A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers

John Plevyak Vijay Karamcheti Xingbin Zhang Andrew A. Chien

Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801
{*jplevyak,vijayk,zhang,achien*}@*cs.uiuc.edu*

Abstract

While fine-grained concurrent languages can naturally capture concurrency in many irregular and dynamic problems, their flexibility has generally resulted in poor execution efficiency. In such languages the computation consists of many small threads which are created dynamically and synchronized implicitly. In order to minimize the overhead of these operations, we propose a hybrid execution model which dynamically adapts to runtime data layout, providing both sequential efficiency and low overhead parallel execution. This model uses separately optimized sequential and parallel versions of code. Sequential efficiency is obtained by dynamically coalescing threads via stack-based execution and parallel efficiency through latency hiding and cheap synchronization using heap-allocated activation frames. Novel aspects of the stack mechanism include handling return values for futures and executing forwarded messages (the responsibility to reply is passed along, like call/cc in Scheme) on the stack. In addition, the hybrid execution model is expressed entirely in C, and therefore is easily portable to many systems. Experiments with function-call intensive programs show that this model achieves sequential efficiency comparable to C programs. Experiments with regular and irregular application kernels on the CM-5 and T3D demonstrate that it can yield 1.5 to 3 times better performance than code optimized for parallel execution alone.

Acknowledgements

The authors recognize the contributions of the other members of the Illinois Concert project: Hao-Hua Chu, Julian Dolby and Mark Gardner. We also thank the anonymous reviewers for their useful comments.

The research described in this paper was supported in part by NSF grants CCR-92-09336, MIP-92-23732 and CDA-94-01124, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and a special-purpose grant from the AT&T Foundation. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Xingbin Zhang is supported by Computational Science and Engineering program of the University of Illinois at Urbana-Champaign. The experiments reported in this paper were conducted on the CM-5 at the National Center for Supercomputing Applications and the T3D at the Pittsburgh Supercomputing Center.

1 Introduction

Irregular and dynamic problems are challenging to express and program efficiently on distributed memory machines. They often do not fit into data parallel programming models, and message passing requires the programmer to deal explicitly with the complexities of data placement, addressability and concurrency control. Fine-grained concurrent languages which provide a shared name space, implicit synchronization and implicit concurrency control can dramatically simplify the programming of irregular applications. Such languages typically express computations as a collection of light-weight threads executing local to the data they compute (the owner computes rule [19]). However, the cost of creating and synchronizing these threads can be high. Given a data layout[1] we propose an execution model which adapts to exploit runtime locality by merging threads, eliminating unnecessary concurrency overhead.

Efficient execution requires optimizing for both sequential efficiency when all the required data is local, and for latency hiding and parallelism generation when it is not. To address both cases, our hybrid execution model combines fine-grained parallel execution from heap-allocated contexts (threads) with larger grained sequential execution on the stack. Distinct versions of the code are generated and specialized for each role by the compiler. The program dynamically adapts to the location of data and available parallelism by speculatively executing sequentially and then falling back to parallel execution. Thus, local, sequential portions of the program accrue the advantages of fast synchronization, resource reclamation and better use of architectural features such as register windows, caches, and stack memory. Parallel portions use multiple threads executing from heap contexts to mask latency and limit the cost of synchronization by reducing the amount of live data that must be preserved during a context switch.

This execution model is general and efficient, integrating parallel and sequential code versions, and providing a hierarchy of calling schema and interfaces for the sequential versions with a range of features. The simplest schema is identical to a standard C function call while the most complex enables user defined communication and synchronization structures to be executed on the stack. By examining the call graph we automatically select the most efficient schema for each portion of the program so that C efficiency is achieved where possible.

In addition, our implementation is portable; written entirely in C, it is not specific to the stack structure on any particular machine. This execution model is used by the Illinois Concert system [5], a compiler and runtime which achieves efficient execution of concurrent object-oriented programs. This system compiles ICC++, a parallel dialect of C++ [14], and Concurrent Aggregates (CA) [8] for execution on workstations, the TMC CM5 [31] and the Cray T3D [10] simply by recompiling and linking with a machine specific version of the runtime.

We have evaluated the performance of the execution model for both sequential and parallel execution on a suite of programs written by different authors with a variety of programming styles. Our results on function-call intensive programs show that the hybrid execution model achieves sequential efficiency comparable to C programs. Additionally, measurements on the CM-5 and the T3D for a regular application kernel (SOR) and two irregular application kernels (MD-Force and EM3D) demonstrate that the hybrid model adapts well to a variety of data placement and synchronization contexts, yielding 1.5 to 3 times better performance than a purely parallel scheme.

In Section 2, we give the relevant background, describing irregular computational structures as well as the basic programming and execution models. Section 3 presents our hybrid stack-heap

[1] While data layout for such languages is an important issue and an area of much research [15], in this paper we focus on efficient execution with respect to a data placement.

execution mechanism. The effectiveness of this mechanism is demonstrated on a number of fine-grained concurrent programs in Section 4. Related work is discussed in Section 5, and we conclude in Section 6.

2 Background

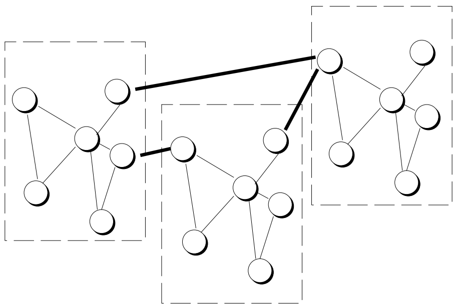

Figure 1: Data (Object) Layout Graph

Irregular and dynamic programs (such as molecular dynamics, particle simulations and adaptive mesh refinement) have a data distribution which cannot, in general, be predicted statically. In addition, modern algorithms for such problems depend increasingly on sophisticated data structures to achieve high efficiency [2, 13, 4]. In this domain, a program implementation must adapt itself to the irregular and even dynamic structure of the data (exploiting locality where available) to achieve high performance. This is particularly important when good data distributions are used which clump parts of data structures together. Figures 1 and 2 show examples of the irregular data and computation mappings that result.

In Figure 1, a set of objects represent the program data, and the arcs between them the relationships between the objects. A good data layout (indicated by the boxes around co-located objects) places groups of tightly coupled objects on the same node. An efficient computation over the data is shown in Figure 2. The tree of threads (activations) is distributed over the machine with portions at the leaves executing on co-located objects. Note that such structures benefit specifically from specialized sequential (those near the leaves) and parallel (those near the root) versions of code.

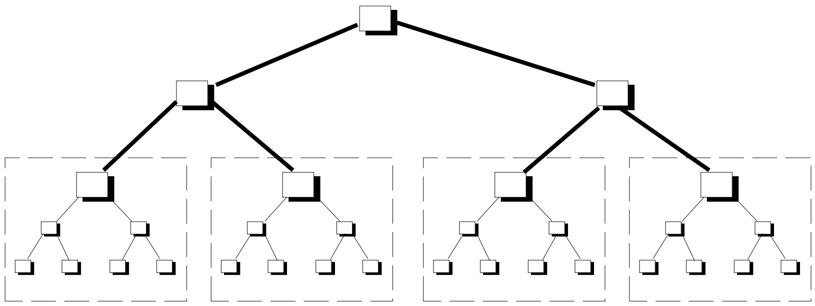

Figure 2: Distributed Computation Structure

The particular programming model we use is a fine-grained concurrent object-oriented model where each method invocation corresponds to a thread. Such a model provides programmers with a number of powerful tools which simplify programming and enable the construction of customized communication and synchronization structures. These tools are implicit synchronization, fine-grained concurrency, location independence, implicit locking, and first class *continuations*. Programmers are encouraged to expose concurrency (non-binding parallelism) in both blocks of statements and loops. Implicit synchronization is provided via implicit *futures* [16], which synchronize lazily on a value or a set of values in the case of a parallel loop. Introducing the futures implicitly solves the problem of future placement, but dramatically increases the density of futures over models where they are inserted manually. Object references are location independent and locking is dictated by data (class) definitions. As a result, both communication and lock operations are also implicit, increasing their frequency as well.

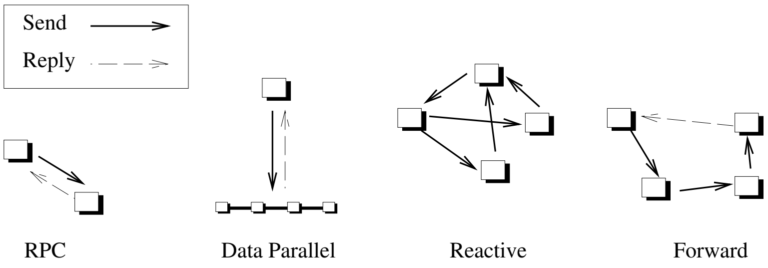

Figure 3: Synchronization and Computation Structures

Our programming model supports a wide variety of synchronization and communication structures including: synchronous (RPC), data (object) parallel, reactive and even custom communication and synchronization structures constructed as convenient for the application. In addition, continuations (the right to determine a future) can be forwarded [21] to another call, passed as arguments and stored in data structures. Graphical representations of these structures, all of which are supported by the ICC++ and CA langauges, appear in Figure 3. The flexibility of this programming model enables the programmer to select the mechanisms most appropriate for the application. While other execution models provide good performance for a particular structure, our hybrid execution model with its hierarchy of invocation schemas provides good sequential performance for all these communication and synchronization structures by enabling them to be executed on the stack.

The flexible parallel-sequential execution model presented in this paper dynamically adapts for parallel or sequential execution and provides a hierarchy of calling schemas of increasingly power and cost. This model is part of the Illinois Concert system [5, 7] which consists of a globally optimizing compiler [28] and runtime [23]. The compiler is capable of resolving interprocedural control and data flow information [27] which enables the use of specialized calling conventions based on the synchronization features required by the called method. The runtime provides a hierarchy of runtime primitives of increasing cost and complexity, enabling the compiler to select the most efficient mechanism for a given circumstance.

3 The Execution Model

The goals of our execution model are to be efficient, flexible, portable, and to support different data layouts by adapting to the location of objects at runtime. Furthermore, while concurrent method invocations cannot be supported using stack allocation alone, for efficiency purposes, stack-based invocations should be used whenever possible. In order to support these goals we have designed an execution model which uses sequential and parallel versions of methods and four distinct invocation schema. These schemas range from cheap, simple and limited to general, complex and more expensive. To avoid confusion, we refer to invocations in the concurrent object-oriented programming model as *method invocations* and C calls as *function calls*.

For each method, there are two versions: a version optimized for latency hiding and parallelism generation (which uses a heap context) and a version optimized for sequential execution (which uses the stack). The heap based version is completely general being capable of handling remote invocations and suspension, but can be inefficient when this generality is not required. The stack version comes in three flavors of increasing generality. These different versions and flavors use different calling conventions to handle synchronization, return values and reclaim activation records. Table 1 illustrates these cases.

Case		Basic Operation
Parallel		Most general schema, all arguments/linkage through the heap; frame reclamation based on function termination
Sequential	Non-blocking	Regular C call/return
	May-Block	Regular call; check return code to either continue computation or peel stack frames to heap
	Continuation Passing	Extension of May-Block which enables forwarding on the stack

Table 1: Invocation Schemas

In the remainder of this section, we describe how method invocations are mapped into C function calls. Because our compiler uses C as a portable machine language, these schemas correspond to the output of our compiler. First we will describe the heap-based parallel invocation mechanism, and then the flavors of stack-based invocation. Finally, in Section 3.3 we will describe *proxy* contexts and wrapper functions which are used to handle certain boundary cases.

3.1 Parallel Invocations

The parallel invocation schema is a conservative implementation of the general case, allocating the activation record as well as passing all arguments and return values through the heap. Parallel invocation create independent threads which store their state in the heap activation record. By storing inactive temporary values in the heap as well, we minimize the cost of suspension. Suspension occurs while waiting for:

- The result of a remote invocation.
- The result of a invocation on a locked object.
- The result a blocking primitive (like I/O).
- The result of an invocation which itself has suspended.

Suspension and fall back from the stack invocation schemas are described below.

The parallel version is optimized for concurrency generation and latency hiding. Invocations are issued in parallel, with the returns occurring in any order, and a set of futures are touched at one time to avoid unnecessary restarts of the activation when not all of the needed values are available. Thus, concurrency is generated both across parallel calls and between caller and callee and latency is masked by enabling several invocations from the same method to proceed concurrently. Figure 4 shows a sample of code which issues several parallel calls and synchronizes on their return with a single touch.

```
parallel_function(...args...) {
     invoke_method(fcn23,&location1,...);
     invoke_method(fcn1,&location2,...);
     invoke_method(fcn7,&location3,...);
     ... continue heap execution ...
     if (!touch(location1,location2,location3,...)) {
        store_state;
        suspend;
     }
     ... use values in location1, location2, location3 ...
}
```

Figure 4: Code Structure for a Parallel Method Version

3.2 Sequential Invocations

There are three different schema for sequential method versions, each requiring a different calling convention (see Figure 5). Because choosing the correct schema can depend on non-local (transitive) properties, our compiler performs a global flow analysis which conservatively determines the blocking and continuation requirements of methods and uses that information to select the appropriate schema [27, 26]. Since only one sequential version of each method is generated, this classification of the methods determines the calling convention which must be used when the method is invoked.

NON-BLOCKING	`return_val = non_blocking_method( ... );`
MAY-BLOCK	`callee_context = may_block_method(&return_val,...);`
CONTINUATION PASSING	`caller_context = cont_passing_method(&return_val,caller_info,...)`

Figure 5: Invocation Schema Calling Interfaces

The criteria for selection of the sequential method versions is as follows. If the compiler can prove that this method and all of its descendant calls cannot block, the **Non-blocking** version is used. In this case, the function return value can simply be used to convey the future value. When the compiler cannot prove that blocking will not occur but the callee does not require a continuation, the **May-block** is used. In this case we optimistically assume the method will not block, and allocate any required context lazily as described in Section 3.2.2. Finally, the **Continuation Passing** version is used if the callee may require the continuation of a future in the caller's as yet uncreated context. In this case we create both context and continuation lazily. Lazy creation of continuations is described in Section 3.2.3.

3.2.1 Non-blocking: Straight C Call

When the compiler determines that a method will not block, a standard C call is used. Since this information is determined by using the call graph, entire non-blocking subgraphs are executed with no overhead. Thus, those portions of the program which do not require the extra power of the programming model are not penalized.

3.2.2 May-block: Lazy Context Allocation

```
callee_context = may_block_method(&return_val,...);
if (callee_context != NULL) {      // fallback code
   context = create_context();
   callee_context->continuation = make_continuation(context[13]);
   save_state_to_heap(context);
   return context;                 // propagate blocking
}
```

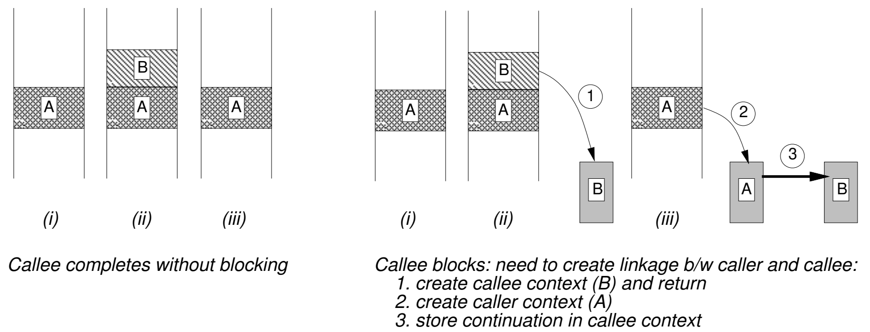

Figure 6: Calling schema for the **May-block** case. The figure on the left shows successful completion, while the figure on the right shows the stack unwinding when the call cannot be completed.

In the may-block case, the calling schema distinguishs the two outcomes – successful completion and a blocked method. If the callee runs to completion, a NULL value is returned, and the caller extracts the actual return value from `return_val`, a pointer to which is passed in as an argument. If the method blocks, the callee context (which itself was just created) is returned. This is necessary because the linkage between caller and callee was implicit in the stack structure, and the caller must insert a continuation for the callee's return value into the callee context to preserve the linkage. Subsequently, the caller will, if necessary, create its own context, revert to the parallel method version, and return its context to its caller. Figure 6 shows an example of the calling schema for the may-block case.

Using this mechanism, a sequence of may-block method invocations can run to completion on the stack, or unwind off the stack and complete their execution in the heap. The fallback code creates the callee's context, saves local state into it, and propagates the fall back by returning this context to its caller which then sets up the linkage.

3.2.3 Continuation Passing: Lazy Continuation Creation

Explicit continuation passing can improve the composability of concurrent programs [33, 6]. However, when continuation passing occurs, invocations on the stack are complicated because the callee may want its continuation. If the call is being executed on the stack, the callee's continuation is implicit. Since one of our goals is to execute forwarded invocations [20] on the stack, lazy allocation of the continuation is essential. As we shall see, allocation of a continuation also implies creation of the context in which the returned value will be stored.

```
Context* root_method(...,return_val_ptr,...) {
  ...
  caller_context = cont_passing_intermed(&return_val,
                              make_caller_info(root_func),...);
  if (caller_context != NULL) {
    save_state_to_heap(caller_context);
    return caller_context;         // propagate blocking
  }
}

Context* cont_passing_intermed(...,return_val_ptr,caller_info,...) {
  ...
  caller_context = cont_passing_method(return_val_ptr,caller_info,...);
  return caller_context;
}

Context* cont_passing_method(...,return_val_ptr,caller_info,...) {
  ...
  if ( can_return_value ) {
    *return_val_ptr = value;
    return NULL;
  } else {                            // need continuation
    caller_context = create_context_from_caller_info(return_val_ptr,caller_info);
    my_context = create_context();
    my_context->continuation = make_continuation(caller_context, caller_info);
    save_state_to_heap(my_context);
    return caller_context;
  }
}
```

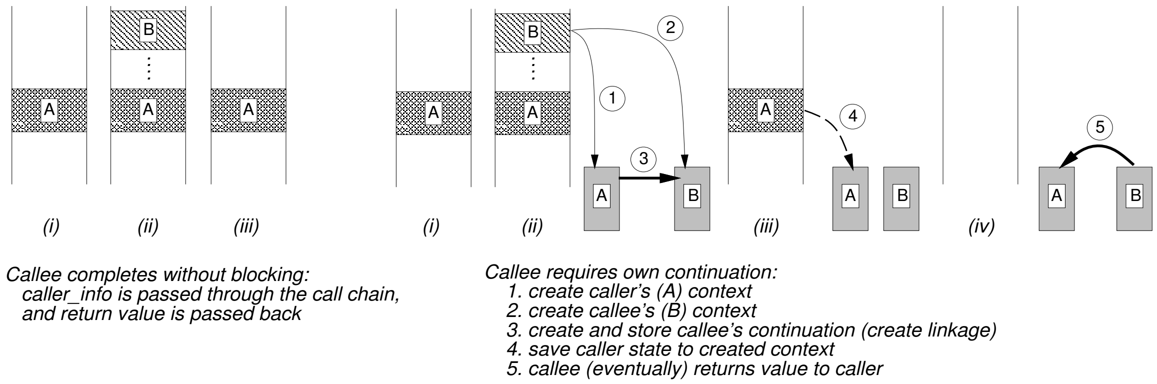

Figure 7: Calling schema for **Continuation Passing**. `root_method` (A) is the root of the continuation forwarding chain, `cont_passing_intermed` an intermediate function, and `cont_passing_method` (B) a function which either returns a value or requires its continuation.

The continuation passing schema (see Figure 7) uses an additional parameter, `caller_info`, which, along with the `return_ptr`, encodes information necessary to determine what to do should the continuation be needed. The `caller_info` information is simply passed along to support local forwarding, but if a method tries to store the continuation or forward it off-node, it must be created. `caller_info` indicates whether the context containing the continuation's future has already been created, the context's size if it has not, the location of the return value within the context, and whether the continuation was forwarded.

The `caller_info` is passed through the call chain, and if the continuation is not created (i.e. the continuation is not explicitly manipulated), the result can be passed on the stack through `return_val_ptr`. The method which replies simply stores the result through `return_val_ptr`, and passes NULL return values back to its caller. The caller of the first continuation-passing method (root of the forwarding chain), receives this NULL value and looks in `return_val` for the result, thus executing the forwarded continuation completely on the stack.

On the other hand, if the continuation is required, `caller_info` is consulted. There are three cases which must be handled by the fallback code. First, if the continuation was initially forwarded, the context must already exist as must the continuation (which is always stored at a fixed location in heap contexts). It is extracted by subtracting the return location offset in `caller_info` from the `return_val_ptr`, adding on the fixed location offset and dereferencing. Second, if the context already exists but not the continuation, the continuation is created for a new future at `return_val_ptr` which is at the return location offset within that context. Finally, if the context does not exist, it is created based on the size information from `caller_info`, and the continuation is created for a future at the return value offset. The callee may now do whatever is desired with the continuation, finally passing the continuation's future's context back to its caller.

3.3 Wrapper Functions and Proxy Contexts

Calling the sequential versions of methods from the runtime or a different schema method can require some impedance matching. For instance, when a message arrives at a node it contains a continuation for the return value. If the appropriate stack-based schema is non-blocking, a wrapper function is used to pass the return value to this continuation. Wrapper functions take either a vector of arguments or a communication buffer and invoke the stack-based version of a method with the appropriate calling convention. In this manner, a remote message can be processed entirely on the stack, and if the continuation is forwarded, it may pass through several nodes, finally respond to the initial caller, all without allocating a heap context.

Figure 8 illustrates how the wrapper functions invoke methods with different calling conventions from a communications buffer. In the case of a non-blocking method, we verify that a value was returned (which will not be the case in a purely reactive computation) and if so pass it to the waiting future by way of the continuation. Similarly for may-block, in addition to any value being returned, the continuation is placed in the callee's context in case the method suspends. Finally, for continuation passing, a proxy context is used along with a `caller_info` which indicates that the context exists and that the continuation was forwarded. Thus, if the continuation is required it is extracted from the proxy context (see Section 3.2.3). The proxy context technique is also used when an arbitrary continuation (perhaps one stored in a data structure) is passed by a user defined method to a function which requires a `return_val` and `caller_info` pair. This can occur with user-defined synchronization structures like barriers.

```
void non_blocking_msg_wrapper(Slot * buff) { // Non-blocking
  result_val = non_blocking_method(buff[0],...)
  if (!EMPTY(result_val)) reply(buff[CONTINUATION],result_val);
}

void may_blocking_msg_wrapper(Slot * buff) { // May-block
  Slot result_val = EMPTY_SLOT;
  Context * callee_context = may_block_method(&result_val,buff[0],...)
  if (!EMPTY(result_val)) reply(buff[CONTINUATION],result_val);
  if (callee_context != NULL) {
    callee_context->continuation = buff[CONTINUATION];
  }
}

void cont_passing_msg_wrapper(Slot * buff) { // Continuation Passing
  Proxy proxy_context;
  proxy_context.return_val = EMPTY;
  proxy_context.continuation = buff[CONTINUATION];
  Caller_Info caller_info = PROXY_CALLER_INFO;
  Context * caller_context = cont_passing_method(&proxy_context.result_val,...)
  if (!EMPTY(proxy_context.result_val)) reply(buff[CONTINUATION],result_val);
}
```

Figure 8: Wrappers for Stack Version Execution by the Runtime

3.4 Summary

We have described a set of method versions and invocation schemas that support the execution of many method invocations in a very general programming model on the stack. Important aspects of this system include customization for parallel or sequential execution, and the lazy allocation of contexts and continuations. The resulting scheme is efficient, flexible, portable and improves overall runtimes as discussed in the next section.

4 Evaluation of Hybrid Execution Model

This section presents a performance evaluation of the hybrid execution mechanisms. First, we examine the base costs of the various calling schemas in terms of dynamic instruction counts. Second, we show the high sequential efficiency achievable by the hybrid model by comparing the sequential performance of codes using our execution model (including forwarding) to the same programs written in C. Finally, we examine parallel performance by measuring the improvement over straight heap-based execution, demonstrating the ability of the execution model to adapt to different data locality characteristics for both regular and irregular parallel codes. The experiments reported in this section were all conducted in the context of the Illinois Concert System on SPARC workstations, the CM-5 and the T3D. The T3D implementation is less mature and we expect the absolute performance to improve.

4.1 Base Overheads

Table 2 presents the cost of the sequential invocation mechanisms for various caller-callee sce-

CALL OVERHEAD

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential: NB	MB	CP
Parallel		130	0	6	8
Seq.	NB	–	0	–	–
	MB	–	0	6	8
	CP	–	0	6	8

FALLBACK OVERHEAD

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential: NB	MB	CP
Parallel		–	0	8	8
Seq.	NB	–	0	–	–
	MB	–	0	79	39
	CP	–	0	140	100

Table 2: Call and fallback overheads (at the caller) for different caller-callee scenarios, expressed in terms of SPARC instructions required in addition to a C function call. NB, MB and CP stand for Non-blocking, May-block and Continuation Passing sequential calling schemas respectively.

narios as the overhead (in SPARC instructions) beyond the cost of a basic C function call[2]. There are two components to this overhead: the first (shown in the left table) corresponds to the situation when the sequential invocation completes on the stack, and the second (shown in the right table) indicates the additional fallback cost when the invocation must be unwound into the heap. Sequential invocations which do not block have cost comparable to a basic C function call and an order of magnitude less overhead than the parallel (heap-based) invocation (130 instructions). The 6–8 additional instructions for sequential calls are due to additional invocation arguments, and passing the return value through memory, rather than in a register.

The fallback overheads vary from 8 – 140 instructions depending on the specific caller-callee scenario. The unwinding costs are different for the different caller and callee combinations because the different schemas place the responsibility for heap context creation and state saving at different places. These fallback overheads make explicit the tradeoff in using the sequential and parallel versions. The maximum fallback cost for any caller-callee pair is comparable to the basic heap-based invocation, so speculative execution using a sequential invocation first is cheaper in almost all cases. The same numbers also show that a sequential method version can incur substantial overhead if it blocks repeatedly incurring multiple fallbacks; thus, reverting to the parallel method after the first fallback is a good strategy, especially if several synchronizations are likely.

4.2 Evaluation of Sequential Performance

Table 3 presents the sequential performance of the hybrid mechanisms for a set of function-call intensive benchmark programs and compares it to equivalent C programs. Using the most flexible hybrid version (*3 interfaces*), all programs run significantly faster than the heap-only versions and achieve close to the performance of a comparable C program.[3] Several programs required all three interfaces to achieve comparable performance. The remaining overhead is due to parallelization. *Seq-opt* shows the performance when this overhead is eliminated.

Since the generated executables can be run directly on parallel machines, they include parallelization overhead in the form of name translation, locality and concurrency checks; the cost of which are discussed in [23]. These checks determine whether or not a invocation can complete immediately, and are used to suspend the caller and to *speculatively inline* [5] invocations on local and

[2] On a SPARC with register windows, a C funtion call costs 5 instructions but it is more likely to be between 10-15 instructions on other processors.

[3] The relative performance of `fib` and `tak` is a result of the comparatively aggressive inlining of our compiler.

Program Description	Parallel Version	Hybrid Parallel-Sequential Versions				C Program
		1 interface	*2 interfaces*	*3 interfaces*	*Seq-opt*	
fib(29)	13.12	1.01	0.95	0.70	0.69	1.10
tak(18,12,6)	152.87	11.37	12.08	6.75	6.71	7.00
qsort(10000)	2.90	0.25	0.28	0.23	0.23	0.16
nqueens(8x8)	3.37	0.77	0.80	0.60	0.56	0.38
list-traversal (128 elements)	1.34	1.05 1.17[a]	0.81 1.00[a]	0.81 0.87[a]	0.81 0.87[a]	0.78

[a]without forwarding optimization supported by Continuation-passing interface.

Table 3: Sequential execution times (in seconds) using the hybrid mechanisms compared with times for a parallel-only and a comparable C program. The hybrid versions represent varying degrees of flexibility: *1 interface* uses only the Continuation-passing interface, while *3 interfaces* uses all three interfaces. *Seq-opt* is a version which eliminates parallelization overheads.

unlocked objects. Since speculative inlining lowers the overall call frequency, it decreases the impact of our hybrid execution model. However, since it is required to obtain good performance from a fine-grained model, we include speculative inlining as part of all our results.

Comparing across the three hybrid versions demonstrates the benefits of a flexible interface. Using a version which provides all three stack interfaces (i.e., the non-blocking, may-block and continuation-passing call schemas) improves performance by up to 30% as compared to when only the most general (continuation passing) interface is always used.[4] It further shows that the hybrid mechanisms can provide C-like performance when all data is locally accessible.

4.3 Evaluation of Parallel Performance

In this section, we consider three application kernels to characterize the parallel performance of the hybrid mechanisms versus a straight heap-based execution, particularly looking at how the performance varies with data locality. We first consider a regular code, SOR, showing that the performance improvement using the hybrid mechanisms increases proportional to the amount of data locality. We then consider two irregular codes – MD-Force and EM3D, demonstrating the ability of the hybrid mechanisms to adapt to available data locality in the presence of irregular computation and communication structures.

4.3.1 Regular Parallel Code: SOR

Successive Over Relaxation (SOR) is an indirect method based on finite differences for numerically solving partial differential equations on a grid. Our algorithm evaluates the new value of a grid point according to a 5-point stencil and consists of two half-iterations: in the first half-iteration we compute the new value for each grid point, and in the second half-iteration, we update the grid point with this computed value. To characterize the impact of data locality, we keep the grid size fixed (1024×1024) and consider various block sizes for a block-cyclic distribution of the grid on an 8×8 grid of processors. These different data layouts result in different ratios of local to remote method invocations and correspond to differing amounts of data locality.

[4]In some cases the performance using two interfaces is worse than that using only one interface: this anomaly arises from an improper alignment of invocation arguments causing them to be spilled to stack instead of being passed in registers.

Data Locality		CM-5 Performance			T3D Performance		
Block Size	*Local vs Remote*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*	*Parallel (secs)*	*Hybrid) (secs)*	*Parallel/ Hybrid*
8×8	0.083:1	135.58	136.85	0.991	48.99	43.50	1.126
16×16	1.167:1	97.16	88.15	1.102	46.77	28.66	1.632
32×32	3.333:1	83.47	52.15	1.601	43.46	20.70	2.099
64×64	7.667:1	60.33	32.43	1.860	34.94	14.97	2.334
128×128	16.333:1	45.80	19.89	2.303	28.46	12.00	2.372

Table 4: Parallel execution times for SOR (1024×1024 grid, 100 iterations) on 64-node configurations of the CM-5 and T3D. The performance of hybrid mechanisms is compared with a parallel-only version for varying amounts of data locality (*Block Size* corresponds to a block-cyclic distribution of the grid, and *Local vs Remote* gives the ratio of local to remote method invocations for the layout).

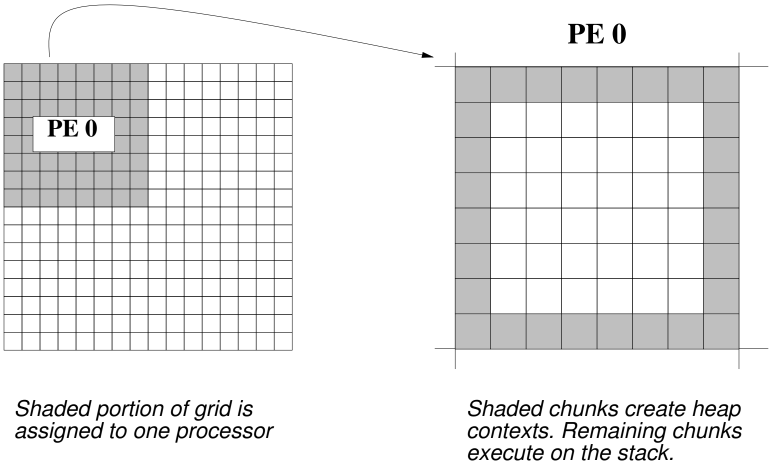

Figure 9: Hybrid mechanisms for the SOR code: heap contexts are only created on the perimeter of the block, all internal chunks execute on the stack.

Table 4 shows the performance of the hybrid mechanisms on 64-node configurations of the CM-5 and T3D for five choices of the block size. The hybrid mechanisms adapt to these different layouts improving overall performance over a parallel-only version by up to 2.4 times. Figure 9 shows the reason for this improvement: in contrast to the straight heap-based parallel version where heap contexts are created in each half-iteration for each grid element, using the hybrid versions, heap contexts need only be created for the grid elements on the perimeter of the blocks assigned to the processor (shown shaded in the figure). Computation on the internal elements can proceed on the stack (shown by clear boxes) and consequently incur significantly reduced overhead.

The results in Table 4 also show that the overall improvement from hybrid mechanisms is directly proportional to the amount of data locality. The speedup of hybrid mechanisms over the parallel version increases from ~1.0 when the fraction of local invocations is 0.077 to ~2.4 when the fraction of local invocations is 0.942.[5] These speedup numbers are in the neighborhood of the theoretical peak values which are determined by the relative costs of useful work, invocation overhead and

[5] For very low locality on the CM-5, the hybrid mechanisms perform worse than the straight heap based scheme because of the large number of fallbacks.

remote communication. For example, factoring out the useful work in the 128×128 SOR block layout on the CM-5, the maximum possible speedup we can achieve is 2.63 given that on average a remote invocation incurs 10 times the cost of a local heap invocation. Our measured value of 2.3 comes close to this maximum.

4.3.2 Irregular Parallel Code: MD-Force

MD-Force is the kernel of the nonbonded force computation phase of a molecular dynamics simulation of proteins [17]. The computation iterates over a set of atom pairs that are within a spatial cutoff radius. Each iteration updates the force fields of neighboring atoms using their current coordinates, resulting in irregular data access patterns because of the spatial nature of data sharing. Our implementation reduces the communication demands of the kernel by locally caching the coordinates of remote atoms and combining force increments.

Data Locality		CM-5 Performance			T3D Performance		
Data Layout	*Local vs Remote*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*
Random	0.38:1	10.71	10.41	1.03	3.94	3.82	1.03
Block	6.05:1	1.46	1.02	1.43	1.32	0.87	1.52

Table 5: Parallel execution times for MD-Force kernel (10503 atoms for 1 iteration) on 64-node configurations of the CM-5 and T3D. The performance of the hybrid mechanisms is compared with a parallel-only version for low-locality random and high-locality block distributions.

Table 5 shows the performance of the MD-Force kernel using two data layouts. The *random* layout uniformly distributes atoms on the nodes, ignoring the spatial distribution of atoms. In contrast, the *spatial* layout adopts orthogonal recursive bisection to group together spatially proximate atoms. Our results show that the hybrid mechanisms improve performance over a parallel-only scheme even for applications with irregular computation and communication structures. Similar to the regular SOR kernel, the performance advantages increase with the amount of data locality. For the *random* distribution, because of poor locality the dominant contributor to the execution time is the communication overhead. Since communication costs remain unchanged by the choice of the invocation mechanisms, we only achieve a speedup of 1.03 in this case. On the other hand for the spatially blocked distribution, the hybrid mechanisms enable the computation to adapt dynamically to data locality, yielding speedups of 1.43 on the CM-5 and 1.52 on the T3D. When run time checks determine that both atoms of an atom pair are local, the computation is small and entirely speculatively inlined. When an atom is found to be remote but its coordinates are in the cache, the computation is larger but completes entirely on the stack without incurring parallel invocation overhead. Otherwise, communication is required, and the stack invocation falls back to the parallel version to enable multithreading for latency tolerance. Even for such invocations, the hybrid mechanisms provide a performance improvement because of a better integration of the communication and computation — the target method can execute directly from the message handler. Thus, these remote invocations avoid the overheads of context creation and thread scheduling which are otherwise present.

4.3.3 Irregular Parallel Code: EM3D

EM3D is an application kernel which models propagation of electromagnetic waves [11]. The data structure is a graph containing nodes for the electric field and for the magnetic field with edges between nodes of different types. A simple linear function is computed at each node based on the values carried along the edges. Three versions of the EM3D code were prepared to evaluate the ability of the hybrid model to adapt to different communication and synchronization structures. Since they are intended to examine invocation mechanisms, elaborate blocking mechanisms were not used. The first version, `pull`, reads values directly from remote nodes. The second version, `push`, writes values to the computing node, updating from the remote nodes each timestep. Finally, in the `forward` version, the updates were done by forwarding a single message through the nodes requiring the update.

Data Locality		CM-5 Performance			T3D Performance		
Algorithm	*Local vs Remote*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*
EM3D	0.0156:1	93.93	68.94	1.362	349.04	336.32	1.037
pull	99:1	7.42	3.34	2.222	29.681	25.96	1.148
EM3D	0.0156:1	543.73	145.36	3.741	494.85	473.59	1.045
push	99:1	11.76	10.96	1.073	41.27	29.47	1.400
EM3D	0.0156:1	180.40	181.6	0.993	602.41	433.65	1.389
forward	99:1	18.86	15.53	1.214	112.79	39.41	2.262

Table 6: Parallel execution times for EM3D (8192 nodes of degree 16 for 100 iterations) on a 64-node CM-5 and a 16-node T3D. The performance for three versions of the algorithm using the hybrid mechanisms is compared with parallel-only versions for random node placement with low locality (0.0156:1) and placement with high locality (99:1).

Table 6 describes the performance of the three versions of EM3D on a 64-node CM-5 and a 16-node T3D. It shows that the hybrid scheme is capable of improving performance for different communication and synchronization structures for both cases of high and low data locality. In the case of low locality, efficiency is increased because off-node requests are handled directly from the message buffer, without requiring the allocation of a heap context. When locality is high, the hybrid mechanism can also execute fully local portions of the computation entirely on the stack. The hybrid mechanisms yield speedups ranging from unity to nearly four times, achieving superior performance in all but one case where the continuation passing schema is used with extremely low locality on the CM-5. In addition to the cost of fallback, this combination produces the worst case for our scheduler on the CM-5.

Overall, the *pull* version provides the best absolute performance since it computes directly from the values it retrieves rather than using intermediate storage. The *forward* version requires longer update messages than *push* but fewer replies. On the CM-5 replies are inexpensive (a single packet), so the cost of *forward*'s longer messages overwhelms the cost of the larger number of replies required by *push*. However, on the T3D the decrease in overall message count enables *forward* to perform better than *push* for low locality. The CM-5 compiler performs better on the unstructured output of our compiler than the T3D compiler. As a result, the cost of the additional operations required by *push* and *forward* has less of an impact on the T3D than messaging overhead. Thus, for high locality, the hybrid mechanism is most beneficial for *pull* on the CM-5 and for *forward* on the T3D where local computation and messaging dominate respectively.

5 Discussion and Related Work

We have described a hybrid execution model for fine-grained concurrent programs. While the mechanisms described in this paper were developed for concurrent object-oriented languages [20, 33] based on the Actor model [18, 9, 1], we believe they are applicable to other programming models that support implicit synchronization and communication. In particular, this model is useful as a compiler target. It has a portable implementation and provides a hierarchy of mechanisms of varying power and cost supporting a wide range of communication and synchronization structures. Moreover, it can adapt to runtime data layout reducing the penalty for imperfect compile time information, irregular computations and poor programmer data distribution. As the target for the Concert system it supports both the ICC++ [14] and Concurrent Aggregates [8] languages.

Several languages supporting explicit futures on shared-memory machines have focused on restricting concurrency for efficiency [16, 24]. However, unlike our programming model where almost all values define futures, futures occur less frequently in these systems, decreasing the importance of their optimization. More recently, lazy task creation [25], leapfrogging [32] and other schemes [22, 3] have addressed load balancing problems resulting from serialization by stealing work from previously deferred stack frames. However, none of these approaches deals with locality constraints arising from data placements and local address spaces on distributed memory machines.

Several recent thread management systems have been targeted to distributed memory machines. Two of them, Olden [29] and Stacklets [12] use the same mechanism for work generation and communication. Furthermore, they require specialized calling conventions limiting their portability. StackThreads [30] used by ABCL/f has a portable implementation. However, this system also uses a single calling convention, and allocates futures separate from the context. Thus, an additional memory reference is required to touch futures. Also, its single version of each method cannot be fully optimized for both parallel and sequential execution.

While the portability of the current implementation of our hybrid execution model has many advantages (especially when considering the short lifespan of some parallel architectures), greater control of code generation would enable additional optimizations. For example, directly managing register spilling and the layout of temporaries in the parallel and sequential versions could reduce the cost of falling back from sequential to parallel computation. Furthermore, modifying the calling convention to support a different stack regimen and multiple return values would reduce the cost of the more general stack schemas. However, we do not expect these optimizations to alter the basic performance trends.

6 Conclusion and Future Work

We have presented a flexible and efficient hybrid execution model for fine-grained concurrent programs. This model adapts to the locality and synchronization structure of the program at runtime by using separate parallel and sequential code versions. In addition, it uses a hierarchy of calling schemas based on C function calls to achieve both high efficiency and portability (no assembly code is required). Performance results on function-call intensive programs show that the hybrid model achieves the sequential efficiency of C programs. Measurements on the CM-5 and T3D for one regular (SOR) and two irregular application programs (MD-Force and EM3D) demonstrate that the hybrid model effectively adapts to available runtime locality, yielding 1.5 to 3 times better performance than a purely parallel scheme.

We are currently working on automating data layout, migration and selection of communication and synchronization structures. Our current system requires the user to specify these aspects of the computation explicitly. By abstracting them at a higher level, we will be able to use the flexibility of our execution model to optimize the implementation with respect to the cost profile of the target platform.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[2] J. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. Technical report, The Institute for Advanced Study, Princeton, New Jersey, 1986.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Andrew Shaw, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. To appear in PPoPP '95, July 1995. Available via anonymous FTP from `theory.lcs.mit.edu` in `/pub/cilk/cilkpaper.ps.Z`.

[4] D. A. Case. Computer simulations of protein dynamics and thermodynamics. *IEEE Computer*, 26:47, 1993.

[5] Andrew Chien, Vijay Karamcheti, and John Plevyak. The Concert system – compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[6] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, MA, 1993.

[7] Andrew A. Chien and Julian Dolby. The Illinois Concert system: A problem-solving environment for irregular applications. In *Proceedings of DAGS'94, The Symposium on Parallel Computation and Problem Solving Environments.*, 1994.

[8] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. Concurrent aggregates language report 2.0. Available via anonymous ftp from cs.uiuc.edu in /pub/csag or from http://www-csag.cs.uiuc.edu/, September 1993.

[9] William D. Clinger. Foundations of actor semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

[10] Cray Research, Inc. *Cray T3D System Architecture Overview*, March 1993.

[11] A. Krishnamuthy et al. Parallel programming in Split-C. In *Proceedings of Supercomputing*, pages 262–273, 1993.

[12] Seth Copen Goldstein, Klaus Eric Schauser, and David Culler. Lazy threads, stacklets, and synchronizers: Enabling primitives for parallel languages. In *Proceedings of POOMA'94*, 1994.

[13] L. Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–48, 1987.

[14] The Concurrent Systems Architecture Group. The ICC++ reference manual, version 1.0. Technical report, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois, 1995. Also available from http://www-csag.cs.uiuc.edu/.

[15] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 1992.

[16] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[17] J. Hermans and M. Carson. Cedar documentation. Unpublished manual for CEDAR, 1985.

[18] C. Hewitt and H. Baker. Actors and continuous functionals. In *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*, pages 367–87, August 1977.

[19] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for FORTRAN D on MIMD distributed-memory machines. *Communications of the ACM*, August 1992.

[20] W. Horwat, A. Chien, and W. Dally. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–9. ACM SIGPLAN, ACM Press, 1989.

[21] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1989.

[22] Suresh Jagannathan and Jim Philbin. A foundation for an efficient multi-threaded Scheme system. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 345–357, June 1992.

[23] Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing'93*, 1993.

[24] D. Kranz, R. Halstead Jr., and E. Mohr. Mul-T: A high-performance parallel lisp. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[25] E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[26] John Plevyak and Andrew Chien. Efficient cloning to eliminate dynamic dispatch in object-oriented languages. Submitted for Publication, 1995.

[27] John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA*, 1994.

[28] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 311–321, January 1995.

[29] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 1995.

[30] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock cpus. In *Joint Symposium on Parallel Processing*, 1994.

[31] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.

[32] David B. Wagner and Bradley G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pages 208–217, 1993.

[33] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990. ISBN 0-262-24029-7.