Efficient Parallel Programming in Poly/ML and Isabelle/ML

David C. J. Matthews
Prolingua Ltd
9/2 Damside
Edinburgh, EH4 3BB
Scotland

Makarius Wenzel
Technische Universität München
Institut für Informatik, Boltzmannstraße 3,
85748 Garching, Germany
http://www.in.tum.de/~wenzelm/

Supported by BMBF project "Verisoft" (01 IS C38)

Abstract

The ML family of languages and LCF-style interactive theorem proving have been closely related from their beginnings about 30 years ago. Here we report on a recent project to adapt both the Poly/ML compiler and the Isabelle theorem prover to current multicore hardware. Checking theories and proofs in typical Isabelle application takes minutes or hours, and users expect to make efficient use of "home machines" with 2–8 cores, or more.

Poly/ML and Isabelle are big and complex software systems that have evolved over more than two decades. Faced with the requirement to deliver a stable and efficient parallel programming environment, many infrastructure layers had to be reworked: from low-level system threads to high-level principles of value-oriented programming. At each stage we carefully selected from the many existing concepts for parallelism, and integrated them in a way that fits smoothly into the idea of purely functional ML with the addition of synchronous exceptions and asynchronous interrupts.

From the Isabelle/ML perspective, the main concept to manage parallel evaluation is that of "future values". Scheduling is implicit, but it is also possible to specify dependencies and priorities. In addition, block-structured groups of futures with propagation of exceptions allow for alternative functional evaluation (such as parallel search), without requiring user code to tackle concurrency. Our library also provides the usual parallel combinators for functions on lists, and analogous versions on prover tactics.

Despite substantial reorganization in the background, only minimal changes are occasionally required in user ML code, and none at the Isabelle application level (where parallel theory and proof processing is fully implicit). The present implementation is able to address more than 8 cores effectively, while the earlier version of the official Isabelle2009 release works best for 2–4 cores. Scalability beyond 16 cores still poses some extra challenges, and will require further improvements of the Poly/ML runtime system (heap management and garbage collection), and additional parallelization of Isabelle application logic.

DAMP'10, January 19, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-859-9/10/01. . . $10.00

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; I.2.3 [*Deduction and Theorem Proving*]: Inference engines

General Terms Design, Experimentation, Measurement, Performance.

Keywords Parallel Standard ML, data parallelism, Poly/ML, Isabelle, theorem proving applications

1. Introduction

There are very many users of Isabelle who are running proofs that take a considerable length of time. Isabelle and the Poly/ML platform that it uses have been engineered to run efficiently on a single processor, but at the start of this project (about two years ago) there was no possibility of exploiting the multi-processors that were increasingly appearing on desktops or laptops. The aim of this project was to allow users to make use of this parallelism in as simple a way as possible.

There were several constraints that applied. Isabelle and Poly/ML together are large projects developed over many years and it is simply not possible to redevelop them as parallel programs from scratch. User code runs on top of this and it would be unreasonable to expect users to redesign their code. They may use legacy features that need to be supported. The aim, therefore, was to develop mechanisms for parallelism which could be implemented efficiently and yet would hide the details from higher levels.

To do this the mechanisms for parallelism were developed as a series of layers providing increasingly more abstract views. The lowest levels are very close to the hardware and operating system and take an imperative approach while the higher levels are much more functional.

The parallel implementation is already part of the standard distribution of Poly/ML 5.2.1[1] and Isabelle2009[2] and users can benefit from running on a multiprocessor. Many users, though, will continue to use uniprocessors or be running other ML programs so it was essential that adding parallelism should not be at the expense of the performance of purely sequential programs.

In the present paper we shall discuss the following main components, with references to related work as we proceed.

1. Poly/ML with native support for operating system threads (§2). There is significant impact on the ML runtime system, but very little on the compiler and basis library.

[1] http://www.polyml.org

[2] http://isabelle.in.tum.de

2. Isabelle/ML as prover-specific programming environment, which now includes infrastructure for high-level concurrency (§3) and parallel value-oriented programming (§4).
3. Isabelle as platform for logic-based applications (§5). End-users usually refer to the Isabelle/Isar source language (for specifications and proofs), or specific tools written in Isabelle/ML.

This non-trivial and quite demanding application of parallel ML programming provides a unique opportunity to evaluate concepts and implementations in reality, beyond small benchmark examples.

2. Parallel Poly/ML

Standard ML does not define any concurrency operations either in the language or in the basis library. How concurrency should be combined with the language has been the subject of research and various proposals have been made, such as CML [22], Facile [8] and the work by Berry et al [3]. Issues such as the expressiveness and suitability of proving correctness have been the primary concerns. Applications of these constructs have focussed largely on concurrency for distributed systems or to provide effective user-interaction in windowing systems. There has been less work done on using concurrency to achieve improved performance. With the arrival of multicore processors attention has been drawn towards trying to exploit parallelism to achieve speed-ups of originally sequential programs and the aim of this work is to make this possible for ML programs, particularly Isabelle.

A concurrency mechanism using communication over blocking channels was added to Poly/ML early on in its development [16] to support concurrent windowing operations. The implementation used a single-threaded run-time system that could run one of a number of threads and time-slice between them. Although the original aim was not to speed up execution there were some experiments with parallel implementations on a shared memory multiprocessor, the DEC Firefly [26] and a distributed memory system [17] which showed that speed-ups could be achieved.

Moving to a fully multi-threaded model running on a shared memory provided an opportunity to reconsider the basic concurrency mechanism in Poly/ML. Rather than use one of the high-level models or invent a new one we chose to offer a fairly low level set of facilities at the ML level influenced by POSIX threads (pthreads). Similar primitives were used in Concurrent CAML Light [6]. Higher-level constructs can be implemented from these in ML itself. This concurrency model provides threads, mutexes and condition variables. These primitives are well understood and there is extensive literature on building more complex structures on top. They can be implemented efficiently; in particular locking and unlocking a mutex. The basic operations are as follows.

- Threads:

```
fork: (unit -> unit) * attribute list -> thread
interrupt: thread -> unit
setAttributes: attribute list -> unit
getAttributes: unit -> attribute list
```

- Mutexes:

```
mutex: unit -> mutex
lock: mutex -> unit
unlock: mutex -> unit
```

- Condition Variables:

```
condVar: unit -> condVar
wait: condVar * mutex -> unit
signal: condVar -> unit
broadcast: condVar -> unit
```

2.1 Interrupts

The original 1990 Definition of Standard ML [18] included an Interrupt exception that was raised “by external intervention”. In Poly/ML this could be raised by means of the SIGINT signal usually generated by the control-C key combination. This was removed in the later 1997 Definition [19] but it is generally useful to break into a computation that is in a loop or taking too long. For this reason it was retained in Poly/ML and Isabelle has continued to use it.

Unlike all other exceptions, which are raised synchronously as the result of a computation, the `Interrupt` exception is asynchronous and can arrive at any time. The thread mechanism of Poly/ML has extended this from a purely user-generated exception by adding functions that allow the `Interrupt` exception to be generated programmatically. The `interrupt` function, for example, attempts to raise `Interrupt` in a specific thread.

Asynchronous interrupts, though, are not generally compatible with multi- threaded programming. If a function has locked a mutex and then receives an `Interrupt` exception it will leave the mutex locked; indeed an interruption during the process of locking or unlocking could leave the mutex in an indeterminate state.

Each thread maintains its own interrupt mode setting, set initially by the parent thread as a thread attribute when the thread is created, but capable of being changed within the thread. This mode setting determines how an `Interrupt` exception is delivered to the thread. When running application code the mode will typically be set to deliver the exception asynchronously. This corresponds to the original behaviour. However, a thread can choose to block itself from receiving an `Interrupt` exception. An interrupt received while the thread is in blocking mode is deferred and may be delivered later if the thread changes the mode. Interrupts can also be set to be delivered synchronously. In this case the `Interrupt` exception will only be raised if the thread calls one of the run-time functions that could block for an appreciable time. This can be useful to allow a thread to wait for a condition variable to be signalled but also have it interruptible.

Using this interrupt state to control the delivery of the interrupt exception allows wrapper functions to be used around code that, for example, lock a mutex, so that the interrupt state is changed temporarily to block interrupts while the mutex is held. The interrupt state is then restored after the mutex is released. In this way existing code can continue to be interrupted asynchronously while code that uses the thread synchronization functions can be executed safely.

The `interrupt` function allows one thread to interrupt another programmatically. There is still the need to be able to generate an exception in application code as a whole as a result of a user request. The `broadcastInterrupt` function sends a broadcast to all threads that are prepared to accept this and delivers the exception according to the current interrupt mode of the thread. As with the interrupt mode, whether a thread accepts broadcasts is controlled by a thread attribute.

Currently, the only thread attributes defined are the interrupt mode and whether a thread wishes to accept broadcasts. The mechanism is extensible, though, and could include properties such as scheduling priority or processor affinity.

2.2 Implementation of multi-threading

The Poly/ML compiler generates native machine code within the heap. There is a separate run-time system (RTS) written in C++ that provides memory management and all interface with the underlying operating system. The existing uni-processor concurrency mechanism supported a degree of multi-threading at the ML level with separate threads each with their own stack. Since only a single thread could run at a time the run-time system itself was single-threaded.

To support multi-threading at the ML level it was necessary to adapt the run-time system to be multi-threaded. Apart from on Windows where native functions are used, the implementation assumes a pthreads library to manage the threads. The initial operating-system thread has a special status and is responsible for various storage management operations in particular garbage-collection. After initialising the memory it creates a thread to run ML code and suspends itself until needed. Threads are created using the pthread library function `pthread_create` or the equivalent `CreateThread` function on Windows. The precise relationship between threads created in this way and multiple processors is not clearly defined, but in practice modern operating systems multiplex threads between available processors. The run-time system relies on the operating system for this and makes no attempt to control the placement of threads on CPU cores.

Like every thread that runs ML code this new thread has both an operating-system stack that it uses when running in the RTS or in "foreign" code through the foreign-function interface (FFI) and an ML stack that is used when running ML code. The ML stack is an object within the ML heap and can be scanned and updated by the garbage-collector.

Although there is a clear relationship between the ML `fork` function and its implementation in the `pthread_create` function the same is not true of the mutex and condition variable functions in ML and their equivalents in the pthread library. These are implemented quite differently. Each thread has a single pthread condition variable and if it needs to suspend itself, either because it is waiting for a condition variable at the ML level or if it has encountered a locked ML mutex, it waits on that variable. Suspending the thread at a well-defined place allows another thread to be able to wake it either as a result of signalling the ML condition variable, unlocking the ML mutex or as a result of sending it an interrupt.

Because all threads are created by calls to the RTS and all interaction with the operating system is through it the RTS is able to maintain a table of threads and their current state. Knowing the state of all threads is particularly important for garbage collection.

Another reason for not implementing the synchronization primitives directly on their pthread equivalents is efficiency and that is because of the division in Poly/ML between ML code and the RTS. The machine code generated by the Poly/ML compiler uses linkage conventions geared towards the efficient execution of small functions and the requirements of the garbage-collector. These are different from the linkage conventions used with C++ functions and calls between the two have to go through an interface level. At the very least all the registers in use in the ML code have to be saved and restored to allow values in them to be modified by the garbage collector. Effectively, the RTS behaves rather like an operating system kernel and switching between ML code and C or C++ code in the RTS requires a context-switch with a measurable overhead. Avoiding unnecessary ML/C context-switches is important for efficiency.

In the common case of locking and unlocking a mutex when there is no contention the implementation is entirely within the ML code. The code uses machine instructions that atomically increment or decrement a value so that contention can be detected. To further avoid unnecessary context switches the `lock` function first checks the state of a mutex and, if it is locked, goes into a tight loop, a spin lock, until either the lock is released or a counter expires. Since it is intended that a thread should only hold a mutex for a short period this can frequently avoid the need to block a waiting thread. Only if this fails will the thread call into the RTS.

2.3 Memory management

ML programs dynamically allocate large amounts of memory which needs to be reclaimed by the garbage collector. The heap is considered as a resource shared between all the threads so potentially allocation requires an interlock between the threads. Interlocking every allocation would add a very significant overhead so instead each thread is given its own segment in which to allocate cells and only when this is exhausted is it necessary to go to the common pool.

The common memory pool is maintained by the RTS so a thread that has exhausted its segment has to make a call into the RTS to allocate more memory. When there is no space left in the heap the thread cannot proceed and must request a garbage collection but a collection cannot begin while any thread is running in the heap. The RTS maintains information about the state of all threads. Those that are blocked, for example for I/O or waiting for another thread or those running "foreign" code through the FFI are known to be out of the heap. The other threads have to be interrupted. The ML code generated by the Poly/ML compiler contains periodic tests for an interrupt request, actually integrated with stack overflow checking. When this is set the ML code makes a call into the RTS. This enables the RTS to interrupt threads that are running ML code and force them to enter the RTS. Only when all threads are out of the heap does the garbage collection begin.

The garbage collector is single-threaded. When a garbage collection is required all ML threads are stopped and the collector runs to completion before releasing the ML threads. This is a potential bottle-neck especially as the number of processors increases (cf. §5.3.1). This could possibly be improved by overlapping garbage-collection with execution of ML using algorithms such as those described by Appel et al [1] and Dijkstra et al [5]. These algorithms have disadvantages; requiring either trap handling on access to data or a modification to the assignment operator. A more promising solution is to run the garbage collector as a distinct phase as at present but to parallelize the collector itself [7, 15].

The choice of segment size can have a bearing on performance: too small and a thread that is allocating frequently has to make repeated calls to allocate new segments; too large and there is wasted space when another thread initiates a garbage collection. Poly/ML uses an adaptive strategy in which each thread has its own preferred segment size. Each time a thread exhausts its segment the segment size is increased and on each garbage collection the sizes of segments for all the threads are reduced. In this way the segment size adapts to the allocation history of the thread. A program with only a single thread will have a very large segment size so that it is exhausted very few times between collections.

Note that the segments are used only to assist with allocation. Once a cell is allocated it is part of the common heap. There is no sense in which a cell is owned by any particular thread. This is in contrast to schemes such as that of Doligez and Leroy [6] where each thread owns — and can garbage-collect — its own heap but at the expense of requiring all assignments to copy the whole of the data structure being assigned.

2.4 ML compiler and library

By far the largest component of Poly/ML that needed to be adapted to enable multi-threading was the run-time system. The remainder of Poly/ML, the compiler and basis library were largely unaffected. It was necessary to add locks in a few places in the library, notably the I/O modules, to make them safe in the presence of accesses by multiple threads. This code is included even if the application does not actually use threads and imposes an overhead that in most cases is negligible. The one exception is perhaps the `TextIO.input1` function that reads a single character from an input stream. Because it is possible for multiple threads to read from the same stream this function requires a lock to be acquired and released and the interrupt mode for the thread changed although the actual work done in reading a single character is small, being little more than loading the character from a buffer and incrementing a counter. All this happens within the compiled ML code and does not require a call into the RTS except to reload the buffer. Measurements when reading a large stream by this method showed that this extra work increased the time to read a character by a factor of roughly five. It is important to note that this is an extreme example and demonstrates that the cost of locking and unlocking a mutex without contention is of the same order as access to (updatable) ML reference variables.

Interestingly, this overhead can be almost entirely avoided by using the functional IO layer of the ML library in which reading a character returns the character and a new stream since in this case a lock is only needed when the buffer is empty and needs to be refilled. In fact, this is a general observation in our ML parallelizing efforts: stateful code with extra synchronization is less efficient than purely functional code that can run unhindered.

The Poly/ML compiler needed almost no modification. Previous work on the compiler had separated out access to the name space of top-level declarations. The compiler has, in simplified terms, the following type:

```
stream * namespace -> (unit -> unit)
```

The stream provides the characters forming the ML program; the namespace is a collection of functions to access names in the various spaces of values, types, structures, etc. used by ML and the result is a function that, when called, performs the effect of the ML program. This almost always has side-effects including entering new declarations into the namespace.

A namespace is an abstraction of the top-level basis in which an ML program is compiled. Traditionally in ML there has just been a single global namespace with each new declaration being added to it. Poly/ML provides an implementation of this but leaves open the possibility of user code adding new namespaces.

The effect of organizing the compiler in this way is that compilations in different namespaces can happen completely independently. It is also possible to have multiple compilations on the same namespace, for example to implement a parallel "make" but if multiple threads can access the same name space in parallel the implementation of the name space must of course be thread-safe. For the default global namespace of the Poly/ML top-level locks were added to ensure this. Isabelle (§5) uses its own tables within purely functional theory contexts that are merged according to the import hierarchy specified by the user.

Parallel compilation is quite important in Isabelle. Formal theory texts may contain embedded ML that needs to be compiled and since the aim of this work was to be able to process theories in parallel it was essential to be able to run multiple instances of the compiler at the same time.

3. Synchronized variables in Isabelle/ML

The fully general pthreads model of synchronized access to (implicitly) shared resources is quite involved. The programmer needs to observe a peculiar protocol of *lock–wait–change–signal–unlock* using a mutex together with a condition variable. Although this is well-established technology, with extensive explanations in text books and online tutorials, it is nonetheless easy to overlook some details in everyday programming. In the spirit of Brinch Hansen's statement that "concurrent programs can be written exclusively in high-level languages" [4], we shall now wrap up synchronization primitives behind higher-order combinators in Isabelle/ML, and thus ensure correctness by construction.

3.1 Programming interface

Synchronized memory access with atomic functional updates is presented as an abstract type in the Isabelle/ML library as follows:

```
type 'a var
val var: 'a -> 'a var
val value: 'a var -> 'a
val guarded_access: 'a var ->
  ('a -> ('b * 'a) option) -> 'b
```

Type `'a var` essentially wraps up a raw ML reference cell, with a mutex and condition variable for the pthread synchronization protocol. Only a single state component of type `'a` is modelled, but complex data structures can be represented via records or datatypes.

Operation `value` provides read access to the internal reference variable. This does not require any pthread synchronization, because of atomic assignments: the machine code produced by Poly/ML already ensures that ML references are accessed by the usual instructions for volatile memory. This guarantees memory consistency in terms of the Poly/ML runtime system: user code always sees *some* ML value, although the actual content depends on the application logic.

The `guarded_access` combinator is our higher-order representation of the idea of a *conditional critical section*, or *monitor* of Hoare and Brinch Hansen [4]. It allows synchronized read and write access, with explicit checking of semantic conditions and implicit signalling of state changes. Note that the guarding predicate and the state update function are rolled into a single partial function, which is modelled via the `option` type of SML.

This means `guarded_access v f` lets the function `f` operate within a critical section on the state `x = value v` as follows: if `f x` produces `NONE` we continue to wait on the internal condition variable, expecting that some other thread will eventually use `guarded_access` to change the content in a suitable manner; if `f x` produces `SOME (y, x')` we are satisfied and assign the new state value `x'`, broadcast a signal to all waiting threads on the associated condition variable, and return the result `y`.

Of course, evaluation of `f x` may raise an exception. Such failure is propagated as usual, without affecting integrity of the internal reference cell. Although external interrupts emerge in Poly/ML as just another exception, special care is required in the `wait` within the critical section. The Isabelle/ML version of `sync_wait` locally modifies the interrupt mode of the current thread, such that the calling ML code will get explicit indication about where an interrupt has happened: either in user code or within the critical *signalled-locked* phase of the pthreads protocol. This extra complexity is hidden in the library implementation: user code may run with asynchronous interrupt handling enabled.

3.2 Examples

The most basic example for `guarded_access` implements the following derived operation of the same Isabelle/ML module:

```
val change: 'a var -> ('a -> 'a) -> unit

fun change v f =
  guarded_access v (fn x => SOME ((), f x))
```

This updates the state unconditionally, always yielding `SOME`. Nonetheless, `guarded_access` will signal all other threads that are waiting on a change to establish some semantic condition.

The next example implements type `MVar` of Concurrent Haskell[3], which represents a buffer that is restricted to 0 or 1 elements. This means operation `take` blocks on empty content and `put` blocks on non-empty content of the buffer. Again, our `guarded_access` combinator allows us to express this succinctly:

[3] `http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-MVar.html`

```
type 'a mvar = 'a option var
fun mvar () = var NONE
fun take v = guarded_access v
  (fn NONE => NONE | SOME x => SOME (x, NONE))
fun put v x = guarded_access
  (fn SOME _ => NONE | NONE => (SOME ((), SOME x))
```

The next example demonstrates a mailbox with unbounded message queue: operation `send` always adds a message without blocking, but `receive` blocks if the mailbox is empty. The implementation uses `type 'a queue` from the Isabelle/ML library, with purely functional operations `empty`, `enqueue` and `dequeue`.

```
type 'a mailbox = 'a queue var
fun mailbox () = var empty
fun send mbox msg = change mbox (enqueue msg)
fun receive mbox = guarded_access mbox (try dequeue)
```

Thanks to careful library design, this achieves some degree of "domain specific language" for concurrent access to shared state.

3.3 Performance

Our stylized view on pthread synchronization primitives is based on simplifying assumptions that work well in many typical situations. We briefly review its suitability for high-performance applications.

First of all, there is usually no problem in assuming that the shared state is represented as a single value of type `'a`, which might be a big record or table structure. Poly/ML performs well in handling large immutable data-structures, and Isabelle uses pure data for complex logical operations routinely. There is no indication that concurrent applications should fall back on mutable substructures such as arrays. In fact, pervasive use of pure data greatly helps to get the parallel application logic right in the first place.

Nonetheless, we do introduce a potential bottle-neck due to the broadcast semantics of `guarded_access`: there is *one* kind of signal sent to *all* waiting threads. Discrimination of state conditions only works by evaluating the given body functions. For example, the scheduler component for futures (cf. §4) was originally built around such a broadcast model. This worked well for 4 cores, as implemented in Isabelle2009 [27], but beyond 8 cores there was a significant waste of CPU cycles by idle worker threads that would continuously check the shared task queue structure. This critical component now uses more fine-grained signalling internally.

4. Value-oriented parallelism in Isabelle/ML

High-level synchronization (§3) helps to implement concurrent ML programs, and is used successfully in various Isabelle components such as the asynchronous "ATP manager", but the inherent complexities of concurrency are still present. Since our main concern is efficient parallel evaluation, we require a more adequate programming abstraction that hides threads and synchronization altogether.

Various sophisticated models for thread-less concurrency have been proposed in recent years. For example, there is support for *software transactional memory* (STM) in Glasgow Haskell [14], which emphasizes modular composition of concurrent program parts. Another model is that of *actors* in Scala [21, 11], which in turn is inspired by light-weight processes in Erlang [2], and unifies a whole spectrum of interacting functional components based on explicit message passing. Haskell, Scala, and Erlang all support true parallelism of the underlying platform, but Isabelle happens to be implemented in Standard ML.

To reconstruct an efficient programming model that fits our needs, we aim at minimizing complexity and maximizing performance in typical symbolic operations of the proof engine. It turns out that the simple concept of *future values* fits particularly well.

Futures can be considered folklore in concurrent functional programming. An early implementation is available in Multilisp [12], but that work points back to even earlier proposals of *eventual values* in Algol. The Alice ML dialect [25] includes futures as one of many built-in concurrency concepts, although its implementation merely works by time-slicing on a single system thread.

Our version of futures in Isabelle/ML closely follows regular evaluation semantics of strict Standard ML, including synchronous exceptions (produced by application code) and asynchronous interrupts (produced by the environment or other ML threads). Beyond that there is no direct communication between evaluations, the focus is on parallelism not concurrency. Nonetheless, futures can influence each other via ML exceptions. As we will see later on (§4.2), this indirect functional interaction can get quite far without re-introducing the complexities of raw concurrency in user code.

4.1 Programming interface for "futures"

The main operations of futures in Isabelle/ML are as follows:

```
type 'a future
val future: (unit -> 'a) -> 'a future
val join: 'a future -> 'a
val cancel: 'a future -> unit
```

The abstract type `'a future` represents a handle to the eventual result of evaluating an expression of type `'a`. Recall that a dummy function `unit -> 'a` is the usual way to represent an unevaluated expression in strict SML: computation commences when it is applied to the unit element.

By invoking `future (fn () => e)` such an expression is turned into an abstract representation of the result of the evaluation process, which is called a *task* internally. The future scheduler ensures that evaluation of `e` is run spontaneously in the background: there is a farm of *worker threads* that continuously pick the next ready task from the queue, and eventually write back the result to a synchronized variable (§3) within the corresponding future handle.

The `join` operation resynchronizes with such parallel evaluation, which could mean waiting until it is finished by another thread or starting its evaluation on the spot. In the latter case, the worker suspends the current task on the ML stack and continues to work on that requirement. Finished results are memoized, so there is no extra cost for joining again later on. In any case, `join (future (fn () => e))` is semantically equivalent to `e`, provided that this is a pure expression without side-effects. Exceptions raised during future evaluation are also propagated to the join point. For example, consider:

```
val x = future (fn () => raise Fail "error")
```

This succeeds, but later attempts to `join x` exhibit the exception.

Note that `future`/`join` resemble `lazy`/`force` in the common way of implementing lazy evaluation on top of strict SML. The difference is that futures may compute independently in the background — this happens even if the final value is never used.

The explicit `cancel` operation tells the system to give up on evaluating a future. If it is not finished yet, it will be forced into `Interrupt` exception state. If it is finished already, then `cancel` has no effect (in conformance with the above evaluation semantics).

If we think of type `'a future` as representing highly stylized "functional threads" with return value of type `'a`, then `cancel` corresponds to the `interrupt` operation (§2.1). Needless to say, efficient implementation of futures must be more sophisticated than mapping evaluation tasks directly on threads. The operating system is capable of handling tens or hundreds of threads, but we would like to work with many hundreds or thousands of futures. For best performance, the number of worker threads should usually be correlated to the number of cores available to the system.

Nonetheless, even thread-less future tasks require some internal organization. It turns out that in realistic applications like the Isabelle prover (§5), the number of tasks can be significantly reduced by fast-path implementations on the following operations:

```
val future_value: 'a -> 'a future
val future_map: ('a -> 'b) -> 'a future -> 'b future
```

Wrapping a constant value as `future_value a` works without the overhead for task management of naive `future (fn () => a)`. Similarly, `future_map f x` allows cheap cascading of evaluations, by reusing the former future task if possible (if evaluation has not been started yet). Otherwise the system will fall back on the naive version `val y = future (fn () => f (join x))`.

Invoking `join x` within the body of `y` as above introduces a functional dependency between the corresponding evaluation processes. If `x` is still untouched, the current thread can continue to work on `x` immediately, just like regular sequential computation would have required anyway. If `x` happens to be in progress by another worker thread, we need to wait for the result. Assuming that dependencies are relatively sparse, this situation happens relatively rarely. Nonetheless, the future scheduler always keeps some inactive workers in reserve to jump in for temporarily stalled threads, to keep the hardware cores saturated.[4]

If `x` is already finished, `join x` can pick up the memoized result directly. This uses `value` (§3) internally, with the minimal overhead of accessing an ML reference cell. Note that due to single-assignment of future results, synchronization is only required on unfinished futures. A finished future stays finished indefinitely: this monotonicity principle keeps things simple and efficient, and justifies the term "value-oriented parallelism".

Some further user-space adjustments are possible. Futures can be created with explicit static dependencies, which allows to enforce certain order of scheduling. By default future tasks are enumerated according to their creation time, and the next task without pending dependencies will be run. This scheme can be also modified by specifying explicit priorities. For example, futures that produce proofs in Isabelle are assigned a low priority.

4.2 Future groups and exception propagation

Our version of futures is able to capture the notion of strict SML evaluation with potential failure via exceptions (either produced by the program, or via the environment as interrupts).

For example, consider expression `e = (x, y, z)` and assume that the subexpression `x` produces a regular value, but `y` and `z` raise exceptions when evaluated. Using sequential SML semantics, the failure of `y` is propagated, and `z` is not attempted at all. There is a deterministic result, which is the *first* exception that is encountered in left-to-right order.

Generalizing this behaviour to parallel evaluation of futures `x`, `y`, `z` means that all subexpressions can be evaluated at some point (in parallel or with arbitrary ordering), and both `y` and `z` get a chance to fail. We want to specify the (nondeterministic) result of `e` as *some* exception that is raised by one of its subexpressions. The implementation can simply propagate the first failure that is encountered at physical runtime, and cancel any pending futures that are considered as related subexpressions.

To provide library functions for such exception propagation in parallel programs, we first require a datatype of SML results:

```
datatype 'a result = Result of 'a | Exn of exn
```

```
fun capture f x = Result (f x) handle e => Exn e

fun release (Result y) = y
  | release (Exn e) = raise e
```

This means the `capture` combinator gets hold of arbitrary ML evaluations, including failures. The `release` operation converts back into the usual runtime semantics of "free" exceptions.

There are some additional tools on captured results, notably `release_first: 'a result list -> 'a list` to release the first program exception encountered here, ignoring `Interrupt` exceptions, unless there is no other choice. An actual list of results is returned iff all elements are of the `Result` variant.

This is typically used with the simultaneous join operation `join_results: 'a future list -> 'a result list` which is part of our library for future values.

The layout of sub-expressions for implicit exception propagation is declared via *future group* identifiers, which can also be nested to express block-structure. Our library provides the following operations for future groups:

```
type group
val new_group: group option -> group
val future_group: group -> (unit -> 'a) -> 'a future
```

Groups can be created via `new_group` with an optional parent for nesting. The combinator `future_group g` is analogous to basic `future` (§4.1), but inserts the corresponding task into the hierarchy of groups at position `g` (we can think of nested group identifiers as paths within a tree structure). In fact, plain `future` involves a default grouping scheme that corresponds to the implicit nesting as futures are created within other futures at runtime.

Whenever some future evaluation fails, its group and all nested subgroups are *invalidated*. This means, any unfinished group member will be permanently cancelled, even those created later on. The latter aspect models the idea, that the overall structure of expressions and subexpressions is somehow static, although it is populated incrementally at runtime.

The following artificial example illustrates parallel exception propagation. We group a diverging and a failing evaluation:

```
fun loop () : int = loop ()

val g = new_group NONE
val x = future_group g loop
val y = future_group g (fn () => 1 div 0)
```

Now `join x + join y` will raise exception `Div`. That result is even deterministic, since the other subexpression does not terminate. Due to the invalidation of groups for all time, we can also swap the order in which `x` and `y` are created, with the same result.

Future combinators are still relatively raw. In practice there are extra library layers for specific applications, say for goal-directed proofs [27]. General purpose list combinators can be implemented as well, notably parallel `map`, `exists`, or `find`.[5]. Note that the latter produces non-deterministic results. In order to achieve a disjunctive parallel evaluation, where the first successful branch cancels any other attempts, we simply use a local exception `Found` that can carry a value. This programming style is not more difficult than regular treatment of exceptions in sequential SML.

[4] Isabelle/ML currently uses a simple adaptive scheme, where the number of workers is dynamically adjusted in the range of $m \ldots 4\,m$, keeping at most m threads active and the others in reserve. Excessive workers will spontaneously yield whenever an overloaded situation is detected.

[5] For example, see `http://isabelle.in.tum.de/repos/isabelle/file/Isabelle2009/src/Pure/Concurrent/par_list.ML`

5. Application: parallel Isabelle

Our main application for parallel ML programming is the Isabelle theorem prover, which is based on the well-known “LCF-approach” [9]. This means proofs are fully foundational in the sense that every single inference is explicitly checked by an inference kernel that implements the basic logic. For Isabelle this is a version of higher-order natural deduction called Pure. The LCF architecture ensures correctness by construction, while allowing users to implement arbitrarily complex proof tools in ML. It also means that significant runtime resources are required for “fully-expansive” proof construction, so multicore programming really matters.

There is a second language layer called Isabelle/Isar (Isar stands for *Intelligible semi-automated reasoning*), which enables end-users to write structured theory and proof documents in a declarative manner. Isar is not computational, but a language for *formal proof expressions*. Isar proofs have rich modular structure that can be exploited for implicit parallelism. Thus most Isabelle users will immediately gain significant speed-up factors, without having to change their proof texts in Isar or proof tools in ML.

5.1 Proof document structure

Isabelle proof documents follow a certain structure that allows various parallel scheduling strategies. Some possibilities are discussed in [27] in further detail; the main observations are as follows.

1. Large Isabelle applications consist of a DAG-structured collection of theories. Independent nodes in that graph can be loaded in parallel. This is analogous to a parallel `make` tool, although everything happens within the same ML process.[6]

2. Theorem statements are explicit and proofs are *irrelevant*, in the sense that a theorem can be accepted as correct and used elsewhere without having checked its proof yet. It is, of course, necessary to finish proofs at some point but this can be done independently via future values.

   For example, consider a long theory text as follows:

   theorem $a_1 : A_1 \ \langle \mathit{proof}_1 \rangle$

   $\vdots$

   theorem $a_n : A_n \ \langle \mathit{proof}_n \rangle$

   Here the top-level statements will be processed in sequential order, but the proofs are treated as independent futures by the Isar interpreter. The Pure inference kernel will re-assemble the results such that the final theorems (with optional proof objects inside) are logically correct, independently of the operational details of proof construction.

3. Isar proofs have a rich sub-structure, where most runtime is spent in terminal justifications (small local proofs, involving potentially complex automated reasoning tools). Here is a stylized Isar proof text for illustration:

   lemma $A \wedge B$
   proof
     show A **by** *auto*
     show B **by** *blast*
   qed

   These **by** steps can be parallelized implicitly, without having to reimplement proof tools like *auto* or *blast* involved here.[7]

It turns out that these parallelization strategies are sufficient to saturate 8 cores reasonably well.

5.2 Efficiency

How much do Isabelle applications actually gain from using multiple cores? Figure 1 shows speed-up factors for various Isabelle sessions, which consist of many definitions, statements, and proofs. Decision-Procs, Hoare-Parallel, MicroJava, and Auth represent the bigger examples in the standard distribution of Isabelle. They introduce hundreds of definitions and thousands of theorems — including complex definitions of inductive predicates and recursive functions that also require proofs. Sequential runtime is the range of minutes:

Decision-Procs	0:14:17
Hoare-Parallel	0:07:09
MicroJava	0:07:03
Auth	0:08:18

There are also Isabelle applications that take several hours, but these are not included in our systematic measurements so far.

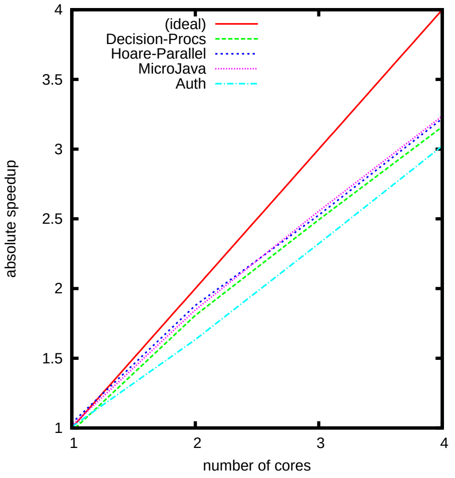

Figure 1. Absolute speed-up $\varepsilon_0/\varepsilon(m)$ for $m = 1 \ldots 4$

Despite rather diverse theory and proof structure (using quite different specification tools and proof styles), all three examples show almost uniform speed-up factors of $1.6 \ldots 1.9$ for 2 cores, and $3.0 \ldots 3.2$ for 4 cores. This looks quite satisfactory, but meaningful performance analysis needs more precise explanations of the physical parameters and methods of measurement.

The test platform is a Linux system with a total of 32 cores (8 times an AMD quad-core Opteron with 2.7 Ghz clock frequency and 0.5 MB cache). There is 64 GB main memory, but we only use 16–32 GB. All measurements use an internal Isabelle snapshot at the time of writing.[8] The official Isabelle2009 version from April 2009 achieves almost the same performance on a Mac Pro with 4 cores [27], but substantial extra tuning was required for further tests shown later.

Each test run specifies m as nominal number of cores: the future scheduler (§4) ensures that at most this number of worker threads are active, but there can be additional threads that are sleeping most of the time. The following runtime parameters are measured, as reported by running `isabelle usedir -t true` in batch mode:

$\varepsilon(m)$	elapsed time (wall-clock)
$\zeta(m)$	CPU time (user and system)
$\gamma(m)$	garbage collection time (included in $\zeta(m)$)

[6] Isabelle is based on the traditional model of *dumped world image*, not separate compilation.

[7] Such plug-in tools are always expected to be thread-safe; usually implementations are purely functional anyway.

[8] See the repository version `http://isabelle.in.tum.de/repos/isabelle/rev/13d00799fe49` from 05-Nov-2009.

We also measure the strictly sequential version, without threads and futures getting in between; resulting parameters are ε_0, ζ_0, γ_0. These baseline figures are used for the absolute speed-up $\varepsilon_0/\varepsilon(m)$ depicted in figure 1. This is what the user perceives in reality. Gaining $\approx$ 3.0 for 4 cores is quite good, but there are also less efficient Isabelle examples, where the speed-up can be lower than 2.0 (usually short running sessions).

This degree of efficiency in a realistic application such as Isabelle is the result of several rounds of fine tuning. Even so, the "law of diminishing returns" begins to apply, the speed-up curve flattening as more cores are used.

5.3 Toward scalability

In order to evaluate scalability from multicore to many-core hardware, we show the measurement for $m = 1 \dots 16$ on the very same test platform, see figure 2.

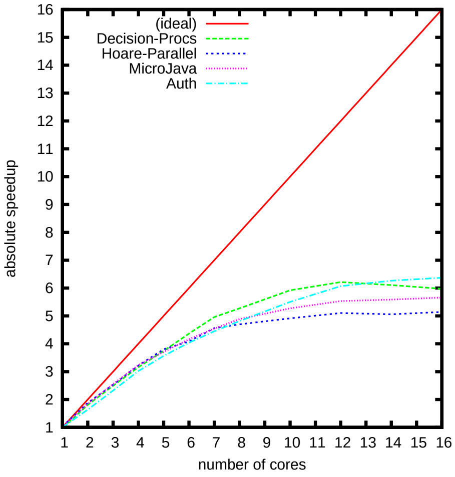

Figure 2. Absolute speedup $\varepsilon_0/\varepsilon(m)$ for $m = 1 \dots 16$

The curve flattens roughly according to *Amdahl's Law* for sub-linear speed-up: $1/(s + p/m)$ where s is the part of the program that is inherently sequential, and p the part that can be parallelized. For $m \longrightarrow \infty$ this would converge to $1/s$, but in reality there is extra overhead for parallelization that even makes the speedup decrease again at some point.

To quantify the relative amount of CPU cycles that are actually spent in the application we also inspect the ratio $\zeta(m)/\zeta_0$, see figure 3. For $m = 8$, this overhead is still only 5–15%, but becomes more noticeable for $m = 16$.

Overall, we observe that our example sessions run reasonably well around 8 cores, and more. The maximum speedup is $\approx 5.0 \dots 6.5$ in this scenario. Right now it does not make much sense to use all 32 cores of this machine. Although we are not yet capable of addressing such a large number of cores, we can analyse the results to see where things might be improved.

5.3.1 Bottle-neck 1: garbage collection

Although the use of local heap segments will establish a degree of locality of data when it is initially allocated (§2.3), actual garbage collection will stop the world and run exclusively on a single thread. Although for the above Isabelle applications garbage collection time is essentially a constant of only 2–5% of total CPU time (thanks to generous use of initial heap space), its relative portion grows with the speed-up factor, reaching about 15–30% for 16 cores as shown in figure 4.

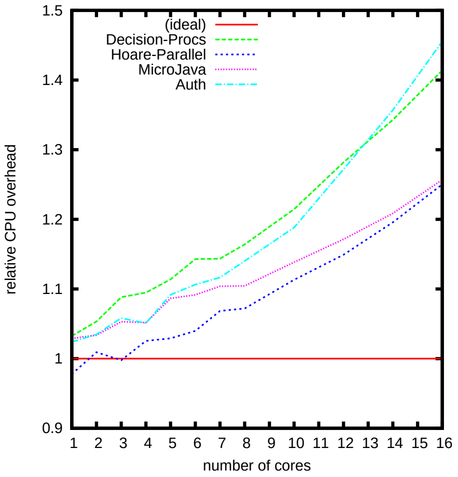

Figure 3. Relative CPU overhead $\zeta(m)/\zeta_0$ for $m = 1 \dots 16$

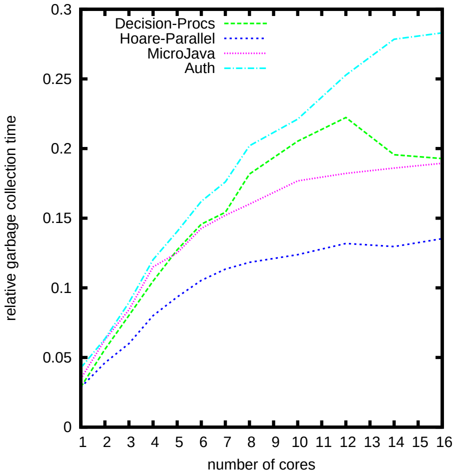

Figure 4. Relative GC time $\gamma(m)/\varepsilon(m)$ for $m = 1 \dots 16$

As discussed in §2.3 there are approaches to parallel GC that have already been implemented in other systems, notably Glasgow Haskell [15]. Although fully parallel heap management is technically difficult to implement, once implemented it is essentially part of the "system software". Thus GC-bound applications would automatically become faster without reorganization of user code.

5.3.2 Bottle-neck 2: insufficient parallelization of the application

Disregarding potential performance losses in the infrastructure, the main limitations are usually due to the application itself. In Isabelle we are relatively fortunate, because of the way LCF-style proof checking works, but for $m \gg 8$ we encounter some weak spots in the parallel organization of individual proof checking tasks.

To analyse the situation, we look more closely at the profile of the MicroJava example, inspecting the state of the task queue of the future scheduler every 500 ms. See figure 5 for $m = 4$, and figure 6 for $m = 16$.

Active workers are those threads that are busy evaluating future tasks. *Running tasks* are in progress by some worker, but have

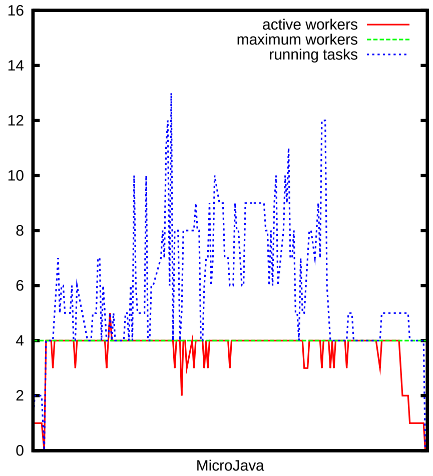

Figure 5. Worker thread utilization for $m = 4$

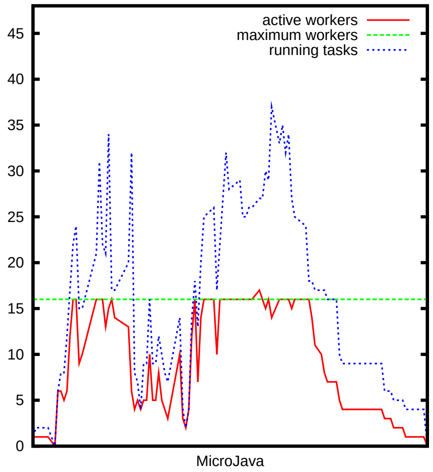

Figure 6. Worker thread utilization for $m = 16$

been temporarily suspended (as regular stack-frame), because some other future needs to be joined into the current evaluation context. Sometimes there are excessively many running tasks, while the active workers drop below the specified boundary m. This indicates that the dependency graph between futures is relatively dense, and too many workers are stalled while waiting for tasks to be finished by other threads. This could be addressed by a more sophisticated adaptive strategy for forking additional replacement workers on demand, similar to the scheduler for Scala actors [10]. On the other hand, further inspection of task queue profiles (data not shown here) indicates that this situation happens only sporadically.

In practice, the main bottle-neck is much more basic: Isabelle cannot always provide a sufficient number of tasks. For $m = 4$ there are typically hundreds or thousands ready tasks available most of the time. This saturation drops sharply for $m = 16$ or more, leading to frequent task queue underflows. Even without further trace details, this phenomenon can be observed in figure 5 and figure 6 by the “ramps” at start and end of the lifetime. For $m = 16$ MicroJava is already running out of independent tasks during the last 30% of its time, and there are further queue underflows in the middle of theory processing.

It remains to be seen how far our approach of implicit parallelization of Isabelle proof checking can be continued, until individual proof tools will have to be reworked. Of course, the Isabelle/ML library infrastructure for parallel programming can be reused in user-code as well.

6. Conclusion

We have presented a parallel programming environment for Standard ML and evaluated its performance for non-trivial applications of theorem proving. Although many details of the underlying Poly/ML and Isabelle/ML platforms had to be reconsidered, the whole project was finished in approximately 1 person year. The two main challenges have been proper interaction of threads with interrupts, and achieving reasonable performance in the end.

Since SML can be used both for functional and imperative programming, we have been lucky that most of the application code in Isabelle was already purely functional. Only very few impure features that had crept in over the years had to be replaced. Thus user code can immediately benefit from the implicit scheduling of value-oriented parallelism provided in our framework.

The inherent advantage of functional programming languages for parallelism had been known for decades, and can now be turned into practical performance figures on current hardware. Mainstream applications in C, C++, Java etc. will be much harder to upgrade. On the other hand, very few parallel functional language implementations have ever reached a reasonably stable state to be used in realistic applications. Apart from Poly/ML, the ML family is particularly weak in this respect. Although there is support for threads and other basic concurrency primitives in OCaml, Mlton, SML/NJ, and Alice, none of the underlying runtime systems work with truly parallel system threads. Nonetheless, there have been many experimental parallel ML implementations that did not become widely available.

Tolmach and Morrisett [20] implemented SML/NJ on several multiprocessors and achieved reasonable speed-ups on some example applications. Their garbage-collector was single-threaded and they noted this as a potential draw-back. Threads were built on top of first-class continuations of the SML/NJ runtime, which is theoretically elegant. However, as they note in their conclusions, this results in poor locality of reference and significantly increases memory-bandwidth requirements.

Reppy et al [23, 24] report on more recent work on parallel Concurrent ML (CML), which is an extension of SML/NJ. This is based on “Manticore”, a novel language for parallel programming. Their focus is on supporting the communication primitives of CML and they demonstrate that they can achieve speed-ups in test examples. The baseline performance compared to the sequential version appears to be relatively low, but scalability to many cores looks promising.

The Haskell community has been very active in research and implementation of a wealth of concepts for concurrency and parallelism. Harris et al [13] report on a shared-memory multi-processor version of Haskell. They show the possibility of speed-ups on two processors on a non-trivial application: the Glasgow Haskell compiler itself. Marlow et al [15] report on improved garbage-collection for Haskell, using a stop-the-world collector that is parallelized internally. This gives noticeable speed-ups on 2–4 cores.

Scala [21] is a very interesting language that unifies higher-order functional and object-oriented programming. Since Scala can coexist natively with Java on the JVM, it can leverage existing parallel and distributed infrastructure. Various server-side JVM implementations support parallel garbage collection routinely. The Scala actors library [11] is inspired by lightweight processes of Erlang [2]. Such thread-less models of independent computational entities can also be used for parallel computations like the future values covered in the present paper. Scala actors have been used successfully in high-performance web applications, but we are not aware of any applications involving more traditional parallel computations.

Future work. As already pointed out in our discussion of parallel Isabelle (§5), there are still various weak spots that need to be addressed if these achievements are to be carried into the next generation of multicore hardware.

At the lowest level, the current sequential garbage collector is an obvious bottle-neck. Parallelizing this is a priority. Whether a fully concurrent garbage collector is required remains to be seen.

Another priority must be to improve the instrumentation of the implementation at all levels so that it is easier to see where the bottle-necks are occurring.

In the longer term a more distributed approach to parallelism will be needed. At present we rely on the hardware providing communication through shared memory and the basic thread operations are geared towards that. As multi-processors grow with more and more cores that will no longer be feasible. Whether the future will be in terms of a cluster with message-passing as the only communications mechanism or as a non-uniform memory architecture (NUMA) remains to be seen. In either case issues of thread placement become more significant. It will be necessary to revisit the lowest level primitives and possibly provide different primitives. Our layered approach to parallelizing Isabelle will hopefully mean that any changes are limited to re-implementation of some of the combinators and background infrastructure.

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[3] D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *17th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1992.

[4] P. Brinch Hansen. Monitors and concurrent Pascal: a personal history. In *ACM SIGPLAN conference on History of programming languages (HOPL-II)*. ACM, 1993.

[5] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[6] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *20th ACM Symposium on Principles of Programming Languages (POPL)*. ACM press, 1993.

[7] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[8] A. Giacolone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.

[9] M. Gordon, R. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Principles of programming languages (POPL)*, 1978.

[10] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Joint Modular Languages Conference*, Springer LNCS, 2006.

[11] P. Haller and M. Odersky. Scala actors: unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[12] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7 (4), 1985.

[13] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *ACM SIGPLAN workshop on Haskell*. ACM Press, September 2005.

[14] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8), 2008.

[15] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *International Symposium on Memory Management*, 2008.

[16] D. C. J. Matthews. Concurrency in Poly/ML. In *ML with Concurrency*, Monographs in Computer Science. Springer Verlag, 1996.

[17] D. C. J. Matthews and T. Le Sergent. Lemma: A distributed shared memory with global and local garbage collection. In H. G. Baker, editor, *Memory Management, International Workshop (IWMM), Kinross, UK*, September 1995.

[18] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[19] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[20] J. G. Morrisett and A. Tolmach. Procs and locks: a portable multiprocessing platform for Standard ML of New Jersey. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 1993.

[21] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne, 2004.

[22] J. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 26. ACM, 1991.

[23] J. Reppy and Y. Xiao. Toward a parallel implementation of Concurrent ML. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, January 2008.

[24] J. Reppy, C. Russo, and Y. Xiao. Parallel Concurrent ML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, September 2009.

[25] A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. Alice through the looking glass. In *Trends in Functional Programming*, volume 5. Intellect Books, Bristol, UK, 2006.

[26] C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. *SIGARCH Comput. Archit. News*, 15(5):164–172, 1987.

[27] M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*. ACM Digital Library, 2009.