February 1989

Report No. STAN-CS-89-1247

(2)

Programming in Qlisp - A Case Study

by

Arkady Rabinov and Igor Rivin

Department of Computer Science

Stanford University

Stanford, California 94305

DARPA INSERT SHEET

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION: unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release: Distribution Unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): STAN-CS-89-1247	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Computer Science Department; 6b OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Stanford University, Stanford, CA 94305	7b. ADDRESS (City, State, and ZIP Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION: DARPA; 8b OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): 1400 Wilson Blvd., Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification): Programming in Qlisp - A Case Study

12 PERSONAL AUTHOR(S): A. Rabinov and I. Rivin

13a TYPE OF REPORT	13b TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

We describe the results of some experiments with Qlisp—an extension of Common Lisp for shared memory multiprocessors. The experiments involved several parallel implementations of the modular univariate polynomial greatest common divisor algorithm in Qlisp on an Alliant FX/8 multiprocessor. These implementations are described and the requisite Qlisp constructs are described and explained (largely by example). The performance of the parallel implementations is analyzed and some areas of future improvement in the current Qlisp implementation and the algorithm are found.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	
22a NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Programming in Qlisp—A Case Study

Arkady Rabinov* Igor Rivin*

Abstract

We describe the results of some experiments with Qlisp—an extension of Common Lisp for shared memory multiprocessors. The experiments involved several parallel implementations of the modular univariate polynomial greatest common divisor algorithm in Qlisp on an Alliant FX/8 multiprocessor. These implementations are described and the requisite Qlisp constructs are described and explained (largely by example). The performance of the parallel implementations is analyzed and some areas of future improvement in the current Qlisp implementation and the algorithm are found.

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

1 Introduction

Qlisp is an extension of Common Lisp by some parallel programming constructs. (See [2] for a detailed description.) The work described in this paper was done with an experimental implementation of Qlisp on an Alliant FX/8 computer.

Since Qlisp is designed to facilitate symbolic programming on parallel computers, and computer algebra is one of the oldest and best understood areas of "symbol processing", we decided to implement a polynomial greatest common divisor (GCD) algorithm based on modular arithmetic to test the expressive power of the language as well as the efficiency of its current implementation.

The plan of this paper is as follows: In Section 2 we briefly describe the Qlisp features we use. In Section 3 we describe the algorithm for calculating GCD. In Section 4 we describe four different ways of making the algorithm parallel. In Section 5 we give the results of timing measurements and explain them.

2 A Qlisp Overview

Qlisp provides syntactic constructs to facilitate run-time parallelism. The primary parallelism construct in Qlisp is `qlet`. This construct is similar to `let` in Common Lisp. Its form is

```
(qlet prop ((x1 arg1)
            ...
           (xn argn))
     body)
```

where *prop* is a proposition evaluated at runtime to control parallelism. If its value is `NIL`, then no parallelism is created and `qlet` turns into a simple `let`: the arguments $arg_1, \ldots, arg_n$ are evaluated, their values are bound to $x_1, \ldots, x_n$ and *body* is evaluated.

If the value of *prop* is the symbol `'eager`, then processes to evaluate $arg_1, \ldots, arg_n$ are spawned, $x_1, \ldots, x_n$ are bound to placeholders for the expected values of $arg_1, \ldots, arg_n$ and a process to evaluate *body* is spawned. If in the process of evaluating *body* the value of some x_i is required, the process evaluating *body* blocks and waits for the process evaluating arg_i to return a value.

If the value of *prop* is anything but `NIL` or `'eager` then $arg_1, \ldots, arg_n$ are evaluated in parallel. After $arg_1, \ldots, arg_n$ are evaluated, their values are bound to $x_1, \ldots, x_n$, and the *body* is evaluated.

prop can be used by the programmer to dynamically control (during evaluation time) whether the forms $arg_1, \ldots, arg_n$ will be evaluated in parallel or serially. Since parallelism incurs a certain amount of overhead, it could be very important to dynamically restrict the growth of the number of parallel processes.

The next construct that Qlisp adds to Common Lisp is `qlambda`. Its syntax is similar to that of the `lambda` in Common Lisp and, like `qlet`, it has the additional form *prop* controlling parallelism associated with `qlambda`. The form of `qlambda` is:

(qlambda *prop* **(***lambda-list***)** *body***)**

This construction has three separate properties:

- It creates a lexical closure (like the `lambda` construct of Common Lisp);
- It is a monitor, or a critical region—when parallel parts of a program try to evaluate more then one copy of a particular `qlambda` simultaneously, the language guarantees that only one will be evaluated, and the evaluation of the next copy will not start until the evaluation of the current copy is finished;
- It is a unit of parallel evaluation; the form `nowait` should be used to force the parallel evaluation of a particular call to `qlambda`.

If *prop* is non-`NIL` a process is spawned to evaluate the `qlambda` form.

A convenient primitive is `spawn`. (`spawn` *form*) creates a process to evaluate *form*. The value computed is ignored, so `spawn` is only used for effect.

Qlisp also provides low level synchronization primitives by way of events and locks and functionality to create, acquire, release and test locks and to create, signal and wait for events. Events can also be used as counting semaphores: a process can wait for a particular event being signaled a certain number of times.

3 GCD computation algorithm

Knuth [3] noticed that the "inherent parallelism of modular arithmetic" lends itself to use in parallel computers. We use modular arithmetic to implement polynomial GCDs (for polynomials in one variable).

Below we use the following (standard) notation: If R is a ring then the ring of polynomials in the transcendental x over R is denoted by $R[x]$. Z denotes the ring of integers. Z/p denotes the quotient of Z by the ideal pZ, where p is a prime.

If P_1 and P_2 are elements of $R[x]$ (where R is a ring), then their *greatest common divisor* (GCD for short) is $Q \in R[x]$ such that Q divides both P_1 and P_2 and any $S \in R[x]$ that divides P_1 and P_2 divides Q also. If $R = Z$ then polynomial GCD is determined uniquely, while if R is a field, GCD is determined only up to multiplication by elements of R^* (the multiplicative group of R).

Let $P_1, P_2 \in Z[x]$. Let $\phi(p) : Z \to Z/p$ be the homomorphism which sends an integer to its equivalence class modulo p, and let $\Phi(p) : Z[x] \to Z/p[x]$ be the induced homomorphism on polynomial rings, which sends $\sum a_i x^i$ to $\sum \phi(p)(a_i)x^i$. The algorithm makes use of the fact that $\mathrm{GCD}(\Phi(p)(P_1), \Phi(p)(P_2)) = \Phi(p)(\mathrm{GCD}(P_1, P_2))$ for *almost* all primes p, assuming that we normalize the leading coefficients of the various GCDs to be the GCD of the leading coefficients of the input polynomials.

One of the problems with using modular arithmetic is that it is not trivial to divide in Z/p. There are two basic different ways of doing this. The first uses Fermat's Little Theorem. (For any integer a and a prime p, $a^p \equiv a \pmod{p}$. Thus $a^{-1} \equiv a^{p-2} \pmod{p}$.) The second approach uses the fact that if $x \equiv a^{-1} \pmod{p}$ then x is the solution of

the Diophantine equation $xa + kp = 1$, which can be solved using some form of Euclid's algorithm. The second way is known to be considerably faster for reasonably sized primes (see [4]), so that's what we use. We use the optimized version of Euclid's algorithm which uses pseudo-remainder sequences ([3]).

Once all the modular GCDs we need are computed we use the Chinese Remainder Theorem to interpolate the GCD of P_1 and P_2 over the integers.

Here is the algorithm (in pseudo-Lisp):

```
(defun poly-gcd (Poly1 Poly2)
  (let ((Bound (gcd-coefficient-bound Poly1 Poly2))
        ;;coefficients of (poly-gcd Poly1 Poly2) will not exceed this
        ;;bound.
        (Avoid (gcd (lead-coeff Poly1)(lead-coeff Poly2)))
        ;;want to Avoid primes dividing the leading coefficients of
        ;;Poly1 and Poly2
        )
    (do ((result? (main-loop Poly1 Poly2 Bound Avoid)
                  (main-loop Poly1 Poly2 Bound Avoid)))
        ((and (divides? result? Poly1)(divides result? Poly2))
         ;;reality check
         result?))))

(defun main-loop (Poly1 Poly2 Bound Avoid)
  (do* ((prime (find-prime Avoid)(find-prime Avoid))
        ;See comment above
        (mod-gcd (mod-gcd Poly1 Poly2 prime)(mod-gcd Poly1 Poly2
                                                     prime))
        (how-big prime (* how-big prime))
        ;our coefficients are correct modulo how-big
        (result mod-gcd (if (< (degree mod-gcd) (degree result))
                            (and (setq how-big (quotient how-big
                                                         prime))
                                 result)
                            (CRT result how-big mod-gcd prime))
                ;See whether PRIME is a bad prime, if so throw it out,
                ;otherwise update the candidate GCD by Chinese
                ;remaindering.
                ))
       ((> how-big Bound) result)
    ;Have we done enough yet?
    ))
```

4 How to Do It in Parallel

Here is the abbreviated first version of the program based on qlet. We will assume that the input polynomials are fixed, so we don't pass them as arguments.

```
(defun gcd1 (primes)
  (let ((prime (car primes)))
    (qlet t
        ((done (gcd1 (cdr primes)))
         (cur-gcd (mod-gcd prime)))
      (chinese cur-gcd done))))
```

In the above we request that the following two computations be done in parallel:

- the calculation of GCD modulo some prime p (we will call it $\texttt{GCD}_p$) by `mod-gcd`;
- the calculation of GCD for the rest of the primes and the ongoing reconstitution of the final answer via the Chinese Remainder Theorem (CRT).

After this is done, we apply CRT to our current $\texttt{GCD}_p$ and to the result of the recursive call to `gcd1` (this result is already lifted to $Z/\prod_{p\in\texttt{(cdr primes)}} p$). It is obvious that while calculations of all $\texttt{GCD}_p$ are done in parallel, each application of CRT starts only after the previous application is finished, and, moreover, the first application of CRT is started only after the last $\texttt{GCD}_p$ is calculated.

How can we improve this? We would like to arrange things in such a manner that after the process finishes with calculating $\texttt{GCD}_p$, it will immediately start applying CRT to this $\texttt{GCD}_p$ and to the result of the previous application of CRT. This previous result has to be passed from the previous invocation of the function GCD1 (see above). On the other hand, this result is not ready yet when the previous invocation of GCD1 calls next the GCD1. To solve this problem, GCD1 passes to its child GCD1 a lambda expression which knows how to retrieve the result of the previous application of CRT. We still need to do something for synchronization: we can start the calculation of $\texttt{GCD}_p$ any time, but the application of CRT must wait till the previous result being prepared by the parent GCD1 is ready. The `qlambda` construct helps us here. `qlambda` is a unit of computation, a monitor and a lexical closure. We use all of these features in a second version of GCD (the code is abbreviated):

```
(defun gcd2 (retriever primes)
  (let ((prime (car primes)))
    (let* (cur-gcd
           (cur-gcd-handler
             (qlambda t (option)
                (if (eq option 'retrieve)
                    cur-gcd
                    (setq cur-gcd (calc-gcd retriever prime))))))
      (nowait
        (funcall cur-gcd-handler 'calculate))
      (new-gcd1 cur-gcd-handler (cdr primes) pr-prod))))
```

For the purpose of synchronization, we want to use the same `qlambda` for calculating `cur-gcd` and pass it to the child for the purpose of retrieval of `cur-gcd`. `retriever` here is an analogous `qlambda` passed by our parent; we pass it to `calc-gcd`, so that it will use it to retrieve the previous result after it calculates $\texttt{GCD}_p$, and before the application of CRT. It

is important that our child be unable to apply qlambda that we pass to it, before we apply it.

Here is the function calc-gcd:

```
(defun calc-gcd (retriever prime)
  (let ((cur-gcd (mod-gcd  prime)))
    (let ((prev-gcd  (funcall retriever 'retrieve)))
      (chinese cur-gcd prev-gcd))))
```

the same algorithm can be easily implemented using the future construct:

```
(defun gcd2 (prev-gcd primes)
  (gcd2 (future (chinese prev-gcd (future (mod-gcd (car primes)))))
        (cdr primes)))
```

And below gcd2 is implemented using qlet 'eager.

```
(defun gcd2 (prev-gcd primes)
  (qlet 'eager ((cur-gcd (mod-gcd (car primes))))
        (qlet 'eager ((cur-chinese (chinese cur-gcd prev-gcd)))
              (gcd2 cur-chinese (cdr primes)))))
```

In GCD2, each application of CRT started as soon as the previous results become ready. It would be better if each application of CRT started as soon as any two previous results were ready. This would require less structured parallelism, and to implement this we need to resort to the low level construct of EVENT. Our approach will be as follows: whenever one GCD_p is ready, it will be thrown into the pool of partially lifted GCDs. For each GCD_p generated (except the very first one) we will spawn the new process which will hunt any two GCDs, combine them into one and put the result back into the pool. When the last process is finished, we have the fully lifted GCD.

To preserve the integrity of *gcd-pool* we use a process closure:

```
(defvar *pool-handler* (qlambda t (handler) (funcall handler)))
```

(This is analogous to bomb-handler in [1].) By passing the appropriate lambda expression as an argument, this process closure can be used either to push one GCD_p onto the *gcd-pool*, or it can be used to retrieve two modular GCD, to be used by application of the Chinese Remainder theorem. Here is the new version of GCD (GCD3).

```
(defun gcd3 (primes)
  (let ((prime (car primes)))
    (spawn
      (let* ((gcd-pair (funcall *pool-retriever*))
        (cur-gcd  (chinese (car gcd-pair) (cdr gcd-pair)))
                (funcall *pool-handler*
                         #'(lambda ()
                                 (push cur-gcd *gcd-pool*)))
```

```
      (signal-event *non-empty*)))     )
    (spawn
      (let ((cur-gcd (mod-gcd prime)))
        (funcall *pool-handler*
                 #'(lambda ()
                     (push cur-gcd *gcd-pool*)))
        (signal-event *non-empty*)))
    (gcd3 (cdr primes))))
```

Here we immediately spawn the process which eventually will apply CRT after retrieving two modular GCDs, using *pool-retriever*. Then we spawn another process to do the calculation of one GCD_p and to push it onto *gcd-pool*.

The first process must wait till two of GCD_p are ready and it uses *non-empty* event as a counting semaphore. Since more than one such process can wait for the count two, and each of them could be woken up, but only one should, we channel the "wait" through another process closure *pool-retriever*.

```
(defvar *pool-retriever*
        (qlambda t ()
           (wait-event *non-empty* :count 2)     ;account for removal
           (signal-event *non-empty* :count -2) ;retrieve
           ;;Notice the unusual use of negative count above
           (funcall
             *pool-handler*
             #'(lambda () (cons (pop *gcd-pool*) (pop *gcd-pool*))))))
```

Therefore such a process first waits for its turn to apply *pool-retriever*, and then the process waits inside *pool-retriever* for the two modular GCDs to become ready.

The use of both *pool-handler* and *pool-retriever* here is strictly as a monitor (protector of particular data objects or synchronizer for particular operations) but not as a unit of parallel execution. Moreover they are executed strictly sequentially with the calling process.

The synchronization in the above program becomes quite messy. What can we do to streamline it?

Our problem is induced by the fact that the processes responsible for the application of CRT are generated too early and have to wait while appropriate GCDs will be ready. Modular GCDs are produced by the application of mod-gcd and by application of chinese, but we have to generate a new CRT process only for half of them (minus 1). Therefore if we give a chance to generate new chinese process to each of them, then each of them will be able to afford to skip the generation of a chinese process when it is too early, knowing that somebody else will generate this process.

Actually, we move this generation into *pool-handler*. Now *pool-handler* saves the first, third and etc. GCD_p, and whenever the second, fourth and etc. GCDs arrive, *pool-handler* spawns a new chinese process giving it the currently arrived even-numbered GCD and the previously arrived odd-numbered GCD.

```
(defvar *pool-handler*
        (qlambda t
           (cur-gcd)
           (if *gcd-pool*
               (let ((prev-gcd (pop *gcd-pool*)))
                 (spawn (chinese-and-push cur-gcd prev-gcd))
                 (push cur-gcd *gcd-pool*)))))
```

Here is the definition of chinese-and-push:

```
(defun chinese-and-push (cur-gcd prev-gcd)
  (let ((new-gcd (chinese cur-gcd prev-gcd)))
    (funcall *pool-handler* new-gcd)))
```

This allows us to write a simpler program to compute GCD as follows:

```
(defun gcd4 (primes)
  (let ((prime (car primes)))
    (spawn
      (let ((cur-gcd (mod-gcd prime)))
        (funcall *pool-handler* cur-gcd)))
    (gcd4 (cdr primes))))
```

5 Results of the Experiments

Figures 1–8 at the end of this paper present timings for the calculation of GCDs of polynomials of different degrees using the four algorithms described above, in the form of speedups obtained versus the degree of polynomials. The speedup is computed as

$$\text{Speedup}(n) = \frac{\text{CPU time using Lucid Common Lisp on one processor}}{\text{CPU time for Qlisp running in parallel on } n \text{ processors}}.$$

The data presented in the figures need to be explained, and in order to do so we performed some additional investigations.

The fact that timings for GCD1–GCD4 are so close suggests that the application of CRT requires only a relatively small amount of CPU time. The investigation confirmed this.

The current implementation of Qlisp provides a history mechanism, which allows us to record the (wall-clock) time during the evaluation of various points of the program. Using this mechanism we measured the overhead caused by the Qlisp parallelism constructs. This overhead appears to be negligible (less than one percent).

Next we analyzed the effect of garbage collection. The analysis showed that in our application for every 100 seconds of CPU time spent on evaluation of the program, we spent 14 seconds doing GC. Since GC is currently done by one processor while the rest are idle, the performance of the whole system is considerably undermined.

To gauge more accurately the effect of garbage collection on performance we performed some experiments with small problems that didn't garbage-collect. Actually, we evaluated

the same GCD_p on each processor. We found that evaluating seven instances of GCD_p on seven processors would increase the time by 30 percent relative to evaluation of one GCD_p on one processor. The evaluation of 4 instances of GCD_p on 4 processors would increase time by 15 percent. This slowdown is obviously due to competing for internal resources. One of these is the cache, that all of the Alliant's processors share (see [5]). Since the current Qlisp conses from a single free-list, another source of overhead is contention for the head of that list. Another, similar, bottleneck is memory allocation—when more memory is needed, a new page is grabbed from the set of free pages, and then this page is zeroed. Since the zeroing is done by a single processor, this makes the execution of the program less parallel than it ought to be.

In view of these considerations, we reimplemented our algorithm using destructive list operations to do polynomial arithmetic. The new implementation uses only a small fraction of the scratch space of the previous, consing implementation. The serial versions of the algorithms have been sped up by factors ranging between two and three. In addition, the memory usage is low enough that garbage collection is not necessary. The speedups obtained are documented in figures 9–16, and are seen to be sharply better (the best speedup has increased from a factor of 4.5 to a factor of 6 on eight processors, a gain in efficiency from 56% to 75%, the speedups on small numbers of processors are very close to optimal). Also, the curves for the four algorithms are now much more distinct, reflecting the fact that the part of computation where the algorithms differ (Chinese remaindering) has become proportionately much more significant. Another point to note regarding the improved performance shown in figures 9–16, is that it goes contrary to the sometimes expressed contention that a more efficient sequential program is harder to speed up through the use of parallelism.

The peculiar shape of of the graphs is caused by a "bin packing" phenomenon. For polynomials of degree 32, 64 and 96, we have only small number of processes (4, 6 and 8 primes), and therefore the relationship between these numbers and the number of processors is very important.

6 Summary

It appears that it was reasonably easy to introduce some parallelism into the chosen algorithm with the help of Qlisp primitives. The granularity of the spawned tasks is perhaps too large to achieve consistently good performance gains, and for problems of practically useful size we will need to expoit the opportunities for parallelism that exist in lower level arithmetic substrate (for example we might parallelize polynomial multiplication and division[1]).

Our first set of programs is also rather memory-intensive (like many in symbolic algebra). This has made it very useful in revealing the inefficiencies of Qlisp in the area of memory management, just as the early computer algebra efforts have done for the Lisps of the time. Efforts to remedy the problem are under way, and the inefficiencies not having to do with garbage collection will disappear in a couple of months. While that is a significant improvement, the numbers indicate that the implementation of a parallel garbage collector

[1] Joe Weening has implemented several data structures to implement polynomials and performed studies as to their suitability for parallel computation. It appears that a hash-table scheme holds the most promise.

is essential in the long run.

7 Acknowledgements

The authors would like to think Joe Weening and other members of the Qlisp group for many helpful discussions.

References

[1] Gabriel, R. and McCarthy, J. Queue-based Multiprocessing Lisp, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming.*

[2] Goldman, R. and Gabriel, R., Preliminary Results with the Initial Implementation of Qlisp. *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming.*

[3] Knuth, D., *The Art of Computer Programming*, v. 2, Reading, Mass., p. 270.

[4] Collins, G. E., Mignotte, M. and Winkler, F., Arithmetic in basic algebraic domains, in Buchberger *et al*, *Computer Algebra, Symbolic and Algebraic Computation*, 2nd ed., Springer Verlag, 1983.

[5] FX/Series Product Summary, Alliant Computer Systems Corporation, 1985.

9 Figures

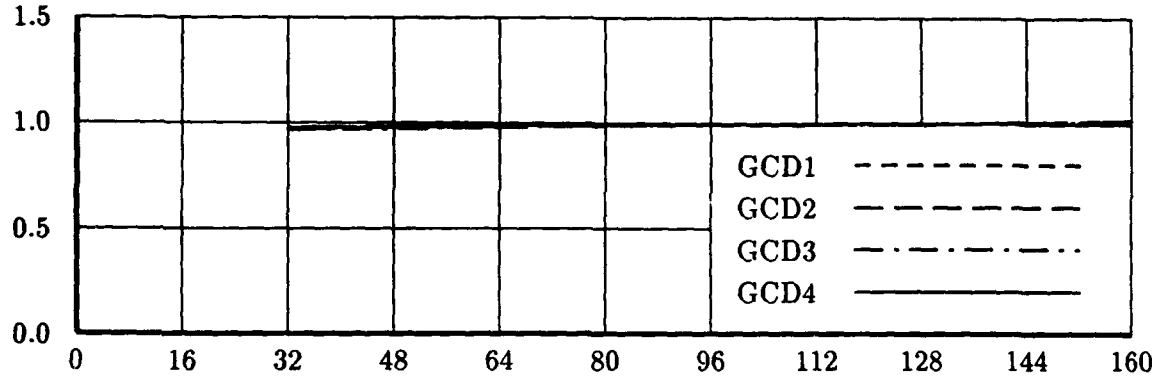

Figure 1: The effective number of processors as a function of the degree of polynomials: **1 processor.**

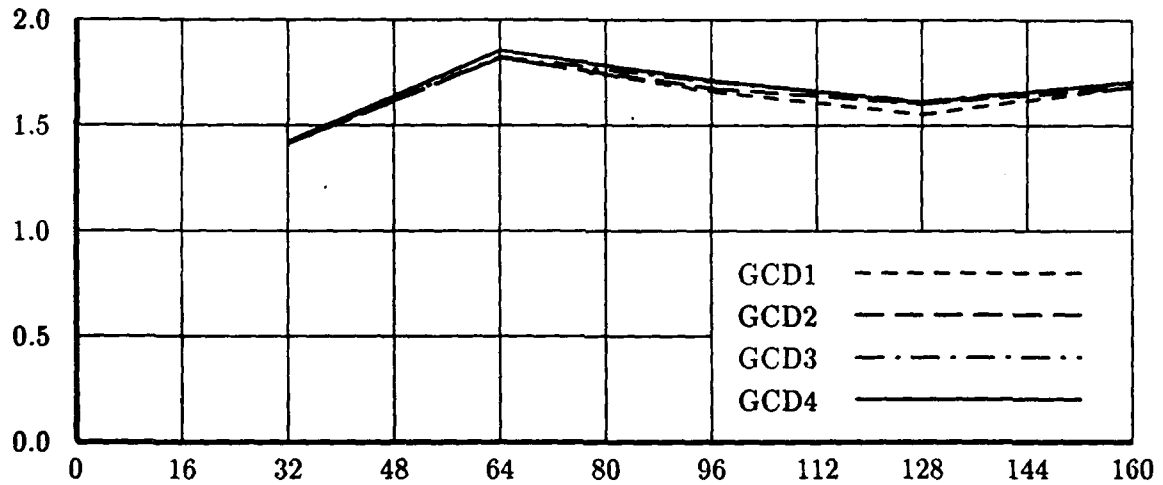

Figure 2: The effective number of processors as a function of the degree of polynomials: **2 processors.**

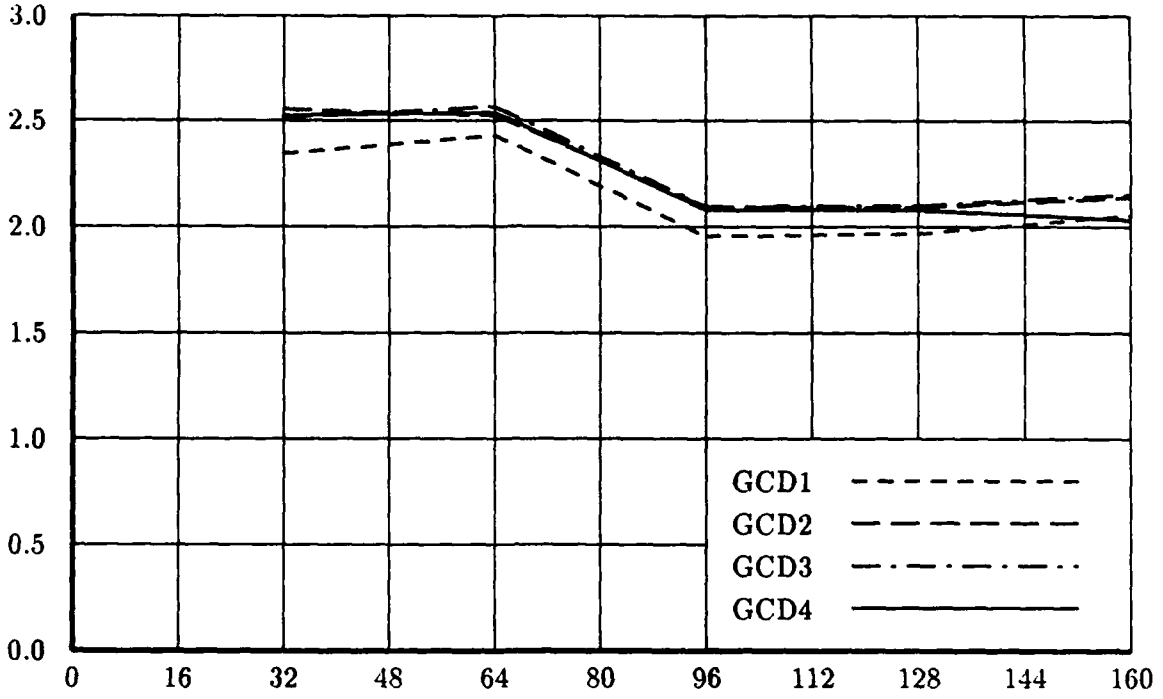

Figure 3: The effective number of processors as a function of the degree of polynomials: **3 processors.**

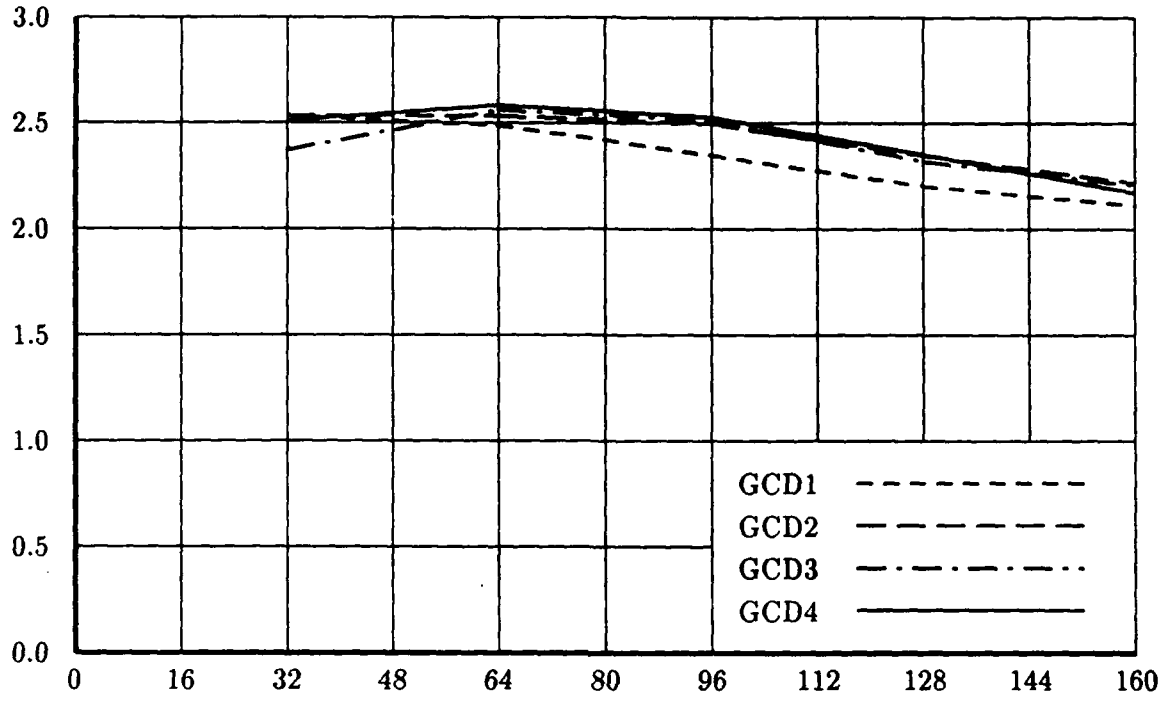

Figure 4: The effective number of processors as a function of the degree of polynomials: **4 processors.**

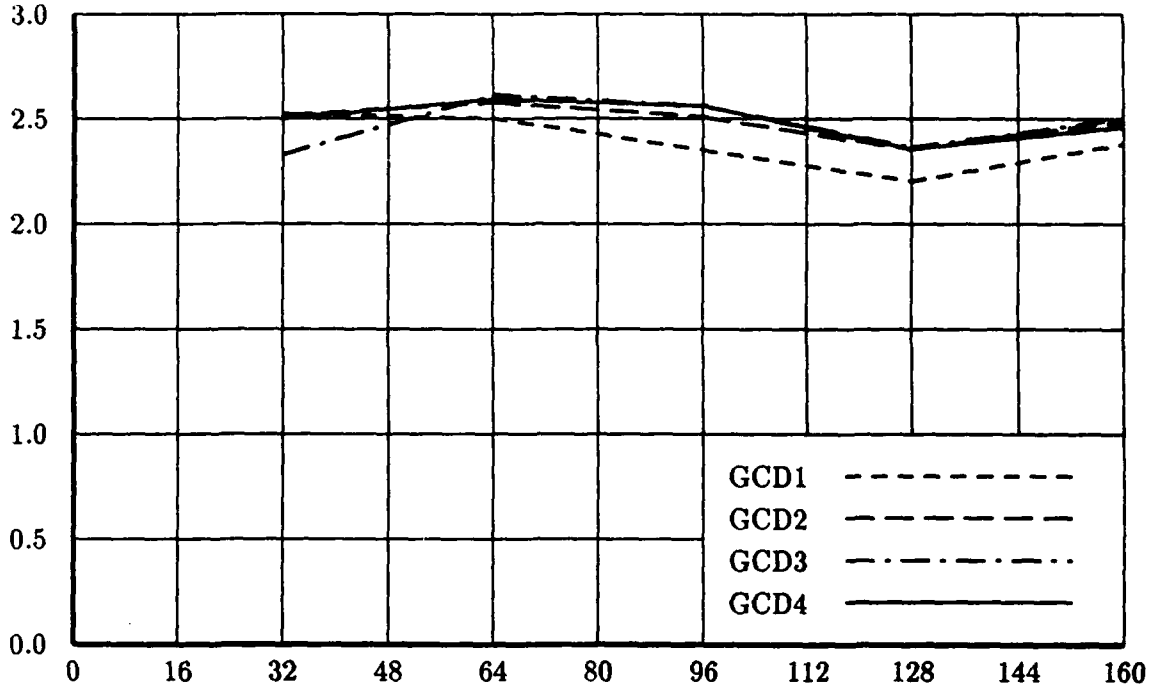

Figure 5: The effective number of processors as a function of the degree of polynomials: **5 processors.**

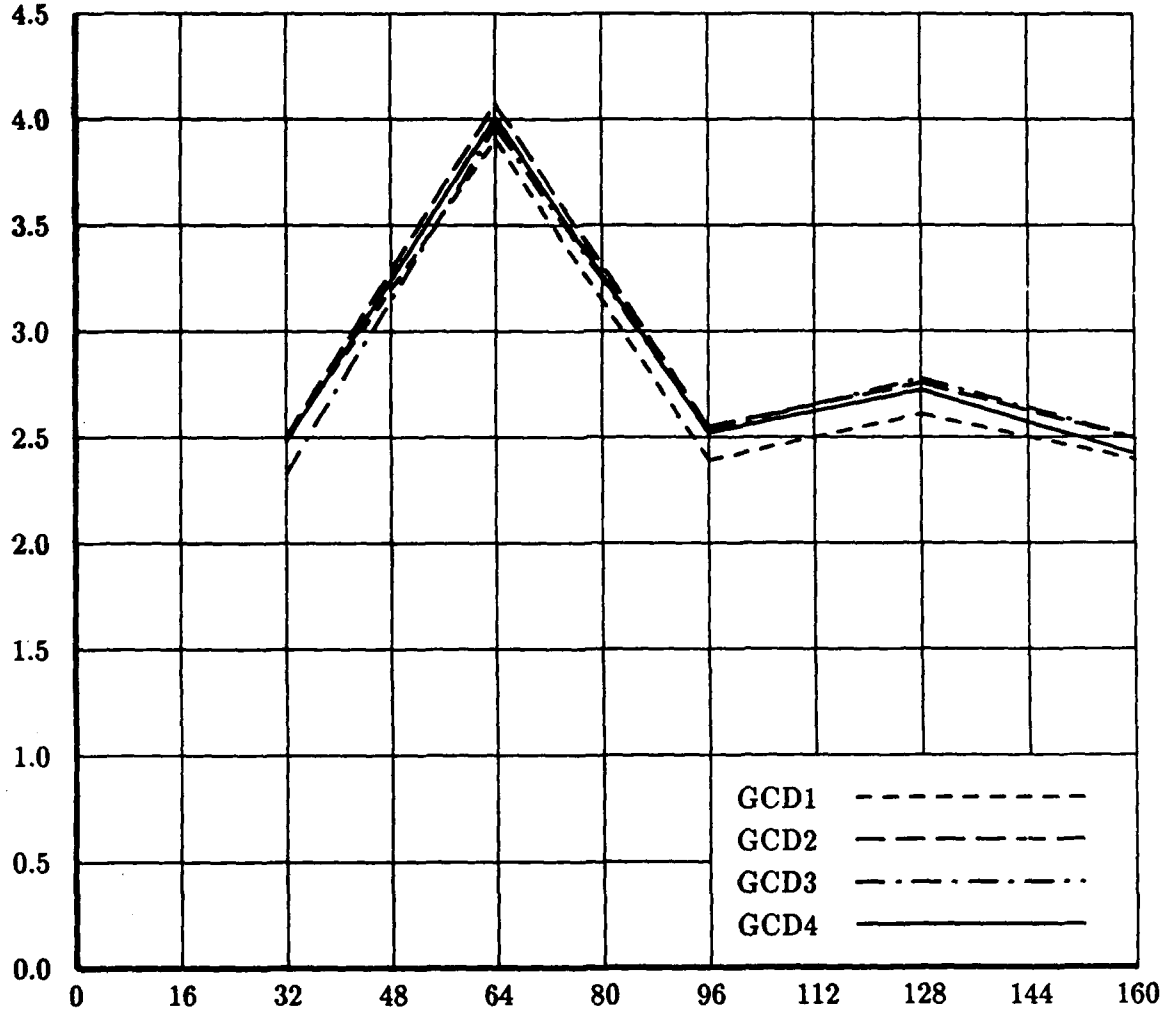

Figure 6: The effective number of processors as a function of the degree of polynomials: **6 processors.**

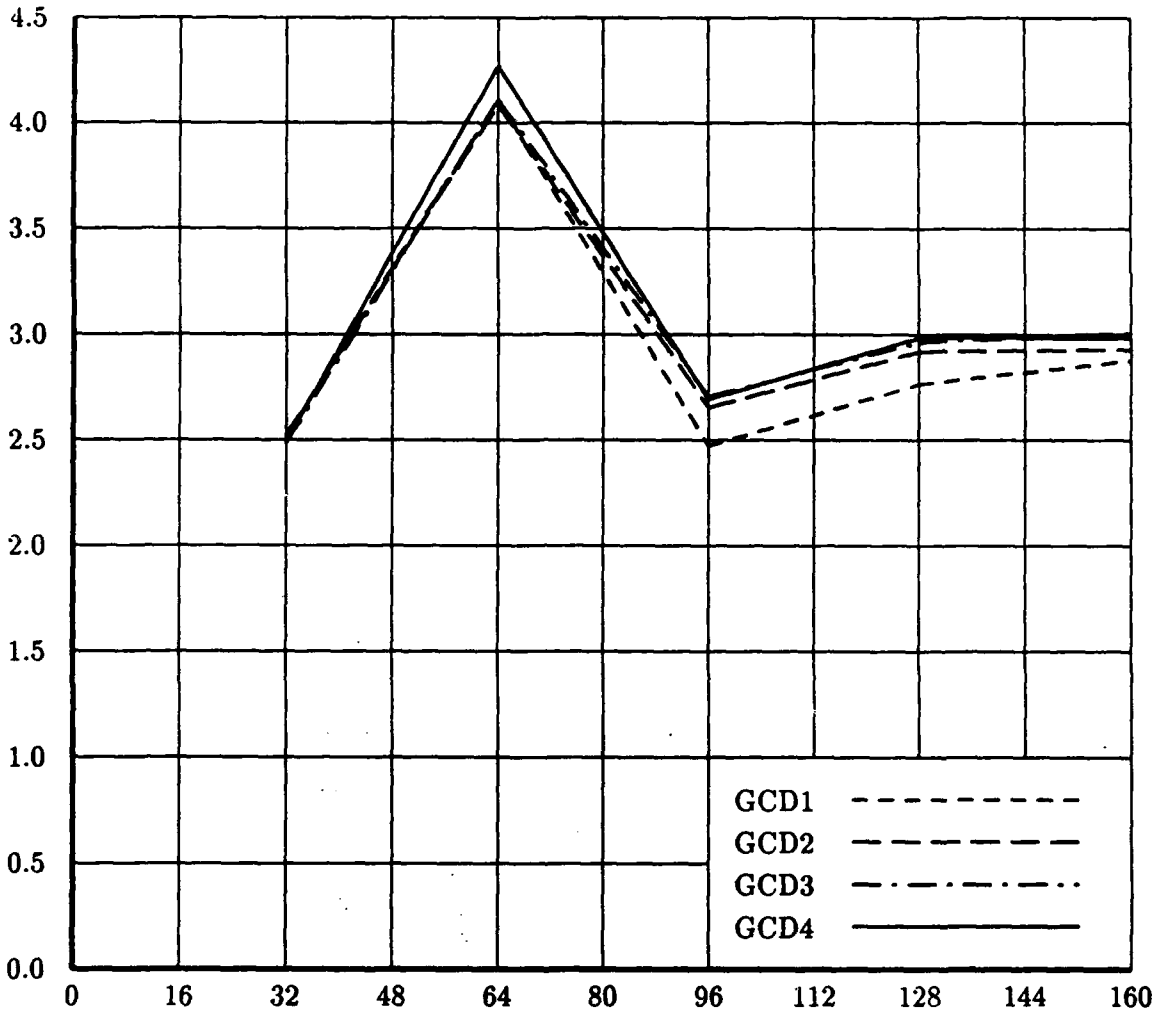

Figure 7: The effective number of processors as a function of the degree of polynomials: **7 processors.**

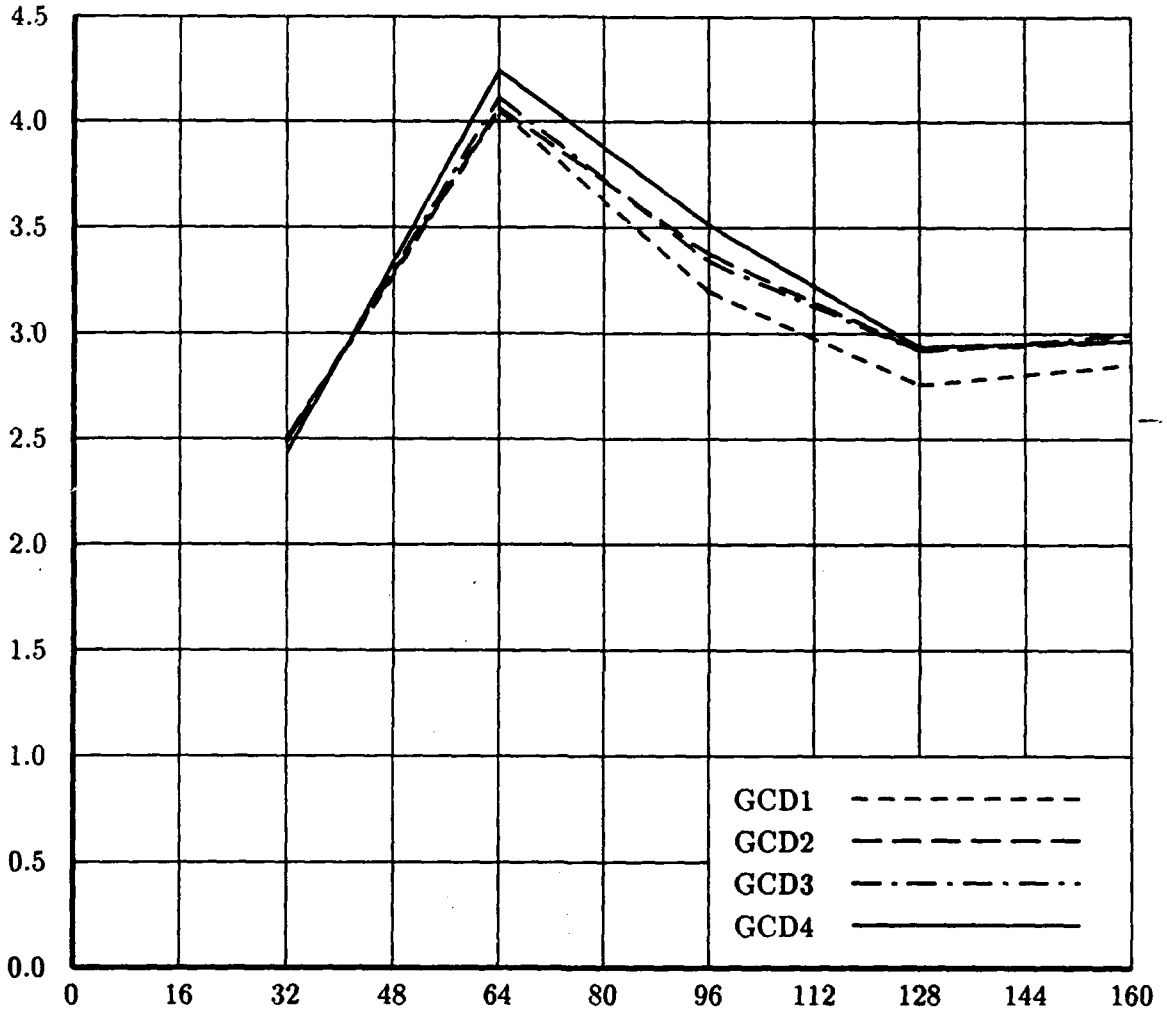

Figure 8: The effective number of processors as a function of the degree of polynomials: **8 processors.**