Lazy Remote Procedure Call and its Implementation in a Parallel Variant of C

Marc Feeley

Dépt. d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, Québec, CANADA
feeley@iro.umontreal.ca

Abstract. *Lazy task creation* (LTC) is an efficient approach for executing divide and conquer parallel programs that has been used in the implementation of Multilisp's `future` construct. Unfortunately it requires a specialized memory management scheme, in particular for stack frames, which makes it hard to use in the context of conventional languages. We have designed a variant of LTC which has a stack management discipline that is compatible with the semantics of conventional languages. This mechanism, which we call *lazy remote procedure call*, has been used to implement a parallel variant of C. A first prototype of our system has been ported to shared-memory multiprocessors and network of workstations. Experimental results on a Cray T3D multiprocessor show that good performance can be achieved on several symbolic programs.

1 Introduction

The `future` construct of Multilisp [Halstead, 1985] has proven to be a convenient and effective means of expressing parallelism in Lisp and in symbolic processing applications. When implemented with *lazy task creation* (LTC), futures can execute parallel divide and conquer programs very efficiently on shared-memory computers as shown by the Mul-T [Mohr, 1991] and Gambit [Feeley, 1993a] systems. LTC dynamically groups tasks together to adjust the effective task granularity of the program thus relieving the programmer from having to think about task granularity, load balancing, number of processors and processor speed. LTC produces a granularity that is both fine enough to keep processors working most of the time and coarse enough to keep task creation overhead low.

These advantages have compelled us to explore the use of LTC for implementing parallel variants of conventional languages. The work reported in this paper is an attempt to adapt LTC to C. We have designed and implemented a new language, called ParSubC (Parallel Subset of C)[1], which:

- is suitable for parallel symbolic processing,
- supports a shared-memory model and a form of message-passing,
- is portable to shared and distributed memory parallel machines,

[1] We have kept this name even though at this point the language is a superset of C.

– is object-code compatible with conventional C compilers.

In parallel symbolic applications, tasks often need to read and write data shared with other tasks. For this reason and to stay close to C's semantics, ParSubC provides a shared-memory model with a single address space and with transparent access to shared data. On distributed-memory machines, this model is implemented with a virtual shared-memory package. Parallelism is introduced with a parallel function call construct. This construct has fork-join semantics and is thus suitable for writing divide and conquer parallel programs. A set of atomic operators is also provided for cases where fork-join synchronization between tasks is not sufficient. The parallel call construct provides a limited message-passing capability which is convenient for expressing certain parallel algorithms and can lead to better efficiency on distributed-memory machines due to bulk transfer. ParSubC's support for the message-passing and shared-memory models combined with its portability means that it is applicable to a wide variety of parallel applications. In typical parallel program development, one of the two models has to be chosen in advance based on the requirements of the application and the target machine and compiler. With ParSubC, a program can freely mix both programming models and is not tied to a particular machine architecture. Object-code compatibility with conventional C compilers is another important feature of ParSubC which allows sequential C code and precompiled libraries to be linked seamlessly with ParSubC code.

This paper does not describe in detail the complete implementation of ParSubC. Instead, the focus is on the implementation of the parallel call construct which is the central problem in ParSubC. As explained in the next section, LTC cannot be used directly to implement ParSubC because the stack management discipline it requires is incompatible with the semantics of C. As a result, we have designed a variant of LTC, which we call *Lazy Remote Procedure Call* (lazy RPC), that is compatible with C's semantics.

2 Lazy task creation

Let us first review a possible implementation of LTC for Multilisp to see why its task management discipline cannot be applied directly to C.

2.1 Implementing LTC for Multilisp

Figure 1 shows the global organization of a LTC based Multilisp system. With each processor is associated an instance of the following three data structures: the run time stack, which is where continuations are allocated, the *lazy-task stack* (LTS), which is a double-ended queue used mostly like a stack, and the *runnable-task queue* (RTQ). The LTS contains pointers to continuations on the local run time stack that are available for execution by other processors (the *stealable* continuations) and the RTQ contains previously blocked tasks that have become runnable. At the top of the stack is the continuation of the task currently

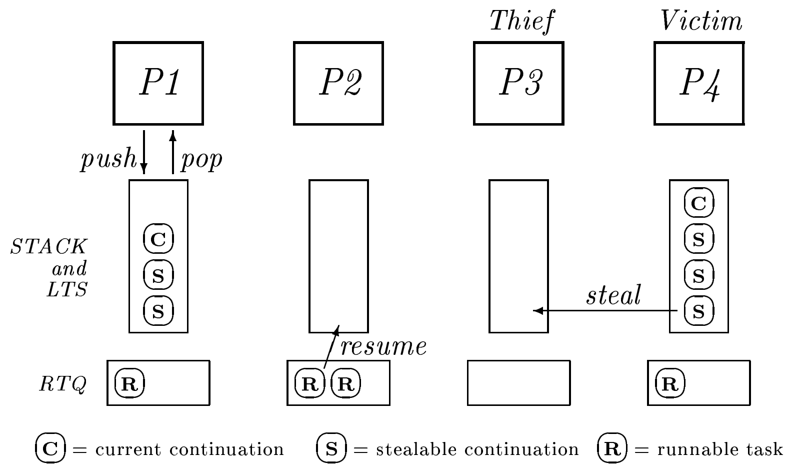

Fig. 1. Global organization of a LTC based Multilisp system.

running on that processor. Let's call k_f the implicit continuation of expression `(future` *expr*`)`. To evaluate this expression a processor simply pushes a pointer to k_f on top of its LTS (this corresponds to pushing the top of stack pointer on the LTS) and then starts evaluating *expr* with a newly allocated continuation k_e. This continuation simply pops the continuation pointer to k_f from the LTS and returns control to k_f. Unless another processor steals k_f from the LTS as explained below, the flow of control that results is identical to that of a function call.

An idle processor gets work by *stealing* a continuation from another processor's LTS or by resuming a task from a RTQ (either the local one or that of another processor). The two processors involved in a continuation steal are the *thief* and *victim*. For implementation simplicity and in order to minimize the number of steals needed to keep the thief working it is the oldest continuation on the victim's LTS that is stolen (i.e. the one at the base of the LTS). When the program is a balanced divide and conquer algorithm, this continuation corresponds to the one with the largest amount of work. The following steps are required to steal continuation k_f. First, the pointer to k_f is removed from the victim's LTS so that no other processor can steal it and k_f is copied to the thief's run time stack. Secondly, an empty placeholder object is created to act as a representative of *expr*'s value and k_e on the victim's run time stack is transformed into a new continuation k'_e that sets the placeholder to the value it receives (i.e. *expr*'s value) and terminates the current task. Finally, the thief invokes its copy of k_f with the placeholder that was created.

The effect of the steal operation is shown in Figure 2. The "before" diagram depicts the state of the stack and LTS at the start of the evaluation of the body of a future nested within 2 other futures. The continuation k_f of the first future evaluated consists of stack frames 1 and 2, and the frame created for k_e

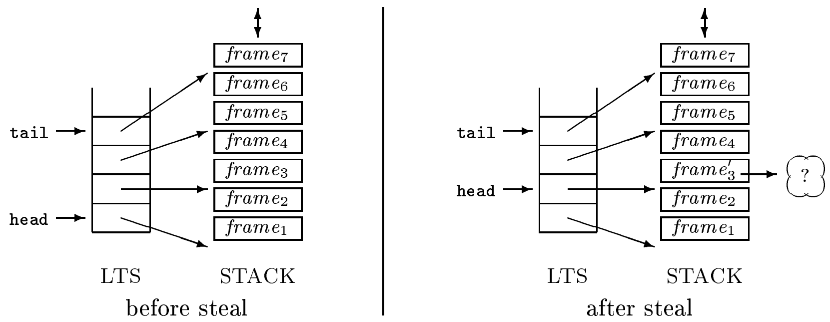

Fig. 2. Victim's run time stack and LTS before and after a steal from the LTS.

is frame 3. The steal operation copies frames 1 and 2 to the thief's stack, and updates frame 3 and the base pointer of the LTS (i.e. `head`). If another steal occurred, frame 3 as updated by the previous steal and frame 4 would be copied to the thief's stack. Note that the active part of the LTS is the section above `head` up to `tail`. The distance between these pointers is the number of stealable continuations. An important invariant is that the pointer under `head` and just above it delimit the stack section containing the frames to copy.

2.2 Task blocking

Trying to dereference (i.e. touch) an empty placeholder causes the current task to be suspended (by copying the current continuation to the heap) and put on a queue of blocked tasks associated with the placeholder. The continuation k'_e in which this placeholder was stored will make the blocked tasks runnable by transferring them to the RTQ just after having set the placeholder to the value returned to k'_e.

Task suspension is more complex when the stack contains stealable continuations. In order to not reduce the amount of parallelism it is important that these continuations remain stealable. Consequently, task suspension is handled similarly to a steal of the topmost stealable continuation to preserve the link between this continuation and the suspended task. An empty placeholder representing the result of the current task is created and the oldest frame of the current continuation (which corresponds to the continuation k_e of the body of the future that created the current task) is transformed into a new continuation k'_e that sets the placeholder to the value it receives.

After a processor has suspended its current task, it can proceed in two different ways. The "tail-biting" approach used in the Encore version of Mul-T [Mohr, 1991] consists of invoking the topmost stealable continuation with the placeholder just created. This approach can be inefficient because the topmost continuation typically contains less work than older continuations and because subsequent blocking is likely. An alternative approach, used in Gambit [Feeley, 1993a], is to immediately copy the stealable continuations to the RTQ in reverse order (so that older continuations are stolen first) and to invoke the oldest continuation. This has the advantage of reducing the likelihood of further blocking by giving more time to compute the placeholder's value before it is needed, and it allows the implementation of the Katz-Weise semantics for first-class continuations [Katz and Weise, 1990].

2.3 Incompatibility with C

There are several reasons why this implementation of LTC can not be applied directly to C. First, placeholders are objects with indefinite extent. They can not be allocated in the run time stack because they can outlive the continuation of the future they were created for. Using a general purpose garbage collector to reclaim heap allocated placeholders is out of the question for a language like C, so the responsibility of deallocating placeholders must rest with the programmer. This however is an error prone solution which we would like to avoid.

Secondly, unlike C, Multilisp uses a uniform representation for objects (i.e. all objects are encoded by a pointer) and objects contain a type tag. It is thus possible for a placeholder to masquerade as a true value and a placeholder can easily be distinguished from a true value. Without these features a placeholder would have to be created in the heap for every future evaluated regardless of whether its continuation gets stolen, thus diminishing the effectiveness of LTC's laziness. The allocation/deallocation overhead this would incur is probably large enough to preclude fine grain tasks.

Finally, stack frames need to be moved to a different address when tasks are stolen, suspended, and resumed. As a consequence, C's "address-of" (i.e. `&`) operator can not be used on local variables. This is a severe restriction because it prohibits common C idioms including "by reference" parameter passing and stack allocation of data.

Our solution to these problems is to hide tasks and placeholders from the user and change the scheduling and stealing algorithms to restrict tasks, continuations and placeholders to have dynamic extent (i.e. nested lifetimes). This makes it possible to allocate them on the stack and to never move them.

3 ParSubC's parallel call

In ParSubC parallelism is introduced explicitly with the parallel call construct which essentially expresses concurrency between a function call and a compound statement. A parallel call can appear anywhere an ordinary function call can appear; a "`!!`" and a compound statement are simply added after the list of arguments, as in:

function`(` arg_1 `,` ... `,` arg_n `) !! {` *declaration-list statement-list* `}`

Conceptually the execution of a parallel call consists of the following steps:

1. The arguments are evaluated.
2. The function's body and the compound statement are executed concurrently.
3. When both executions are done, the continuation of the function call is executed with the result of the function.

This ordering avoids some race conditions, in particular: between the evaluation of the arguments and side effects in the compound statement, and between the compound statement and side effects in the call's continuation. In the parallel call

```
x = sin(y) !! { y = cos(x); };
```

there is no race condition for reading and writing variables `x` and `y`. Since procedures are viewed as functions with `void` result in C, the parallel call construct can also be used for procedure calls. The creation of more than two tasks can be obtained by cascading parallel calls as in the call

```
f(x) !! { g(y) !! { h(z); }; };
```

which expresses concurrency between the procedure calls to `f`, `g`, and `h`.

```
int sum( int *X, int n )
{
  if (n == 1)
    return *X;
  else
    {
      int s1, s2;
      s1 = sum(X,n/2) !!
           { s2 = sum(X+n/2,n-n/2); };
      return s1 + s2;
    }
}
```

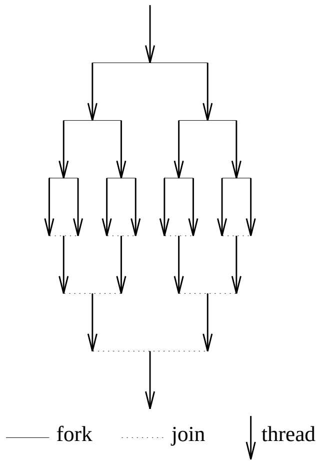

Fig. 3. Parallel sum of an array in ParSubC and resulting spawning graph for array of length 8.

Figure 3 shows a simple application of ParSubC's parallel call in a recursive function that computes the sum of the elements of array `X` of length `n`. The array is split in two sub-arrays of equal size and the sum is performed concurrently on both sub-arrays. The recursion ends when the array is of length 1. The resulting spawning graph shows the fork-join dependences between the threads of control that are created. By convention the left thread at fork points corresponds to the function call and the right thread to the compound statement. Divide and conquer parallel algorithms such as this are a typical way to use ParSubC's parallel call.

4 Lazy remote procedure call

Lazy RPC borrows from LTC the idea of deferring the creation of new tasks until some processor needs work. However, instead of stealing continuations to create tasks, lazy RPC creates tasks by stealing function calls. The LTS is managed in the same way as in LTC but it contains pointers to function call descriptors on the stack rather than pointers to continuations.

When a parallel call is executed, a function call descriptor is created on the stack and the LTS is updated to indicate that this call is stealable. Execution then continues with the compound statement. If the function call was not stolen by another processor when the compound statement is done, then the function call is removed from the LTS and the call is performed immediately.

4.1 Join points

Execution can not proceed past the join point if the function call was stolen and it has not completed yet. This situation is similar to the touching of an empty placeholder in LTC, but an important difference is that there are no stealable calls left on the LTS since older calls are stolen first. Rather than busy wait for the call to complete, the victim processor will steal a call from another processor (if one is available). When the victim processor has finished executing this function call it checks the status of the call it is waiting on and proceeds past the join point if it is done.

The code sequence required to implement a parallel call with lazy RPC is shown in Figure 4 in a C-flavored pseudocode. The exact layout and size of descriptors varies from function to function, but they contain the following fields: a pointer to a function descriptor (`function`), the arguments of the call (`arg1`,...), and the result of the function if there is one (`result`). The `function` field serves a dual purpose. Initially it points to a function descriptor which contains information on the function to call including the entry point and the space occupied by the arguments and the result. This information is required to steal and execute the call and to return the result. The `function` field is also used as a flag to indicate completion of the stolen call.

4.2 Stealing function calls

The procedure `steal_and_execute_function_call` is responsible for stealing calls when a processor is idle or waiting for a call to complete. When the program is started all processors execute this procedure in an infinite loop, except for the one executing the program's `main` function. Each LTS is checked in turn until

```
{
  descriptor d;                   /* allocate function call descriptor on stack */

  d.function = &f_descr;          /* initialize function call descriptor */
  d.arg1     = x*y;
  d.arg2     = g(x);

  *++tail = &d;                   /* push descriptor to LTS */

  { statements }                  /* execute compound statement */

  tail--;                         /* join point, pop descriptor from LTS */

  if (head > tail)                /* was call stolen? */
    {
      head--;
      while (d.function != 0) /* is call still in progress? */
        steal_and_execute_function_call();
    }
  else
    d.result = f( d.arg1, d.arg2 );

  /* result of call is available in d.result */
}
```

Fig. 4. Pseudocode for the parallel call `f(x*y,g(x)) !!` { *statements* }.

a victim processor with stealable calls is found (i.e. `head<tail`). `Head` is then incremented, the corresponding function call descriptor is fetched (except for the result field), and finally the function is called with the appropriate arguments. When the function returns, the result is transferred to the descriptor on the victim's stack and the `function` field is set to zero to indicate completion of the call. Note that the thief can perform these operations directly on a shared-memory machine or by exchanging messages with the victim on a distributed-memory machine. In this latter case, three messages in all are required per successful steal: the steal request, the call's descriptor, and the call's result.

A race condition exists between the thief stealing a call and the victim popping a call from the LTS. To avoid conflict, the pair of instructions performed by the victim at the join point for decrementing `tail` and testing `head>tail`, must not be executed at the same time as the test `head<tail` and increment of `head` performed by the thief. This mutual exclusion problem can be solved in a variety of ways including the use of polling to handle steal requests [Feeley, 1993b], disabling interrupts during the critical sections, and serializing access to the LTS with hardware locks or software locks [Feeley, 1993a].

4.3 Why defer the function call?

When a parallel call is executed, lazy RPC defers the function call and immediately begins the execution of the compound statement. The alternative of deferring the execution of the compound statement, which would be closer to what LTC does, is not as good on distributed-memory machines. In general the compound statement must access the current stack frame to obtain the data it needs and to return results to the code following the join point. Since we do not want to move stack frames, a stolen compound statement will have to access the stack frame remotely. In addition, a message is needed to signal termination of the compound statement. When stealing function calls, a single message carries all the arguments required and the signal of termination can be piggybacked on the message carrying the result.

4.4 Overhead of a parallel call

It is important that the cost of the parallel call construct be low to allow fine grain parallelism. To evaluate this cost we will measure the overhead of executing a parallel call compared to a sequential call and assume that the call is not stolen. In other words, we want to know how much more expensive it is to execute the function `par` than it is to execute `seq`:

```
void par(int a, int b); { x = f(a) !! { y = g(b); }; }
void seq(int a, int b); { x = f(a); y = g(b); }
```

Some steps in the execution of a parallel call can be ignored since they are also needed when a sequential call is used: the evaluation of the arguments and the execution of the compound statement. The allocation and deallocation of the function call descriptor can also be ignored because it can be done with the normal stack pointer adjustments at function entry and exit.

Depending on the compiler's calling convention, the initialization of the function call descriptor is not necessarily all overhead. If the compiler passes arguments on the stack it is easy to choose a layout for the descriptor's argument fields which matches the calling convention. A similar statement holds for the `result` field if function results are returned on the stack. If arguments and results are passed in registers (as is the case on most RISC machines) then each argument will account for an additional write and read from the descriptor. No overhead is counted for the assignment to `d.result` since it can easily be bypassed (the result will be immediately consumed by the code that follows the parallel call). To be thorough however we must take into account the variation in the creation of the caller's stack frame. The above example shows this well. Arguments `a` and `b` are passed in registers to the functions `par` and `seq`. However, `seq` must store `b` to the stack before calling `f` and fetch it later because its value is needed for the call to `g`. This is not the case for `par` which consumes `a` and `b` before any call that could overwrite them. Consequently, for the example above, the only overhead associated with the initialization of the descriptor and call of the function is the initialization of the descriptor's `function` field.

The remaining overhead is the pushing and popping of the descriptor on the LTS and the test `head>tail`. The total overhead per call can thus be as low as these 4 operations which translate into about 12-15 RISC instructions (including

4 writes and 3 reads) if `tail` and `head` are kept in memory. This compares well with the overhead of LTC. A `future` would generate these same instructions except for the initialization of the descriptor's `function` field. However, the `touch` operation needed in LTC for synchronization is not needed in lazy RPC so the total overhead is equal or slightly lower for lazy RPC (depending on how placeholders are represented and tested for).

5 Discussion

5.1 Compatibility with conventional languages

The stack management discipline of lazy RPC is well suited for conventional languages. Since the stack does not have to be managed specially when executing sequential code, parallel code can be linked seamlessly with precompiled libraries and object files produced by conventional compilers[2].

Since stack frames are never moved, the address of a local variable stays the same throughout its life. It is thus straightforward to implement C's `&` operator if the environment supports a shared address space. On a shared-memory multiprocessor, each processor can allocate its stack in a different region of the shared address space so that references will transparently be directed to the right frame. On a distributed-memory machine, each processor can allocate its stack in local memory if a virtual shared-memory package is used to implement the shared address space.

5.2 Loss of parallelism

Lazy RPC's policy for handling join points is problematic because it creates artificial control dependences between threads. If a processor is waiting at join point J for a stolen call A to complete and it steals call B, then J becomes dependent on the termination of B. This case is shown in Figure 5 where each solid oval corresponds to a call that has been stolen and each black dot is the point of execution in a thread. If A terminates before B some parallelism will be wasted because execution past J, which is possible in principle, must wait for the termination of B. The fact that B can itself be waiting at a join point (because a call was stolen from it) only compounds the problem. This loss of parallelism however will only be important if there is little parallelism in the program. Our experimental results indicate that the loss is small on fine grain divide and conquer programs (see Section 6).

5.3 Stack size

With LTC the space needed for the stack and LTS per processor to run a parallel program is within a constant factor of the space needed for a sequential execution

[2] Of course there may be incompatibilities with things like `longjmp`, `fork`, and threads packages.

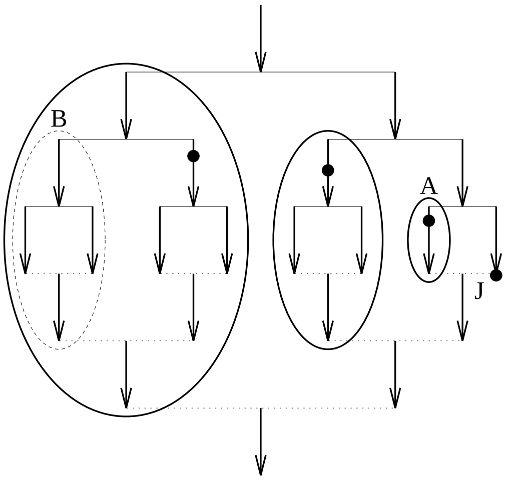

Fig. 5. Call B is stolen because stolen call A has not completed when control reaches join point J.

of the program. This is not the case for lazy RPC because all blocked threads (those waiting at a join point) consume space on the stack and LTS. To simplify the analysis we will only search for an upper-bound on the size of the LTS (for typical programs the size of the stack is within a constant factor of the size of the LTS, where the constant depends on the program). The number of entries on the LTS (N_{LTS}) is at most d times $s + 1$ where d is the nesting depth of parallel calls and s is the maximum number of calls stolen from a processor. According to [Mohr, 1991], if p is the number of processors, $s < pd$ for LTC with "polite stealing" (which requires that a processor attempt to steal from a given processor only after having tried to steal from all other processors). This upper-bound implies that for lazy RPC $N_{LTS} \leq pd^2$. Unfortunately, there is a subtle error in Mohr's proof[3] so this upper-bound is not correct.

Fortunately, it has been observed that N_{LTS} is usually close to d in practice (see Section 6). It is therefore sufficient to use a stack that is a constant factor larger than is needed for a sequential execution (S_{seq}) and to make the stealing of calls at join points conditional on the space left on the stack to guarantee that the stack never overflows. The join point becomes a busy-wait when the space left is less than S_{seq}.

6 Experimental results

In order to evaluate lazy RPC a prototype ParSubC compiler was built. This compiler was written with experimentation in mind, not absolute performance.

[3] It incorrectly assumes that a processor with an empty LTS has no work to share with other processors. It could simply be that the processor has not yet pushed work on the LTS.

Each ParSubC source file is compiled to a file of C code which is subsequently compiled by the machine's native C compiler and linked with the ParSubC run-time system. The C code produced is slower than a direct C compilation because the stack is explicitly represented as an array (all accesses to local variables are compiled to array reference so the native C compiler can not allocate these variables to registers). There is also an overhead for the handling of steal requests. Each processor has a "steal" flag that is raised when a steal request is pending. This flag is checked at every function call and inside every loop. For a purely sequential program running on one processor, the slowdown factor caused by this code generation is anywhere between 2 and 20. For this reason it is not possible to draw strong conclusions regarding absolute performance from this prototype. In the near future, we plan to implement a more direct ParSubC to C translation that will impose no overhead for the sequential parts of the program.

The ParSubC compiler has been ported to a variety of shared-memory multiprocessors (including the BBN Butterfly, SGI Challenge, and Cray T3D) and network of UNIX workstations using a virtual shared-memory package implemented on top of TCP/IP. Here we only report on the Cray T3D port because it is representative of the ports to shared-memory multiprocessors and because the port to network of workstations is undergoing changes to improve its performance on communication intensive programs.

6.1 The Cray T3D implementation

The Cray T3D that was used is a 512 processor machine at the Pittsburgh Supercomputer Center. Each processor is a 150MHz DEC Alpha microprocessor with 64MB of local memory. The processors are interconnected by a 3D torus network. By configuring the virtual memory circuit attached to each processor (the "annex"), it is possible to implement a coherent shared address space for up to 16 processors [Numrich, 1994]. When this is done, a cache miss to local memory costs about 30 cycles and a remote memory reference costs 3 to 4 times that depending on the distance of the remote processor. The annex also supports an atomic swap operation and we have used it to implement spin locks.

Shared-memory is used to implement the steal operation. The thief stores its processor number in the victim's memory, raises the victim's steal flag, and starts waiting for a response by busy-waiting on a response variable in local memory. A lock associated with the victim's LTS is used to prevent interference from other processors. When the victim detects the steal request it removes the oldest call from its LTS (if there is one) and stores a pointer to the descriptor in the thief's response variable. The thief then prepares the call on its stack by reading the descriptor on the victim's stack and executes the call. Finally, the thief stores the result in the descriptor on the victim's stack and zeros the `function` field. This protocol was designed to avoid the memory traffic that would occur if busy-waits were done on remote flags.

6.2 Benchmarks

Most of the benchmarks we have used are small programs ($<$ 200 lines of ParSubC) translated from Multilisp. We have also used a 3000 line program (`nucleic`) that computes the 3D structure of a nucleic acid [Feeley *et al.*, 1994][Hartel *et al.*, 1996]. All these programs are based on divide and conquer parallel algorithms. These programs are briefly described in Appendix A.

The benchmarks were run on 1 to 16 processors and the average run time (real time in seconds) of 10 executions was taken giving T_1 to T_{16}. We also ran sequential versions of the programs (by simply replacing "`!!`" by "`;`") using one processor (T_{seq}). The parallel call granularity (g) corresponds to the average run time per parallel call (i.e. T_1/n where n is the number of parallel calls executed). Table 1 gives g, T_{seq}, T_1, and the speedup (T_1/T_p) for each benchmark.

Program	g (μsecs)	T_{seq}	T_1	Speedup (T_1/T_p) when $p=$ 2	4	8	16
`queens`	12	9.60	10.22	2.00	4.00	7.99	15.94
`fib`	4	4.97	5.66	1.99	3.96	7.90	15.72
`fibr`	4	4.97	5.66	1.99	3.96	7.90	15.74
`sum`	6	2.60	3.00	1.92	3.75	7.33	14.13
`nucleic`	66	2.32	2.25	1.87	3.57	6.89	12.55
`scan`	7	1.57	1.72	1.87	3.52	6.02	8.58
`poly`	15777	7.76	8.06	1.70	3.00	5.21	6.08
`mm`	401	9.26	9.03	1.70	3.07	4.71	4.80
`abisort`	20	4.49	4.62	1.65	2.62	3.39	3.63
`tridiag`	24	4.55	4.64	1.61	2.63	3.40	3.19

Table 1. Benchmark granularity, run time and speedup.

6.3 Lazy RPC overhead

The overhead of lazy RPC is the difference between T_1 and T_{seq}. As expected the overhead is inversely proportional to the parallel call granularity: it is between 10% and 15% for very fine grain programs (`fib`, `fibr`, `sum` and `scan`), and below 6% for larger grain programs. Similar overheads are reported for LTC in [Feeley, 1993a] for the Multilisp version of these benchmarks.

6.4 Speedup

The speedup obtained for `queens`, `fib` and `fibr` is very close to linear. Because these benchmarks have imbalanced spawning trees it shows that lazy RPC is balancing the load very evenly. Appreciable speedup is also obtained for `sum`, `nucleic` and `scan`. The main reason for the lower speedup is that these benchmarks access shared data and the average cost of a memory reference increases with the number of processors (the probability that a reference is remote increases and the average distance increases).

The mediocre speedup of the remaining benchmarks (`poly`, `mm`, `abisort` and `tridiag`) is due to memory contention. For simplicity, all global variables are allocated in the memory of the first processor. This means that all accesses to global data go to the same processor, which leads to a serialization of the accesses by the interconnection network. Since these benchmarks access global arrays very frequently, contention quickly becomes the dominant bottleneck and no further speedup is possible when the network reaches saturation. We confirmed that this was not a load balancing problem by checking that the idle time is small using our profiling tool. This tool processes an event log produced during execution to compute various statistics on the program including the activity of the processors as a function of time and the average time spent in each state (working, stealing, and idle). The output of this tool for `tridiag` run with 16 processors is given in Figure 6. It shows that processors are idle for only about 1% of the total run time. Similarly low idle time was observed for the other benchmarks. This leads us to believe that the loss of parallelism due to lazy RPC's scheduling policy is small when there is ample parallelism to begin with.

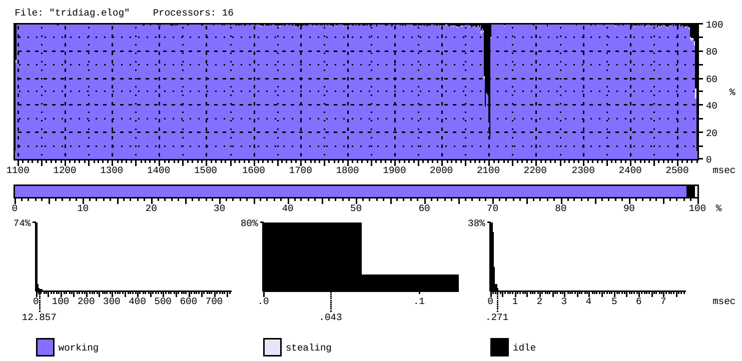

Fig. 6. Execution profile of `tridiag` benchmark with 16 processors.

On a cache coherent multiprocessor, contention becomes less important because data gets distributed on demand to the caches of the processors that access it and there is sufficient locality. On an 8 processor SGI Challenge cache coherent multiprocessor we have observed much better speedup than the Cray T3D for the benchmarks suffering from contention. The speedup with 8 processors is above 7 for `poly` and `mm`, and above 6 for `abisort` and `tridiag`[4].

[4] Because the machine we used for this experiment is shared by many users and is always busy, the speedup was impossible to measure exactly but on an idle machine it would be at least as high as indicated.

6.5 Stack size

To evaluate the stack size needed for the benchmarks, we measured the maximal N_{LTS} over all processors and over 10 runs. This maximum was computed locally by each processor at the pushing of a descriptor on the LTS and a global maximum was computed at the end of the run. The results are given in Table 2.

	Maximal N_{LTS} when				
Program	$p=$ 1	2	4	8	16
`queens`	43	43	43	43	43
`fib`	15	15	19	23	30
`fibr`	29	29	29	29	29
`sum`	18	18	19	23	31
`nucleic`	101	101	101	88	89
`scan`	17	17	18	19	25
`poly`	9	9	10	12	14
`mm`	14	14	15	22	22
`abisort`	14	14	16	20	25
`tridiag`	17	17	18	22	23

Table 2. Size of the LTS as a function of the number of processors.

For $p = 1$, the maximal N_{LTS} equals d, the nesting depth of parallel calls. Most programs show an increase in N_{LTS} as more processors are used. The increase is at its highest for `fib` which at 16 processors has $N_{LTS} = 2d$. For some programs N_{LTS} stays constant (`queens` and `fibr`) or even decreases (`nucleic`). This variation in N_{LTS} is related to the lopsidedness of the spawning tree. N_{LTS} tends to increase with p more rapidly when the spawning tree is lopsided on the side of the call (i.e. the call does more work than the compound statement) because there will be a high probability that a stolen call is not yet finished when the corresponding compound statement ends. It is thus likely that new stolen calls will be added to the stack. This is the case for `fib` which computes the call `fib(n-1)` in parallel with a compound statement which computes `fib(n-2)`. For `queens`, `nucleic` and `fibr` (which is like `fib` but with the recursive calls exchanged), the spawning tree is lopsided on the side of the compound statement so join points rarely end up stealing calls.

7 Related work

One of the earliest attempts to adapt the `future` construct to C was done at Tera Computer Company [Callahan and Smith, 1989] (to our knowledge this system has never been implemented). The authors propose to extend C with a placeholder type qualifier, a task spawning construct, data sharing directives, and synchronization variables. Since no restriction is imposed on the allocation and deallocation of placeholders and tasks, dangling pointers are a definite possibility.

To exploit fine grain parallelism, load based inlining was considered to implement the spawning construct. There is no mention of LTC. A new fixed size stack is allocated for every task created, making it easy to support the `&` operator, but also causing considerable memory consumption and task spawning overhead. The extensions to C are not as natural as ParSubC's because the user must indicate how a spawned task shares data with its parent task.

Cid [Nikhil, 1994] shares with ParSubC the desire for minimal extensions to C and compatibility with existing code and compilers. Cid is mainly intended for programming distributed-memory machines. Its programming model is more complex than ParSubC's because it exposes the features of the architecture to give the programmer more control. Like lazy RPC, a thread is created by forking a function call. Unlike lazy RPC, the call executes concurrently with the continuation of the fork and an explicit join variable must be used to synchronize the call with the consumer(s) of the result. The user can specify that the call be forked on a specific processor or on any processor. The latter case is handled by maintaining on each processor a stealable-call queue (independent from the stack) which contains call descriptors (including the arguments and a pointer to the join variable). Because the stack and stealable-call queue are decoupled, Cid does not handle non-stolen calls as efficiently as lazy RPC. The overhead of a non-stolen call includes a thread suspension on a join variable and a subsequent thread resumption. A form of load-based inlining is available in Cid to reduce the number of non-stolen calls but this is not as efficient as creating tasks lazily [Mohr, 1991].

Olden [Rogers *et al.*, 1995] is another system that has adapted LTC to C for programming distributed-memory machines. Instead of migrating data between processors when non-local data is accessed, Olden migrates the task to the processor containing the data. Like LTC, a stack of stealable continuations is maintained on each processor. Unlike LTC, tasks are only stolen locally and from most recent to oldest. When a task is migrated or a task blocks (because an empty placeholder is dereferenced), the most recent stealable continuation on that processor is invoked. Because processors don't steal tasks from other processors, load balancing is not automatic. Task distribution is directly dependent on the distribution of the data and the uniformity of the work to be performed on the data. The task scheduling policy adopted by Olden requires that a spaghetti stack [Bobrow and Wegbreit, 1973] be used to manage the stack. This adds overhead to all function calls and is not compatible with the stack management of traditional compilers. Finally, the `&` operator can not be used on local variables because stack frames move as a result of task migration (the compaction of the spaghetti stack needed to avoid stack overflows also moves stack frames).

The leapfrogging technique [Wagner and Calder, 1993] is closely related to lazy RPC. It was specifically designed to be compatible with conventional compiler technology and to require no syntactic extension to C++. A subclass of the class `Future` must be declared for each function to be called in parallel. Instances of this class correspond to lazy RPC call descriptors but they are first class objects so the user is responsible for their allocation and deallocation (with the usual caveats). The constructor of this class accepts the parameters of the call, saves them in the descriptor, and adds the descriptor to the processor's stealable-call queue. Stealable calls are removed from this queue in an arbitrary order, so the queue is more expensive to manage than lazy RPC's LTS. When a task dereferences a call descriptor that has not been stolen, the descriptor is removed from the queue it is in (possibly on another processor because there are no restrictions on the order of dereferencing descriptors), the call is executed and the result stored in the descriptor. Like lazy RPC, blocking is implemented by stealing calls but leapfrogging places a number of constraints on the call to steal. When a task executing a parallel call C_1 blocks because it has dereferenced the descriptor of a parallel call C_2 that has been stolen by processor P but has not yet completed, a call is stolen from P that is at a deeper nesting level in the spawning tree than C_1 and C_2. This constraint insures that the stack size needed is no more than for a sequential execution of the program. However it requires the maintenance of a nesting depth and a search in P's stealable-call queue, it exposes less parallelism than lazy RPC, and tends to steal smaller and smaller calls, leading to a higher number of stolen calls and a higher overhead per parallel call.

8 Conclusion

We have proposed a variant of lazy task creation (LTC) called *lazy remote procedure call* (lazy RPC) that is specifically designed for divide and conquer parallel programs and is compatible with conventional languages and C in particular. Several memory management problems have been avoided by hiding tasks and placeholders from the user and adopting a task scheduling policy that restricts tasks and placeholders to nested lifetimes. A single stack is maintained per processor and stack frames are never moved, making it easy to implement C's `&` operator. Rather than adopt an unusual stack management discipline (e.g. spaghetti stack) or move stack frames, we handle task blocking by stealing parallel calls from other processors. Lazy RPC is applicable to fine grain parallelism because the overhead of a non-stolen parallel call is very low (about 10-20 machine instructions, which compares favorably with LTC).

We have implemented a prototype compiler based on lazy RPC on a Cray T3D shared-memory multiprocessor. Experimental results with 16 processors show that lazy RPC's scheduling policy does not incur an appreciable loss of parallelism compared to LTC when there is ample parallelism in the application. The results also show that the stack space needed is reasonably small (within a factor of 2 of that required for a sequential execution of the program).

9 Acknowledgments

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. Francis L'Écuyer and Mario Latendresse helped with the implementation of the prototype compiler and the experiments.

A Brief Description of Benchmark Programs

- `queens`: Compute the number of solutions to the 12-queens problem.
- `fib`: Compute `fib(30)` using the doubly recursive algorithm.
- `fibr`: Like `fib` but the parallel call is `fib(n-2)` instead of `fib(n-1)`.
- `sum`: Compute the sum of a vector of 500000 integers.
- `nucleic`: Compute 3D structure of a 23 nucleotide section of a nucleic acid. The computation is essentially a depth first search with constraints.
- `scan`: Compute the parallel prefix sum of a vector of 131072 integers in place.
- `poly`: Compute the square of a 2000 term polynomial of x (represented as an array of coefficients) and evaluate the resulting polynomial for a certain value of x.
- `mm`: Multiply two matrices of integers (150 by 150).
- `abisort`: Sort 32768 integers using the adaptive bitonic sort algorithm [Bilardi and Nicolau, 1989].
- `tridiag`: Solve a tridiagonal system of 262143 equations.

References

[Bilardi and Nicolau, 1989] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing*, 12(2):216–228, April 1989.

[Bobrow and Wegbreit, 1973] D. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16(10):591–603, 1973.

[Callahan and Smith, 1989] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Papers from the Second Workshop on Languages and Compilers for Parallel Computing*, pages 95–113. University of Illinois at Urbana-Champaign, 1989.

[Feeley *et al.*, 1994] M. Feeley, M. Turcotte, and G. Lapalme. Using Multilisp for solving constraint satisfaction problems: an application to nucleic acid 3D structure determination. *Lisp and Symbolic Computation*, 7(2/3):231–246, 1994.

[Feeley, 1993a] M. Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University Department of Computer Science, 1993. Available as publication #869 from département d'informatique et recherche opérationnelle de l'Université de Montréal.

[Feeley, 1993b] M. Feeley. Polling efficiently on stock hardware. In *Proceedings of the 1993 ACM Conference on Functional Programming Languages and Computer Architecture*, 1993.

[Halstead, 1985] R. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[Hartel *et al.*, 1996] P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. Van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "pseudoknot" a float-intensive benchmark. To appear in *Journal of Functional Programming*, 1996.

[Katz and Weise, 1990] M. Katz and D. Weise. Continuing into the future: on the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[Mohr, 1991] E. Mohr. *Dynamic Partitioning of Parallel Lisp Programs.* PhD thesis, Yale University Department of Computer Science, October 1991.

[Nikhil, 1994] R. S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Languages and Compilers for Parallel Computing*, pages 376–390, August 1994.

[Numrich, 1994] R. W. Numrich. *The Cray T3D Address Space and How to Use It.* Cray Research Inc., 1994.

[Rogers *et al.*, 1995] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[Wagner and Calder, 1993] D. B. Wagner and B. G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.