s'_1)..., (s_n, s'_n) : ((\Sigma_1 \triangleright_b \Sigma'_1)||...||(\Sigma_n \triangleright_b \Sigma'_n))}$$

$$\frac{\rho \vdash s : \Sigma \qquad \nexists l : \Sigma(l) = \top}{\vdash \langle \rho, s \rangle}$$

Figure 3: Determinism-checking semantics.

Figure 3 specifies the overall structure of our rules for checking determinism. They essentially match the informal intuition given earlier: determinism is ensured by restricting the sequential and parallel composition of operations on each location. The formal specification of these rules uses a state $\sigma = \Sigma(l)$ to track the effects of the operations in each statement s on a location l. These effects are computed by the $\rho \vdash s : \Sigma$ judgment. By convention, $\sigma = \bot$ represents a location untouched by s and $\sigma = \top$ represents a determinism violation. Figure 3 guarantees a few general properties of determinism checking: the states for each location l are independent of each other and depend only on the operations performed on l and the trace's sequential and parallel structure.

Each type specifies its rules for enforcing determinism by specifying the *ops*, $\triangleright$, $\triangleright_b$ and $||$ state computation functions and operators (these extend naturally to apply to environments). As locations are independent of each other, these can be viewed as being defined independently for each type. The *ops* function maps each operation to a state, $\triangleright$ specifies the effects of the sequential composition of operations and $||$ the effects of parallel composition. Finally, $\triangleright_b$ specifies how to combine the effects of the two parts of a parallel loop iteration separated by `andthen`.

Figures 4 and 5 specify the sharing rules for writeonce and scan types; we omit here the rules for plain and reduce types as they are straightforward. The writeonce rules are simple: a write is forbidden after any other operation. The parallel composition rules are identical to the sequential ones, and thus enforce that any write in a parallel loop must happen before any read in the parallel

$$
\begin{array}{rcl}
\sigma & ::= & \bot < R < W < \top \\
o & ::= & rd \mid wr \\
ops(\tau, o) & = & \begin{cases} R & \text{if } o = rd \\ W & \text{if } o = wr \end{cases} \\
s_1 \triangleright s_2 & = & \begin{cases} \top & \text{if } s_1 \neq \bot, s_2 = W \\ s_1 \sqcup s_2 & \text{otherwise} \end{cases} \\
s_1 \triangleright_b s_2 & = & s_1 \triangleright s_2 \\
s_1 || s_2 & = & s_1 \triangleright s_2
\end{array}
$$

Figure 4: Determinism rules for $\tau =$ writeonce.

$$
\sigma ::= \quad \bot \rightarrow \begin{matrix} R \\ A \end{matrix} \rightarrow \begin{matrix} R_X \\ S \\ A_X \end{matrix} \rightarrow W \rightarrow \top
$$

$$
\begin{array}{rcl}
o & ::= & rd \mid wr \mid acc \\
ops(\tau, o) & = & \begin{cases} R & \text{if } o = rd \\ W & \text{if } o = wr \\ A & \text{if } o = acc \end{cases} \\
s_1 \triangleright s_2 & = & s_1 \sqcup s_2 \\
s_1 \triangleright_b s_2 & = & \begin{cases} R_X & \text{if } s_1 \sqcup s_2 = R \wedge s_1 \neq \bot \\ A_X & \text{if } s_1 \sqcup s_2 = A \wedge s_2 \neq \bot \\ W & \text{if } s_1 \sqcup s_2 = S \wedge \neg(s_1 = A \wedge s_2 = R) \\ s_1 \sqcup s_2 & \text{otherwise} \end{cases} \\
s_1 || s_2 & = & \begin{cases} \top & \text{if } s_1 \sqcap s_2 \neq \bot \wedge s_1 \sqcup s_2 = W \\ s_1 \sqcup s_2 & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5: Determinism rules for $\tau =$ scan(op).

loop's implicit sequential order.

Consider the trace shown earlier. From the rule for **par** in Figure 3, we know that we must check the two statements separately and combine them with $||$. Both consist of only a single write, so they will both generate $\Sigma(x) = W$. To combine them, we compute $W||W = \top$, and so we have found a violation.

Scan locations have three operations, read (rd), write (wr) and accumulate (acc). Scan locations require a number of extra states beyond those corresponding directly to individual operations (R, W, A) to enforce the scan rules: if accumulates are performed in parallel with reads, then all accumulates must be before the `andthen` and all reads must be after the `andthen`. The R_X (respectively A_X) state tracks reads that occur before an `andthen` (respectively accumulates that occur after an `andthen`) and thus cannot happen in parallel with accumulates (respectively reads). The S state is used for scan locations on which parallel accumulates and reads are "in progress". The lattice for σ defines

the sequential composition rules for scan locations. The parallel composition rules are actually identical to the sequential ones, except that sequential compositions that map to the write state represent determinism violations when they occur in parallel (except in the trivial case where one state is $\perp$). The `andthen` composition rules also match the sequential rules, but with three exceptions to enforce the scan rules: reads before an `andthen` map to R_X, accumulates after an `andthen` map to A_X, and only accumulates before an `andthen` combined with reads after an `andthen` map to S. Note also that two nested parallel loops cannot both accumulate and read the same scan location as either the inner loop's reads occur before, or its accumulates occur after, the outer loop's `andthen`.

The restrictions described here for writeonce and scan locations cannot be expressed as parallel commutativity rules on the read, write and accumulate operations. It is thus not possible to express these kinds of sharing with the commutativity declarations found in Jade [31], DPJ [7] and Galois [20].

4. Yada Prototype

Our Yada prototype targets 32 and 64-bit Intel® processors on Linux and Mac OS X machines (it would not be hard to port to other architectures and operating systems). It is composed of a Yada to C++ compiler, and two backends: a serial, determinism-checking backend (for debugging) and a parallel (unchecked) backend (for performance).

4.1. Compiler

The Yada compiler is implemented as an extension to the CIL [26] C frontend, and generates C++ code. Sharing types are specified as type qualifiers on existing base types. These qualifiers both specify the sharing type and select amongst the available implementations: for instance, `int reduce_atomic(+)` selects the atomic-update implementation of the `int reduce(+)` sharing type (Section 4.3.2). Parallel loops are specified by an annotation on regular C `for` loops (an error is reported if the loop doesn't correspond to an iteration over an integer range). Parallel execution of n statements is translated to the obvious parallel loop over the $1..n$ range.

Generating C++ simplifies the implementation of sharing types: a sharing type is implemented as a C++ class with a constructor and destructor to automate the management of its internal state, e.g. to initialise a lock, and methods that implement the type's operations. To ensure that constructors and destructors are called for heap-allocated data, Yada supports C++'s `new` and `delete` syntax.

Parallel loops are compiled by generating a closure for the body. The closure function and closure state are then passed to a runtime library function that executes the parallel loop. The decision of how to execute the parallel loop is left completely to the runtime. The closure state is a structure with a field for each local variable accessed in the loop. These fields are pointers to locals

unless a variable's sharing type indicates that an optimisation is possible. For instance, most sharing types place the local's value in the closure if the body contains only reads (but this is not a safe transformation for `writeonce` locals). We do not detail the closure generation or these optimisations here as they are straightforward. We defer discussion of two-part loops using `andthen` to Section 4.3.3, which presents the implementation of `scan(op)` sharing types.

After closures have been generated for all parallel loops, the compiler then rewrites all accesses to sharing types. These are replaced by a call to an inlined method on the C++ class representing the sharing type.

4.2. Determinism Checking Backend

Yada's sequential backend essentially performs the checking rules outlined in Section 3 online during the execution of the program and reports any determinism violations as they occur. This implementation has not been optimised for performance and currently causes a large slowdown (20x-75x), which is somewhat acceptable when debugging, but clearly far from ideal. We believe this overhead can be significantly reduced by a better implementation.

When using the determinism-checking backend, for each operation o on a shared location l, the compiler inserts a runtime call to `detcheck_`o`(&`l`)`. The runtime uses a hash-table (indexed by the address of l) to track l's state as of the last access at each parallelism nesting level. The `detcheck` function then updates these states using either the $\triangleright$ or $||$ operators as appropriate. If any state becomes $\top$, a determinism violation is reported.

Finally, the determinism checker needs to interact with the execution of parallel loops using `andthen` (to emulate the $\triangleright_b$ operator), and with memory allocation, including allocation of locals on the stack, to avoid reporting invalid errors due to address reuse.

4.3. Parallel Backend

The parallel backend is built over the Intel®Threading Building Blocks (TBB)[8] library version 2.2, which supports nested parallelism using a work-stealing scheduler. Simple parallel loops map to a single TBB `parallel_for` that invokes the body closure. We discuss two-part parallel loops using `andthen` along with the implementation of `scan()` sharing types below.

At startup, the user specifies the number P of threads to create to run the Yada program. To help implement the sharing types, each thread has an identifier `myproc` between 0 and $P - 1$.

The most interesting part of the parallel backend is the implementation of the sharing types. The implementation of these types is greatly simplified (and their performance enhanced) by the fact that they can assume that the sharing rules described in Section 2.1 and formalized in Section 3 are respected. In particular, locations without a sharing type specifier ("plain" sharing) only

[8] Retrieved from `http://www.threadingbuildingblocks.org` in November 2009.

allow read-read parallelism and so can be compiled and optimised as if the code were sequential.

We present a brief overview of the implementation of each sharing type t as it applies to a location l of type pointer-to-t. Each implementation description starts with a reminder of the rules for each type as presented in Section 2.1.

4.3.1. writeonce

A location l of type `t writeonce *` has two operations: $*l$ (read) and $*l = v$ (write). The only legal parallel combinations are read||read and write||read (the write occurs before the read in Yada's implicit sequential execution order).

Yada provides two implementation of `writeonce`. The first, `writeonce_busy`, implements t `writeonce` as a structure containing a value of type t and a `written` flag. Reads simply busy-wait until `written` is true. The second implementation, `writeonce_sleepy`, adds a lock and condition variable. When `written` is false, reads wait on the condition variable (which is signaled by writes) rather than busy-waiting. `writeonce_busy` has lower space overhead, but should only be used when waits are expected to be unusual or of short duration. Currently, the programmer must manually choose between these implementations.

4.3.2. reduce

A location l of type `t reduce(op) *` has three operations: $*l$ (read), $*l = v$ (write) and $*l$ *op*$= v$ (accumulate). The only legal parallel combinations are read||read and accumulate||accumulate.

Yada also provides two implementations of `reduce`, and the programmer must manually specify which to use. The first, `reduce_atomic`, is best when contention on l is low: in the compiled program, l simply has type t, reads and writes are unchanged, and accumulate is compiled to an atomic operation implemented as an atomic instruction for the supported combinations of *op* and t (C operators on integer types) and a compare-and-swap loop for all other cases.

The second, `reduce_rep`, is best when contention on l is high: l is represented by a structure containing l's value and a P element array that allows threads to perform independent, unsynchronised accumulates on l. A subsequent read of l (which cannot happen in parallel with the accumulates) must update l's value based on the contents of the array. Figure 6 gives the pseudocode of `reduce_rep`'s implementation.

To avoid performance problems caused by false sharing, the `values` array actually stores one element per cache line.[9] To save memory and improve locality, an array of n `t reduce_rep(op)` locations is represented slightly differently: `value` becomes an n element array, and `values` is a two-dimensional $P \times n$ array where the n elements for each thread are stored contiguously.

[9] Cache line size is configured in the compiler on a per-target basis.

```
template<t> class reduce {
  t value;
  t *values;
  reduce() { values = 0; }
  ~reduce() { if (values) delete values; }

  void write(t v) { // No parallelism
    if (values) { delete values; values = 0; }
    value = v;
  }
  void accumulate(t by) { // Parallelism with itself
    if (!values) allocate_reduce_values();
    values[myproc] op= by;
  }
  t read() { // Parallelism with itself
    if (values) reduce();
    return value;
  }

  void allocate_reduce_values() {
    ... allocate values ...
    ... handle races when called in parallel ...
  }
  void reduce() {
    ... update value from values, delete values ...
    ... handle races when called in parallel ...
  }
};
```

Figure 6: Replicated reduce implementation for operator *op*.

4.3.3. scan

Like `reduce`, a location l of type `t scan(op) *` has three operations: $*l$ (read), $*l = v$ (write) and $*l$ *op*$= v$ (accumulate). However, scan locations allow both accumulates and reads in the same parallel loop, under the condition that the accumulate happens in the loop's first part (before `andthen`), and the read happens in the loop's second part (a two-part parallel loop can also perform just reads or just accumulates, but these cases need no additional handling). To simplify scan implementation, our prototype restricts accumulates to the innermost parallel loop. We have not found this restriction problematic in practice.

Essentially, a parallel loop of the form

```
forall (i = s; i < e; i++)
  body_a;
andthen:
  body_b;
}
```

that accesses scan locations $l_1, \ldots, l_n$ is executed as:

```
l1->setup(e - s); ... ln->setup(e - s);
forall (i = s; i < e; i++)
  body_a;
l1->prefix(); ...; ln->prefix();
forall (i = s; i < e; i++)
  body_b;
l1->cleanup(e - s); ...; ln->cleanup(e - s);
```

Each location l is represented by a structure containing a `value` field containing l's value before the parallel loop, and an $n = e - s$ `values` array that collects the values accumulated by `body_a` in each iteration. The `prefix` operation then performs a parallel-prefix operation on `values`. Finally, reads in `body_b` read the element of `values` that corresponds to the loop iteration. After the loop completes, `cleanup` updates l`->value` to reflect l's final value and the array is discarded. This implementation is summarised in Figure 7.

The set of locations accessed by a parallel loop is not necessarily discoverable at compile-time. Therefore, the Yada prototype simply tracks this set dynamically (the details are straightforward).

This implementation has one significant problem: when the body of the forall loop is simple (performs very little computation) and the iteration space is large, the overheads of scan locations swamp any gain from increased parallelism. This is true of the examples we saw in radix sort in Section 2, and in fact of all the uses of scan in our evaluation (Section 5). In particular, the requirement of an n element array for an n iteration parallel loop significantly increases memory-bandwidth consumption in simple loops.

In many cases, and in all cases in our benchmarks, we can fix this performance problem by an alternate compilation strategy while leaving the scan implementation nearly unchanged. If `body_a`, the first half of the parallel loop, has side effects **only** on scan locations, then we can execute our parallel loop as:

```
nchunks = 4 * P; // 4 is arbitrary, but works well
l1->setup(nchunks); ... ln->setup(nchunks);
forall (chunk = 0; chunk < nchunks; chunk++)
  for (i = s(chunk); i < e(chunk); i++)
    body_a;
l1->prefix_exclusive(); ...; ln->prefix_exclusive();
forall (chunk = 0; chunk < nchunks; chunk++)
  for (i = s(chunk); i < e(chunk); i++) {
    body_a;
    body_b;
  }
l1->cleanup(nchunks); ...; ln->cleanup(nchunks);
```

where $s(i)$ and $e(i)$ compute the loop bounds for the ith chunk and `prefix_exclusive` computes the exclusive-parallel-prefix rather than the parallel-prefix (where the

```
template<t> class scan {
  t value;
  t *values;
  scan() { values = 0; }

  void write(t v) {
    value = v;
  }
  void accumulate(t by) {
    values[current_loop_index] op= by;
  }
  t read() {
    if (values)
      return value + values[current_loop_index];
    else
      return value; // no accumulate in this loop
  }

  void setup(int n) {
    values = new t[n];
  }
  void prefix() {
    ... perform parallel-prefix on values using op ...
  }
  void cleanup(int n) {
    value op= values[n - 1];
    delete values;
    values = 0;
  }
};
```

Figure 7: Scan implementation (simplified) for operator *op*. `current_loop_index` is a thread-local variable containing the value of the innermost parallel loop iteration.

ith result element does not include the contribution of the element at i):

$$\text{prefix\_exclusive}(A, op)[i] = A[0]\ op\ \ldots\ op\ A[i-1]$$

This reduces the loops' parallelism to four times the number of threads, which is typically enough for load-balancing,[10]and greatly reduces both the memory size of a scan location and the computational cost in the parallel-prefix step between the two parallel loops (in fact, we execute `prefix_exclusive` sequentially in our prototype, as we currently use at most 8 threads). This alternate execution strategy is valid because the re-execution of `body_a` in the second parallel loop leaves each of the scan locations with its correct value.

[10]The value four was selected empirically to reduce overhead while maintaining sufficient load-balance on our benchmarks.

Program	Lines of code		
	Seq.	Par.	Yada.
quick sort	34	184(T)	57
radix sort	46	61(S)	68
merge sort	40	170(T)	114
sample sort	73	164(S)	84
shell sort	29	39(T)	33
bitonic sort	117	131(T)	118
pure bitonic sort	39	47(T)	39
cholesky	30	58(S)	45
ebarnes	2066	2107(S)	1944
bzip2	-	2613(P)	1596
ammp	13484	13521(O)	13546
mg	1742	1774(O)	1792

Table 1: Benchmark Sizes (the top eight are algorithms, the bottom four are applications). The parallel sizes are followed by a letter in parenthesis indicating the parallel application's programming style: (T) for the TBB library, (S) for SPMD-style, (P) for raw use of pthreads and (O) for OpenMP.

5. Evaluation

We evaluated Yada on eight algorithms and four applications, covering a reasonably diverse set of programming domains: sorting algorithms (7), scientific algorithms and applications (4), and a file-compression tool (bzip2). We investigated the sorting algorithms to see if Yada could express parallelism and get useful speedups in a diverse set of algorithms. We chose the scientific applications and pbzip2 because we could compare Yada's performance with existing parallel implementations and because most of them exhibited non-trivial data-sharing patterns. For all algorithms and applications we either wrote sequential and parallel versions, or, when possible, reused the existing sequential and parallel implementations.

The sorting algorithms are quick sort, radix sort, merge sort, sample sort, shell sort and two variations on bitonic sort. The sequential implementation is a natural implementation of the algorithm. The parallel implementations are implemented either using the Intel®Threading Building Blocks (TBB) library (quick sort, merge sort, shell sort and the bitonic sorts) or a custom library (inspired by and derived from the Split-C [19] parallel C dialect) for writing SPMD-style parallel programs (radix sort and sample sort).

The scientific algorithm is a cholesky decomposition. The sequential code is the natural implementation of the algorithm. The parallel version uses presence booleans for each matrix cell to wait for results computed by other threads. The scientific programs are ammp, a molecular dynamics simulator included in the SPEC2000 (in a sequential version) and SPECOMP2001 (in a parallel OpenMP version) benchmark suites, mg, a multigrid solver for the 3-D scalar

Program	Number of foralls	writeonce	reduce()	scan()
quick sort	2			2
radix sort	5		1	2
merge sort	2			
sample sort	3			2
shell sort	1			
bitonic sort	3			
pure bitonic sort	1			
cholesky	1	1		
ebarnes	4		1	
bzip2	4	2		
ammp	7		12	
mg	11		2	

Table 2: Yada parallel annotation statistics.

Poisson equation from the NAS parallel benchmark suite [3] (available in both sequential and OpenMP versions, and ported to C by the Omni project[11]), and ebarnes, a Barnes-Hut particle simulator. The parallel ebarnes code builds its oct-tree in parallel using locks, while the sequential version uses a divide-and-conquer approach. We note that mg is typical of many SPMD-style scientific programs: it basically exhibits only parallel read-sharing. Such programs are deterministic, and easy to translate to Yada.

The pbzip2 utility parallelises compression and decompression of bzip2 files. It uses a single file-reading and file-writing thread, and n compression threads to both parallelise the compression work and overlap I/O and computation. We use the regular bzip2 utility as the reference sequential implementation.

The parallel versions of sample sort, cholesky and ebarnes are ports of Split-C [19] programs.

5.1. Parallelisation with Yada

Table 1 compares the size of sequential, Yada and parallel versions of our benchmarks. We omit the sequential size for bzip2 as it is not directly comparable to the parallel or Yada versions.

Generally, Yada code is close in size to the sequential code and in some cases (bzip2, quick sort, merge sort, sample sort) significantly smaller than the parallel code. For bzip2, this is mostly due to the removal of the parallel version's work queue and synchronisation logic. The difference in the three sorting algorithms is due to a combination of relative verbosity of code built using the TBB library

[11] Retrieved from `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp` in November 2009.

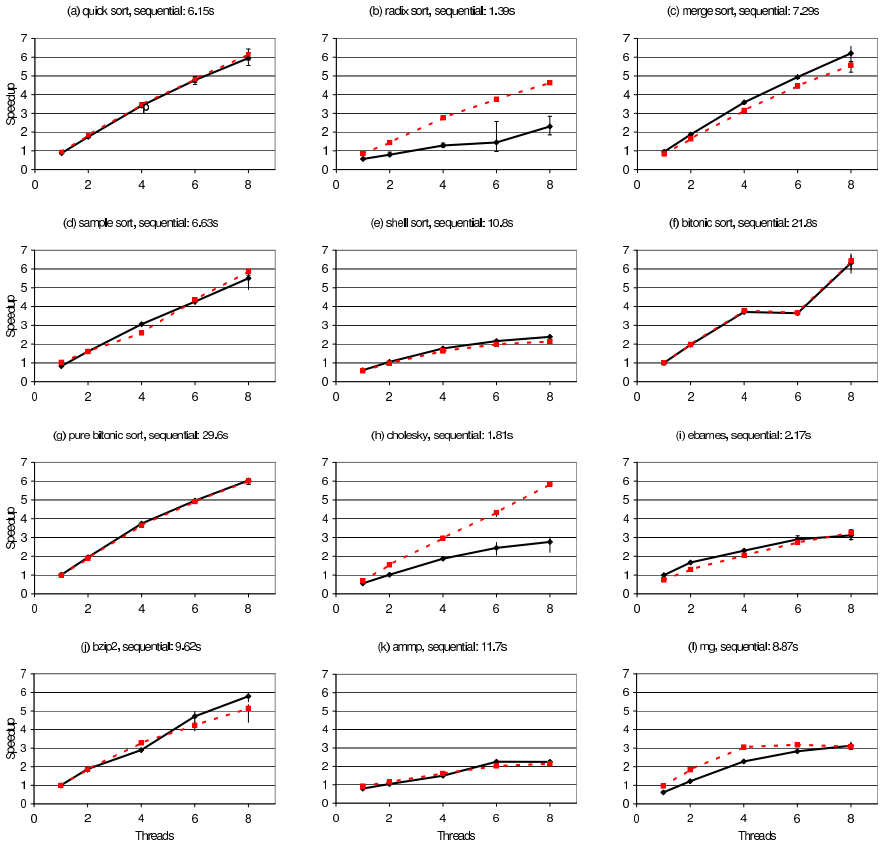

Figure 8: Speedups vs sequential implementations. Solid line: Yada, dotted line: Parallel.

and, for merge and quick sort, the inclusion of a sequential version of the sorting algorithm to handle the base case of the recursive sorts efficiently. The Yada code is able to avoid this extra copy by conditionally selecting between parallel and sequential execution of its loops — when sequential execution is selected the code and performance is essentially identical to the sequential code in these benchmarks.

Table 2 presents statistics on the number of parallel loops (and parallel statement sequences) and sharing types used in the Yada version. All the applications and half of the algorithms need to use at least one sharing type, and all the sharing types are used in at least two benchmarks. This provides reasonable evidence that our sharing types capture styles of deterministic data sharing commonly found in parallel programs. It is worth noting that for most benchmarks, as we saw with radix sort in Section 2, the Yada version is very similar to the sequential version. The two exceptions are bzip2, where the Yada

code is based on the parallel version and merge sort, where the parallel merge algorithm is completely different than the sequential one. The same similarity with the sequential code holds for the OpenMP-based parallel codes.

We next briefly describe the changes and annotations required to convert each benchmark to Yada. Quick sort parallelises both the step of splitting the array around the pivot and performing the two recursive sorts in parallel. The natural sequential code to split an array `A` around a pivot, leaving the results in array `B`, is:

```
int j = 0, k = n;
for (i = 0; i < n; i++) {
  if (A[i] < pivot) B[j++] = A[i];
  else if (A[i] > pivot) B[--k] = A[i];
}
```

This loop is paralleliseable in Yada using two scan sharing types because the updates to j and k can easily be separated from the reads by an `andthen` pseudo-label, in a similar fashion to the examples for radix sort in Section 2:

```
int scan(+) j = 0, k = n;
for (i = 0; i < n; i++) {
  if (A[i] < pivot) j++;
  else if (A[i] > pivot) --k;
andthen:
  if (A[i] < pivot) B[j - 1] = A[i];
  else if (A[i] > pivot) B[k] = A[i];
}
```

This pivot-splitting loop is only run in parallel at the outermost levels of the quick sort recursion because using scans has significant overheads and only makes sense when other sources of parallelism are not available.

Radix sort was examined in detail in Section 2.

Merge sort parallelises both the recursive calls to merge sort, and the subsequent merge. The parallel calls touch independent data so no sharing types are required.

Sample sort can be viewed as a generalisation of quick sort: a small, fixed-size sample of the array is sorted and used to split the array according to k pivots. These k sub-arrays are sorted in parallel then combined back into a single array. The Yada version parallelises all these steps except the array sampling and selection of the pivots. In particular, the splitting of the array into k buckets is very similar to the last loop of radix sort (line 16 of `radix_C` in Figure 1) and also directly translatable into a Yada `forall` loop using an array of `int scan(+)` variables.

The shell and bitonic sorts contain loops that can easily be rewritten to touch independent data and are therefore simple to parallelise.

The cholesky matrix decomposition uses `writeonce_busy` variables to track whether each matrix element's value has been computed — these essentially correspond to the presence booleans in the parallel version.

The parallelisation of the oct-tree construction in ebarnes is based on the sequential divide-and-conquer version, and simply parallelises the conquer step. Attempts to parallelise the divide step failed to give any parallel speedup, as the changes required swamped any gains from parallelism: the sequential version of the Yada code ran approximately 8x slower than the original sequential divide-and-conquer oct-tree construction. The particle updates are easily parallelised as they are independent of each other.

The OpenMP version of the ammp molecular dynamics simulation uses locks to protect concurrent updates to the force and potential fields of multiple atoms. However, it turns out that these concurrent updates all take the form of additions and subtractions and are thus easily expressed in Yada by declaring twelve fields in the ATOM structure with `reduce(+)`. With these annotations and the elimination of some redundant assignments, the Yada ammp code deterministically expresses the same parallelism as the OpenMP version.

The parallelisation of mg is straightforward: it involves only independent updates, except for two scalar variables of type `double reduce(+)` and `double reduce(max)`.

Bzip2 is a data compression algorithm. Its compression code reads a file into equally-sized blocks, which are then compressed separately and concatenated together into an output file. The Yada pseudocode for compression is shown below; decompression behaves similarly.

```
struct buffer {
  char * writeonce_sleepy buf;
  unsigned int writeonce_sleepy bufSize;
} *in, *out;
void compress() {
  for (int i = 0; i < numBlocks; i++)
    ...  read parts of the file into in[i] ...
  |;| // in parallel with...
  forall (int i = 0; i < numBlocks; i++)
    ...  compress in[i] and write it into out[i] ...
  |;| // in parallel with...
  for (int i = 0; i < numBlocks; i++)
    ...  write out[i] into a file ...
}
```

The middle loop operates on independent data and can easily be parallelised. To overlap the I/O with compression, our Yada code uses two writeonce locations within an array. As soon as the first chunk has been read and written into the `in` array, the corresponding `forall` loop iteration can compress it in parallel with the rest of the file reads. Similarly, once a chunk has been compressed and written into the `out` array, it can be written to a file. This simple structure replaces pbzip2's work-queue (about 360 lines of code), used to communicate between the reader and the compressor, and its array+busy-wait communication (about 110 lines) between the compressors and the writer. Furthermore, testing of the Yada version revealed that performance was significantly more stable

when the writer ran sequentially after the readers and compressors, which we accomplished by replacing the last |;| with a normal ;.

5.2. Performance

Our tests were performed on a dual quad-core Intel®Xeon®X5570 machine with 6GB of memory, running Linux 2.6.24, gcc 4.1.2 (which includes OpenMP support) and version 2.2 of Intel's®TBB library. The performance of Yada on our benchmarks is summarised in Figure 8. The graphs report the performance of the Yada and parallel code for 1, 2, 4, 6 and 8 threads as a speedup relative to the sequential version. Each benchmark is run 10 times and its execution time averaged. The error bars (when visible) show the speedups of the fastest and slowest of the 10 parallel executions (always computed with respect to the average sequential time, but the sequential time is a lot less variable than some of the parallel times).

The inputs for the programs were as follows: the sorting algorithms sorted an array of 50 million (2^{25} for the pure bitonic sort) randomly generated 32-bit integers, cholesky decomposition was applied to a 2048x2048 matrix, ebarnes simulated 2^{20} particles distributed on a sphere over 4 time steps (oct-tree construction took significant time on this input, speedups reach 6x on other distributions), bzip2 compressed a 50 million byte tar ball composed of text files, ammp was run on the SPEC2000 training input and mg was run on its 256x256x256 "class B" input.

Yada achieves a speedup on all benchmarks. Furthermore, in most cases, and in all our test applications, Yada's performance is comparable to the parallel version. In two cases, radix sort and cholesky, Yada has significantly worse performance than the parallel code. For cholesky, the difference is due to the fact that our `writeonce_busy`-based presence booleans take two bytes vs the one byte in the parallel version. For radix sort, the parallel version is very similar to a hand-optimised version of the Yada code. A better Yada compiler should be able to match the performance of the parallel code.

We also investigated what input sizes are required to get parallel speedups with Yada. Figure 9 shows the speedup vs sequential code of four of our sorting algorithms. For quick, merge and sample sorts, sorting 10,000 elements or more provides significant speedups. Unsurprisingly given its poor scaling on large inputs, the Yada radix sort only breaks even at one million elements.

Finally, it is worth noting that the parallel speedups on several of the benchmarks are around 3 on 8 cores. This is comparable to the 2.5x speedup (from 10 GB/s on one core to 25GB/s on eight cores) obtained on the Stream [23] bandwidth-testing benchmark. This suggests that these benchmarks' sequential and parallel performance is mostly bandwidth limited, and hence their parallel speedup cannot much exceed the ratio of the maximum parallel bandwidth to the maximum sequential bandwidth.

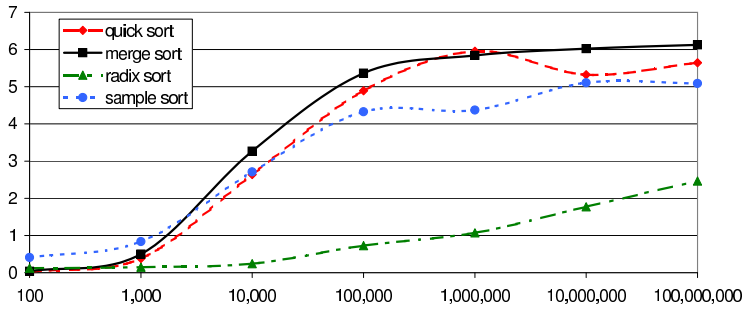

Figure 9: Speedup vs input size of sorting algorithms on 8 cores.

6. Related Work

There has been a significant amount of work on deterministic and mostly deterministic parallel languages. One approach to guaranteeing determinism is to forbid side-effects, i.e. program in a pure functional language. This is the approach taken most notably by NESL [4] and SISAL [24], and in some recent languages [9, 17]. We believe that providing determinism in imperative languages provides many of the benefits of parallel functional programming while being more familiar to most programmers. Data-parallel languages, such as High Performance Fortran [16] and ZPL [10], are also deterministic by construction. They achieve good performance on regular problems, but make it difficult or impossible to express less structured parallelism, e.g. the overlapped I/O and computation in the parallel bzip2 benchmark. As we discussed in the introduction, a number of parallel languages [6, 7, 1, 31, 2] were designed to "feel sequential". However, they mostly only allow parallel read sharing, or relax determinism to allow more general programs. They would thus have trouble ensuring that many of our benchmarks execute deterministically.

Several previous languages and systems have concentrated on restricting sharing to ensure determinism. Terauchi and Aiken's capability calculus [33] uses splittable capabilities to statically ensure determinism in programs with message queues, Kahn networks and shared locations. Fortran M [11] uses a similar capability-based approach to check accesses to shared locations, and also includes deterministic message queues and writeonce variables. C**'s *reduction assignments* [21] and MSA's `accumulate` arrays [13] provide similar functionality to our `reduce()` shared types. The Galois system [20] focuses on unordered (non-deterministic, but still sequential) loop execution with potentially concurrent updates to shared objects. Conflicts are automatically detected (based on method commutativity annotations) and lead to rollback of one of the iterations. In contrast, Yada focuses on checking that iterations do not conflict (and hence that the loop will execute in parallel). More generally, Yada supports sharing types not expressible using the commutativity annotations provided by Galois, DPJ and Jade.

Some techniques check determinism in existing code. Dynamic analysis [15, 32] can be used to check whether an execution of a program is deterministic. Burnim and Sen [8] allow programmers to write assertions that specify the exact amount of determinism required (which may be somewhat relaxed) and check these assertions through testing. Static analysis [30] techniques can check sequential programs for commutativity and parallelise them if all of their operations commute. These approaches work on existing code, but they can only report errors, not ensure that programs always execute deterministically.

Another approach is to force repeatability onto programs. Record and replay techniques record the thread interleavings of an execution and allow deterministic replays. Recent advances [25] have reduced the overheads of such approaches, but they still incur significant slowdowns compared to normal execution. DMP [14] requires processes to hold tokens when accessing memory and pass them deterministically. It requires hardware support to be efficient. Kendo [27] achieves good performance without specialized hardware by constructing a deterministic logical time, but it only guarantees a deterministic order of lock acquisitions and so requires a race detector to achieve true determinism. While all of these approaches achieve some level of determinism, none help programmers understand their programs: small changes in a program can lead to large changes in its execution.

Finally, decoupled software pipelines [29] are an alternate way of parallelising our two-part parallel loops separated by `andthen`: one thread can run the first part, communicating the partial results of `scan()` locations via special hardware support to a second thread executing the second part of the loop. However, this provides only a maximum two-fold speedup, unlike Yada's two-stage loops that allow parallelism up to the number of loop iterations n. Conversely, decoupled software pipelines support parallel execution even when the updates are not performed with an associative operator (and extend to k-way parallelism if the loop can be split into k independent parts). Adding speculation to decoupled software pipelines [34] can introduce extra parallelism when certain dependence edges are rarely realised.

7. Conclusion and Future Work

Our experience with Yada has shown that it is practical to express realistic algorithms and applications in a deterministic programming language while still getting speedups that are competitive with implementations in non-deterministic programming environments. We believe that this style of parallel programming that feels much closer to sequential programming, and which allows programs to be debugged sequentially, will be much more approachable to mainstream programmers.

Yada's design is not targeted at obtaining the absolutely best possible performance from a parallel machine. However, when that is a goal, we still believe that Yada is a useful tool. First, when targeting a shared-memory machine, Yada should be a good first step in parallelization, as it allows exploration of different parallelization strategies, easy detection of when and where parallel

actions may step on each other's data, and the use of the sharing types to handle common ways of resolving these data sharing issues — reductions, scans, and data-flow (easily expressed with `writeonce`). This Yada program can then serve as the basis for a non-deterministic parallel program written, e.g., with OpenMP. Second, even when targeting a distributed memory machine, where the Yada version may be less obviously useful as a first step, Yada can still serve as an exploratory tool for how to parallelize the algorithm. Finally, it is worth repeating that Yada is actually quite expressive, as mentioned in the introduction: any SPMD program that uses only barrier synchronization and reduction and scan operators can easily be written in Yada. This includes all the NAS 3.3 parallel benchmarks except UA.[12]

Yada is still a work in progress. From a language design perspective, we have mostly concentrated on computational algorithms. Real programs often have user interfaces, act as servers responding to outside requests, etc. We plan to investigate whether Yada is suited to handling the concurrency often present in such programs, possibly via new sharing types to support deterministic, parallel I/O. Similarly, as argued by Pingali et al [20], the best known approach for parallelising some algorithms is non-deterministic. It would be desirable for Yada to provide effective mechanisms for interacting with non-deterministic code without losing its advantages of easy understanding and debugging. Finally, we will provide mechanisms to help detect determinism violations due to interactions with external sequential and parallel libraries.

We will also continue working on Yada's implementation to reduce the overheads of sharing types and of parallel loops and to optimise the performance of the determinism checker, in particular to allow parallel execution. To help programmers tune their Yada programs we need to provide feedback on performance tradeoffs in switching between sharing type implementations and on sequential vs parallel loop execution.

Finally, from a theoretical perspective, we aim to find a general proof framework for showing how the restrictions on sharing types preserve determinism.

References

[1] OpenMP application program interface version 3.0. http://www.openmp.org/mp-documents/spec30.pdf retrieved in June 2010.

[2] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *PPoPP '09*, pages 85–96.

[12] Determined by inspection of the usage of locks, atomics and critical sections (in the OpenMP versions) to protect shared data: all uses except those in UA can be expressed with `reduce(+)` or `reduce(max)`. UA would be expressible with the addition of sets with a parallel "add-to-set" operation, effectively a variation on `reduce`.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. 5(3):66–73, 1991.

[4] G. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21:102–111, 1994.

[5] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, Nov. 1990.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP'95*, pages 207–216.

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA'09*, pages 97–116.

[8] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *ESEC-FSE'09*, pages 3–12.

[9] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel Haskell: A status report. In *DAMP'07*.

[10] B. L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 2001.

[11] K. M. Chandy and I. Foster. A notation for deterministic cooperating processes. *IEEE Trans. Parallel Distrib. Syst.*, 6(8):863–871, 1995.

[12] J. Demmel. Private commmunication.

[13] J. DeSouza and L. V. Kalé. MSA: Multiphase specifically shared arrays. In *LCPC'04*.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS'09*, pages 85–96.

[15] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA'97*, pages 1–11.

[16] H. P. F. Forum. High performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, 1993.

[17] A. Ghuloum. Ct: channelling NESL and SISAL in C++. In *CUFP'07*, pages 1–3.

[18] G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475, Aug 1974.

[19] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93*, pages 262–273.

[20] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI'07*, pages 211–222.

[21] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory system support for parallel language implementation. In *ASPLOS'94*, pages 208–218.

[22] C. Lin and L. Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2008. ISBN 978-0321487902.

[23] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers, Oct. 1995. http://www.cs.virginia.edu/ mccalpin/papers/bandwidth/bandwidth.html.

[24] J. McGraw, S. Skedziewlewski, S. Allan, R. Oldehoeft, J. Galuert, C. Kirkham, and B. Noyce. SISAL: Streams and iteration in a single assignment language, 1986. Memo M-146, Lawrence Livermore National Laboratory.

[25] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS'09*, pages 73–84.

[26] G. C. Necula, S. McPeak, and W. Weimer. CIL: Intermediate language and tools for the analysis of C programs. In *CC'04*, pages 213–228.

[27] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS'09*, pages 97–108.

[28] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[29] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT'2004*.

[30] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, 1997.

[31] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[32] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP'09*, pages 394–409.

[33] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Program. Lang. Syst.*, 30(5):1–30, 2008.

[34] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT'2007*.