Parallel List Mapping Operations

Ian A. Mason, Joseph D. Pehoushek, Carolyn Talcott, Joseph S. Weening
Stanford University*

Abstract

One approach to producing tools for parallel Lisp computation is to design parallel versions of standard Lisp tools such as iteration, mapping, and loop constructs. Although these constructs appear sequential there are many applications in which sequentiality is not essential—the construct is merely used as a means of expressing that an action or function must be applied to each element of a list or each number in some range, possibly accumulating the results in some manner. Parallel versions of such tools would allow programmers to write parallel programs using familiar constructs, with little concern about synchronization, scheduling, or granularity.

In this paper we present parallel versions of the Common Lisp mapping functions `mapc`, `mapl`, `mapcar`, `mapcan`, `mapcon`, and `maplist`. The central idea involves dynamically and recursively subdividing the list, while minimizing idle time and process creation time. Both theory and experiment show that our method permits speedup which is limited only by Amdahl's Law, or by the number of processors, whichever is less.

1 Introduction

Qlisp is a parallel extension of Common Lisp. The original design is due to Gabriel and McCarthy [1]. The basic model of computation is shared-memory, queue-based (the 'Q' in Qlisp) multi-processing. When tasks are created they are put in a queue and when a processor is idle it removes a task from the queue and executes it. The basic premise of the Qlisp programming model is that the decision of whether or not to create parallel tasks should in general be done at run time, not at compile time, and should depend on parameters of the problem being solved. The goal is to create enough tasks to keep available processors busy, but to avoid flooding the system with waiting tasks. The initial idea was that parallelism would be controlled by parameters that corresponded to problem size. Parallel subtasks would be created until some cutoff point was reached; for example, if recursion has reached some cut-off depth, or the remaining tasks are smaller that some cut-off size, then tasks are no longer created. It was expected that a large factor (a thousand or even a million) in number of tasks spawned could be tolerated before seriously degrading performance.

However, experiments showed that finding cut-off parameters and tuning was much trickier than expected, especially in a complex program that could be part of a larger parallel system. Although the cut-off method does work for special cases, it is not suitable

*Research supported by Defense Advanced Research Projects Agency contract N00039-84-C-0211

for building components for use in arbitrary applications. In the process of trying to discover good cut-off parameters, an alternative approach called "dynamic partitioning" was discovered [4, 5].

On tree-like computations, the dynamic partitioning approach has proven highly successful in terms of speedup, while remaining relatively insensitive to tuning details. The basic idea is simple: the decision of whether or not to create parallel tasks is based on the number of tasks waiting in the local queue (each processor has its "own" task queue), and not on program parameters. The result is a relatively small amount of idle time and process creation time.

Dynamic partitioning has been used to parallelize several Lisp constructs. In this paper we demonstrate the use of dynamic partitioning to implement parallel list mapping operations.

2 Parallel mapping functions

Given a list mapping operation whose iterations are independent computations, our goal is to execute this computation as efficiently as possible on a shared-memory multiprocessor. The following variables characterize our parallel machine. Those that represent time are all multiples of some basic time unit, whose exact value is not important.

p is the number of processors.

s is the amount of time needed to create ("spawn") a process.

d is the amount of time needed to evaluate the `cdr` function.

We use the following to describe a specific instance of the use of a mapping function.

n is the length of the list being mapped over.

c is the time needed to apply the function to each list element. (We assume c is constant in this paper.)

The above descriptions combine some of the primitive operations needed to evaluate a mapping function. All of the work done in stepping from each iteration to the next (testing for the end of the list, calling `cdr`, and whatever else is needed) is subsumed in the parameter d, and all of the work needed to create and schedule a process is contained in s.

Our potential speedup is limited by Amdahl's Law, which predicts a maximum speedup on any parallel program based on its inherently sequential component. In our case, the list data structure requires n `cdr` operations to be performed in order, since each cons cell contains the pointer to the next one. So the minimum time for any parallel mapping function is nd, the time needed to perform the n `cdr` operations.

A straightforward sequential version of the mapping function takes time $n(c+d)$, since we perform one function application on each element of the list, and step from each element to the next. Therefore Amdahl's Law limits the speedup to

$$\frac{T_{seq}}{\min T_{par}} = \frac{n(c+d)}{nd} = \frac{c+d}{d}.$$

Unless we change the list data structure to something else, there is no way to overcome this limitation.

```
(defun qmapa (fn list)
  (if (null list)
      nil
      (qvalues (funcall fn (car list))
               (qmapa fn (cdr list)))))
```

Figure 1: The function `qmapa`.

A simple way to parallelize the computation is shown in the function `qmapa`. Using the Qlisp form `qvalues`, which creates processes for each of its argument forms, waits for them to finish, and returns their values, the main loop of this function creates a new process for each iteration of the loop; this process will perform the c units of work required to apply the function to one element of the list. Even if enough processors are available to handle the processes that are created, the minimum time for `qmapa` is $n(s+d)$, and by the argument above, its maximum speedup is now $(c+d)/(s+d)$ instead of $(c+d)/d$. If the spawning time s is large, this is a significant loss.

The function `qmapa` has other problems. If there are not enough processors to handle all of the processes as they are created, then proper scheduling of the processes becomes important. Also, the amount of memory needed to hold data structures describing the waiting processes can become a serious obstacle.

Our experience in Qlisp programming has shown that programs that work by top-down recursive splitting (such as the Quicksort algorithm for sorting) are easy to parallelize. Such computations can be viewed as a tree of processes, where the root represents the entire computation, and each process's children are subcomputations that may be executed in parallel. We have studied in some depth the particular case where each node in the tree has two children, the work performed at each node is roughly constant, and a "dynamic partioning" method is used to avoid creating many more processes than are necessary to keep the parallel machine busy [5].

Dynamic partitioning, in its simplest form, uses a separate queue of processes for each of the p processors. When the program allows a new process to be created, a processor does so only if its own queue is empty, as indicated by the function `dynamic-spawn-p`. Processes are inserted only into a processor's own queue. When it is idle, a processor first tries to take work from its own queue; if the queue is empty, it cycles among the other processors' queues, removing a process from the first non-empty one that it finds. If there are p processors and the computation tree has height h, this results in $O(p^2h^4)$ processes being created.

Function `qmapb` uses a modified divide-and-conquer method, dividing only when it spawns a process. Initially, `qmapb` computes the length of the list n. It is the job of the inner function `map-loop` to perform the actual calls to the function being mapped, as well as to check to see if it is reasonable to split the task into two equal sub-tasks. The answer to the latter question is provided by a call to `dynamic-spawn-p`. This predicate returns `T` if the local task queue (the current processor's queue of things to do) is empty, and `NIL` otherwise.

```
(defun qmapb (fn list)
  (labels
      ((map-loop (k list)
         (cond ((or (null list) (= k 0))
                nil)
               ((not (dynamic-spawn-p))
                (funcall fn (car list))
                (map-loop (1- k) (cdr list)))
               ((= k 1)
                (funcall fn (car list)))
               (T (let ((k2 (halve k)))
                    (qvalues  (map-loop k2 list)
                              (map-loop (- k k2)
                                        (nthcdr k2 list))))))))
    (map-loop (length list) list))
  list)

(defun halve (k) (ash k -1))
(defun double (k) (ash k 1))
```

Figure 2: The function `qmapb`.

When the predicate causes a partition, the algorithm divides the list into two parts of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, creates subprocesses to perform the mapping function on these sublists, and combines the results of these processes. There is also a test prior to spawning, insuring non-trivial processes.

Dynamic partitioning applied to `qmapb` yields a significant reduction in the overhead due to process spawning, compared to `qmapa`, which spawned n processes. The height h of the computation tree is $O(\log n)$, so for a fixed number of processors p, the number of processes spawned is at most $O(\log^4 n)$, in the worst case, using the analytical result previously mentioned. In practice, the average number of spawns is $O(\log^2 n)$, but in either case, this function grows much more slowly than n.

However, there is still a problem—idle time. We divide idle time into three components.

- At the beginning of the computation, only one processor is busy. Other processors remain idle until enough processes have been created to make them busy.

- Once all of the processors become busy, the machine reaches a "steady state" where there is very little idle time. (This is true for the algorithms we are describing, but it is not true in general for all programs.)

- The steady state ends when the computation has passed the point when any new processes can be created, and all of the queues used by the dynamic scheduler are empty. Then, once a processor becomes idle it remains idle for the rest of the computation. This is because no new process can be created for it, and whenever another processor finishes a process, allowing its parent to resume, that processor is available to run the parent.

Of the three components of idle time in `qmapb`, the first is the most significant. To compute the length of the list requires n `cdr` operations, which takes time nd. All of this is done on one processor, while the others wait, since this cannot be parallelized. Additionally, the time until all p processors are busy is $O(nd \log p)$, due to the large number of calls to `cdr` near the beginning of the computation. Even if the rest of the computation is done in the fastest possible time, which we observed above to also be nd, the minimum time for the parallel algorithm is at least $n(2d + d \log p)$, and hence the potential speedup is less than half of the limit imposed by Amdahl's Law.

The idle time at the end is not as large. During the "steady state" period, all p processors remain busy. (Here we assume that p is not more than $(c+d)/d$, the speedup limit imposed by Amdahl's Law.) As long as some of the processes are performing the mapping operation on lists of length greater than 1, the steady state continues, since such processes can be partitioned whenever needed to provide work for a processor that has become idle. After the steady state period, therefore, all processors are either idle, are applying the function to lists of length 1, or are combining the results of subcomputations.

Only c time units (a constant number) can be spent in finishing the work on lists of length 1. The combination of subcomputations takes time proportional to the height of the computation tree, which is $O(\log n)$. Therefore the idle time at the end of the computation is $O(\log n)$. As n increases, this becomes insignificant compared to both the idle time at the beginning (which is at least nd) and the overall runtime (at least n/p).

The function `qmapb` eliminated one obstacle to achieving the optimal speedup given by Amdahl's Law, namely the overhead of process creation, but the excessive idle time at the beginning of the computation still stands in the way. We now describe an improved function `qmapc` that reduces this idle time.

Rather than precompute the length, n, of the list, we use a parameter k as an initial estimate, and divide the work into two tasks. The first task applies the function to the first k elements of the list, while the second task is a recursive call with the length estimate k doubled. The repeated doubling of k insures that the end of the list is reached after $\log n$ tasks have been spawned. This virtually eliminates the idle time at the beginning of the computation (assuming the initial value of k is small). However it does not insure that the machine reaches a steady state, in particular the last task spawned is as large as all the others combined. By using the dynamic partioning method within each of these $\log n$ tasks we can insure that a steady state is reached, and maintained as long as possible. Each of these tasks divides into equal sized subtasks whenever the dynamic partitioning predicate is true. While the predicate is false each task simply performs the desired mapping operations.

We begin by describing the simpler non-value accumulating mapping functionals `qmapc` and `qmapl`, concentrating on the former for ease of exposition. The `qmapc` program has two local functions `map-loop` and `map-rest`. The function `map-rest` spawns the first $\log n$ tasks. Each of these tasks consists of a call to the second local function `map-loop`, which is identical to `map-loop` in `qmapb`. In this version of the program we take 1 to be our initial estimate of the length of the list to be processed.

The `qmapc` function is written using `macrolet` to capture the uniformities between this function and the related function `qmapl`. The definition of `qmapl` is obtained by modifying the macro `*map-apply*` so that it expands to `(funcall fn list)`.

```
(defun qmapc (fn list)
  (macrolet
      ((*map-apply* (fn list) `(funcall ,fn (car ,list))))
    (labels
        ((map-loop (k list)
           (cond ((or (null list) (= k 0))
                  nil)
                 ((not (dynamic-spawn-p))
                  (*map-apply* fn list)
                  (map-loop (1- k) (cdr list)))
                 ((= k 1)
                  (*map-apply* fn list))
                 (T (let ((k2 (halve k)))
                      (qvalues  (map-loop k2 list)
                                (map-loop (- k k2)
                                          (nthcdr k2 list)))))))
         (map-rest (k list)
           (when list
             (qvalues (map-loop k list)
                      (map-rest (double k)
                                (nthcdr k list))))))
      (map-rest 1 list)))
  list)
```

Figure 3: The function qmapc.

```
(defun list-2-cycle (list)
  (when list
    (let ((cycle (last list))) (setf (cdr cycle) list) cycle)))

(defun cycle-2-list (cycle)
  (when cycle
    (let ((first-cell (cdr cycle))) (setf (cdr cycle) nil) first-cell)))
```

Figure 4: The functions list-2-cycle and cycle-2-list.

To extend this technique to the value returning mapping functionals, `mapcar`, `mapcan`, `mapcon` and `maplist`, we need to accumulate and pass along the values of the respective calls to the function. To do this efficiently we use cyclic lists in the following way. Rather than have `map-loop` return the list of accumulated values that would then have to be `cdr`-ed down to be attached to the remaining result. The program `map-loop` is written so as to return the last cell in this list, modified so that the `cdr` points to the first cell of the list. We shall call such a cyclic representation (or modification) of a list a cycle. The transformations from lists to cycles, `list-2-cycle`, and from cycles to lists, `cycle-2-list`, explicitly explains this representation.

The functions `map-loop` and `map-rest` are modified so as to return cycles, which in the case of `map-loop` entails adding a new argument, `cycle`, representing the cycle upto the current point in the loop. This also entails that the cycles returned by spawned tasks must be remembered and linked together. This linking is performed by the function `link-cycles`. It takes two cycles as arguments and links them together to form a third cycle. The resulting cycle encodes the list obtained by `nconc`-ing the list encoded by the first cycle onto the list encoded by the second cycle. In otherwords a call to `(link-cycles cycle-1 cycle-2)` is equivalent to a call to `(list-2-cycle (nconc (cycle-2-list cycle-1) (cycle-2-list cycle-2)))`.

Similarly when `map-loop` applies the function to the appropriate argument it must splice the resulting list into the cycle accumulated so far, i.e. the value of `cycle`. This is accomplished by the program `splice-cycle` which takes a cycle, and a list and returns the same cycle that would result from a call to `(link-cycles cycle-1 (list-2-cycle list))`.

These modifications result in the function `qmapcar`. Again the actual program is written using `macrolet` so as to capture the uniformities between this program, `mapcar`, and its sister programs `mapcan`, `maplist` and `mapcon` whose definitions are obtained by modifying the macro `*map-apply*` suitably. In particular for `mapcan` the macro definition expands to `(splice-cycle cycle (funcall fn (car list)))`, for `maplist` it expands to `(splice-cycle cycle (cons (funcall fn list)))`, and for `mapcan` it expands to `(splice-cycle cycle (funcall fn list))`.

3 Analysis of `qmapc`

The function `qmapc` outperforms `qmapb` in several respects. Here we will show that the idle time at the beginning of the computation, which was the main source of overhead in `qmapb`, becomes negligible as n increases.

The key idea is to show that enough work to keep p processors busy is found in $O(p \log p)$ time, instead of the $O(n \log p)$ that we needed for `qmapb`. If the lowest-level processes are large enough, the first p iterations of the function provide this work, and our method of doubling the process size at the beginning of the computation ensures that these processes are created in $O(p \log p)$ time.

It may happen that some of the initial processes finish before the steady state is reached, and in that case the initial idle time is longer. Eventually, though, the doubling of the segment size produces a process large enough so that all p processors remain busy while the beginning of the next segment is found. The size of this segment is some constant multiple of p, so the time needed to reach it is $O(p)$, and the time to partition it into p processes is

```
(defun qmapcar (fn list)
  (macrolet
      ((*map-apply* (fn list cycle)
         `(splice-cycle ,cycle
           (cons (funcall ,fn (car ,list)) nil))))
    (labels
        ((map-loop (k list cycle)
           (cond ((or (null list) (= k 0)) cycle)
                 ((not (dynamic-spawn-p))
                  (map-loop (1- k)
                            (cdr list)
                            (*map-apply* fn list cycle)))
                 ((= k 1) (*map-apply* fn list cycle))
                 (T
                  (let ((k2 (halve k)))
                    (multiple-value-bind (second third)
                        (qvalues  (map-loop k2 list nil)
                                  (map-loop (- k k2)
                                            (nthcdr k2 list)
                                            nil))
                      (link-cycles cycle
                                   (link-cycles second third))))))))
         (map-rest (k list)
           (when list
             (multiple-value-bind (first second)
                 (qvalues (map-loop k list nil)
                          (map-rest (double k) (nthcdr k list)))
               (link-cycles first second)))))
      (cycle-2-list (map-rest 1 list)))))
```

Figure 5: The function `qmapcar`.

$O(p \log p)$. This is the initial idle time of the computation.

For small values of n, the input list may be exhausted before the situation described above holds. The result is therefore true asymptotically as n increases. In the next section, our experimental results show how large n needs to be as a function of the work performed in each iteration of the mapping function.

4 Experimental results

Our experiments were done with an implementation of Qlisp, based on Lucid Common Lisp, running on an Alliant FX/8, a shared-memory multiprocessor with eight processors. The parallelism features of Qlisp were developed at both Lucid and Stanford.

In the graphs below, we present the speedup of `qmapc` over `mapc`. The other mapping functions have similar speedup graphs. Each experiment consisted of mapping the function

```
(defun work (m) (if (<= m 0) 0 (work (1- m))))
```

over a list containing n copies of a number m. Thus m represents the granularity and n the problem size for a more general list mapping operation.

In the graphs below, the x axis indicates the length of the list, and the y axis is the speedup of `qmapc` compared to four measures of the sequential time, to indicate how much work is spent in various overhead activities.

The dark circle is the ratio of the time for a pure sequential `mapc` to the time for `qmapc`. This is the "true speedup" achieved by our program. The light circle is the ratio of `qmapc` running on one processor to `qmapc` running on eight processors. This includes no process creation time (since the dynamic partitioning method will avoid almost all spawning on one processor), but includes testing the partitioning predicate and maintaining the counter `k`.

The dark triangle adds to the light circle the time spent in process creation and scheduling when `qmapc` is run on eight processors. The light triangle adds all additional overhead except idle time; it was computed by subtracting measured idle time from the time of `qmapc` on eight processors. The difference between the dark and light triangles can be attributed to extra work done by the parallel program, i.e. extra calls to `cdr` that are not done on a single processor.

When the work done per element is large, optimality is approached fairly quickly. In the smallest case, `(work 0)`, 1000 elements is not enough to display a clear asymptote; from running large examples (lists of 1 or 2 million elements) we know that the graph for `(work 0)` has only reached about half of its best possible speedup, which is at least 5.0. Note that the speedup for some small lists appears to be zero, implying infinite slowdown. The reason for this is that `mapc` took less than 1 millisecond to do the given computation, and so its time was recorded as 0 milliseconds.

As part of the experimental data, we include the following table of times for basic operations. Most of these times were deduced by looping over a simple operation many times.

Operation	μsecs	Operation	μsecs
(setf s i)	0.9	(incf s)	1.9
car, cdr	1.2	funcall	9.0
(null X)	1.2	cons	12.5
dynamic-spawn-p	1.2	create task	20.0

References

[1] Richard P. Gabriel and John McCarthy. *Queue-based multiprocessing Lisp.* In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pages 25–44, Austin, Texas, August 1984.

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

[3] Takayasu Ito and Manabu Matsui. *A parallel language PaiLisp and its kernel specification.* Proceedings of the US/Japan Workshop on Parallel Lisp, Sendai, Japan, June 1989.

[4] Joseph D. Pehoushek and Joseph S. Weening. *Dynamic Partitioning in Parallel Lisp.* Computer Science Technical Report to appear. Stanford University, Stanford, CA, 1990.

[5] Joseph S. Weening. *Parallel Execution of Lisp Programs.* Computer Science Technical Report STAN-CS-89-1265, Stanford University, Stanford, CA, June 1989.