Summer 8-1-1992

Portable, Efficient Futures ; CU-CS-609-92

David B. Wagner
University of Colorado Boulder

Bradley G. Calder
University of Colorado Boulder

Portable, Efficient Futures

David B. Wagner Bradley G. Calder

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-609-92 August 1992

University of Colorado at Boulder

Technical Report CU-CS-609-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

David B. Wagner Bradley G. Calder

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

Portable, Efficient Futures

David B. Wagner Bradley G. Calder

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

August 1992

Abstract

A *future* is a language construct that allows programmers to expose parallelism in applicative languages such as MultiLisp [8] with minimal effort. We propose that futures are natural for considerably wider application than this. In particular, they are well-suited for dynamic programming problems because they eliminate the need for explicit synchronization.

In this paper we describe a C++ implementation of futures that is efficient, portable, and easy to use. Our system employs a new technique, *leapfrogging*, that reduces blocking due to load imbalance. The utility of leapfrogging is enhanced by the fact that it is completely platform-independent, is free from deadlock, and places a bound on stack sizes that is at most a constant times the maximum depth of the computation tree.

We report on the performance of our system on recursive, iterative, and dynamic programming problems. Along the way, we illustrate several coding techniques that can be used by the programmer to improve performance, and we quantify this improvement.

1 Introduction

A *future* is a language construct that enables programmers to identify parallelizable computations in an otherwise sequential program. Rather than waiting for the result of such a computation, the program receives a "placeholder" for that result and is able to continue executing. The placeholder behaves just like any other variable until an attempt is made to use its value; at that point, if the computation is not finished then the thread of execution trying to obtain the value will be blocked until the value is ready.

The principal design rationale behind futures is that "the programmer takes on the burden of identifying *what* can be computed safely in parallel, leaving the decision of exactly *how* the division [of work] will take place to the runtime system" [9]. This is very similar to parallelizing a Fortran loop by inserting `DOALL`-style compiler directives. However, whereas Fortran compiler directives are naturally suited for exposing parallelism in looping constructs, futures are naturally suited for exposing parallelism in recursive constructs. Thus, the bulk of research on futures has been in the context of functional-style languages such as Lisp [5, 9].

The following example from [9] shows how futures can be used in Mul-T, a dialect of Scheme. The algorithm recursively sums the leaf values of a binary tree. Figure 1 shows the sequential version of this algorithm, and Figure 2 shows the "futurized" version. Simply by inserting the

```
(define (sum-tree tree)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (sum-tree (left tree))
         (sum-tree (right tree)))))
```

Figure 1: Mul-T version of sequential `sum-tree`.

```
(define (psum-tree tree)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (future (psum-tree (left tree)))
         (psum-tree (right tree)))))
```

Figure 2: Mul-T version of parallel `sum-tree`.

word `future` before the first recursive call to `sum-tree`, the programmer has indicated that the two recursive calls can be executed in parallel.

Another nice feature of futures is that all synchronization is implicit. The programmer never explicitly checks to see if the value of a computation is ready; if it is not, the thread of control trying to use that value is transparently blocked. Because of this, futures are as powerful as general fork-join, yet much easier to use.

From this example it is clear that futures provide a very minimal-effort method of parallelizing a recursive program. Consider, for example, how much simpler it would be to parallelize a loop having cross-iteration dependencies using futures than using the `POST-WAIT`-style mechanism of some Fortran dialects. This suggests that futures have another, unexplored domain of application: computation graphs with complex precedences, such as dynamic programming computations.

There are significant challenges to making a futures runtime system efficient when the emphasis is on *minimal programmer effort*. For example, the programmer might identify very fine-grained computations as candidates for parallel execution. Actually executing each of these computations as a separate task might be counterproductive, since the overhead of creating tasks and synchronizing might overwhelm the time saved by parallelization.

Another potential problem is that of unbalanced computation graphs. The computation graph might be structured such that the result of a future is almost never ready when called for; this leads to extra overhead, either in the form of idle processor time or context switching. Even worse, a careless implementation can lead to very subtle deadlock problems in realistic (not contrived) computations.

A very elegant, but complicated, solution to these problems was presented by Mohr, Krantz and Halstead [9]. Their system, which we will explore in more detail in the next section, is based on *continuations*. Our main concern with their method is that it seems to be very "wedded" to a particular compiler and architecture, which raises some concerns about portability. This is due not to sloppy implementation by any means, but rather is a consequence of their design decision to use continuations as the fundamental mechanism for dynamically partitioning work.

In this paper we present an alternative implementation of efficient futures that also is very portable. Our implementation is `C++`-based, although that is strictly a matter of convenience rather than a limitation of our techniques. The crucial point is that our implementation provides efficiency similar to that of ultra-specialized implementations such as [9], but is immediately portable to any machine that provides a Cthreads [2] runtime environment, and can be ported with very little effort to any system that provides process creation, shared memory, and lock synchronization primitives.

A secondary contribution of our work is that we explore in detail the problem of applying the futures paradigm to non-recursive applications with complex precedence graphs. In this context, we explore the tradeoff between programmer effort (in terms of code restructuring and providing hints to the runtime system) and performance. Furthermore, we introduce the novel concept of an *unbound future*, which allows for a very natural coding style for computations of this type.

The remainder of this paper is organized as follows. Section 2 provides a more detailed look at the efficiency issues involved in implementing futures and describes our solution to these problems. Section 3 starts with a a discussion of `C++`-related syntactic issues, and then delves into what is going on "behind the scenes." We also state and prove the conditions under which our implementation is guaranteed to be deadlock free, as well as some other nice properties. We conclude Section 3 with some remarks about portability. In Section 4 we present performance data to demonstrate our claims of efficiency, and we quantify the effect of programmer effort on this efficiency. Finally, Section 5 summarizes our conclusions and discusses directions for future research.

2 Performance Issues

2.1 Basic issues

The `sum-tree` example provides a good illustration of how difficult it is to get good load distribution and minimal blocking using futures, because it contains a plethora of potential parallelism. A naive approach to parallelizing `sum-tree` would use a separate task for each future, so that the number of tasks used would be equal to the number of leaves in the tree. This excess task creation will greatly diminish the speedup of the parallelized program, since the granularity of task creation will certainly be much larger than the granularity of a leaf computation.

The ideal approach to parallelizing the `sum_tree` algorithm would be to expand the tree "breadth-first by spawning tasks until all processors are busy, and then expanding the tree depth-first within the task on each processor" [9], which is abbreviated "BUSD." The BUSD approach is optimal for a balanced computation tree; however, optimal load balancing is much harder to realize in the general case.

In order to achieve BUSD execution, the programmer could limit the creation of tasks to a fixed depth in the algorithm itself. Below that depth the algorithm would execute the recursive calls sequentially, rather than futurizing them. In this way, the programmer takes responsibility for matching the granularity of the tasks to the overheads of the runtime system. This is the approach taken by the Qlisp system [6, 7]. In our opinion, this is not as desirable as having the system do it for the programmer automatically.

Furthermore, although use of a static cutoff point can sometimes approximate the optimal solution, in general it will not always be possible to come up with a simple heuristic for an unbalanced computation tree. If there are exactly as many tasks as processors and some tasks finish much faster than others, then those processors are underutilized. Thus, failure to create

enough tasks can be as bad as creating too many of them. Frequently, the only way to find the optimal cut-off depth is through experimentation.

One possible attack on the granularity problem is called *load-based inlining*. The idea is to dynamically monitor the load and create a new task to compute a future only when processors are idle. Otherwise, the future is executed inline by the task that creates it. This keeps the number of tasks created close to the number of processors available. Qlisp [6, 7] provides the programmer with primitives that inspect the state of the system, and allows the programmer to specify an arbitrary predicate to control the inlining of futures.

One problem with load-based inlining is that once the decision has been made to inline a future, the decision cannot be revoked. This could result in some processors sitting idle while others work on large tasks, especially if the task granularity is unpredictable. Another problem is that keeping track of processor availability creates extra overhead for the task scheduler.

Thus, there appears to be a fundamental tradeoff between achieving good task granularity and achieving good load balance.

2.2 WorkCrews

The "obvious" solution to the granularity vs. load balancing dilemma is to note that there is really no need to create a separate task for every future. Instead, a WorkCrew-style approach can be adopted [11]: a future is implemented as a passive object that contains enough information to carry out the computation. These passive objects are then picked up by worker tasks and executed. The number of workers is fixed and is equal to the number of processors. The advantage of using passive futures and active workers is that the overhead of creating a future is reduced so much that inlining, with its associated load balancing problems, rarely is required for good performance.

Unfortunately, there are other performance problems with a WorkCrew-style implementation. Foremost among these is that of workers blocking to wait for an unfinished future. In the one-task-per-future implementation, either (a) the blocked task can idle its processor until the desired result is available, or (b) it can context switch to some other task. The former lowers processor utilization, thus the latter alternative would be preferred unless futures were of extremely small granularity. Unfortunately, in the WorkCrew approach there may not be any other task to switch to! A heuristic solution to this is to set the number of workers to be larger than the number of processors, but there is likely to be no good justification for any particular choice of this number.

Thus, we conclude that a naive WorkCrew approach would be considerably faster than the one-task-per-future approach for balanced computations, but could be arbitrarily bad for unbalanced computations.

2.3 Lazy Task Creation

Mohr et al. [9] have devised a clever scheme that they call *Lazy Task Creation* that solves the problems just discussed. Their technique is built into a compiler for Mul-T, a parallel dialect of Scheme.

In Mul-T, `(K (future X))` is interpreted to mean, "Start evaluating X in the current task, but save enough information so that its continuation K can be moved to a separate task if another processor becomes idle" [9]. This effectively makes the decision to inline any future a revocable one, so the default behavior is that every future is inlined. There are exactly as many tasks as processors, and continuations are stolen only when a processor becomes idle.

Lazy Task Creation is implemented by maintaining a list of continuations on the producer task's stack. Then when a continuation is stolen, the scheduler cuts the stack apart and replaces the inlined future's stack frame with a placeholder. The continuation-based strategy also provides a strategy for dealing with the problem of a task blocking on some unfinished future. When such a case arises, the scheduler selects another continuation from the blocked task's stack to work on and cuts the stack again. The authors claim that the costs of doing this are low enough to meet their performance goals. For more details see [9].

Obviously, Lazy Task Creation is a completely general approach with the potential for very good performance. If the load is well-balanced, continuation-stealing will happen infrequently, if at all, and the computation will proceed much faster than if the futures had not been inlined. On the other hand, if the computation is seriously unbalanced, then it is crucial that the implementation make the time required to steal a continuation competitive with having created a separate future in the first place.

The principal drawback to the Lazy Task Creation alternative, as we see it, is that it requires some very intricate implementation and is *very* non-portable. Using a traditional call-stack implementation, continuation stealing requires the copying of stack frames, which is platform-dependent as well as expensive. To solve the latter problem, Mohr et al. implemented a linked-list call-stack, which obviously does not conform to the call-stack standards of very many existing architectures. This will likely limit its use to a narrow community of users.

As mentioned earlier, our experience has convinced us that futures have a much wider domain of application than they have usually been credited with, and our motivation is thus to "spread the word" about them to as wide a community of users as possible. The main contribution of this research is that we have developed a very simple technique, *leapfrogging*, that provides very good performance while being extremely simple to implement. Our system is written in C++ but the leapfrogging technique is not language- or compiler-dependent.

2.4 Leapfrogging

The leapfrogging technique is built into a WorkCrew-style implementation. In this implementation, futures are passive objects rather than active tasks. There are a fixed number of worker tasks at all times,[1] and each worker has a corresponding FIFO work queue into which futures are inserted and removed. A worker can remove a future from any work queue, but by default inserts futures only into its own.[2]

When a worker finishes evaluating a future and has nothing in its work queue, it looks in other workers' work queues in order to find work. This is termed *stealing*. Since futures typically are not inlined (because the cost of creating them is so small), if there is work available anywhere in the computation, an idle worker will eventually find it.

Stealing work is the principal way of balancing the load among the workers, but it does not solve the problem of workers blocking on unresolved futures. *Leapfrogging* is our solution to the problem of worker blocking. A prototypical leapfrogging scenario is depicted in Figure 3.

Suppose that a worker, X, creates a future F1 and begins executing its continuation K1 (Figure 3(a)). Now worker Y steals F1 and the scenario becomes that of Figure 3(b). (In the

[1] Our current implementation is built on top of a Cthreads [2] runtime library, so each worker is implemented as a Cthread. Ideally each Cthread runs on a different processor, but the implementation works correctly in any case.

[2] The user can optionally specify a particular work queue in which to insert the new Future, to aid in load-balancing. See section 3 for examples.

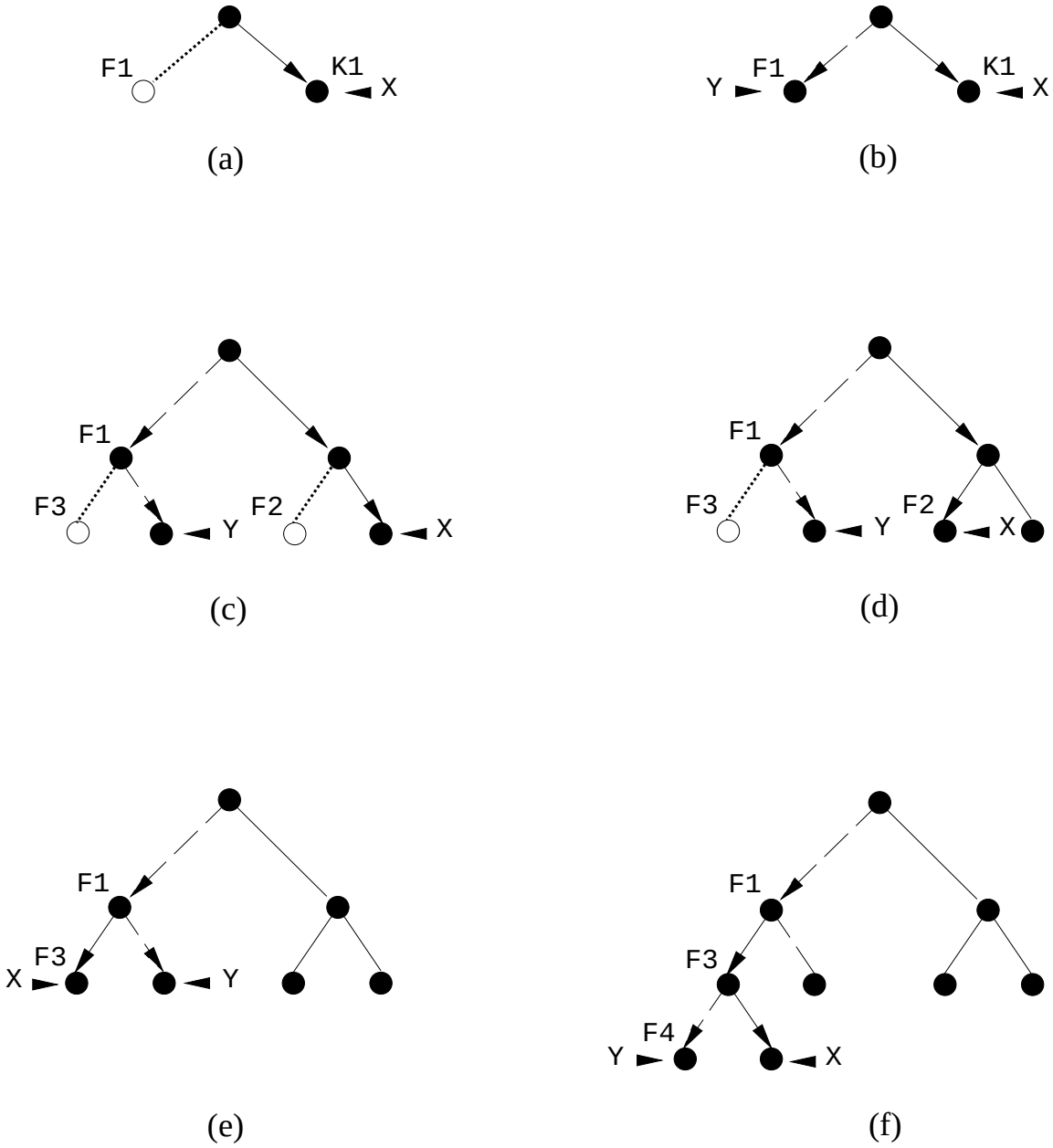

Figure 3: Typical leapfrogging scenario.

figure, the clear nodes represent uncomputed futures, and the opaque nodes represent work that has been or is currently being executed. The current position of each worker is marked with an arrowhead, and the flow of control of each worker is differentiated by line styles.)

Sometime later, the computation might appear as shown in Figure 3(c). Both workers have created additional futures and are further down in the computation tree. At this point, suppose that X returns to the next higher level in the tree and requires the value of F2, which has not yet been computed. Since no other worker has claimed F2, X would inline it (Figure 3(d)).

Eventually, X will return to the root of the tree and require the value of F1. If the load is unbalanced, it may be the case that F1 is not ready. Rather than block itself, X takes the first future in Y's work queue that is descended from F1 (F3) and starts executing it. The result is shown in Figure 3(e).

The motivation for leapfrogging is that X cannot proceed until Y has completed evaluating F1, so X is indirectly helping itself if it evaluates F3. The assumption implicit in this statement is that F1 depends on the value of F3; in other words, that a future does not create other futures unless it is going to use their values. This is certainly true for recursive, AND-parallel computations. For other types of computations, this assumption may be violated.[3]

Of course, it is perfectly plausible that after X leapfrogs Y but before X finishes F3, Y might require the value of F3. In this case, if X has created yet another future (F4) then Y can leapfrog over X, leading to the situation depicted in Figure 3(f). This can continue for the entire depth of the tree, if necessary; by now it should be apparent why we chose the term "leapfrog" to describe this mechanism.

The action of leapfrogging prevents workers from idling when there is work to do. The difference between stealing and leapfrogging is that the former is done by an idle worker (with an empty stack), whereas the latter is done by a worker that is trying to resolve a future (whose stack is in some intermediate stage of a computation). Note, however, that all of the futures on a worker's stack always lie on a single path from the root of the computation tree to a leaf, although there may be any number of "gaps" along the way. This implies that no worker's stack will ever exceed the maximum stack size that a sequential execution of the program might encounter.[4]

It might seem to be more beneficial in the long run to allow worker X to choose a future as high up in the computation tree as possible, because it would likely have a larger granularity. However, this would violate the conditions necessary to ensure the bounded stack property.

2.5 Comparison with Lazy Task Creation

The creating of a future in our system is similar to Mohr et al. idea of creating a continuation because our futures are just passive objects on a work queue, and if no other processor steals the futures the worker that created them executes them. The main differences are that in our system, a newly created future is saved for later execution and the worker that created the future continues executing the current function (the continuation). In Mul-T's lazy task creation approach the future is executed immediately and the continuation is saved on the task's stack.

Although the costs of creating a future in our system and of creating a continuation in Mul-T appear to be comparable, the cost of *stealing* a continuation is much higher than the cost of stealing

[3] For example, this assumption will not hold for OR-parallel computations, e.g., certain search algorithms. We do not consider OR-parallelism in this paper.

[4] We will prove this statement more carefully after we have discussed some details of our implementation in Section 3.

a future because of the stack manipulations that are necessary to accomplish the former. On the other hand, if the computation tree is balanced then both methods will tend to provide BUSD execution, so that stealing (of continuations or futures) will be a relatively rare event.

Mul-T may have an advantage over our system in the case of a balanced computation tree. Consider Figures 3(a) and 3(b) once again, and suppose that the amounts of computation in F1 and K1 are approximately the same. In our system, since worker X will get a "head start" on worker Y, it is quite likely that X will finish K1 before Y finishes F1, causing X to leapfrog over Y, or to block if there is no available future in Y's work queue.

In Mul-T, the roles of X and Y in the diagrams would be reversed, with X executing F1 and Y executing K1. Assuming that X again finishes before Y, X does not need to wait for Y, since Y has stolen the continuation. In other words, it will be Y, and not X, that must eventually return to the next higher level in the tree. We note, however, that the cost of leapfrogging is small (only slightly larger than the cost of stealing a future), so in the case that there is an available future in Y's work queue, our system is really not at a disadvantage relative to Mul-T.

On the other hand, if the computation tree were unbalanced it might transpire that Y would finish before X. In our system, this would free Y to look for work elsewhere. But the Mul-T strategy in this case is to have Y steal a continuation *from itself*. However, we cannot envision a natural scenario in which there would be any continuations in Y's continuation list under such circumstances. It is not clear, then, whether or not Mul-T would have any choice but to block Y.

In some sense, the two scenarios described above are the dual of each other, and no compelling reason has been demonstrated for preferring either system over the other.

Performance considerations aside, the main strength of Mul-T's continuation-based approach lies in its freedom from deadlock. In section 3.3 we prove that, under very weak (and entirely reasonable) assumptions about the behavior of the computation, our technique also is deadlock-free.

All in all, then, the two systems have comparable functionality and (probably) comparable performance. But we maintain that our approach is much easier to implement and, as a corollary, is more portable.

3 Implementation Details

3.1 Syntactic Issues

Our entire implementation of futures is encapsulated in a single `C++` class called, not surprisingly, `Future`. The syntax is not very elegant and could certainly benefit from a simple preprocessor; however, in its current form it is not difficult to use, and syntactic issues are not the main focus of this research.

Figure 4 shows a `C++` version of the `sum_tree` algorithm. The `Future` class allows the programmer to re-write this algorithm as shown in Figure 5.[5] Each time `psum_tree` is executed, a future is created for the right subtree while the current task evaluates the left subtree inline. Thus the evaluation of both subtrees can proceed in parallel.

The interface to the `Future` class is shown in Figure 6.

[5] In this and all subsequent examples, `function_t` is presumed to have been previously defined as the `C++` type signature of the function being futurized.

```
int sum_tree(NODE *tree) {
   if (!tree->left) {
      return(tree->value);
   } else {
      return (sum_tree(tree->left) + sum_tree(tree->right));
   }
}
```

Figure 4: C++ `sum_tree` sequential code.

```
int psum_tree(NODE *tree) {
   if (!tree->left) {
      return(tree->value);
   } else {
      Future right((function_t) psum_tree, tree->right);
      return (psum_tree(tree->left) + (int) (void *) right);
   }
}
```

Figure 5: C++ `sum_tree` parallel code.

To begin a futurized parallel computation, the program issues a call to `Future::begin()`, in which it specifies the number of workers to use and, optionally, a limit on the work queue sizes. `Future::begin()` creates only `numworkers-1` *additional* threads of execution, because the thread doing the call is considered to be worker 0. Thus, `numworkers` should be less than or equal to the number of processors available to the computation.

After calling `begin()` and creating one or more futures, the main thread simply continues executing. If and when it references some unresolved future it will automatically start executing as a worker until that future is resolved. On the other hand, if it never references an unresolved future then it does not wait for the computation to finish. Either way, there will be exactly `numworkers` threads in execution.

To terminate the other workers, the main thread calls `Future::end`.

A `Future` can be created and bound to a computation in a single step by the constructor functions. Alternatively, creation and binding can be decoupled: class `Future` provides a constructor that takes no arguments to create an unbound `Future`, and overloaded function-call operators to bind such a future to a computation. There is also an overloaded assignment operator to bind a value directly to a `Future`.[6] In the `psum_tree` example, the future for the right subtree is created and bound by the constructor.

Note that one of the constructors and one of the function-call operators take an additional parameter, `pid`. This can be used to specify the integer index (in the range `0...numworkers`) of the

[6] We will give examples of the use of these features in Section 4.

```
class Future {
  public:   // Called directly by application program

    // Start up and shut down a parallel computation
    static int begin(int numworkers, int queue_limit);
    static void end();

    // Constructors:
    // Create an unbound Future
    Future();
    // Create and bind a computation to this Future
    Future(function_t func, void *args);
    // As above, but specify which queue to insert in
    Future(function_t func, void *args, int id);

    // Bind a computation to an unbound Future
    void operator() (function_t func, void *args);
    void operator() (function_t func, void *args, int id);
    // Bind a value to an unbound Future
    Future &operator=(void *v);

    // Resolve the future and return a value
    // (block if necessary)
    inline operator void*();

  protected:    // For use by derived classes

    // the value of the future
    void *value;
    // Evaluate func(args), store result internally
    virtual void evaluate();

    // Place future on a workq
    void init_future();
    void init_future(int id);
    // Resolve but don't return anything
    void resolve();
};
```

Figure 6: Interface to the `Future` class.

worker in whose work queue this future should be inserted. This gives the programmer the *option* of providing load-balancing help to the runtime system.

A future is resolved by type-coercing it to the type of the return value. The overloaded type-coercion operator implements the necessary synchronization. The default future class provides only a single type-coercion operator, `(void*)`, which represents any type that can fit into a pointer. This type can in turn be successfully coerced into a pointer to any type or into any basic subtype except a double-precision floating point number.

Programs that are to be futurized will not always pass in a single four byte parameter, nor return a four-byte result. One way to accommodate this would be to package all parameters into a structure and pass a pointer to this structure as the single parameter to the Future. Likewise, instead of returning a large piece of data, return a pointer to the data. Unfortunately, this requires a redeclaration of the function that is to be futurized.

A more convenient way to handle such cases is to add a subclass of `Future`. Creating a subclass allows one to pass in as many parameters as desired to the futurized function, and to return a result of arbitrary type. This means the declaration of the futurized function can remain unchanged.

As a concrete example, suppose the leaves of the tree in the `sum_tree` example contained double-precision floating point numbers; then clearly we would want `sum_tree` to return a value of that type. Figure 7 shows a subclass of future that futurizes the hypothetical double-precision `sum_tree` program. Note that we have hard-coded the function to be called into the constructor for the subclass; we could have easily created a more generic subclass by passing the function as a parameter to the constructor.

A subclass of Future should contain the following elements:

- Local storage for the parameters to the futurized function and its return value (if any).
- A constructor or function-call operator for initialization. This member should accept the same parameters as does the function being futurized. The routine then assigns the passed in parameters to the mirror image instance variables. Finally the routine calls `Future::init_future()` in order to insert the future into the work queue. (The optional integer argument can be used to specify *which* work queue.)
- An `evaluate()` member function. `evaluate()` is a virtual member function that is called by a worker to compute the value of a `Future`. Thus, the redefined `evaluate` routine simply calls the futurized function with the instance variables as parameters, and saves the return value (if any) in another instance variable.
- Zero or more type coercion operators. Each coercion operator should call `Future::resolve()`, which performs all necessary synchronization. Once resolved, the future has been computed, and the type coercion operator returns the instance variable that is used to store the return value.

  Caveat: If the derived class stores the return value in `Future::value`, then coercion operators are optional. However, if the derived class uses some derived class instance variable to hold the return value (as in our example), then the derived class *must* provide at least one type coercion operator, because the value returned by `Future::operator void*()` will be undefined.

Thus, futurizing a `C++` program is not as elegant as futurizing a Mul-T or Qlisp program, but it is by no means difficult.

```
double psum_tree(NODE*);

class Fsum_tree : public Future {
    NODE *tree;
    double val;

  public:
    Fsum_tree(NODE *t) {
      tree = t;
      init_future();
    };
    void evaluate() {
      val = sum_tree(tree);
    };
    operator double() {
        resolve();
        return val;
    }
};

double psum_tree(NODE *tree) {
   if (!tree->left) {
      return(tree->value);
   } else {
      Fsum_tree right(tree->right);
      return (sum_tree(tree->left) + (double)right);
   }
}
```

Figure 7: Subclass of Future to parallelize `sum_tree`.

One limitation of our implementation is that passing a `Future` to a function in which the corresponding formal parameter is not of type `Future` will cause the compiler to invoke a type coercion operator, which will force resolution of the future before the call occurs. Lisp-based languages do not have this problem because data values are dynamically typed. To get around this problem it is necessary to change the type of the formal parameter to `Future` (or the appropriate subclass). The function could still be called with a non-`Future` actual parameter because `Future::operator=()` would provide an implicit type conversion from the non-`Future` to the `Future`.

3.2 Synchronization

A future can be in one of four states: unbound (UB), not executing (NE), executing (E), or finished (F). The meanings of these states are:

UB — The future has no computation bound to it. This is the state of a future between its creation (using the parameterless constructor) and the use of one of the function call operators.

NE — The future has an associated computation, but no worker is currently executing it.

E — The future is being evaluated by some worker, but the value is not ready.

F — The future has been computed.

The progression through the states is either UB → NE → E → F, or NE → E → F, depending on which constructor was used to create the `Future`.

Every future contains the private fields `workerid` and `creatorid`. `Creatorid` is set to the integer index of the worker in whose work queue the future was placed when it was bound to a computation. This is usually the id of the worker doing the binding, but may be some other worker if the optional parameter `pid` is specified (in either the constructor or the function-call operator). This field is meaningless if the state of the future is UB.

`Workerid` is set to the integer index of the worker that is evaluating the future, if any. This field is meaningless if the state of the future is UB or NE.

Every future also contains a `depth` field indicating its depth in the computation graph, and each worker keeps track of a value `current_depth`, representing that worker's current depth in the graph.[7] (The depth of the main routine is by definition 0.) Each time a worker creates a new future, it sets the depth of that future to `current_depth` + 1.

When a worker executing a future `f` tries to resolve a future `q`, the action taken depends on the state of `q`:

- If `q.state` is UB, the worker spin-waits for a change in state, and then proceeds to the appropriate case.

[7] This assumes either that a futurized function is never statically inlined (i.e., called directly) by the programmer, or that the implementation has some way of tracking such calls. Our implementation does not do this at the present time. A general solution to this problem is to have the programmer encapsulate the futurized function in a subclass of a special base class, and to overload the *member selection operator* for that base class. This would enable the runtime system to track every call to the function, whether inlined or futurized. We have chosen not to pursue this at present because the version of C++ that we are using (Gnu 1.40.0) does not implement member selection in accordance with the current C++ standard [4].

- If `q.state` is NE, the worker finds the work queue in which `q` resides using the `q.creatorid` field, and locks that work queue. If `q.state` is still NE, the worker dequeues `q`, sets `q.workerid` to its own id, sets `q.depth` to max(`q.depth, current_depth+1`), sets `q.state` to E, and unlocks the work queue. If not, it proceeds to the appropriate case.
- If `q.state` is E, the worker attempts to leapfrog worker `q.workerid`. The worker is allowed to leapfrog if it can find some future `g` in worker `q.workerid`'s work queue with a depth that is *strictly greater* than `max(f.depth, q.depth)`. If so, it attempts to dequeue `g` and evaluate it. It continues this behavior until `q.state` changes to F.
- If `q.state` is F, the worker obtains the value.

The constraint `g.depth` > `max(f.depth, q.depth)` in the leapfrogging case is called the *leapfrog depth rule*. We emphasize this because it is a necessary condition to guarantee freedom from deadlock and bounded stack sizes. We prove these properties formally in the next section.

There is one work queue for each worker and each work queue has its own spinlock. There are no locks on the futures themselves; all state changes take place while one of the work queue locks is being held. Although this raises the utilization of the work locks, it reduces the number of locks that must be obtained in the no-contention case, which is the common case. Furthermore, it obviates having to create and initialize a new spinlock each time a `Future` is created.

There is no blocking synchronization anywhere in the implementation (unless the program is being run on a uniprocessor). When it becomes necessary to wait for a state change, a worker spin-waits on `f.state`. This is quite efficient on a multiprocessor with coherent caches.

When `f.state` is E, we may run into a problem: if there is no future with a large enough depth field in the creator worker's work queue, then the resolving worker is blocked. This would be a problem if the granularity of the future that the creator worker is evaluating is larger than normal *and* this future creates no additional futures for a long time. For the programs that we have looked at this has not been the case, because non-leaf futures tend to create other futures on a regular basis, and leaf futures tend to be small.

3.3 Deadlock freedom and stack size bounds

Given two very simple (and reasonable) assumptions about program behavior, we can guarantee the following desirable properties about our implementation:

P1: The computation will be deadlock-free.

P2: The maximum stack size (measured in stack frames) of any worker will be at most a constant factor times the maximum depth of any node in the computation graph.

The reason for the "constant factor" disclaimer in P2 is that, while the maximum number of activations of a futurized function on a worker's stack will never exceed the maximum depth of the computation graph, there may be an extra stack frame or two *per activation* because of calls to `Future::operator<type>` and `Future::evaluate`.

Before proceeding, we define the following notation: `f1`$\leadsto$`f2` means that future `f1` binds a computation to future `f2`;[8] `f1`$\rightarrow$`f2` means that future `f1` depends on the value of future `f2`; and $\overset{*}{\leadsto}$ and $\overset{*}{\rightarrow}$ are the transitive closures of $\leadsto$ and $\rightarrow$, respectively.

[8] More precisely, `f1`$\leadsto$`f2` means that `f1.workerid = f2.creatorid`. This distinction is subtle but important, because our implementation allows one worker to insert futures into the work queues of other workers (presumably

The assumptions that allow us to guarantee P1 and P2 are:

A1: The precedence graph for the computation is acyclic.

A2: If `f1`$\rightsquigarrow$`f2` then `f1`$\rightarrow$`f2`.

The necessity of A1 is obvious; however A2 is more subtle. A2 implies that a future does not create "dangling" futures, i.e., futures that are still unevaluated when their creator returns. This assumption is certainly true of all recursive computations, and many non-recursive computations as well. A2 is crucial to the proof of Lemma 2, below.

Before proceeding, we formally state the following, somewhat obvious result:

Lemma 1 *If* `f` *and* `g` *are two futures on the same worker's stack, and if* `f` *is below* `g` *on that stack, then* `f.depth` $<$ `g.depth`.

Proof Assume `f` is at the top of the stack. There are only two ways to add the future `g` to the stack: either by attempting to resolve `g` while it is in state NE, or by attempting to resolve some other future `h` in state E, and thereby end up leapfrogging onto `g`. If `g` is stolen, `g.depth` is set to max(`g.depth, f.depth+1`) $>$ `f.depth`.[9] If `g` is executed via leapfrogging, then by the leapfrog depth rule `g.depth` $>$ `f.depth`. Now by induction, `g.depth` is also greater than the depth of **any** future that is below it on the stack.

Lemma 2 *If a worker* W *currently has a future* `f` *on the top of its stack,* `f.depth` $= d$*, and if* `g` *is some future in* W*'s work queue, then*

- *if* `f`$\overset{*}{\rightsquigarrow}$`g` *then* `g.depth` $\leq d+1$.
- *otherwise,* `g.depth` $\leq d$.

Proof Assume that `f`$\overset{*}{\rightsquigarrow}$`g` and `g.depth` $> d+1$. Since `f.depth` $= d$, there must be some other future `h` $\neq$ `f` such that `f`$\overset{*}{\rightsquigarrow}$`h`$\rightsquigarrow$`g`. Now since `g` is still in the work queue then `g` cannot yet have begun execution; hence by A2 `h` cannot have completed execution. Also, since `f`$\overset{*}{\rightsquigarrow}$`h`, `h` must be above `f` on W's stack. This contradicts the assumption that `f` is at the top of W's stack.

Now assume that it is not the case that `f`$\overset{*}{\rightsquigarrow}$`g`. Let `e` be the parent of `g`, i.e., `e`$\rightsquigarrow$`g`. By A2, `e` has not yet completed execution, hence `e` is below `f` on W's stack. But then Lemma 1 implies that `e.depth` $<$ `f.depth` $= d$. Therefore, `g.depth` $=$ `e.depth`$+1 \leq d$.

Lemma 2 assures us that the only futures that can be in worker W's work queue with depth greater than the currently executing future are descendants of the currently executing future. This allows us to prove:

---

for load balancing purposes). This capability, while a practical benefit, allows the construction of virtually any pathological scenario by a "devil's advocate." By ignoring this capability in the following discussion, we simplify the proofs considerably.

[9] As noted in footnote 7 on page 13, this may not be true (in our current implementation) if the programmer statically inlines calls to the futurized function. However, this statement is still true (vacuously) for recursive computations, since in that case futures are stolen only by workers with empty stacks. For non-recursive computations with dependences, we require that the programmer refrain from static inlining, but this is not a great imposition. We re-iterate that we could solve this problem by overloading the `C++` *member selection operator*, but we have chosen not to do so for portability reasons.

Lemma 3 *If* `f` *and* `g` *are two futures on the same worker's stack, and if* `f` *is below* `g` *on that stack, then* $\texttt{f} \xrightarrow{*} \texttt{g}$.

Proof Assume `f` is at the top of the stack. There are only two ways to add the future `g` to the stack: either by attempting to resolve `g` while it is in state NE, or by attempting to resolve some other future `h` in state E, and thereby end up leapfrogging onto `g`. Stealing can happen only if $\texttt{f} \rightarrow \texttt{g}$, therefore we need not consider that case any further.

Assume, then, that `g` is brought onto the stack via a leapfrog. Let `g.creatorid` = X, and let `h` be the future in X's stack over which `f` leapfrogged, i.e., `h.workerid` = X and $\texttt{f} \rightarrow \texttt{h}$. We need to prove that this implies that $\texttt{f} \rightarrow \texttt{g}$. If $\texttt{h} \xrightarrow{*} \texttt{g}$ we are done, so assume that this is not the case. The proof is *by induction on the total number of leapfrogs that have occurred in the entire computation.*

Suppose that no leapfrogs have yet occurred. Let $\texttt{k} \rightsquigarrow \texttt{g}$, which implies $\texttt{k} \rightarrow \texttt{g}$. Since no leapfrogs have yet occurred, then if `h` is below `k` on the stack then $\texttt{h} \rightarrow \texttt{k}$. But that would imply $\texttt{h} \rightarrow \texttt{k} \rightarrow \texttt{g}$, which we assumed was not the case. Therefore, `h` must be above `k` in the stack, and thus, since `h` is still in execution, `h` could not have begun execution until after `g` was placed in X's work queue. At the time that `h` begins execution, Lemma 2 implies that `g.depth` $\leq$ `h.depth`. This means that `f` could not have leapfrogged over `h` onto `g`. This is a contradiction, thus $\texttt{h} \xrightarrow{*} \texttt{g}$, which implies $\texttt{f} \xrightarrow{*} \texttt{g}$.

Our inductive hypothesis is that after n leapfrogs have taken place, the statement of the lemma is still true. Now consider the $n+1$-st leapfrog. Let `k` be as in the previous case. By the inductive hypothesis, `h` cannot be below `k` on the stack, which leads to the same contradiction as before. Therefore $\texttt{h} \xrightarrow{*} \texttt{g}$, which implies $\texttt{f} \xrightarrow{*} \texttt{g}$. By induction, $\texttt{f} \xrightarrow{*} \texttt{g}$ for all n.

Now by induction on the distance between `f` and `g` in the stack, if `f` is anywhere below `g` on the stack, then $\texttt{f} \xrightarrow{*} \texttt{g}$.

Theorem 1 *P1 is true.*

Proof Assume that a deadlock occurs. Then there must be a cycle of *workers* $\text{W} \rightarrow \text{X} \rightarrow \text{Y} \rightarrow \ldots \rightarrow \text{W}$. Let the futures `w`, `x`, `y`, ...be the futures that are at the top of the stacks of `W`, `X`, `Y`, ..., respectively. There must be other futures $\texttt{x}'$, $\texttt{y}'$, ..., $\texttt{w}'$ in each of the respective stacks such that $\texttt{w} \rightarrow \texttt{x}'$, $\texttt{x} \rightarrow \texttt{y}'$, ... By Lemma 3, $\texttt{w}' \xrightarrow{*} \texttt{w}$, $\texttt{x}' \xrightarrow{*} \texttt{x}$, ..., and therefore $\texttt{w} \rightarrow \texttt{x}' \xrightarrow{*} \texttt{x} \rightarrow \texttt{y}' \xrightarrow{*} \texttt{y} \xrightarrow{*} \texttt{w}' \xrightarrow{*} \texttt{w}$. Thus there must be a cycle in the computation graph, which contradicts A1.

Theorem 2 *P2 is true.*

Proof Note that the stack of a worker can never become larger than it would in the sequential case unless that worker leapfrogs. However, the leapfrog depth rule guarantees that a worker can leapfrog only onto a future whose depth is strictly greater than the depth of the worker's current future. Thus, the maximum number of futurized function calls on the stack is at most the maximum depth of the entire computation graph, and P2 has been shown.[10]

Lest the reader think these proofs are mere exercises in vacuous reasoning, we point out that if the implementation did not enforce the leapfrog depth rule then P1 and P2 could be violated in perfectly reasonable computations.[11]

[10] Once again, the proof fails if the programmer statically inlines some function calls. However, we have yet to encounter any examples in which leapfrogging causes stacks to grow excessively large.

[11] When we say "perfectly reasonable computations", we mean that we actually observed each of these behaviors in one of our benchmarks prior to the implementation of the leapfrog depth rule. These benchmarks, described in Section 4, were MVA (P1) and Queens (P2).

As an example of a violation of P1, consider an iterative computation with a loop dependence. Suppose that the main worker creates one future per iteration, in iteration order. Call these futures `f1, f2, ..., fmax`. Note that `f1`$\rightarrow$`f2`$\stackrel{*}{\rightarrow}$`fmax`. For simplicity, we assume that max=3 and that there are only two workers. Then the following is a deadlock scenario:

1. Worker0 begins executing `f1`.
2. Worker1 steals `f2` and begins executing it.
3. `f2` requires the value of `f1`. Since `f1` is not yet ready and since `f3` is in Worker0's work queue, Worker1 leapfrogs and begins executing `f3`.
4. `f3` requires the value of `f2`. Unfortunately, `f2` cannot finish executing because `f3` is on top of it on Worker1's stack. Therefore, the computation is deadlocked.

We call the dilemma in which Worker1 finds itself a *stack dependence*. Clearly, the leapfrog depth rule makes stack dependences impossible.

Note that Lazy Task Creation [9] would solve the analogous problem by allowing Worker1 to dissect its own stack in order to extract the continuation for `f2`. Thus LTC takes a *detection and recovery* approach to the deadlock problem, whereas our system practices deadlock *avoidance*.

As every operating systems student knows, there is a third solution to the deadlock problem, namely *prevention*. Stack dependence deadlocks like the one in the example could be prevented by creating futures in a way such that there were no backward dependences in the work queue. The problem with doing so is that it turns an iterative computation into a recursive computation, creating very large stacks that take up an enormous amount of memory and take a long time to unwind. It might also be impossible to create the futures that way because one might not know all dependences ahead of time.

As an example of a violation of P2, consider a recursive program in which each recursive call is assigned to a separate future, and in which some recursive call creates more than one future. Again, for simplicity we assume that the number futures created by that call is two, and that the number of workers is two. The the following scenario violates P2:

1. Worker0, executing f0, creates futures `f1a` and `f1b`.
2. Worker1 steals `f1a` and begins executing it.
3. `f1a` (Worker1) creates future `f2a` and `f2b`.
4. f0 (Worker0) requires the value of `f1a` (Worker1). Since `f1a` is not yet ready and since `f2a` is in Worker1's work queue, Worker0 leapfrogs and begins executing `f2a`.
5. `f1a` (Worker1) requires the value of `f2a` (Worker0). Since `f2a` is not yet ready and since `f1b` is in Worker0's work queue, Worker1 leapfrogs and begins executing `f1b`.

If the program were never to make another recursive call (create another future), then at this point the number of futurized function activations on Worker1's stack is larger than the greatest recursion depth of the program. Thus P2 is violated. Furthermore, note that this scenario could continue indefinitely if the program continued to create futures, with the workers taking turns leapfrogging each other.

In conclusion, we have shown that, under the assumptions given, the leapfrog depth rule not only is a sufficient condition for P1 and P2 to hold, but also is a necessary one.

3.4 Portability

It should be clear from the discussion in the previous subsections that our techniques are not platform- or language-dependent. In particular, we do not require knowledge of stack-frame layout.

The *syntax* requires `C++` in order to work, but an interface to runtime system could be constructed for any language.

The current implementation is based on Cthreads. We chose a Cthreads platform because it makes the code instantly portable to any other environment that support the Cthreads interface, which seems to be a major influence on the emerging Posix threads standard.

The parallel processing requirements of the software are satisfied by any system that provides a mechanism for thread creation (Cthreads, Posix threads, even heavyweight processes with shared memory such as are provided by Dynix's `m_fork()`) and locks.

The techniques are most efficient using spin-waiting locks, but the implementation can be modified easily to work with blocking locks. All lock routines are accessed indirectly through macros that are declared in a single header file. For example, on the Sequent we changed the macros to allow the program to use native Dynix spinlocks rather than the slower locks provided by our Cthreads runtime system. This took only a few minutes and resulted in a noticeable improvement in performance.

4 Experimental Results

The results presented in this section were obtained from executions of our benchmark programs on a Sequent Symmetry model B. We recorded the timings for each benchmark using 1, 2, 4, 8 and 16 processors. In doing multiple timings for each run we found that the timings were very consistent on all benchmarks.

We graph not only at the speedup of our futurized implementations relative to the sequential implementation (denoted by (S) on the graphs), but also relative to the futurized implementation running with one worker on one processor (denoted by (F) on the graphs).

4.1 A synthetic benchmark

In parallelizing future programs, granularity is a big concern. Unless the granularity of the futurized function is larger than the overhead of creating a future, efficiency will suffer. In order to test out how our futures perform with different granularities we ran a synthetic benchmark, similar to the one in [9].

The synthetic benchmark is a modification of the `psum-tree` program shown in Figure 5. The only difference is that before a leaf returns its value, a for loop is executed that delays the leaf node for a specified number of machine instructions. Inside the loop one add instruction is done, so one execution of this loop is six machine instructions (on the Sequent). The average number of machine instructions per future is actually half of this number, because the granularity loop is executed only in the leaf futures and they account for only half of the futures in the computation tree.

The granularity benchmark was run with 16 processors (and thus 16 workers). Efficiency is the sequential time divided by the product of the number of processors and the parallel time. As can be seen in Figure 8, small granularities leads to poor efficiency because of the overhead associated

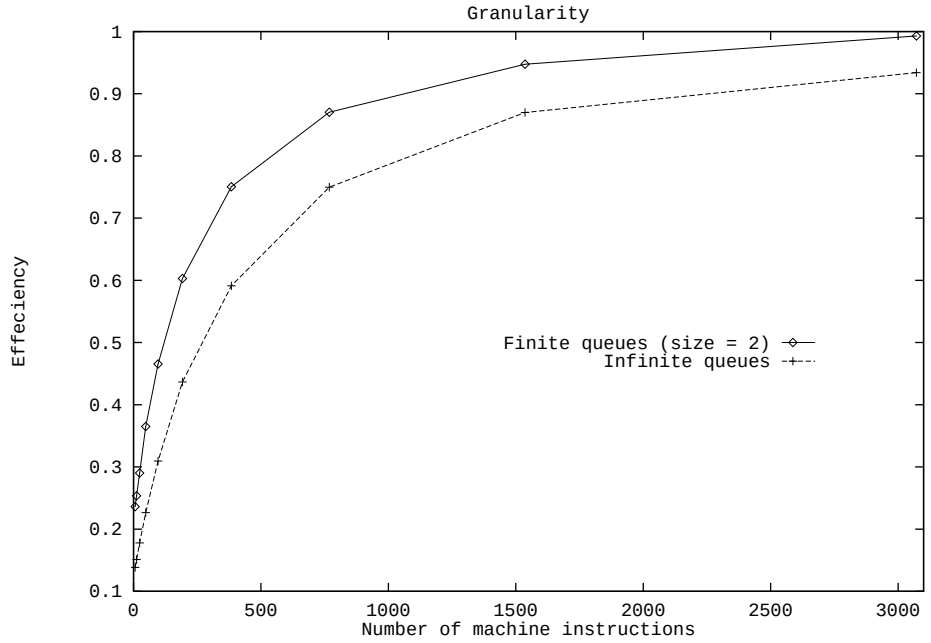

Figure 8: Efficiency vs. granularity on the synthetic benchmark.

with creating futures, although load-based inlining yields a substantial improvement. However, efficiency exceeds 90% for granularities larger than about 750 instructions.

Our performance is comparable to but not quite as good as the Encore implementation of Lazy Task Creation reported in [9]. However, this is somewhat misleading, because certain futures-related overheads, which are built into the Mul-T compiler, are present in the sequential times reported in that paper. This reportedly causes blowups of the execution times of their sequential programs in the range of 1.4 to 2.2 [9, Table 1], which tips the balance strongly in favor of our system.

4.2 Recursive computations

Futures were originally designed and used in a recursive language [8], so naturally futures are well suited to recursive computations. The type of recursive computations futures can be applied to are those with two or more recursive calls; these recursive calls must operate on disjoint data.

We have found that recursive programs can benefit from load-based inlining, using a work queue limit of around two to four. A small work queue limit is reasonable because our recursive programs create futures at a steady rate. If future creation were extremely bursty then such a small work queue limit could cause poor performance. The only way to *guarantee* good performance with load-based inlining is through experimenting with some work queue limits until one is found that best fits the application.

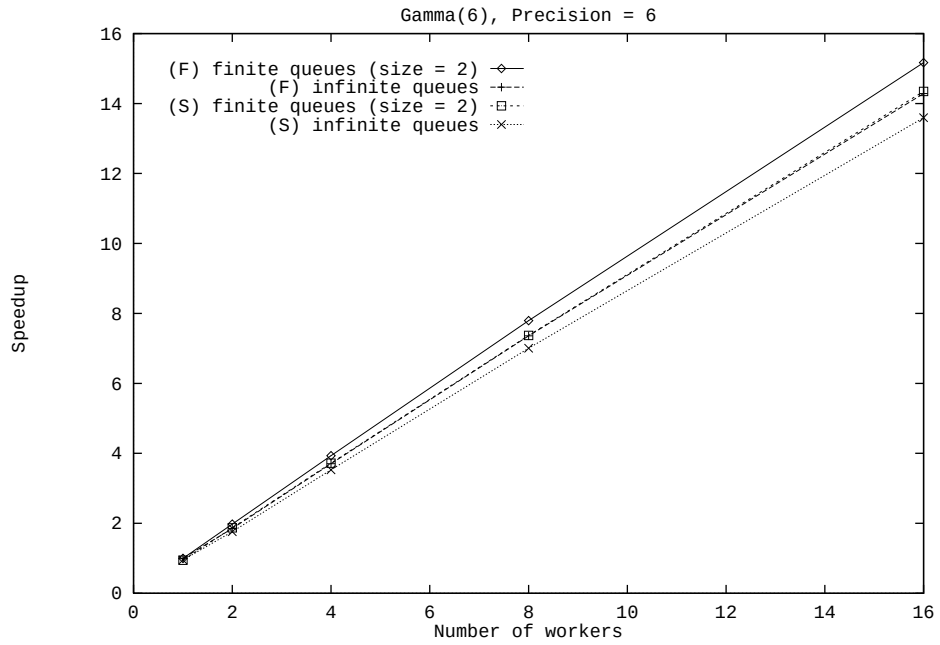

Figure 9: Speedup for Gamma.

The first non-synthetic benchmark we will discuss is Gamma, a recursive, adaptive quadrature algorithm that computes the gamma function:

$$\Gamma(n) = \int_0^\infty x^n e^{-x} dx$$

The basic subroutine in Gamma computes the area under the integrand over a given interval using trapezoidal rule. If the curvature of the integrand in the interval is above a certain threshold, the interval is bisected and recursive calls are made. Because the integrand in the gamma function has widely varying curvature, the computation tree is highly unbalanced.

An unbalanced computation tree leads to a great deal of leapfrogging. Thus, Gamma is a good test of the efficiency of this technique. Figure 9 shows that we get close to linear speedup, and we do not lose much speedup in comparison to the sequential program.

In the graph there are two different futurized executions being compared, one that uses load-based inlining and another that has infinite work queues. For the load-based inlining we fixed the work queue sizes at two, a number we arrived at through trial and error. The execution with finite queues does outperform the execution with infinite queues, but the difference is small enough that we conclude that load-based inlining is not vital for this benchmark.

Our next benchmark, Queens(N), counts all the possible solutions to the problem of placing N queens on an NxN chess board without any queen attacking another. The algorithm starts on the first row and goes through the row trying to find squares on which a queen may be placed. If and when it finds such a square, a queen is placed there and a recursive call is made for the next row. This program creates some interesting choices for the programmer when he or she futurizes it.

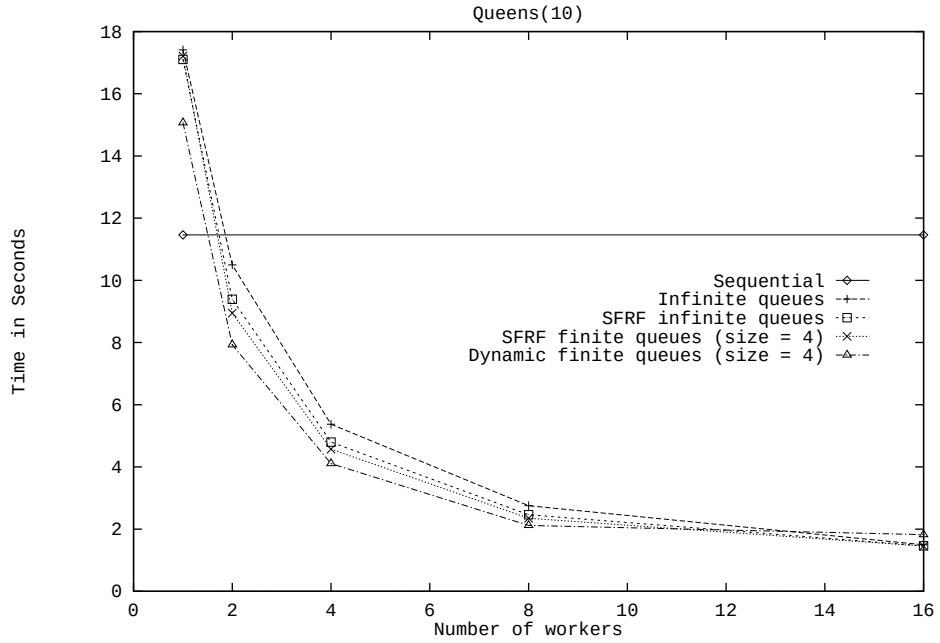

Figure 10: Elapsed time for Queens(10).

First of all, the programmer does not know in advance how many futures a given recursive call might create. So the programmer has a couple of options:

- Declare an array of futures upon function entry to hold the maximum number of futures that can possibly be produced.
- Allocate futures dynamically as they are needed.

The problem with the first approach is that, even though there is no dynamic allocation going on, the compiler has to call the `Future::Future()` constructor for every element of the array.

The second approach will not scale well if the programmer uses the built-in `new` and `delete` operations for free store management, since all of the workers would be spending a lot of time contending for the centralized lock that protects the free store. We can avoid this problem by overloading the new and delete operators for class `Future` to maintain a separate linked list of free futures for each worker. This scales very well in comparison to using the built in allocation routines.

The curve labeled "Dynamic" in Figure 10 uses our overloaded `new` and `delete` operators. All of the other curves in the graph take the approach of having the compiler allocate all of the futures (whether needed or not) at function entry. The dynamic method outperforms the others up to about eight processors, and then falls behind slightly at sixteen.

Another choice faced by the programmer is whether to create a future for every recursive call, or for only some of them. Since we want to get as much parallelism as possible it makes sense to make each recursive call a future. This choice is the "Infinite queue" curve in Figure 10. The

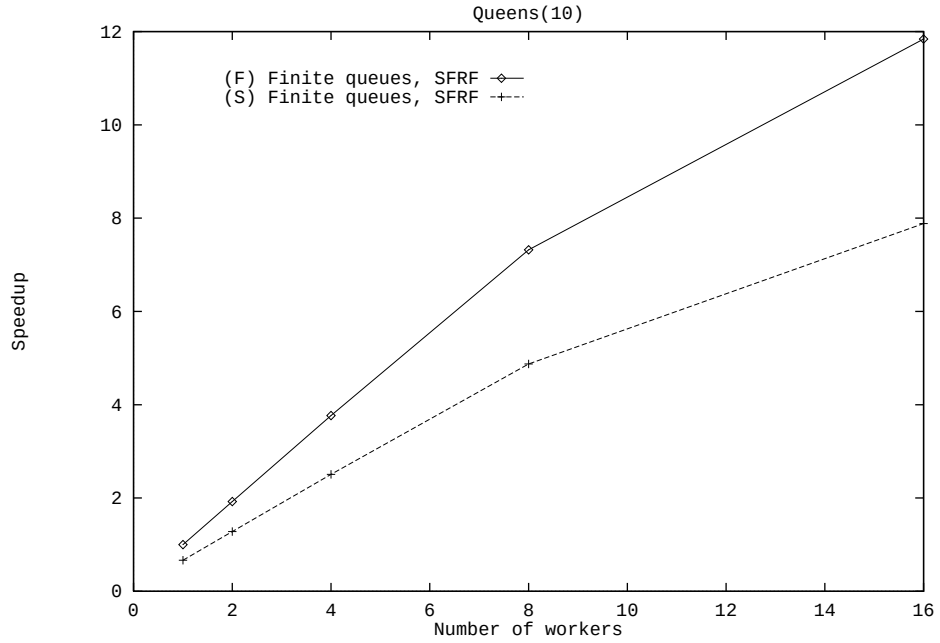

Figure 11: Speedup for Queens(10).

problem with this approach is that all of the futures created may be stolen by other workers before the current worker gets a chance to evaluate any of them. In such a scenario, the worker will almost certainly either block or leapfrog when it tries to sum up the return values from the futures.

A better approach is what we call *save first recursive future* (SFRF). Using SFRF, when the program encounters the first recursive call it saves the parameters and sets a flag rather than creating a future. All subsequent recursive calls for that row are turned into futures. Once the entire row has been evaluated for possible queen placings, the flag is checked and if it is set the recursive call is done. This approach, with finite and infinite queues, is marked on the graph with "SFRF."

Figure 11 shows the speedup of the queens program using SFRF with a queue size limit of four, and allocating an array of futures at the beginning of each recursive call. The futurized code when compared with itself (F) gets almost linear speedup up to eight processors. A possible reason for the smaller slope between eight and sixteen processors is that the granularity of the futures varies from $O(n)$ to $O(n^2)$. Thus, during execution there might be times where there is not enough work to keep all 16 processors busy.

In comparing the (F) speedup curve to the (S) speedup curve one notices a significant difference. The running time of the futurized program on one processor is significantly larger than the running time of the sequential program. Most of the difference comes from two sources. First, the futurized function is allocating an (overly large) array of futures at the start of each call, and second, the futurized function must resolve the $O(n)$ futures it created before returning. The latter is necessary in order to correctly total up the number of successful solutions to the problem.

```
void matrix_mult(double **InA, double **InB,
                 double **C, int N) {
   int i, j, k;
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         C[i][j] = 0.0;
         for (k=0;k<N;k++) {
            C[i][j] += InA[i][k] * InB[k][j];
         }
      }
   }
}
```

Figure 12: Sequential Matrix Multiply.

4.3 Iterative computations without dependences

Futures can be used to handle iterative computations as well as recursive ones. The amount of work to convert from the sequential program to the futurized program is about the same as in the recursive case. Iterative computations can be broken down into two cases: Those computations without any dependences and those that have dependences. The former are the topic of this subsection, and the latter are the topic of the next subsection.

Iterations without any dependences can easily be converted to use futures. A simple example of such a program is matrix multiplication. Figure 12 shows a sequential version of matrix multiply.

This type of iterative program can be futurized by encapsulating the body of the iterative algorithm into a function and changing one or more of the index variables into input parameter(s). For example, one can easily create futures that calculate one element of the result matrix at a time, as shown in Figure 13.

However, in doing this division of work one needs to take into account the granularity of the resulting futures in order to get optimal speedup. Unless the granularity of each future is larger than the cost of creating it, the program's performance will suffer. As it turns out, calculating each element of the result matrix as a future is too small a granularity. Better results are obtained by creating futures that calculate an entire row of the result matrix each.

Once the program is converted to Future form, there are several ways the main worker (executing the main program) can distribute the work. The main worker can:

- Create one future per iteration and append to its own work queue
- As above, but deal out the futures to the workers in a cyclic fashion
- Create only one future per worker with the work divided evenly

Putting all the futures into the main worker's work queue is the worst of the three approaches, because all of the workers will be competing constantly for access to that queue when looking for

```
void
matrix_mult_elem(double **InA, double **InB,
            double **C, int i, int j, int N) {
   int k;

   C[i][j] = 0.0;
   for (k=0;k<N;k++)
      C[i][j] += InA[i][k] * InB[k][j];
}

class Fmatrix_mult : public Future {
public:
   double **InA, **InB, **InC;
   int i, j, n;

   void
   operator() (double **A, double **B, double **C,
               int I, int J, int N)
   {
      InA = A; InB = B; n = N; InC = C; i = I; j = J;
      init_future();
   };

   void evaluate() {
      matrix_mult_elem(InA,InB,InC,i,j,n);
   };
};
```

Figure 13: Future subclass for calculating a single element of the result matrix in a NxN matrix multiply computation.

```
main() {
   Fmatrix_elem FM[N][N];
   for (i=0;i < N; i++) {
      for (j=0; j<N; j++) {
         count++;
         FM[i][j](A,B,C,i,j,N,(count%numworkers)+1);
      }
   }
}
```

Figure 14: Dealing out the matrix multiply futures.

work to do. This contention also slows down the main worker, who is trying to insert more work into its work queue.

We provide the option of specifying the work queue to which a newly created future is to be appended. This allows the main routine to *deal out* the futures, as shown in Figure 14. Note that `Fmatrix_mult::operator()`, not shown, is modified to take an extra integer parameter, which it passes to `init_future()`.

Dealing is no more difficult than not dealing and is always preferable. Dealing allows the main worker to distribute the futures evenly to the other workers' work queues as the futures are created. This will nearly eliminate work queue contention. A drawback to this approach arises when the workers consume the futures faster than the main worker can distribute them. This will result in extra time spent by the workers searching the work queues for work, and lower processor utilization.

The best approach of all is to divide the work evenly into exactly `numworkers` futures, and to deal one of these futures to each worker. This technique, which we call *block dealing*, is quite easy to do when there are no dependences. We change `Fmatrix_mult::evaluate()` to compute each iteration that a particular instance of `Fmatrix_mult` is responsible for computing. This approach is optimal because the work is distributed evenly among all the workers using the minimum possible number of futures.

For our benchmark we multiplied two 128x128 matrices having double-precision floating point elements. We first tried having each future compute one element of the result matrix. This can be seen in Figure 15 as the line labeled "matrix-elem." We then tried a larger granularity be having each future compute an entire row of the result matrix. This is labeled "matrix-row." Clearly, the larger granularity of matrix-row achieves better performance than matrix-elem.

Curiously, we found that we were getting superlinear speedup when comparing either parallel algorithm against the sequential algorithm. The reason for this anomaly appeared to be more cache misses and register spills in the sequential execution than in the parallel executions. Re-running the computations using manifest constants in place of array accesses eliminated the superlinear speedup, confirming our hypothesis.

In order to level the playing field, so to speak, we tiled [12] the sequential program (Figure 16) and the matrix-row program (Figure 17). The tiled programs were run using a value of 16 for `Gran`. In Figure 15, the curves labeled (T) represent the tiled versions of the programs. Tiling ended up giving us the near-exact linear speedup that we expected, as can be seen in Figure 18.

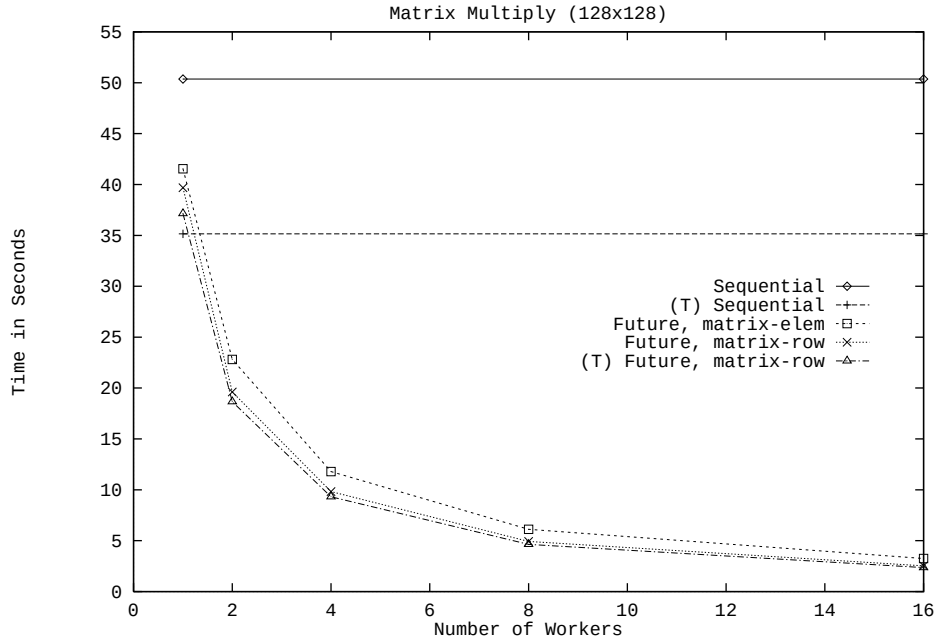

Figure 15: Elapsed time of matrix multiply.

```
void matrix_mult(double **InA, double **InB, double **C, int N) {
  int i, j, k, I,J,K;
  for(I = 0; I < N; I += Gran)
    for (J = 0; J < N; J += Gran)
      for (K = 0; K < N; K += Gran)
        for (i = I; i < I + Gran; i++)
          for (j = J; j < J + Gran; j++)
            for (k = K; k < K + Gran; k++)
              C[i][j] = C[i][j] + InA[i][k] * InB[k][j];
}
```

Figure 16: Tiled sequential matrix multiply.

```
void matrix_mult(double **InA, double **InB, double **C, int i, int N) {
  int j, k, J, K;
  for (J = 0; J < N; J += Gran)
    for (K = 0; K < N; K += Gran)
      for (j = J; j < J + Gran; j++)
        for (k = K; k < K + Gran; k++)
          C[i][j] = C[i][j] + InA[i][k] * InB[k][j];
 }
```

Figure 17: Tiled parallel matrix-row.

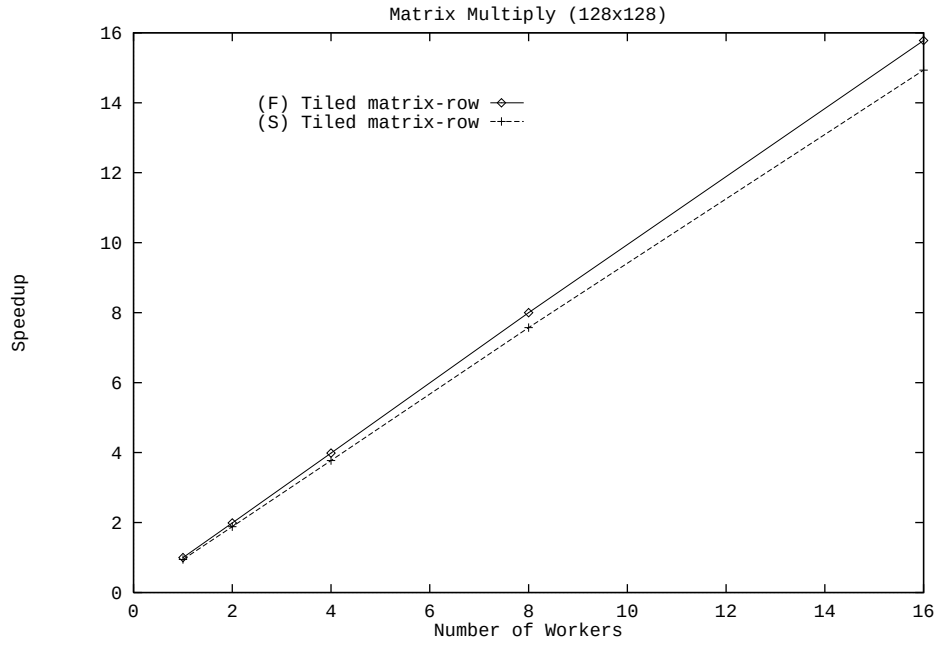

Figure 18: Speedup of matrix multiply.

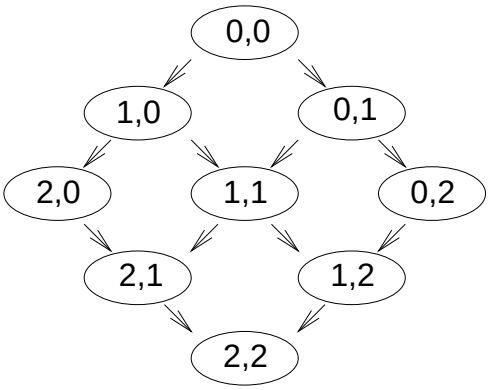

Figure 19: Example precedence graph for 2cMVA with population (2,2).

4.4 Iterative computations with dependences

We now leave the simple world of dependence-free iterative computations and enter the complicated world of dependence-rich iterative computations. An example of an iterative computation with a rich dependence set is two-class Mean Value Analysis (2cMVA) [10]. 2cMVA has the prototypical structure of a dynamic programming problem; Figure 19 shows an example of a 2cMVA *precedence* graph. The parameters that impact the performance of the 2cMVA problem are its *population* (N_1, N_2) and the number of *queueing centers* NQ.[12] The population specifies the dimensions of the precedence graph, while the granularity of each node in the graph is directly proportional to NQ.

A code skeleton for sequential 2cMVA is shown in Figure 20. The `dopop` (*do pop*ulation) routine is an encapsulation of the computation of one node in the precedence graph, which computes a result of type `QLengths`. (Note that the return value from `dopop` is not really necessary since the Q array is declared globally.) In typical dynamic programming style, the computation is iterated in some topologically-sorted order of the precedence graph.

The iteration in Figure 20 is along the diagonals of the precedence graph. It is equally correct to iterate along the horizontal levels of the graph, as shown in Figure 21. The code makes use of the fact that all nodes within a level have the property that the sum of their population components is a constant, `ntot`. Also note that all nodes within a level are independent of each other. This technique, which is generally referred to as *loop skewing* [13, Ch. 8], is the key to parallelizing this type of computation.

Using our `C++` futures it is quite straightforward to parallelize this computation. Figure 22 shows the code for `dopop` and `main`. Each element of QF is a programmer-defined future that will eventually resolve to the corresponding element of Q. (The declaration of class `QLFuture`, which is derived from `Future`, is omitted for brevity.) The main routine is dealing out individual loop iteration to the workers.

Note that by declaring QF in this way, all elements of QF are initially in state unbound. The interesting thing about this is that the computation will work correctly no matter in what order the main routine binds the futures to iterations, because of the following two facts:

1. a reference to an unbound future will block until a computation is bound to it.

[12] We shall not attempt to define there terms here; the interested reader is referred to [10].

```
QLengths Q[N0][N1];

QLengths*
dopop(int n0, int n1)
{
    for (q = 0; q < NQ; q++) {
        if (n0 > 0) {
            QLengths &prevpop = Q[n0-1][n1];
            ... = prevpop ...
        }
        if (n1 > 0) {
            QLengths &prevpop = Q[n0][n1-1];
            ... = prevpop ...
        }
    }
    ... computation with local variables ...
    QLengths &thispop = Q[n0][n1];
    thispop-> ...  = ...
    return thispop;
}

main() {
    for (n0 = 0; n0 <= N0; n0++) {
        for (n1 = 0; n1 <= N1; n1++) {
            dopop(n0,n1);
        }
    }
}
```

Figure 20: Code skeleton for sequential 2cMVA.

```
main() {
    for (ntot = 0; ntot <= N0+N1; ntot++) {
        for (n0 = max(0, ntot-N1); n0 <= min(ntot,N0); n0++) {
            n1 = ntot-n0;
            dopop(n0,n1);
        }
    }
}
```

Figure 21: Loop-skewed 2cMVA.

```
QLengths Q[N0][N1];
QLFuture QF[N0][N1];

QLengths*
dopop(int n0, int n1)
{
    for (q = 0; q < NQ; q++) {
        if (n0 > 0) {
            QLengths &prevpop = *(QLenghts*)QF[n0-1][n1];
            ... = prevpop ...
        }
        if (n1 > 0) {
            QLengths &prevpop = *(QLengths*)QF[n0][n1-1];
            ... = prevpop ...
        }
    }
    ... computation with local variables ...
    QLengths *thispop = Q[n0][n1];
    thispop-> ...  = ...
    return thispop;
}

main() {
    for (ntot = 0; ntot <= N0+N1; ntot++) {
        for (n0 = max(0, ntot-N1); n0 <= min(ntot,N0); n0++) {
            n1 = ntot-n0;
            count++;
            QF[n0][n1](n0,n1,(count%numworkers) +1);
        }
    }
}
```

Figure 22: Futurized MVA (dealing out individual `dopop` iterations).

2. since all futures will have `f.depth` = 1, there can be no leapfrogging and hence no chance of a deadlock.

However, it behooves the programmer to bind them in the order shown (loop-skewed) in order to get the best possible performance.

As our final attack on this problem, we demonstrate how to block-deal the work to the workers. The precedence graph shown in Figure 19 gives no clue how to do this. Our solution is shown in Figure 23.

The block-deal algorithm is tricky because it requires the dependences of the program to be known, as in [3]. A deadlock will result if an iteration that is currently being executed is dependent on an iteration later in the same worker's block of work. The most straightforward way to insure the correct ordering is to embed the original iteration-spawning code in *every* `BlockFuture`. `BF[i]` simply enumerates all of the `QLFutures` in the original order, stopping to evaluate every `i`'th one.

In contrast, the advantage of dealing individual iterations is that the programmer doesn't have to know (or understand) the dependences and the program is guaranteed to be deadlock free.

The performance of futurized MVA for four different combinations of population and NQ is shown in Figures 24–27.

Figure 24 shows that for a large MVA problem with fairly large granularity ($NQ = 30$) we obtain good speedup, even relative to the sequential program. The graph shows only the results of block dealing because there was very little difference between that technique and iteration dealing.

On the other hand, Figure 25 shows that for small problems, the choice of work distribution technique can have a sizable effect on speedup, although performance is not terribly exciting in either case. (Note the change in scale on the y-axis; also note that the *lowest* of the three curves is the iteration dealing curve.)

The poor performance on the small MVA problem could be due one or both of the following factors:

- The small value of NQ results in a small iteration granularity. Thus the overheads of future creation and synchronization have a larger impact on performance.

- The small population reduces the dimensions of the precedence graph. Although a 21x21 graph seems rather large, the enforcement of precedences causes the average *available* parallelism of this problem to be quite a bit smaller than the diameter of the graph. This can be understood by re-examining the precedence graph in Figure 19. Although this is a 3x3 graph, the average available parallelism is only $(1 + 2 + 3 + 2 + 1)/5 = 1.8$. For small problems, the "narrow" sections of the precedence graph represent a greater proportion of the running time of the problem.

In an effort to determine whether the poor performance on the small problem was due mainly to the smaller granularity (NQ) or the smaller precedence graph, we ran two more experiments using the remaining two possible combinations of granularity and problem dimensions. The results are not conclusive, since performance is virtually identical in these two cases (Figures 26, 27). We do note that there is a larger difference relative to the sequential program in the small-granularity, large dimension problem (Figure 26).

```
QLengths Q[N0][N1];
QLFuture QF[N0][N1];

class BlockFuture : public Future {
    int id;

    void operator()(int i) {
        id = i;
        init_future((i+1)%numworkers);
    }

    void evaluate() {
        int count = 0;
        for (ntot = 0; ntot <= N0+N1; ntot++) {
          for (n0 = max(0, ntot-N1); n0 <= min(ntot,N0); n0++) {
            int n1 = ntot-n0;
            if (count%numworkers == pid)
                QF[n0][n1] = dopop(n0,n1);
            count++;
          }
        }
    }
};

main() {
    BlockFuture BF[numworkers];
    for (i=0; i < numworkers; i++) {
       BF[i](i);
    }
}
```

Figure 23: Futurized MVA (dealing out blocks of **dopop** iterations).

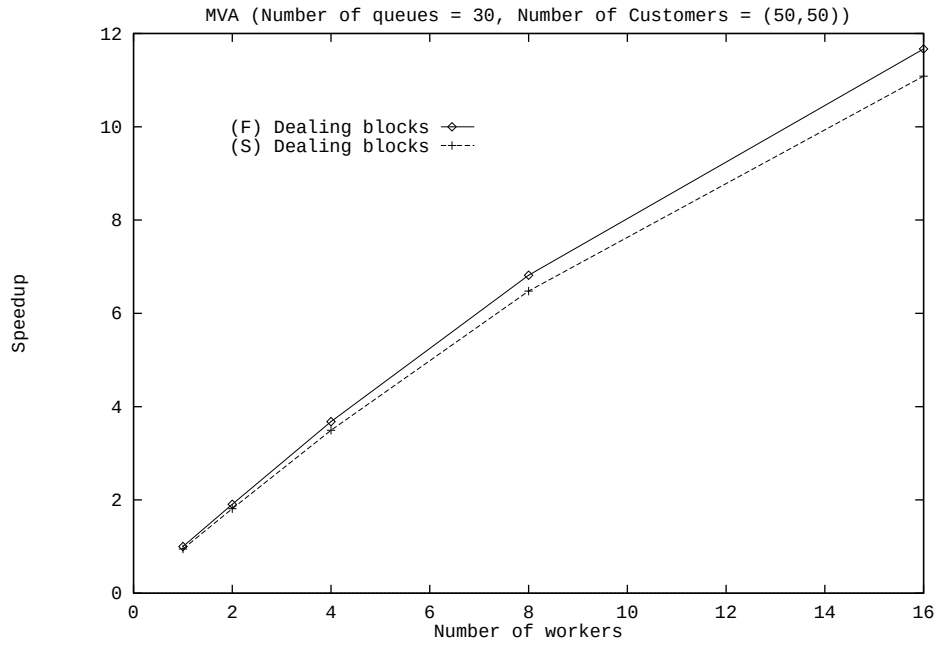

Figure 24: Speedups for MVA with large granularity and large problem dimensions.

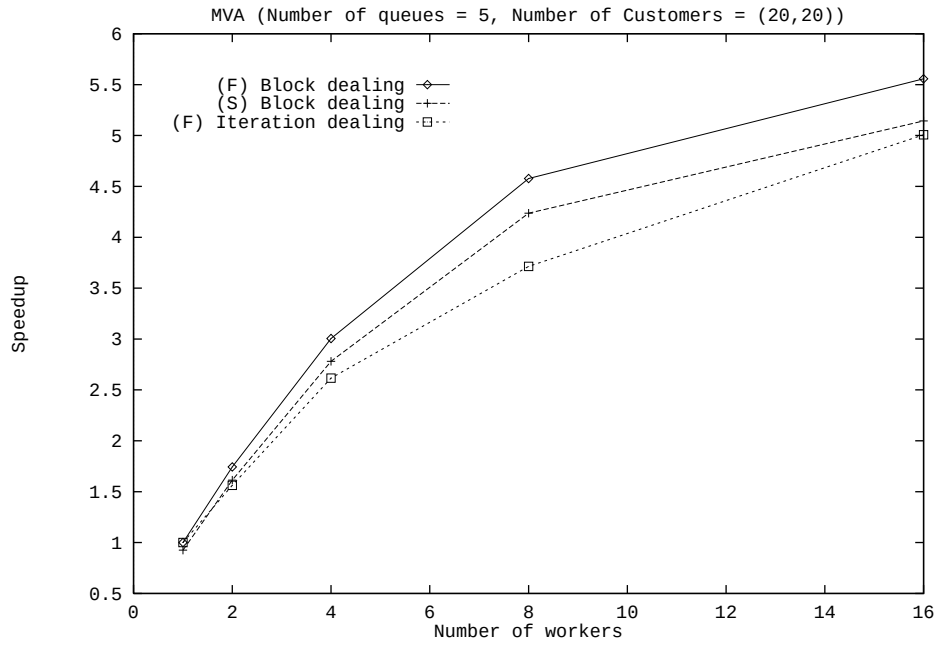

Figure 25: Speedups for MVA with small granularity and small problem dimensions.

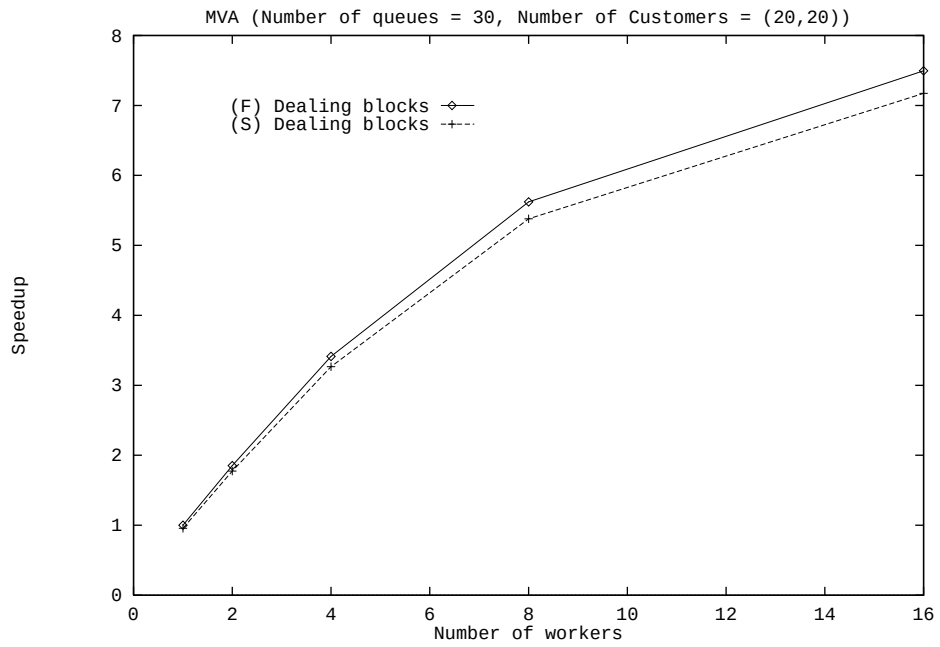

Figure 26: Speedups for MVA with large granularity and small problem dimensions.

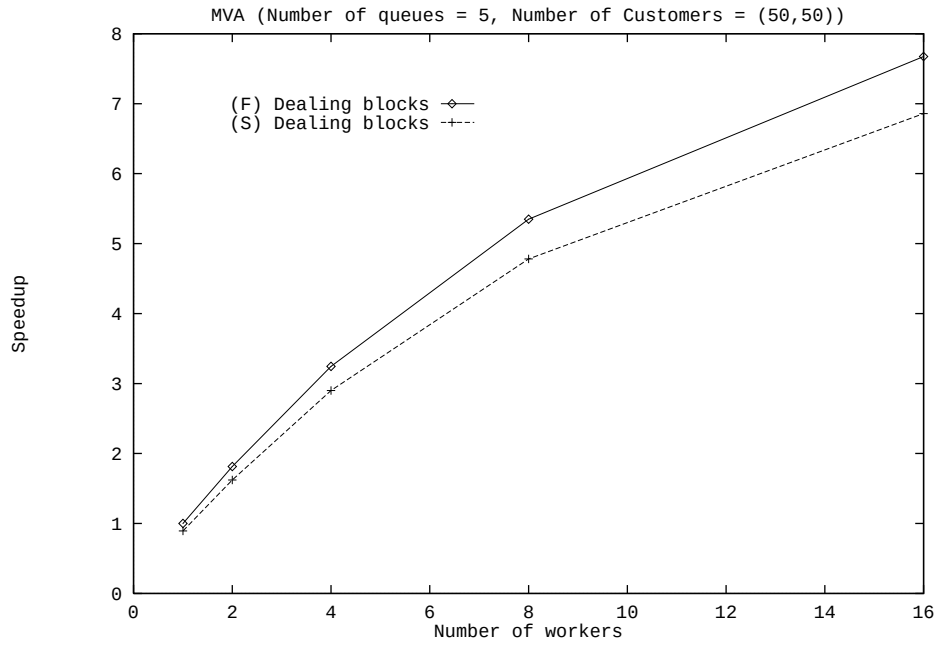

Figure 27: Speedups for MVA with small granularity and large problem dimensions.

5 Conclusions and Future Directions

The principal contribution of this research is an efficient, portable, C++ implementation of futures. Our technique for handling the problem of worker blocking, *leapfrogging*, balances the load well and is completely platform-independent. In addition, it is provably free from deadlocks, and worker stack sizes are kept within a constant factor of the maximum depth of the computation tree.

We also have demonstrated the use of our system on a wide variety of computations, both recursive and non-recursive. We feel especially strongly that our system makes parallelizing dynamic programming computations about as painless as possible. This is due to a combination of the implicit synchronization provided by all futures with the use of *unbound* futures, which are unique to our implementation. Using *unbound* futures, the programmer can enumerate all iterations of the problem in any order without sacrificing correct execution (although efficiency may suffer). This is a big advantage in case not all precedences are known in advance, or if they are known but are so complicated that it is difficult for the programmer to enumerate them in some topologically sorted precedence order.

The syntax used by our interface certainly could be improved. We believe it would increase the transparency and flexibility of the interface to overload the C++ *member selection operator*, as is done in the ES-kit system [1]. We have refrained from doing so because the compiler we are using does not follow the current standard for this operator [4]. This option requires further exploration. Another option that we are considering is writing a simple preprocessor.

Our system is currently implemented using Cthreads [2] to manage parallelism and synchronization. However, the implementation is easily portable to any platform that provides thread creation primitives, shared memory, and mutual-exclusion locks.

Acknowlegements

The authors would like to thank Mike Fox, for working out some of the syntax design issues in embedding futures in C++; Dennis Colarelli, for suggesting that we tile the matrix multiply code; and the members of the Computer Science Department at the University of Washington, Seattle, who put up with our use of their Sequent multiprocessor.

The authors would also like to acknowledge the National Science Foundation, which supported this research under award #CCR-9010672.

References

[1] Chatterjee, A., Khanna, A., and Hung, Y. ES-Kit: an object-oriented distributed system. *Concurrency: Practice and Experience 3*, 6 (Dec. 1991), 525–539.

[2] Cooper, E., and Draves, R. C-Threads. Tech. Rep. CMU-CS-88-154, Carnegie-Mellon University, Feb. 1988.

[3] Eager, D. L., and Zahorjan, J. Chores: Enhanced run-time support for shared-memory parallel computing. Tech. rep., University of Washington, 1991.

[4] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[5] Gabriel, R., and McCarthy, J. Queue-based multi-processing lisp. *Proc. 1984 ACM Symp. Lisp and Functional Programming* (Aug. 1984), 25–44.

[6] Goldman, R., and Gabriel, R. Preliminary results with the initial implementation of Qlisp. In *Proc. 1989 Conf. on Lisp and Functional Programming* (July 1989), ACM SIGPLAN, ACM, pp. 143–152.

[7] Goldman, R., and Gabriel, R. Qlisp: Parallel procesing in Lisp. *IEEE Software* (July 1989), 51–59.

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

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

[10] Reiser, M., and Lavenberg, S. Mean value analysis of closed multichain queueing networks. *JACM 27*, 2 (Apr. 1980), 313–322.

[11] Vandevoorde, M. T., and Roberts, E. S. Workcrews: An abstraction for controlling parallelism. *Int. Journal of Parallel Programming 17*, 4 (1988), 347–366.

[12] Wolfe, M. More iteration space tiling. In *Proc. of Supercomputing '89* (Reno, NV, Nov. 1989), pp. 655–664.

[13] Wolfe, M. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.