Published in: Halstead, R., Ito T. (editors), ''Parallel Symbolic Computing: Languages, Systems, and Applications'', Springer-Verlag Lecture Notes in Computer Science 748, November, 1993, pp. 94-107.

A Message Passing Implementation of Lazy Task Creation

Marc Feeley

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

Abstract. This paper describes an implementation technique for Multilisp's `future` construct aimed at large shared-memory multiprocessors. The technique is a variant of lazy task creation. The original implementation of lazy task creation described in [Mohr, 1991] relies on efficient shared memory to distribute tasks between processors. In contrast, we propose a task distribution method based on a message passing paradigm. Its main advantages are that it is simpler to implement, has a lower cost for locally run tasks, and allows full caching of the stack on cache incoherent machines. Benchmarks on a 32 processor BBN TC2000 show that our method is more efficient than the original implementation by as much as a factor of 2.

1 Introduction

Multilisp [Halstead, 1985] extends the Scheme [IEEE Std 1178-1990, 1991] programming language with a simple and elegant parallel programming paradigm. Parallelism is specified explicitly in the program by the use of the `future` special form. The expression

(`future` *expr*)

is called a *future* and *expr* is its *body*. Parallelism follows a tasking model. When a task evaluates a future, it spawns a *child* task to evaluate the future's body. The *parent* task then starts executing the future's continuation. Thus, concurrency exists between the evaluation of the body and the continuation.

Conceptually, the value returned to the continuation is the value eventually computed for the body by the child task. This paradox is usually avoided by using a *placeholder* object to represent the body's eventual value. The placeholder is initially empty and is assigned a value (is *determined*) only when the child is done. When a placeholder is passed to a strict operation, such as `car` and the predicate position of an `if`, the placeholder must be dereferenced (*touched*) to obtain the desired value. If the placeholder is not yet determined, the current task is suspended until the placeholder is determined by the child.

In addition, Multilisp uses a shared-memory model. This means that tasks can directly access any piece of data regardless of where it was created.

Eager task creation (ETC) is a straightforward implementation of futures which has been used in several parallel Lisp systems [Halstead, 1984, Steinberg *et al.*, 1986, Goldman and Gabriel, 1988, Miller, 1988, Zorn *et al.*, 1988, Swanson *et al.*, 1988, Kranz *et al.*, 1989]. The evaluation of a future immediately creates a heap allocated task object to represent the child task and makes it available to all processors by enqueuing it on a global queue of runnable tasks: the *work queue*. Task objects are essentially composed of a continuation which represents the task's state. This continuation is initially set up to evaluate the future's body and then determine the appropriate placeholder with the result. When a processor becomes idle it removes a task from the work queue and resumes it by invoking its continuation. To reduce contention and improve locality, the work queue is usually distributed across the machine and by default processors spawn tasks and resume tasks from their local work queue. When an idle processor has an empty work queue, it must obtain the task to resume from some other processor's work queue. This task transfer between the *thief* and *victim* processors is called a *steal*. ETC is illustrated in Fig. 1 (the dark circles represent tasks).

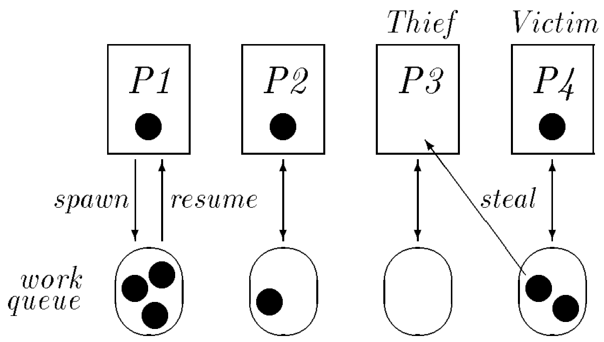

Fig. 1. Eager task creation.

The main drawback of ETC is the high task management overhead. Assuming that each task created eventually runs and terminates, the following task management operations are required for every task:

At task spawning:
1. Creating the task object and the placeholder.
2. Creating a closure for the future's body.
3. Enqueuing the task on the work queue (which requires locking and then unlocking the queue).

At task resumption:

4. Dequeuing the task from the work queue (again locking and unlocking the queue).
5. Invoking the task's continuation.

At task termination:

6. Determining the placeholder (which must also be locked and then unlocked to avoid races with tasks touching the placeholder).
7. Transferring the tasks suspended on the placeholder to the work queue.

The total cost of these operations can easily be in the hundreds of machine instructions. The performance of existing implementations of ETC seem to confirm this. The Mul-T system was carefully designed to minimize the cost of ETC [Kranz *et al.*, 1989] and it takes roughly 130 machine instructions per task on the Encore Multimax (the actual cost depends on the number of closed variables, their location etc.). Other systems have an even higher cost (both portable standard Lisp on the BBN butterfly GP1000 [Swanson *et al.*, 1988] and QLisp on an Alliant FX/8 [Goldman and Gabriel, 1988] take roughly 1400 instructions).

Due to this high cost, the overhead of exposing parallelism (i.e. of adding futures to a sequential program) will depend strongly on the task granularity. The performance of fine grain programs will be poor due to the relatively small proportion of the total time spent doing useful work.

2 Lazy Task Creation

Lazy task creation (LTC) is an alternative implementation of futures proposed in [Kranz *et al.*, 1989] and subsequently implemented and studied by Eric Mohr on an Encore Multimax [Mohr, 1991]. The implementation of LTC described in this section is essentially that used by Mohr and will be called the *shared-memory* (SM) protocol because it assumes the existence of a global shared memory. LTC reduces the cost of evaluating a future by postponing, and in many cases completely avoiding, the creation of the task. In essence, a task is only created when some other processor needs work. LTC achieves this by adopting a stack-like scheduling policy which, in the absence of idle processors, produces the same order of execution as the sequential version of the program (i.e. with the futures removed). The evaluation of the future's body is immediately started and the future's continuation, which logically corresponds to the parent task, is suspended. As shown in Fig. 2, each processor maintains its suspended continuations in a stack-like data structure called the *lazy task queue* (LTQ)[1]. When the body's evaluation is done, the most recently suspended continuation on the LTQ is removed and invoked. This is necessarily the continuation of the future corresponding to the body. Note that there is no need to create and determine a placeholder since the continuation can directly consume the value returned by the body.

In LTC, a steal is performed by first removing the oldest continuation from the victim's LTQ and then constructing the corresponding task object and placeholder, and transferring the task to the thief. For proper linkage between the stolen task and its child, the thief must invoke the stolen task's continuation

[1] This structure is really a double-ended queue which supports push, pop, and steal operations.

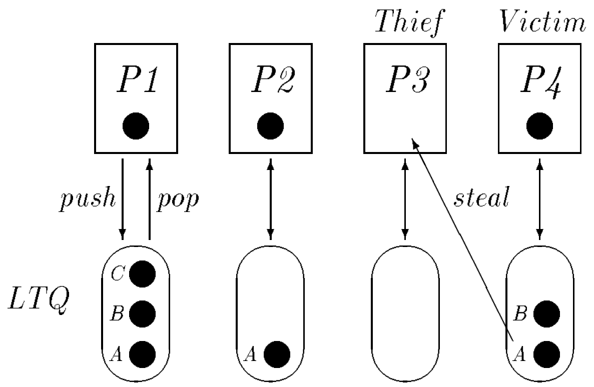

Fig. 2. Lazy task creation.

with the newly created placeholder. The placeholder must also be stored in the stolen task's child so that it can get determined correctly. Referring to Fig. 2, the placeholder created by *P3* when it steals continuation A must be "attached" to continuation B. A reference to this placeholder, which is called the child's *goal* placeholder, can be stored at a location preallocated for the victim because there is exactly one goal placeholder per LTQ. When the child's continuation (i.e. B) returns, the goal placeholder must get determined with the returned value. Note that attaching the placeholder to the child's continuation is important to allow subsequent steals.

Stealing the oldest rather than the youngest task has the following benefits:

1. It tends to reduce the number of steals because the older tasks generally contain more work (the thief will thus stay busy longer before its next steal). This is especially true in programs using divide-and-conquer parallel algorithms. If the algorithm is well balanced, the amount of work in the stolen task will be comparable to the total work left on the victim's LTQ.
2. Access to the LTQ is more efficient because the two ends of the LTQ can be accessed concurrently. The victim can push a continuation to its LTQ while a thief is simultaneously stealing a task.

The stack-like scheduling policy of LTC permits an inexpensive implementation of the continuation push and continuation pop operations. The key idea is that the continuations do not have to be copied or moved from the stack. The LTQ is really a double-ended queue of pointers into the continuation stack. Figure 3 shows the state of the stack and LTQ for *P4* before and after the steal (note that the links between the stack frames are purely conceptual). Initially, the LTQ's head and tail pointers (i.e. `HEAD` and `TAIL`) are equal and a pointer to the bottom of the stack is put under `HEAD`. A useful invariant is that a pointer to the bottom of the LTQ's oldest continuation is always under `HEAD`. The procedure linkage mechanism pushes and pops activation frames from the stack in the normal way. Each frame contains a return address to be jumped to when the frame is deallocated from the stack. When a future is evaluated, the continuation is pushed by simply pushing a pointer to the current activation frame on the LTQ (this increments the `TAIL` pointer). Popping a continuation, which is performed after the execution of the future's body, needs to be done carefully because another processor may be simultaneously stealing the same continuation. If the LTQ is not empty (i.e. `TAIL` $\neq$ `HEAD`), `TAIL` is decremented and the continuation on the stack invoked with the body's value. If the LTQ is empty, it means that the future's continuation was stolen so the body's value is used to determine the task's goal placeholder and the processor goes idle. Figure 4 gives the C-flavored pseudocode for the evaluation of `(f (future (g x)))` when the SM protocol is used. The boxed section represents a critical section that must be performed atomically with respect to the steal operation.

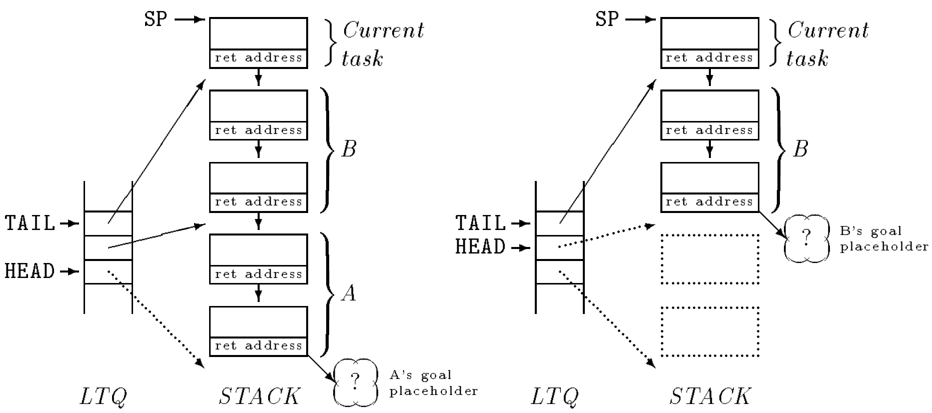

Fig. 3. The lazy task queue before and after a steal.

```
*++TAIL = SP;                   Push future's continuation
val = g( x );                   Execute future's body
if (TAIL != HEAD)
  {                             Attempt to pop the continuation
    TAIL--;
    f( val );                   Execute future's continuation
  }
else
  determine( goal, val );       Determine goal placeholder and go idle
```

Fig. 4. SM protocol's pseudocode for the evaluation of `(f (future (g x)))`.

The thief processors must similarly test the LTQ for an available continuation. If one is available the thief takes a copy of the appropriate frames directly into its stack (`HEAD[0]` and `HEAD[1]` are the boundaries of the section to copy), stores the goal placeholder with the child, and then increments `HEAD`. Before it can proceed the thief must figure out how to resume the frames it has obtained, in other words where is the code of the continuation. The code that needs to be executed is the one following the continuation pop sequence. The simplest approach is to save this address on the LTQ when the continuation is pushed. Note that this address could be computed from the return address passed to the future's body (for example, the end of the continuation pop sequence could be a constant offset away). Unfortunately, the thief has no way of finding this return address unless severe restrictions are put on the procedure calling convention, the format of frames, and the locations where return addresses can be stored[2].

The total overhead (in number of memory operations) of a non-stolen future is thus the cost of one continuation push and one successful continuation pop: 4 memory writes (2 to the LTQ and 2 to `TAIL`), 4 memory reads (1 of `HEAD` and 3 of `TAIL`), and 2 lock operations. The lock operations, which are expensive on some machines, can be avoided by using "software lock" algorithms such as [Peterson, 1981]. Algorithms specially tailored for LTC are described in [Mohr, 1991] and [Feeley, 1993a]. In addition, `TAIL` can be cached in a register since it is only mutated by the LTQ's owner. Another trick is to indicate that a continuation is no longer available by clearing the corresponding entry in the LTQ. This allows thief processors to check if the LTQ is empty by testing `HEAD[1]`$= 0$. These optimizations bring down the cost to: 3 memory writes to the LTQ and 1 memory read of `HEAD`.

Because of its special nature, the LTQ can't be used for reactivating tasks that were suspended on placeholders. For this purpose it is better if each processor has a separate ready queue that holds task objects. Both the ready queue and the LTQ must be searched by thief processors but it is preferable to check the ready queue first since less work will be required to obtain a task.

3 Hidden Cost of Sharing the Stack

An unfortunate requirement of the SM protocol is that all processors must have access to the LTQ and stack. Making these structures accessible to all processors has a cost because it precludes the use of the more efficient caching policies on machines that do not have coherent caches. The stack and the LTQ only need to be read by thief processors so they can be cached (by their owner) using the write-through caching policy. This however is not as efficient as the copy-back caching policy normally used in single processor implementations of Lisp. For typical Lisp programs, caching of the stack will likely be an important factor since the stack is one of the most intensely accessed data structures.

But how large is the performance loss due to a suboptimal caching policy? To better understand the importance of caching on performance, the memory access behavior of a few benchmark programs was analyzed (the Gabriel benchmarks were used in addition to a few medium to large sequential programs and 12

[2] For example, return addresses can't be put in a register, even temporarily, because the thief could not access them.

parallel programs). The run time of a program can be broken down into the time spent accessing data in memory and the time spent on "pure computation". Memory accesses can further be broken down into: accesses to the stack and accesses to the heap. Thus, a program is described by the three parameters S (stack), H (heap), and C (pure computation) which represent the proportion of total run time spent on each category of instructions ($S + H + C = 1$).

The measurement of S, H, and C was done by running the programs on a cache-incoherent NUMA shared-memory multiprocessor (the BBN butterfly TC2000) with the Gambit system [Feeley and Miller, 1990]. All these programs were run on a single processor. The run time of each program was measured in three different settings. The first run was with the stack and heap located in non-cached local memory. The second run was with the stack located in remote memory (on another processor) so that each access to the stack would cost more. The final run was with the heap in remote memory. The three run times are respectively T, T_S, and T_H. The relative cost of a remote access with respect to a local access is 4.2 on the TC2000 so we obtain the following system of linear equations

$$\begin{array}{llll} S & + H & + C & = 1 \\ 4.2S & + H & + C & = T_S/T \\ S & + 4.2H & + C & = T_H/T \end{array}$$

This system is easily solved to find the value of S, H, and C. The results are given in the plot in Fig. 5. Each program is a point in S-H space. The value for pure computation is $C = 1 - S - H$.

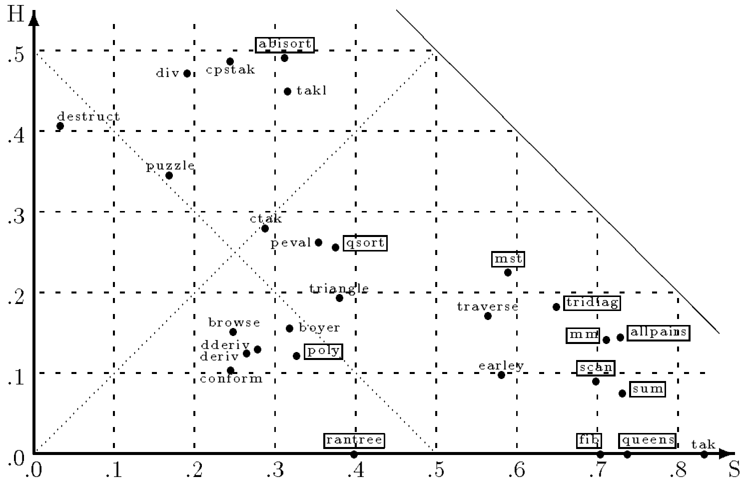

Fig. 5. Relative importance of stack and heap accesses on TC2000.

A few interesting observations can be made from this figure. Firstly, the proportion of time spent accessing memory is high. Most programs spend more time accessing memory than doing pure computation (i.e. all the programs above the $S + H = .5$ line). This is reasonable since symbolic applications typically do a lot of data movement. Secondly, most of the programs access the stack more often than the heap (i.e. all the programs below the $S = H$ line). This tendency is even more pronounced for the parallel benchmarks (the boxed names in the plot). This is to be expected since the majority of these parallel benchmarks are based on recursive divide-and-conquer algorithms.

The high S value of the programs is a sign that the access time to the stack is an important factor in overall performance.

4 The Message Passing Protocol

The message passing (MP) protocol implements LTC in such a way as to allow full caching of the stack and LTQ. In the MP protocol, the thief initiates a steal by sending a *steal request* message to the victim. The victim eventually gets interrupted and calls its *steal request handler*. This handler checks the LTQ and, if a continuation is available, recreates the oldest task and sends it back to the thief. Otherwise a failure message is sent back to the thief which must then try stealing from some other processor. The victim then resumes the interrupted computation.

There are several advantages to this protocol. Firstly, it relies less on an efficient shared memory. The stack and LTQ are private to each processor and can be cached with copy-back caching since only one processor can access them.

Secondly, it is possible to handle the race condition between the continuation pop and steal operations more efficiently than the SM protocol because all operations on the LTQ are performed by its owner. Preventing the race is as simple as inhibiting interrupts for the duration of the continuation pop sequence. This can be achieved by adding a pair of instructions around the sequence that disables and then reenables interrupts. The method used by Gambit is to detect interrupts via polling and to never poll for interrupts inside the continuation pop sequence. There are other methods that have no direct overhead. For example, in the *instruction interpretation* method [Appel, 1989] the interrupt handler checks to see if the interrupted instruction is in an "uninterruptible" section (i.e. a continuation pop sequence). If it is, the rest of the section is interpreted by the interrupt handler before the interrupt is serviced. Other zero cost techniques are described in [Feeley, 1993b].

Thirdly, it is no longer necessary to save the future's return address on the LTQ. Instead, the return address can be found in the first stack frame above the stolen continuation[3]. For this to work properly, it is important that the interrupt handler be called as a subproblem (so that the return address will have been

[3] This is done by scanning the stack upwards until the first return address is found. This assumes that return addresses are specially tagged or at least can be distinguished from other object pointers.

moved to the stack if it was in a register at the moment of the interrupt). This is fairly easy to do when the system detects interrupts through polling because the call to the handler is a subproblem call. For a system that uses hardware interrupts it is more complex but still possible.

Finally, the check for an empty LTQ in the continuation pop sequence can be avoided. The trick is to have the victim change the return address of the stolen task's child at the moment of the steal so that the child will branch directly to the right place when it returns. Locating this return address on the stack was described in the previous paragraph. In its place is put the address of a stub that determines the goal placeholder and causes the processor to go idle. The continuation pop sequence is only executed if the LTQ is not empty.

The pseudocode for the MP protocol for evaluating `(f (future (g x)))` is given in Fig. 6. The cost per non-stolen future is thus only 2 memory writes to copy-back cached memory.

```
*++TAIL = SP;     Push future's continuation
val = g( x );     Execute future's body
TAIL--;           Pop the continuation
f( val );         Execute future's continuation
```

Fig. 6. MP protocol's pseudocode for the evaluation of `(f (future (g x)))`.

5 Potential Problems

The most serious problem with the MP protocol is that the thief must busy wait for the reply to its steal request. The total time wasted, the *steal latency*, is the sum of the time needed by the victim to detect the steal request (T_{detect}) and the time to create the stolen task and send it back (T_{steal}). Little can be done to decrease T_{steal} but if interrupts are detected with polling, T_{detect} can be decreased by polling more frequently. However, this increases the cost of polling so in practice some balance must be found between these two costs.

A related problem is that the speed at which work gets distributed to the processors is dependent on the steal latency. Distributing work quickly is crucial to fully exploit the program's parallelism. It is especially important at the beginning of the program (or more precisely a transition from sequential to parallel execution) because all processors are idle except one.

Finally, the cost of failed steal requests is a concern because the victim pays a high price for getting interrupted but this serves no useful purpose. The victim might get requests at such a high rate that it does nothing else but process steal requests. For example, a continuous stream of steal requests will be received by the victim if it is executing sequential code and all other processors are idle. The problem here is that processors are too "secretive". No indication of the

LTQ's state is shared with other processors so the only way for a thief to know if the victim has some work is to send it a steal request. A simple solution is to have each processor regularly save out `HEAD` and `TAIL` in a predetermined shared memory location. Before attempting a steal, the thief checks the copy of `HEAD` and `TAIL` in shared memory to see if a task might be available. This snapshot only reflects a previous state of the LTQ but, if it is updated frequently enough, its correlation to the current state will be high. If the snapshot indicates a non-empty LTQ it is thus likely that the steal attempt will succeed. Gambit always keeps `HEAD` in shared memory so it does not need to be saved out (this does not affect performance because the victim accesses `HEAD` infrequently). `TAIL` is saved out on every interrupt poll.

6 Results

To evaluate and compare the SM and MP protocols, some experiments were conducted on the BBN butterfly TC2000. This shared-memory multiprocessor has incoherent-caches and a non-uniform memory access cost. Accesses to the cache are 3.8 times faster than to local memory and accesses to local memory are 4.2 times slower than remote memory.

Each parallel benchmark program was compiled with Gambit[4] with each protocol and then the run time was measured for several executions with 1 to 32 processors. Polling was done at a rate sufficient to make T_{detect} roughly comparable to T_{steal} (the instructions added for polling caused an average overhead of 12%). The stack and LTQ were write-through cached for the SM protocol and copy-back cached for the MP protocol. A description of the programs and some comments about their performance can be found in Appendix A.

Figure 7 contains the speedup curves for these programs. Speedup is expressed relatively to the sequential version of the program (i.e. with futures and touches removed) run with a copy-back cached stack. This means that the value of the speedup on one processor is the inverse of the overhead of exposing parallelism with futures (O_{expose}). For the MP protocol, the highest overhead (21%) is for `queens`. The overhead is much higher for the SM protocol which has a run time larger by a factor of two. This big difference is due mostly to the caching policy but, because this program is fine grained, the cost of the continuation push and pop operations is also an important factor. Note that the cache on the TC2000 is really slow when compared to the caches of modern processors (which are easily 20 to 50 times faster than main memory). We expect a much larger difference between the SM and MP protocols on future processors.

The speedup of the SM protocol is consistently lower than that of the MP protocol. For each protocol, the speedup curve starts off at $1/O_{expose}$ on 1 processor (for their respective O_{expose}) and as the number of processors increases the curves tend to get closer. Programs with good speedup characteristics (such as `fib` and `sum`) maintain a roughly constant distance between the speedup curves.

[4] A back-end generating C code was used.

In other words, the ratio of their run time stays close to the ratio of their O_{expose}. On the other hand, programs with poor speedup characteristics (e.g. `mst` and `qsort`) have speedup curves that become colinear at a high number of processors. This can be explained by the progressive increase of administrative work being performed by the program. Suboptimally caching the stack and LTQ does not affect the administrative costs. The relative importance of suboptimally caching the stack will thus decrease as the programs spend more and more time being idle and/or accessing remote memory.

A more detailed analysis of the SM and MP protocols, including experiments on a 90 processor BBN GP1000, can be found in [Feeley, 1993a].

7 Conclusion

We have proposed a message-passing protocol for implementing lazy task creation. The performance of this protocol was compared with the original protocol which relies on an efficient shared memory. Experiments on a 32 processor cache-incoherent machine show that the overhead of exposing parallelism with the future construct is typically less than 20% when using the message-passing protocol. This is much better than the shared-memory protocol which can have an overhead as high as a factor of 2. This difference is mostly due to the fact that the message-passing protocol can cache the stack in the most efficient way (copy-back caching) whereas the shared-memory protocol must use write-through caching. The experiments also indicate that the latency for detecting steal request messages is not critical. A latency comparable to the task creation cost is sufficient to get good performance.

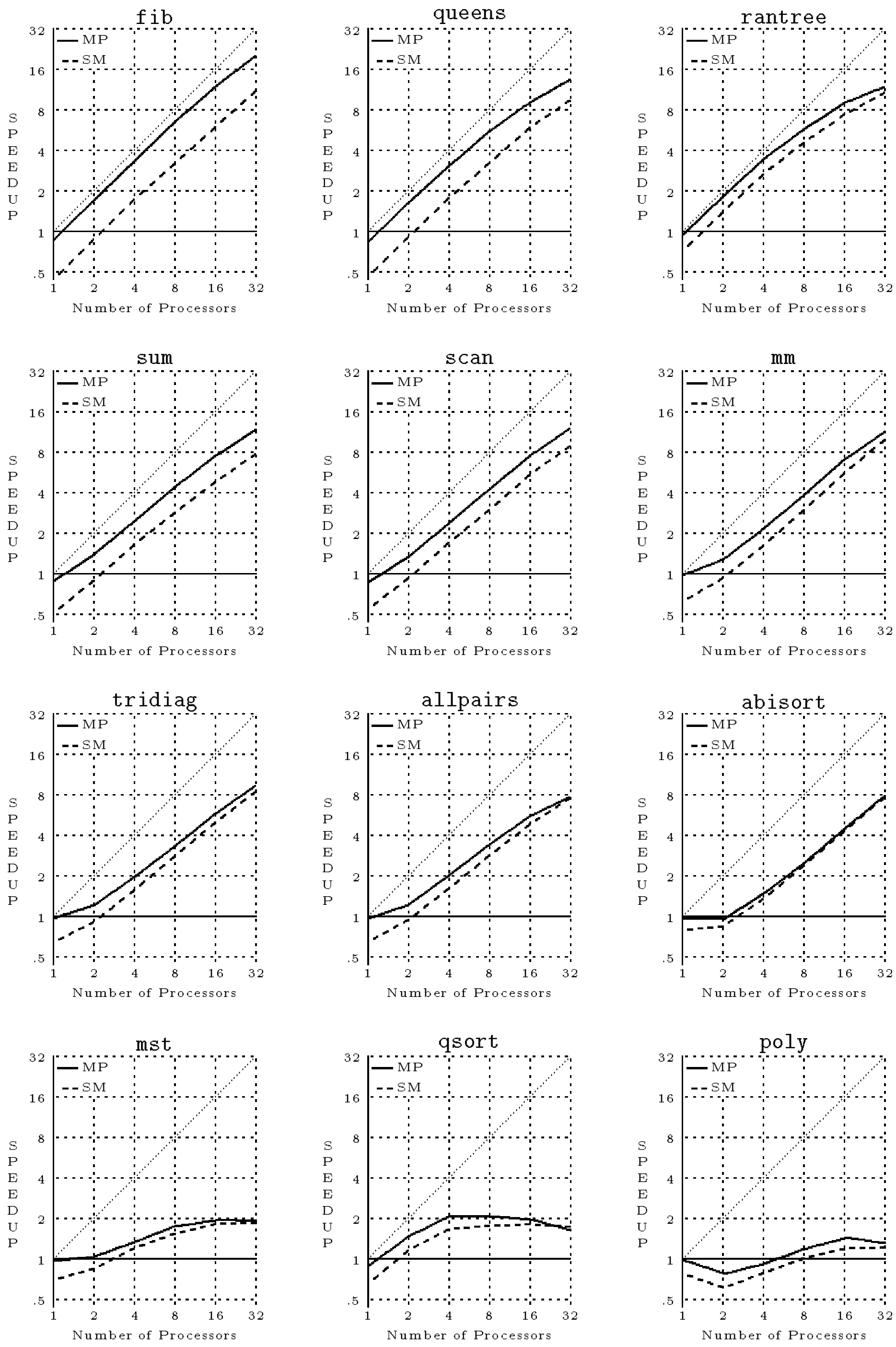

Fig. 7. Speedup curves for MP and SM protocols on TC2000.

A Description of Parallel Benchmarks

The benchmark programs can be roughly classified in three groups, according to their speedup characteristics.

1. **Parallel and compute bound.** These programs do not access memory. The speedup curve is initially close to linear speedup, and gradually distances itself from it as the number of processors increases (in other words the first derivative of the curve starts at 1 and the second derivative is negative). The flattening out of the curve as the number of processors increases is consistent with Amdahl's law.
   - `fib` – Compute `(fib 25)` using the doubly recursive algorithm.
   - `queens` – Compute the number of solutions to the 10-queens problem.
   - `rantree` – Traverse a random binary tree with 32768 nodes.
2. **Parallel and memory accessing.** These programs access memory to various extents. The speedup curves for these programs is "S" like (i.e. the second derivative is initially positive and then negative). A good example is `abisort`. The initial bend in the curve is explained by the increase in cost for accessing shared user data which is distributed evenly across the machine. A memory access has a probability of $\frac{n-1}{n}$ of being to remote memory (where n is the number of processors), so the average cost of an access to shared user data is $\frac{L+R(n-1)}{n}$, where R is the cost of a remote memory access and L is the cost of a local memory access. The bend in the curve is consequently more pronounced for programs which spend a high proportion of their time accessing the heap (e.g. `abisort`, `allpairs`, `mm`, and `tridiag`).
   - `abisort` – Sort 16384 integers using the adaptive bitonic sort algorithm [Bilardi and Nicolau, 1989].
   - `allpairs` – Compute the shortest path between all pairs of 117 nodes using a parallel version of Floyd's algorithm.
   - `mm` – Multiply two matrices of integers (50 by 50).
   - `scan` – Compute the parallel prefix sum of a vector of 32768 integers (in place).
   - `sum` – Compute the sum of a vector of 32768 integers.
   - `tridiag` – Solve a tridiagonal system of 32767 equations.
3. **Poorly parallel.** These are programs whose algorithms don't contain much parallelism or that contain a form of parallelism that is not well suited for LTC. The speedup curves for these programs are mostly flat. The curve generally starts going down after a certain number of processors because no more parallelism can be exploited but other costs, such as contention and memory interconnect traffic, increase.
   - `mst` – Compute the minimum spanning tree of a 1000 node graph using a parallel version of Prim's algorithm.
   - `poly` – Compute the square of a 200 term polynomial of x (represented as a list of coefficients) and evaluate the resulting polynomial for a certain value of x.
   - `qsort` – Sort a list of 1000 integers using a parallel version of the Quicksort algorithm.

The source code for these programs is available via anonymous FTP as the file `/pub/parallele/multilisp-bench.tar` on the FTP server `ftp.iro.umontreal.ca`.

References

[Appel, 1989] A. W. Appel. Allocation without locking. *Software Practice and Experience*, 19(7):703–705, July 1989.

[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.

[Feeley and Miller, 1990] M. Feeley and J. S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[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.

[Goldman and Gabriel, 1988] R. Goldman and R. P. Gabriel. Preliminary results with the initial implementation of Qlisp. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 143–152, Snowbird, UT, July 1988.

[Halstead, 1984] R. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, TX, August 1984.

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

[IEEE Std 1178-1990, 1991] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[Kranz *et al.*, 1989] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 81–90, June 1989.

[Miller, 1988] J. S. Miller. Implementing a Scheme-based parallel processing system. *International Journal of Parallel Processing*, 17(5), October 1988.

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

[Peterson, 1981] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[Steinberg *et al.*, 1986] S. Steinberg, D. Allen, L. Bagnall, and C. Scott. The Butterfly Lisp system. In *Proc. 1986 AAAI*, volume 2, Philadelphia, PA, August 1986.

[Swanson *et al.*, 1988] M. Swanson, R. Kessler, and G. Lindstrom. An implementation of portable standard Lisp on the BBN Butterfly. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 132–141, Snowbird, UT, July 1988.

[Zorn *et al.*, 1988] B. Zorn, P. Hilfinger, K. Ho, J. Larus, and L. Semenzato. Features for multiprocessing in SPUR Lisp. Technical Report Report UCB/CSD 88/406, University of California, Computer Science Division (EECS), March 1988.

This article was processed using the LaTeX macro package with LLNCS style