Parallel Sparse Polynomial Division Using Heaps

Michael Monagan *
Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.
mmonagan@cecm.sfu.ca

Roman Pearce
Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.
rpearcea@cecm.sfu.ca

ABSTRACT

We present a parallel algorithm for exact division of sparse distributed polynomials on a multicore processor. This is a problem with significant data dependencies, so our solution requires fine-grained parallelism. Our algorithm manages to avoid waiting for each term of the quotient to be computed, and it achieves superlinear speedup over the fastest known sequential method. We present benchmarks comparing the performance of our C implementation of sparse polynomial division to the routines of other computer algebra systems.

Categories and Subject Descriptors: I.1.2 [Symbolic and Algebraic Manipulation]: Algebraic Algorithms

General Terms: Algorithms, Design, Performance

Keywords: Parallel, Sparse, Polynomial, Division, Heaps

1. INTRODUCTION

Modern multicore processors let you write extremely fast parallel programs. The cores share a coherent cache with a latency of nanoseconds, where communication can occur at roughly the speed of the processor. The challenge now is to design fast parallel algorithms that execute largely in cache and write only their result to main memory.

In [11] we presented such a method for sparse polynomial multiplication. Given polynomials f and g with $\#f$ and $\#g$ terms, we construct $f \times g = \sum_{i=1}^{\#f} \sum_{j=1}^{\#g} f_i \cdot g_j$ by creating, sorting, and merging all the products in parallel, entirely in the cache. We based the algorithm on Johnson's method [7] which we found to be a fast sequential approach in [12, 13].

Johnson's algorithm computes $\sum_{i=1}^{\#f} f_i \cdot g$ using a binary heap to perform an $\#f$-ary merge. The products $f_i \cdot g_j$ are constructed on the fly so only $O(\#f)$ scratch space is used. It begins with $f_1 \cdot g_1$ in the heap, and after merging $f_i \cdot g_j$ it inserts $f_i \cdot g_{j+1}$. When $j = 1$ it also inserts $f_{i+1} \cdot g_1$. This assumes that f and g are sorted in a monomial ordering.

*We gratefully acknowledge the support of the MITACS NCE of Canada and NSERC of Canada

PASCO 2010, 21–23 July 2010, Grenoble, France.

In our parallel algorithm each core multiplies a subset of the terms of f by all of g. Those subproblems were chosen because Johnson's algorithm is $O(\#f\#g \log \#f)$. The cores write their intermediate results to circular buffers in shared cache, and a global instance of Johnson's algorithm merges the buffers to produce the result. Superlinear speedup was obtained from the extra local cache in each core.

This paper obtains a similar result for sparse polynomial division. This is a considerably harder problem, because in multiplication the polynomials f and g are known up front. For division, we are given the dividend f and the divisor g, and we construct each new term of the quotient q from the largest term of $f - q \cdot g$. This produces a tight dependency among the terms of the quotient, and adds synchronization and contention to the multiplication of q and g.

We are not aware of a comparable attempt to parallelize sparse polynomial division. Our algorithm is asynchronous and does not wait between the computation of q_i and q_{i+1}. In [15], Wang suggests parallelizing the subtraction of $q_i \cdot g$ and synchronizing after each new term of the quotient. No data is provided to assess the effectiveness of this approach but we believe the waiting would be a problem. It appears the CABAL group [10, 14] has also tried this approach. For dense polynomials, Bini and Pan develop a parallel division algorithm based on the FFT in [1], and in [8], Li and Maza assess parallelization strategies for dense univariate division modulo a triangular set.

Our paper is organized as follows. In Section 2 we discuss the division algorithm and the challenges of parallelization. We describe our solutions and present the algorithm. Then in Section 3 we present benchmarks of our implementation. We compare its performance and speedup to the sequential routine of [13], the parallel multiplication codes of [11], and the division routines of other computer algebra systems.

2. SPARSE POLYNOMIAL DIVISION

Consider the problem of dividing two sparse multivariate polynomials $f \div g = q$ in $\mathbb{Z}[x_1, \ldots, x_n]$. In general there are two ways to proceed. In the *recursive* approach we consider them as polynomials in x_1 with coefficients in $\mathbb{Z}[x_2, \ldots, x_n]$. We divide recursively to obtain a quotient term q_i, then we subtract $f := f - q_i g$. The recursive coefficient operations could be performed in parallel as suggested by Wang in [15].

One problem with this method is the many intermediate pieces of storage required. Memory management is difficult to do in parallel while preserving locality and performance. For exact division the polynomial f is also reduced to zero, so the construction of $q \cdot g$ in memory is wasteful.

In the *distributed* approach we impose a monomial order on $\mathbb{Z}[x_1, \ldots, x_n]$ to divide and cancel like terms. We divide the largest term of f by the largest term of g to construct the first term q_1 of the quotient, and repeat the process for $f - q_1 g$ to obtain q_2, and so on, until either $f - \sum q_i g = 0$ or the division fails. There may be very little work between the computation of q_i and q_{i+1}, which makes this approach difficult to parallelize.

But it has a critical advantage for division. Using a heap we can merge the terms of $q \cdot g$ in descending order without constructing large objects in memory. For example, when a new term q_i is computed we can insert $q_i \cdot g_2$ into the heap, and when this term is used we would replace it with $q_i \cdot g_3$. This is Johnson's "quotient heap" algorithm, where a heap of size $\#q$ is used to merge $\sum_{i=1}^{\#q} q_i \cdot (g - g_1)$. It uses $O(\#q)$ memory in total, far less than the $O(\#f + \#q\,\#g)$ memory used by the recursive approach.

One nice feature of the quotient heap algorithm is that a new term q_i completely determines a row $q_i \cdot g$ in the heap. If we could distribute the q_i to different processors it would be easy to parallelize division. However one problem is that a new term of the quotient could be computed at any time. We may also use $q_i \cdot g_2$ immediately to compute q_{i+1}. This suggests an alternative partition of the work.

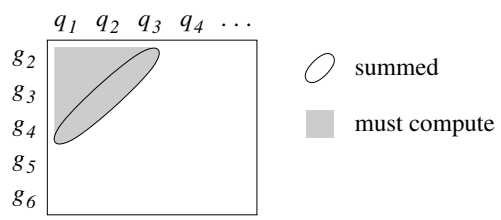
The "divisor heap" algorithm of Monagan and Pearce [12] computes $\sum_{i=2}^{\#g} g_i \cdot q$ instead. That is, elements of the heap walk down the quotient and multiply by some divisor term. Distributing terms of g to the threads solves two problems. First, we can divide the work in advance with good locality and suggest the number of threads. Second, $\{g_2, g_3, \ldots, g_k\}$ may be merged by the processor computing quotient terms so that their products are known without delay. Our entire algorithm is designed to avoid waiting in a typical division, and this is one of two situations we address.

One may ask whether there is a loss of efficiency because the divisor heap algorithm performs $O(\#f + \#q\#g \log \#g)$ monomial comparisons. This is not optimal when $\#q < \#g$. In [13] we present a sequential division algorithm that does $O(\#f + \#q\#g \log \min(\#q, \#g))$ comparisons. However our divisor heap is run on subproblems with $\#g/p$ by $\#q$ terms where p is the number of threads, so the threshold becomes easier to meet as the number of threads increases.

2.1 Dependencies

We begin with an example that shows the main problem encountered in parallelizing sparse polynomial division. Let $g = x^5 + x^4 + x^3 + x^2 + x + 1$ and $f = g^2$. To divide f by g we will compute $q_1 = f_1/g_1 = x^5$, $q_2 = (f_2 - q_1 g_2)/g_1 = x^4$, $q_3 = (f_3 - q_1 g_3 - q_2 g_2)/g_1 = x^3$, and so on. Each new term of the quotient is used immediately to compute subsequent terms, so q_k depends on the triangle of products $g_i \cdot q_j$ with $i + j - 1 \le k$, as shown below for q_4.

Figure 1: Dependency in Dense Univariate Division.



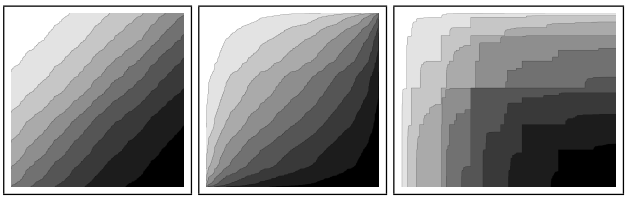
In parallel division the products $g_i \cdot q_j$ are merged using multiple threads. Our problem is to divide up the products in a way that mostly prevents having threads wait for data. For example, in the computation above we compute q_k and then immediately use $q_k g_2$ to compute q_{k+1}. It would make sense to do both operations in the same thread. Otherwise, one thread will compute q_k and stop to wait for $q_k g_2$ while the thread computing $q_k g_2$ waits for q_k and then carries out its task. Waiting serializes the algorithm because the round trip latency is longer than it takes to compute terms.

In the dense example (see Figure 1) we might be able to multiply $g_6 \cdot q$ in a separate thread without waiting for any of its terms, because after we compute q_k we need to merge $\{g_2 q_k, g_3 q_k, g_4 q_k, g_5 q_k\}$ before $g_6 q_k$ is used. We could merge other terms as well but those four have distinct monomials. The second thread may still have to wait for q_k if it doesn't have enough other work to do.

The structure of sparse polynomial multiplication is that $g_i q_j > g_{i+1} q_j$ and $g_i q_j > g_i q_{j+1}$ when the terms of q and g are sorted in a monomial ordering. In general this is called $X + Y$ sorting, see Harper et al. [6]. We are exploiting this structure to get parallelism in the multiplication of q and g. The approach is a recognized parallel programming pattern called *geometric decomposition*. For details see [9].

Our algorithm partitions the products $\{g_i q_j\}$ into regions that are merged by different threads. The $X + Y$ structure provides a lower bound on the amount of work that is done before a term from an adjacent region is needed. The work is used to conceal the latency of communication so that our threads can run independently and do not have to wait.

Figure 2: Common X+Y Sort Orders.



Whatever partition we choose will have to interact nicely with the construction of the quotient q, but there is no way to know the dependencies of q in advance. So we identified three common cases by experiment, see Figure 2. To create each graphic, we sorted the products $\{g_i q_j\}$ for $1 \le i \le \#g$ and $1 \le j \le \#q$ and shaded them from white to black. The image shows the order that terms are merged, and the first row shows when we construct each term of the quotient.

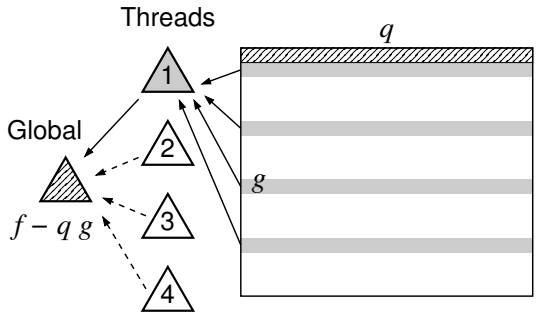
The triangular dependencies of dense univariate divisions (see Figure 1) are apparent in the first image, although the structure is found in sparse problems too. In this case $O(k)$ terms are merged between the computation of q_k and q_{k+1}. Merging and quotient computation both occur at the same regular rate, so this is the easiest case to parallelize. In the hyperbolic case the quotient is computed rapidly, with very little work between the computation of q_k and q_{k+1}. There we must avoid waiting for $\{g_2 q_k, g_3 q_k, \ldots\}$ to be computed since those terms will be needed immediately. The last case is the hardest one to parallelize. Polynomials with algebraic substructure tend to produce blocks which must be merged in their entirety before any new quotient term is computed. In the next section we describe our solution.

2.2 Parallel Algorithm

Our parallel division algorithm borrows heavily from our multiplication algorithm in [11]. To each thread we assign a subset of the partial products $\{g_i \cdot q\}$. These are merged in a heap and the result is written to a buffer in shared cache. A global function is responsible for merging the contents of the buffers and computing new terms of the quotient. This function is protected by a lock.

Unlike in the parallel multiplication algorithm, the global function here is also assigned a strip of terms along the top $(g_1 + \cdots + g_s) \cdot q$. This allows it to compute some quotient terms and stay ahead of the threads. It uses g_1 to compute quotient terms and the terms $(g_2 + \cdots + g_s) \cdot q$ are merged. Then the strip $(g_{s+1} + \cdots + g_{2s}) \cdot q$ is assigned to thread 1, the next strip of s terms is assigned to thread 2, and so on, as in Figure 3 below. The strip height s is derived from the number of terms in g, refer to Section 2.3 for details.

Figure 3: Parallel Sparse Division Using Heaps.



The threads merge terms from left to right in the style of a divisor heap of Monagan and Pearce [13]. Each iteration of the main loop extracts all of the products $g_i \cdot q_j$ with the largest monomial, multiplies their coefficients to compute a sum of like terms, and inserts their successors $g_i \cdot q_{j+1}$ into the heap to set up the next iteration of the algorithm.

A major problem is that after $g_i \cdot q_j$ is extracted from the heap and merged, we may find that q_{j+1} does not yet exist. For example, towards the end of a division there will be no more quotient terms. The threads need some way to decide that it is safe to continue without $g_i \cdot q_{j+1}$ in the heap.

In the sequential division algorithm this is easy because $g_1 \cdot q_{j+1} > g_i \cdot q_{j+1}$ in the monomial order. This guarantees q_{j+1} is constructed (by dividing by g_1) before any products involving it need to be in the heap. We can safely drop the products missing q_{j+1} as long as they are reinserted before they could be merged. For example, in our algorithm in [13] we set bits to indicate which g_i have a product in the heap. When a new quotient term q_{j+1} is computed we check if g_2 has a product in the heap and insert $g_2 \cdot q_{j+1}$ if it does not, and when we insert $g_i \cdot q_j$ with $i < \#g$, we also insert the next product for g_{i+1} if it is not already in the heap.

In the parallel algorithm the computation of the quotient is decoupled from the merging of products, so this strategy does not work. It becomes difficult to maintain consistency in the algorithm and expensive synchronization is required. Eventually we made a compromise – if a thread encounters $g_i \cdot q_{j+1}$ and q_{j+1} is missing, the thread must wait for q_{j+1} to be computed or be relieved of the task of merging $g_i \cdot q$. The idea is to have the global function steal rows from the threads to allow them to proceed.

Figure 4: The Global Function Steals Rows.

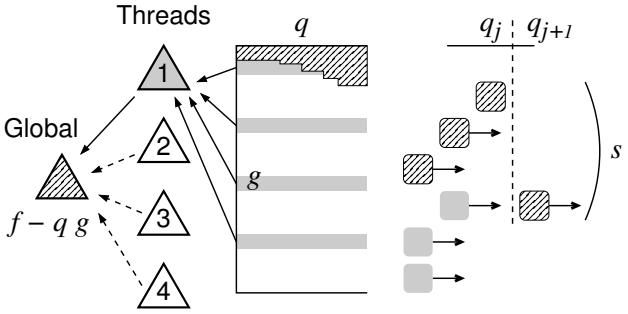


Figure 4 shows the global function in more detail. At the beginning of the computation it is assigned a strip of $s = 4$ terms. It uses g_1 to construct quotient terms and it merges $(g_2 + g_3 + g_4) \cdot q$ using a heap. After merging $g_2 \cdot q_j$, it sees that q_{j+1} has not been computed. It steals $g_5 \cdot (q_{j+1} + \cdots)$ by incrementing a global bound that is read by all threads. This bound is initially set to 4, and it will be updated to 5. When new quotient terms are computed, the current value of the bound is stored beside them for the threads to read.

Two possibilities can now occur in the Figure 4 example. If the thread merging $g_5 \cdot q$ reaches $g_5 \cdot q_{j+1}$ before q_{j+1} has been computed, it checks the global bound and the number of terms in the quotient. With no more quotient terms and a global bound greater than or equal to 5, it drops the row from its heap. Otherwise, if q_{j+1} is computed first, a bound of at least 5 is stored beside q_{j+1}. The thread sees this and again drops the row from its heap.

Stealing rows in the global function allows the threads to continue merging terms without any extra synchronization. If used aggressively it also eliminates waiting, at the cost of serializing more of the computation. This is a bad tradeoff. We prefer to steal as few rows as possible with a reasonable assurance that waiting will not occur.

2.3 Implementation

It is a non-trivial matter to sit down and implement this algorithm given the main idea. With sequential algorithms one expects the performance of implementations to vary by a constant factor. This is not the case for complex parallel algorithms since implementation details may determine the scalability. These details are a critical aspect of the design.

Our main challenge in designing an implementation is to minimize contention. This occurs when one core reads data that is being modified by another. In the division algorithm the quotient is a point of contention because we compute it as the algorithm runs and it is used by all of the threads.

We manage contention by using one structure to describe the global state of the algorithm. Shared variables, such as the current length of the quotient and the bound are stored on one cache line and updated together. Each thread reads these values once and then continues working for as long as possible before reading them again. This optimization may reduce contention by up to an order of magnitude.

We first used the trick of caching shared variables in the circular buffers of the parallel multiplication algorithm [11]. Those buffers are reused here. They reach 4.4 GB/s on our Intel Core i7 920 with this optimization, but only 1.2 GB/s without it. This shows just how high the cost of contention is for only two threads, and with more threads it is worse.

We now present the algorithm. The first function sets up the global state and creates the threads. When the threads terminate, it could be because the algorithm has completed or because the global function has stolen every row. In the latter case we continue to call the global function until the division is complete.

Just like our multiplication algorithm [11] we run at most one thread per core to avoid context switches. For X cores we compute $t = \sqrt[3]{\#g}$, create $p = min(t/2, X)$ threads, and give each thread strips of size $s = t^2/p$ terms. This value is a compromise between large strips which are fast and small strips which uniformly distribute the work.

The next function is the local merge that we run on each thread. It creates a heap and tries to add the first product. If the necessary quotient term does not exist yet, it tries to enter the global function and compute more quotient terms. It also discards any products stolen by the global function.

A product $g_i \times q_j$ has been stolen if q_j exists ($j < t$) and $i \leq bound(q_j)$, or if q_j does not exist ($j \geq t$) and $i \leq b$. The function will block in the case $j \geq t$ and $i > b$, i.e. when q_j does not exist and the row has not yet been stolen.

An important detail of the algorithm is that it must use memory barriers to ensure correctness. For example, as the algorithm runs, the global function computes new quotient terms and steals rows by incrementing a bound. Imagine if both were to happen in quick succession. A thread may see the bound modified first and discard a row before it merges all of the terms. Memory barriers enforce the correct order.

We use a simple rule: *'first written, last read'* to logically eliminate race conditions from our program. With this rule threads can read a volatile global state and act consistently as long as the variables are monotonic. Here the number of rows stolen and quotient terms computed only increase.

The global function is shown on the next page. It inserts terms from the buffers to update the global heap G, but at the start of the division there is no quotient and $mergeG$ is set to $false$. It performs a three way comparison to decide which of the dividend, local heap, and global heap have the largest monomial that must be merged. We write this step in a clear but inefficient way. Our implementation performs at most two ternary comparisons that return $<$, $>$, or $=$.

The global function then merges the maximal terms. The local heap case contains additional logic to add stolen rows. After merging $g_i \times q_j$, we check to see if g_{i+1} has a term in the heap. If not and $i+1 \leq bound(q_j)$ we insert the row for g_{i+1} starting at q_j. Otherwise $g_{i+1} \times q_j$ will be merged by a thread so we set $mergeG := true$ to start the global heap.

The global function can steal a row if $g_i \cdot q_j$ is merged by the local heap and q_{j+1} does not exist, or if terms from the global heap are merged when the local heap is empty. This second case is needed at the end of the division when there are no more quotient terms. The global function must keep stealing rows to allow the threads to progress.

The general idea is to maintain a gap of $s-1$ monomials between the global function and all the threads. When the global function merges the last term of row g_i, it steals row g_{i+s-1} if it has not already done so. This allows a thread to merge to the end of row g_{i+s}. Once all of its assigned terms have been merged, the global function steals a row for each distinct monomial it encounters. This allows the threads to continue merging terms without any extra synchronization, as long as they send zero terms to the global function to be merged.

Algorithm: Parallel Sparse Polynomial Division.
Input: $f, g \in \mathbb{Z}[x_1, \ldots, x_n]$, number of threads p.
Output: quotient $q = f/g$, boolean saying if division failed
Globals: heap F, heap G, set Q, lock L, quotient q,
booleans $terminate, failed, mergeG$,
slack S, gap s, bound b.

```
F := an empty heap ordered by < with max element F_1
     for merging the top strip in the global function
G := an empty heap ordered by < with max element G_1
     for merging the results from all the threads
Q := a set of p empty buffers
     from which we insert terms into G
L := an unheld lock to protect the global function
terminate := false      // set to terminate threads
mergeG := false         // set to merge terms from G
failed := false         // set if exact division fails
q := 0                  // the quotient q = f/g
b := p                  // rows owned by global function
s := b                  // initial height of the top strip
S := 0                  // "slack" before a row is stolen
for i from 1 to p do
  spawn local_merge(i, p)
wait for all threads to complete
while not terminate do
  merge_global()
return (q, failed)
```

Subroutine: Local Merge.
Input: thread number r, total number of threads p.
Output: a subset of terms of $q \cdot g$ are written to B.
Locals: heap H, set E, monomial M, coefficient C
rows stolen $b1$, number of quotient terms $t1$.
Globals: quotient q and divisor g in $\mathbb{Z}[x_1, \ldots, x_n]$,
rows stolen b, number of quotient terms t,
lock L, boolean $terminate$

```
H := an empty heap ordered by < with max element H_1
E := {}       // terms extracted from H
t1 := 0       // number of quotient terms
b1 := p       // number of rows stolen
// {g_1, ..., g_p} owned by global function, we start at g_{p+r}
(i, j) := (p + r, 0)      // try to insert g_{p+r} × q_1
goto check_term:
while |H| > 0 do
  // merge all products with largest monomial M
  M := mon(H_1); C := 0; E := {};
  while |H| > 0 and mon(H_1) = M do
    (i, j, M) := extract_max(H)
    C := C + cof(g_i) · cof(q_j)
    E := E ∪ {(i, j)}
  insert term (C, M) into the buffer B
  // for each extracted term insert next term into heap
  for all (i, j) ∈ E do
    // insert first element of next row
    if j = 1 and i + p ≤ #g and bound(q_1) < i + p then
      insert g_{i+p} × q_1 into H
    check_term:
    // loop until g_i × q_{j+1} can be inserted or discarded
    while j = t1 and i > b1 do
      if trylock(L) then
        global_merge()
        release(L)
      else
        sleep for 10 microseconds
      b1 := b       // update rows stolen
      read_barrier()
      t1 := t       // update number of quotient terms
      if terminate then return
    if j < t1 and bound(q_{j+1}) < i then
      insert g_i × q_{j+1} into H
close(B)
return
```

```
Subroutine: Global Merge.
Output: terms of the quotient are written to q.
Locals: coefficient C, monomial M, buffer B,
        booleans stealG, stealL.
Globals: heaps F and G, sets P and Q, polynomials f, g, q,
        rows stolen b, number of quotient terms t,
        booleans terminate, failed, mergeG,
        index k into f, initial height s, slack S.
  if terminate then return
  if mergeG then       // insert terms into global heap G
    for all B in Q do
      if B is not empty then
        extract next term (C, M) from buffer B
        insert [B, C, M] into heap G
        Q := Q \ {B}
      else if not is_closed(B) then goto done:
  // 3-way comparison of dividend, local heap, global heap
  // u, v, w is set to true or false to merge terms from each
  C := 0;  u := (k ≤ #f);  v := (|F| > 0);  w := (|G| > 0);
  stealG := w and not v;  stealL := false;
  if u and v  and mon(f_k) < mon(F_1) then u := false
  if u and w and mon(f_k) < mon(G_1) then u := false
  if v and u  and mon(F_1) < mon(f_k) then v := false
  if v and w and mon(F_1) < mon(G_1) then v := false
  if w and u and mon(G_1) < mon(f_k) then w := false
  if w and v and mon(G_1) < mon(F_1) then w := false
  if not (u or v or w) then       // no terms to merge
    terminate := true             // division complete
  if u then       // merge a term from the dividend
    C := C + cof(f_k)
    M := mon(f_k)
    k := k + 1
  if v then       // merge terms from local heap F
    P := {}
    M := mon(F_1)
    while |F| > 0 and mon(F_1) = M do
      (i, j, M) := extract_max(F)
      C := C + cof(g_i) · cof(q_j)
      P := P ∪ {(i, j)}
    for all (i, j) ∈ P do
      if j < #q then
        insert g_i × q_{j+1} into F
      else stealL := true
      if i < #g and g_{i+1} has no term in F then
        if i + 1 ≤ bound(q_j) then
          insert g_{i+1} × q_j into F
        else     // start merging global heap
          mergeG := true
  if w then       // merge terms from global heap G
    Q := {}
    M := mon(G_1)
    while |G| > 0 and mon(G_1) = M do
      (B, K, M) := extract_max(G)
      C := C - K
      Q := Q ∪ {B}
  if C = 0 then goto done:
  // compute a new quotient term
  if LM(g) | M and LC(g) | C then
    q_{t+1} := (C/LC(g), M/LM(g))
    bound(q_{t+1}) := b
    write_barrier()    // commit term to memory
    t := t + 1         // make term visible
    S := b - s         // set slack
    if #g > 1 and g_2 has no product in G then
      insert g_2 × q_t into G
  else                 // division failed
    terminate := true
    failed := true
done:    // steal row if local heap empty or product dropped
  if (stealG or stealL) and b < #g then
    if S > 0 then S := S - 1      // reduce slack
    else b := b + 1               // steal a new row
  return
```

3. BENCHMARKS

We retained the benchmark setup of [11] to allow for easy comparison of the parallel sparse polynomial multiplication and division algorithms. We use two quad core processors: an Intel Core i7 920 2.66GHz and a Core 2 Q6600 2.4GHz. These processors are shown below. The Core i7 has 256KB of dedicated L2 cache per core. We get superlinear speedup by using more of the faster cache in the parallel algorithms. In all of the benchmarks our time for one thread denotes a sequential time for an algorithm from [13].

Core i7 **Core 2**

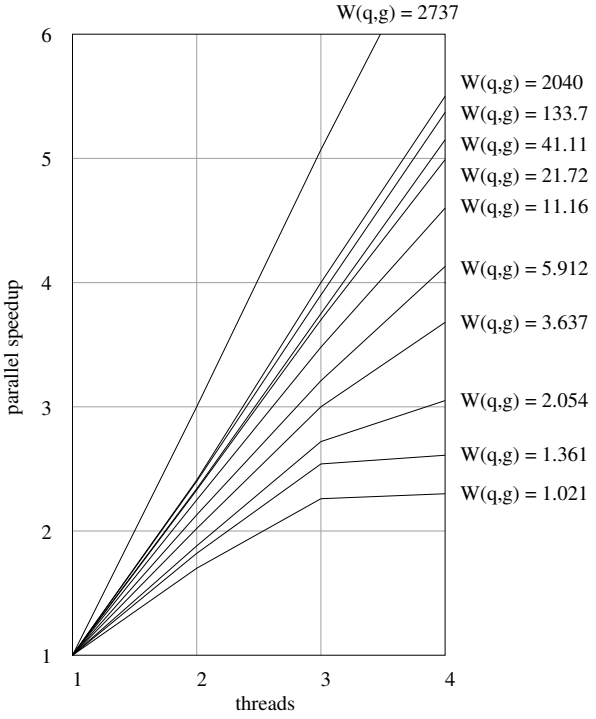
3.1 Sparsity and Speedup

We created random univariate polynomials with different sparsities and multiplied them modulo 32003 as in [11]. The polynomials have 8192 terms. We then divide their product by one of the polynomials modulo 32003. The graph shows the speedup obtained at different sparsities on the Core i7.

For division we measure sparsity as the *work per term* to multiply the quotient and the divisor. That is, for $f/g = q$ $W(q, g) = (\#q \cdot \#g)/\#(q \cdot g)$. This makes our graph below directly comparable to the one for multiplication in [11].

Figure 5: Sparsity vs. Parallel Speedup over $\mathbb{Z}_p$

(totally sparse) $1 \le W(q, g) \le 4096.25$ (totally dense)



The results in Figure 5 are generally good, but the curve for extremely sparse problems flattens out. We are not able to fully utilize all the cores to maintain parallel speedup as $W(q, g) \to 1$. Otherwise, our results here are comparable to those for parallel multiplication in [11]. We obtained linear speedup in the completely dense case.

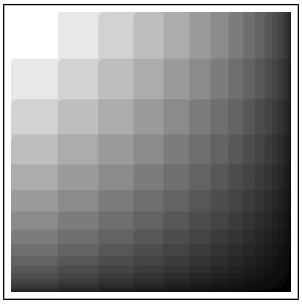
Our throughput here is limited by the dependencies of q, which are triangular in shape. The computation of quotient terms is thus tightly coupled to the merging in the threads, and our global function can not stay ahead. This forces the threads to wait for quotient terms.

3.2 Dense Benchmark

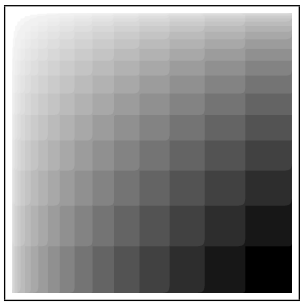
Let $g = (1 + x + y + z + t)^{30}$. We compute $f = g \cdot (g + 1)$ and divide f/g. The quotient and divisor have 46376 terms and 61 bit coefficients. The dividend has 635376 terms and 128 bit coefficients. This problem is due to Fateman [3].

Unlike [11], we also test graded lexicographical order with $x > y > z > t$. This choice of order produces the monomial structure below. The upper left block is 5456×5456 terms consisting of all the products of total degree 60. It must be merged in its entirety to compute the 5457^{th} quotient term, which forces our global function to steal 5455 rows. Despite this difficulty, the performance of our algorithm was good.

Figure 6: Fateman Benchmark



graded lex order (tricky)



lexicographical order

In addition to our software sdmp, we timed Magma 2.16, Singular 3-1-0, and Pari 2.3.3. Magma now also uses heaps to do polynomial multiplication and division. Singular uses a divide-and-conquer algorithm to multiply and a recursive sparse algorithm to divide. Pari uses recursive dense and it supports division only in the univariate sense.

Table 1: Dense benchmark $\mathbb{Z}_{32003}$, $W(f,g) = 3332$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	11.68 s	5.87x	15.10 s	5.42x
	3	16.52 s	4.15x	21.94 s	3.73x
	2	27.83 s	2.46x	37.07 s	2.21x
	1	68.59 s		81.93 s	
sdmp (grlex)	4	11.20 s	6.12x	15.37 s	5.43x
	3	15.94 s	4.30x	21.22 s	3.93x
	2	27.56 s	2.49x	35.01 s	2.38x
	1	68.59 s		83.54 s	
Singular 3-1-0	1	152.65 s		105.26 s	
Magma 2.16-7	1	134.29 s		299.29 s	
Pari 2.3.3	1	795.22 s		438.62 s	
Core 2	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	13.86 s	4.25x	17.82 s	3.80x
	3	19.06 s	3.09x	23.93 s	2.83x
	2	29.82 s	1.97x	35.24 s	1.92x
	1	58.91 s		67.69 s	
sdmp (grlex)	4	13.93 s	4.34x	18.42 s	3.74x
	3	19.19 s	3.15x	23.97 s	2.87x
	2	27.58 s	2.19x	35.06 s	1.96x
	1	60.50 s		68.87 s	
Singular 3-1-0	1	273.05 s		150.36 s	
Magma 2.16-7	1	139.98 s		446.57 s	
Pari 2.3.3	1	942.78 s		520.15 s	

Table 2: Dense benchmark $\mathbb{Z}$, $W(f,g) = 3332$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	11.33 s	6.25x	15.18 s	5.78x
	3	16.30 s	4.34x	21.94 s	4.00x
	2	28.01 s	2.53x	37.03 s	2.37x
	1	70.81 s		87.68 s	
sdmp (grlex)	4	11.50 s	6.15x	15.57 s	5.72x
	3	16.33 s	4.33x	21.36 s	4.17x
	2	28.31 s	2.50x	35.34 s	2.52x
	1	70.75 s		89.11 s	
Singular 3-1-0	1	817.43 s		296.75 s	
Magma 2.16-7	1	359.98 s		441.43 s	
Pari 2.3.3	1	651.02 s		354.82 s	
Core 2	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	14.20 s	4.25x	17.88 s	4.28x
	3	19.48 s	3.10x	24.15 s	3.17x
	2	30.35 s	1.99x	35.29 s	2.17x
	1	60.38 s		76.59 s	
sdmp (grlex)	4	14.27 s	4.24x	18.59 s	4.20x
	3	19.69 s	3.07x	24.20 s	3.22x
	2	28.11 s	2.15x	35.39 s	2.20x
	1	60.50 s		78.09 s	
Singular 3-1-0	1	1163.49 s		349.06 s	
Magma 2.16-7	1	361.42 s		597.51 s	
Pari 2.3.3	1	692.59 s		382.74 s	

Tables 1 and 2 present times to multiply and divide with coefficients in $\mathbb{Z}/32003$ and $\mathbb{Z}$. The parallel heap algorithms generally achieve superlinear speedup on the Core i7 due to their use of extra L2 cache. On the Core 2 architecture the speedup is still fairly good. The sdmp times are similar for $\mathbb{Z}$ and $\mathbb{Z}_p$ because our integer arithmetic assumes word size coefficients. Magma and Singular use faster representations for $\mathbb{Z}_p$ when p is less than 24 or 31 bits.

3.3 Sparse Benchmark

Our last benchmark is a sparse problem with an irregular block pattern. Let $g = (1 + x + y + 2z^2 + 3t^3 + 5u^5)^{12}$ and $q = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^{12}$. We compute $f = q \cdot g$ and divide f/g in lexicographical order $x > y > z > t > u$. The quotient q and the divisor g have 6188 terms and their coefficients are 37 bits. The dividend f has 5.8×10^6 terms and its coefficients are 75 bits.

Table 3: Sparse benchmark $\mathbb{Z}_{32003}$, $W(f,g) = 6.577$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.547 s	2.67x	0.589 s	3.40x
	3	0.658 s	2.22x	0.707 s	2.83x
	2	0.915 s	1.60x	1.004 s	1.99x
	1	1.462 s		2.006 s	
Singular 3-1-0	1	10.520 s		20.860 s	
Magma 2.16-7	1	4.710 s		66.540 s	
Pari 2.3.3	1	113.786 s		65.314 s	
Core 2	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.663 s	2.67x	0.741 s	3.16x
	3	0.813 s	2.18x	0.858 s	2.73x
	2	1.081 s	1.64x	1.196 s	1.96x
	1	1.774 s		2.343 s	
Singular 3-1-0	1	16.940 s		26.140 s	
Magma 2.16-7	1	5.770 s		127.750 s	
Pari 2.3.3	1	132.388 s		74.991 s	

We were surprised that the speedup for division could be higher than for multiplication, but the sequential algorithm for division seems to have lower relative performance. This could be due to the extra work it performs to maintain low complexity. Unlike the parallel algorithm, the method from [13] is highly efficient if the quotient is small.

Table 4: Sparse benchmark $\mathbb{Z}$, $W(f,g) = 6.577$.

Core i7	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.584 s	2.65x	0.675 s	3.23x
	3	0.738 s	2.10x	0.791 s	2.76x
	2	1.002 s	1.54x	1.102 s	1.75x
	1	1.548 s		2.182 s	
Singular 3-1-0	1	25.660 s		32.400 s	
Magma 2.16-7	1	7.780 s		80.200 s	
Pari 2.3.3	1	59.823 s		34.566 s	

Core 2	threads	$f = q \cdot g$		$q = f/g$	
sdmp	4	0.752 s	2.33x	0.817 s	3.02x
	3	0.903 s	1.95x	0.951 s	2.60x
	2	1.205 s	1.46x	1.289 s	1.91x
	1	1.759 s		2.468 s	
Singular 3-1-0	1	36.840 s		40.090 s	
Magma 2.16-7	1	9.930 s		137.460 s	
Pari 2.3.3	1	65.362 s		37.582 s	

4. CONCLUSION

We presented a fast new parallel algorithm for division of sparse polynomials on multicore processors. The algorithm was designed to achieve very high levels of performance and superlinear speedup on a problem that could be considered inherently sequential. Our benchmarks show that with few exceptions, this was achieved in practice. This has made us cautiously optimistic towards parallel computer algebra.

Our next task is to integrate the routines into the Maple computer algebra system. By parallelizing basic operations at a low level, we hope to obtain noticable parallel speedup for users and library code at the top level.

Acknowledgements

We gratefully acknowledge the MITACS NCE of Canada and NSERC of Canada for funding this work, and we thank the anonymous referees for their helpful comments.

5. REFERENCES

[1] D. Bini, V. Pan. Improved parallel polynomial division. *SIAM J. Comp.* **22** (3) 617–626, 1993.

[2] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symb. Comp.*, **24** (3-4) 235–265, 1997

[3] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *ACM SIGSAM Bulletin*, **37** (1) 4–15, 2003.

[4] M. Gastineau, J. Laskar. Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries. *Proc. ICCS 2006*, Springer LNCS 3992, 446–453.

[5] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 3.1.0 – A computer algebra system for polynomial computations, 2009. `http://www.singular.uni-kl.de`

[6] L.H. Harper, T.H. Payne, J.E. Savage, E. Straus. Sorting X+Y. Comm. ACM 18 (6), pp. 347–349, 1975.

[7] S.C. Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, **8** (3) 63–71, 1974.

[8] X. Li and M. Moreno Maza. Multithreaded parallel implementation of arithmetic operations modulo a triangular set. *Proc. of PASCO '07*, ACM Press, 53–59.

[9] T. Mattson, B. Sanders, B. Massingill. Patterns for Parallel Programming. Addison-Wesley, 2004.

[10] M. Matooane. Parallel Systems in Symbolic and Algebraic Computation. Ph.D Thesis, Cambridge, 2002.

[11] M. Monagan, R. Pearce. Parallel Sparse Polynomial Multiplication Using Heaps. *Proc. of ISSAC 2009*, 295–315.

[12] M. Monagan, R. Pearce. Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. *Proc. of CASC 2007*, Springer LNCS 4770, 295–315.

[13] M. Monagan, R. Pearce. Sparse Polynomial Division Using a Heap. *submitted to J. Symb. Comp.*, October 2008.

[14] A. Norman, J. Fitch. CABAL: Polynomial and power series algebra on a parallel computer. *Proc. of PASCO '97*, ACM Press, pp. 196–203.

[15] P. Wang. Parallel Polynomial Operations on SMPs. *J. Symbolic. Comp.,* **21** 397–410, 1996.