# Parallel Sparse Polynomial Multiplication 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 high performance algorithm for multiplying sparse distributed polynomials using a multicore processor. Each core uses a heap of pointers to multiply parts of the polynomials using its local cache. Intermediate results are written to buffers in shared cache and the cores take turns combining them to form the result. A cooperative approach is used to balance the load and improve scalability, and the extra cache from each core produces a superlinear speedup in practice. We present benchmarks comparing our parallel routine to a sequential version and 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, Multiplication

## 1. INTRODUCTION

As multicore computers become standard there arises a need to exploit the additional computing power provided by Moore's law. This paper develops a parallel algorithm for multiplying sparse distributed polynomials that is designed to extract maximum performance from modern processors where multiple identical cores share a large L3 cache.

Let $f$ and $g$ be polynomials stored in a sparse distributed format that is sorted with respect to a monomial order $<$. We shall write the terms of $f$ as $f = f_1 + f_2 + \cdots + f_{\#f}$ and $g$ as $g = g_1 + g_2 + \ldots + g_{\#g}$. Our task is to compute the product $h = f \times g = \sum_{i=1}^{\#f} \sum_{j=1}^{\#g} f_i g_j$.

The fastest sequential algorithm is due to Johnson in [7]. It uses a binary heap to simultaneously merge each $f_i \times g$. Elements of the heap contain a pointer to $f_i$ and $g_j$ and the monomial of $f_i g_j$. After a product $f_i \times g_j$ is extracted from the heap and added onto the end of the result, we compute and insert $f_i \times g_{j+1}$. The algorithm is fast because the heap has at most $\#f$ elements so it often fits in the CPU cache.

For sparse multiplications that produce $O(\#f\#g)$ terms Johnson's heap algorithm uses $O(\#f\#g \log \min(\#f, \#g))$ monomial comparisons which is the best complexity known. For dense multiplications that produce $O(\#f + \#g)$ terms Monagan and Pearce [11, 12] show how to modify the heap so that only $O(\#f\#g)$ comparisons are performed, like in a divide-and-conquer multiplication.

Previously we found that a heap could be the fastest way of multiplying sparse polynomials. In [11] we found that it beat Yan's geobucket strategy from [21] despite performing twice as many monomial comparisons, a result we attribute to its use of cache. In [12] a heap beat all of the computer algebra systems tested on both sparse and dense problems, including systems with a recursive dense representation.

**Figure 1: A Symmetric Multiprocessor System.**

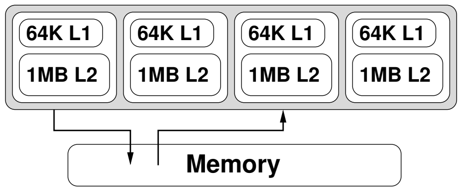
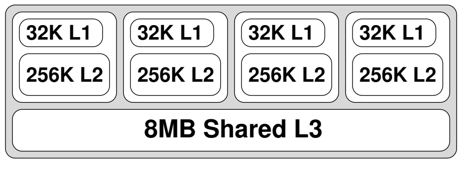


Figure 1 shows a symmetric multiprocessor machine with shared memory. The four cores may be separate processors or part of the same physical chip, but they communicate in random access memory. For problems that are CPU bound this is often sufficient. For example, if each core performs a dense polynomial multiplication using a classical algorithm it does $O(\#f\#g)$ multiplications to produce $O(\#f + \#g)$ terms. Most of the work is independent, and the relatively small results can be merged efficiently in shared memory.

In a memory bound computation the cores cooperate to process a massive amount of data. The speed of memory is already a bottleneck so you can not get a parallel speedup if RAM is also used for communication. Figure 2 shows the Intel Core i7 processor introduced at the end of 2008. Each core now has a smaller L2 cache for its own computations, but a large shared L3 cache provides a way to transfer data between cores. AMD processors also use this design, which is what we need to run memory bound parallel algorithms.

**Figure 2: The Core i7 Quad Core CPU.**

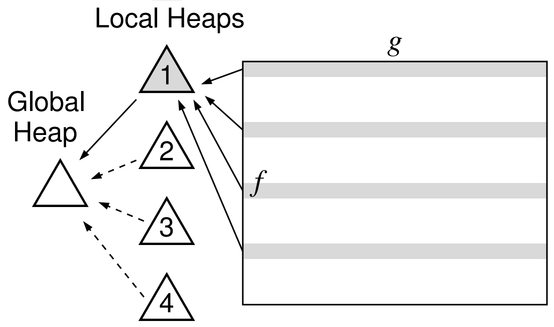


Sparse polynomial multiplication is memory bound when the product has $O(\#f\#g)$ terms. Our algorithm, which we present in §2, was designed for processors like the Core i7. In §3 we present benchmarks, and in §4 we discuss previous work and the prospects for multicore computer algebra.

## 2. ALGORITHM

In designing the parallel algorithm we chose to give each core the task of merging some of the $\{f_i g\}$ with a heap. In addition to being the fastest method, it makes the best use of the small cache in each core. Our heaps use five words of memory per term of $f$, and only three words if the problem is dense, so if each term of $f$ is two words (128 bits) we can multiply by 18000 terms with the L2 cache on the Core i7. Figure 3 shows the structure of the algorithm. The shaded strips are products $f_i \times g$ that we assign to the first core.

**Figure 3: Parallel Multiplication Using Heaps.**



We partition the problem into strips so that the threads start and stop at the same time. This is needed to achieve linear speedup. We divide the work as follows. Let $X$ be the actual number of CPU cores. We compute $t = \sqrt[3]{\#f}$ and create $p = \min(t/2, X)$ threads. Each thread is given $\#f/p$ terms of $f$ in $t$ blocks of size $t^2/p$ to multiply by $g$. Large blocks improve the cache performance [8] and increase the effectiveness of the "chaining" optimization from [12].

The results from all the threads are merged together in a global heap. If a separate thread were used for this task it would expose our program to context switches, so this task is shared among the threads. After computing some terms each thread tries to acquire a lock for the global heap. If it succeeds the thread enters a critical section to merge some of the intermediate results. Otherwise it continues merging terms from its local heap. We give more details in §2.2.

Finally we must consider how to transfer data from the threads to the global heap. We will have each thread write its result to a finite circular buffer that is read by the global merge. The speed of these buffers is critical. Larger buffers reduce synchronization but steal cache from the algorithm. On sparse problems the buffers' cost largely determines the parallel speedup. This problem was difficult to solve so we explain our solution in detail.

### 2.1 Buffer Implementation

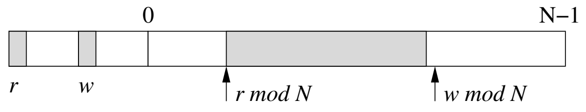
To efficiently transfer data between the cores, we need to understand how the processor in Figure 2 works. Consider a word of data stored at a memory address $x$. When a core loads the value at $x$ into a register the data is fetched from memory and a copy is kept in the L1 cache. For an inclusive cache extra copies are kept in the L2 and L3 caches as well. Then if the core modifies the value at $x$, the new value is written to the cache(s) and main memory is updated later.

If a second core now modifies the value at $x$, for example by setting the value to zero, a *cache coherence protocol* [17] is used to invalidate any copies of the data in the caches of the other cores. The next time a core reads $x$ it must fetch the data from L3. This process is costly, and it is the main reason why sharing variables in multiple threads is slow.

Now consider the problem of transferring data from one thread to another using a finite array. This is the classical "producer and consumer" problem often seen in textbooks. A typical solution involves semaphores that are modified by both threads whenever data is written or read. In practice this approach can not be that fast.

Our buffer is based on [15], which allows the number of items written and read to be updated independently. In the structure below, $w$ and $r$ are counters that are initialized to zero. Their values modulo the buffer size give an index into the array where the data is stored. When the buffer size is a power of two this division may be done with bitwise *and*. We have modified the buffer to place $w$ and $r$ on different cache lines so that false sharing does not occur.

**Figure 4: Circular Buffer Implementation.**



```
#define N     65536   /* size in words (512 K) */
#define MASK  (N-1)   /* (i & MASK) = i mod N  */
#define LINE  64      /* bytes per cache line  */
struct fifo {
     long r;          /* words read */
     char pad1[LINE - sizeof(long)];
     long w;          /* words written */
     char pad2[LINE - sizeof(long)];
     long data[N];
};
```

For good performance the reader must avoid accessing $w$ and the writer must avoid accessing $r$. The value of $w - r$ is the number of words that are already in the buffer. The reader can read all of those words before accessing $w$ again. Likewise, $N - w + r$ is the number of words that can be written to the buffer before the writer must access $r$ again. On average this reduces synchronization by a factor of $N/2$.

Note that the approach requires some way to atomically store a *long*. For most architectures this is an inherent part of the instruction set and C code will suffice. Otherwise an atomic operation such as *compare and swap* could be used. Another problem is that $w$ and $r$ can only be incremented after data is written or read, but compilers and processors may reorder loads and stores. Memory barriers are needed to enforce the correct order, as shown below.

```
void fifo_put1(struct fifo *f, long v) {
     f->data[f->w & MASK] = v;        /*  f->w mod N  */
     write_barrier();                 /* finish write */
     f->w++;
}

long fifo_get1(struct fifo *f) {
      long v = f->data[f->r & MASK];  /*  f->r mod N  */
      read_barrier();                 /* finish read  */
      f->r++;
      return v;
}
```

## 2.2 Scheduling

Our next task is to balance the work done by the threads between the local and the global heap. Suppose one thread produces a block of terms that are all strictly greater than the terms being produced by the other threads. This block is consumed by the global heap immediately. In that case we want the thread producing the block to do less work on the global heap to avoid starvation, while the other threads do more work on the global heap to compensate.

It is important to do this load balancing without a lot of communication because the task is already memory bound. Our threads act independently, in a manner similar to [16]. Each thread uses the number of terms in its own buffer to determine the role it should play during the computation.

Let $p$ be the number of threads and let $N$ be the buffers' capacity in terms. In the main loop for each thread we will know $k$, the number of terms in the buffer the last time we checked. We will compute $\min(N-k, N/p)$ terms and add them to the buffer, update $k$, and try to acquire the lock for the global heap. When we get the lock we will compute up to $\min(k, N/p)$ new terms of the result. Otherwise, we will continue merging terms from the local heap.

The ratio $N/p$ ensures that the threads synchronize with a minimum frequency proportional to the number of cores. This does not limit scalability in practice, since we already use at most $(\sqrt[3]{\#f})/2$ threads. Our primary concern is the actual performance with 2 to 64 cores, and experimentation lead us to use this approach.

One tricky problem that we solved was sharply degraded performance on a system under load. With other programs running in the background, our performance is proportional to the number of available cores. This is accomplished by a short sleep of 10 microseconds when a thread's buffer is full and it can not acquire the lock for the global heap. Rather than busywait we briefly yield to the operating system. The length of time controls how aggressive our program is when other programs are running and was chosen by experiment. We trust the operating system to run the threads at about the same rate and to maintain processor affinity. If no other programs are running the algorithm rarely blocks.

## 2.3 Pseudo Code

We now present the algorithm, beginning with the main routine that is run on each thread. This function merges a subset of the partial products $f_i g$ and writes the result to a buffer $B$. The input includes the total number of threads $p$ and a unique number $r \in \{1, 2, \ldots p\}$. The threads multiply every $p^{th}$ term of $f$ by $g$, starting with $f_r$. We removed the cache blocking on $f$ to simplify the presentation.

Products in the heap are represented as triples $[\,i, j, M_{ij}\,]$ and denoted $f_i \times g_j$. $M_{ij}$ is the monomial of $f_i g_j$, which we compute when inserting the product into the heap. We also refer to that monomial as $mon(H_1)$ where $H_1$ is a product. Finally, $mon(f_i)$ and $cof(f_i)$ denote the monomial and the coefficient of a polynomial term $f_i$.

The main loop proceeds in three stages. First, the largest element of the heap is extracted, giving the monomial $M$ of the next term. Coefficients are multiplied and accumulated in the variable $C$ as products with the same monomial are extracted and merged. We keep track of the products that are merged by adding them to a set $Q$.

Second, using $Q$ we insert the successors of each product that was merged in stage one. For $f_i \times g_j$ we insert $f_i \times g_{j+1}$ if it exists, and for $j = 1$ we insert the next term of $f$ times $g_1$ if it exists. This avoids adding new elements to the heap until they must be compared. Combined with the chaining optimization, it reduces the complexity of dense univariate multiplications to $O(\#f\#g)$ [11, 12]. Finally, if the current term is non-zero we insert it into the buffer $B$.

In the third stage we recompute the number of terms in the buffer and try to acquire the lock for the global heap. Both operations read data from other cores, so this stage is executed infrequently. We use a counter $k$ that decrements as the buffer is filled, starting from at most $N/p$. When the buffer is full and the global heap lock can not be acquired, we sleep for 10 microseconds before trying again.

Before each thread terminates, it must signal that it has finished sending terms and is not simply blocked. We store this information in the buffer structure, and write $close(B)$ for the setting of this data.

**Subroutine: Local Heap Merge.**

```
Input:    $f, g \in \mathbb{Z}[x_1, ..., x_n]$, monomial order $<$, buffer $B$,
          thread number $r$, total number of threads $p$.
Output:   terms of the product are written to $B$.
Locals:   heap $H$, set $Q$, monomial $M$, coefficient $C$.
Globals:  lock $L$, buffer capacity $N$ (in terms).
  $H :=$ an empty heap ordered by $<$ with max element $H_1$
  insert $[\,r, 1, mon(f_r) \cdot mon(g_1)\,] = f_r \times g_1$ into $H$
  $k := N/p$
  while $|H| > 0$ do
    $(i, j, M) := extract\_max(H)$
    $C := cof(f_i) \cdot cof(g_j)$
    $Q := \{(i, j)\}$
    while $|H| > 0$ and $mon(H_1) = M$ do
      $(i, j, M) := extract\_max(H)$
      $C := C + cof(f_i) \cdot cof(g_j)$
      $Q := Q \cup \{(i, j)\}$
    for all $(i, j) \in Q$ do
      if $j < \#g$ then
        insert $f_i \times g_{j+1}$ into $H$
      if $j = 1$ and $i + p \leq \#f$ then
        insert $f_{i+p} \times g_1$ into $H$
    if $C \neq 0$ then
      insert the term $(C, M)$ into the buffer $B$
    $k := k - 1$
    while $k = 0$ do
      $k = |B|$
      if $trylock(L)$ then
        $global\_heap\_merge(min(k, N/p), <)$
        $release(L)$
      else if $k = N$ then
        sleep for 10 microseconds
      $k = min(N - k, N/p)$
  $close(B)$
  return
```

Our next routine is the global merge which can be called by any thread. When it is first called, some buffers may be empty and the global heap does not yet exist. The function begins by building the global heap, and it aborts if a buffer is empty but is still sending terms. Once every buffer has a term in the heap it computes the next term of the result.

We will use a global set $P$ to initialize the global heap $G$ similar to how $Q$ was used to update $H$ in the last routine. $P$ is a set of buffers that is pre-initialized by the outermost routine. The heap elements are now of the form $[\,B, C, M\,]$, where $C$ and $M$ are the coefficient and monomial of a term and $B$ is a pointer to the buffer they came from. There is a test for $|G| = 0$ since we may discover that all buffers have stopped sending terms, in which case we should quit.

```
Subroutine: Global Heap Merge.
Input:    max iterations $i$, monomial order $<$.
Output:   terms of the result are written to $h$.
Locals:   coefficients $K, C$, monomial $M$, buffer $B$.
Globals:  heap $G$, set $P$, polynomial $h$.
  while $i > 0$ do
    $i := i - 1$
    for all $B$ in $P$ do
      if $B$ is not empty then
        extract the next term $(C, M)$ from the buffer $B$
        insert $[B, C, M]$ into $G$
        $P := P \setminus \{B\}$
      else if not $is\_closed(B)$ then
        return
    if $|G| = 0$ then return
    $(B, C, M) := extract\_max(G)$
    $P := \{B\}$
    while $|G| > 0$ and $mon(G_1) = M$ do
      $(B, K, M) := extract\_max(G)$
      $C := C + K$
      $P := P \cup \{B\}$
    if $C \neq 0$ then
      append the term $(C, M)$ to $h$
  return
```

Finally we present the outermost routine that sets up the algorithm and runs the threads. Once they finish it merges any remaining terms and returns the product $h$. There are some important things to note. The buffers must be passed by reference so that threads write to the objects in $P$. The lock protects $P$ as the threads are spawned.

```
Algorithm: Sparse Polynomial Multiplication.
Input:    $f, g \in \mathbb{Z}[x_1, \ldots, x_n]$, monomial order $<$,
          number of threads $p$.
Output:   $h = f \cdot g$.
Globals:  heap $G$, set $P$, lock $L$, buffer capacity $N$ (in terms),
          result polynomial $h$.
  $G :=$ an empty heap ordered by $<$ with max element $G_1$
  $P :=$ a set of $p$ empty buffers
  $L :=$ an unheld lock
  $h := 0$
  $lock(L)$
  for $i$ from 1 to $p$ do
    spawn $local\_heap\_merge(f, g, <, P_i, i, p)$
  $release(L)$
  sleep until all threads complete
  $global\_heap\_merge(\#f\#g, <)$
  return $h$
```

Our implementation has two additional optimizations to improve the performance. First, the merge of all remaining terms is performed in *local_heap_merge* by the thread which merges $f_{\#f} \times g$, i.e., the thread assigned the last term of $f$. Second, only $p - 1$ threads are spawned; the parent thread is used to run the last *local_heap_merge*.

## 2.4 Implementation

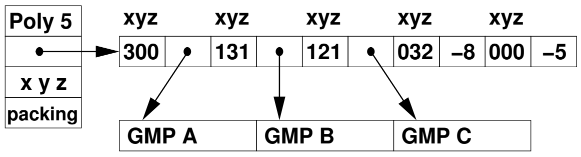
We implemented the algorithm in C using Posix threads. In our software (sdmp), we represent polynomials as sorted arrays of monomial and coefficient pairs. Monomials can be one or more words, and we have an option to pack multiple exponents into each word to increase speed and reduce the storage cost. We use word operations to compare, multiply, and divide monomials. The coefficients are always integers.

An integer coefficient $x$ is stored as $2x + 1$ if $|x| < 2^{\beta - 2}$ where $\beta$ is the base of the machine. For example $x = 5$ is stored as 11. We use assembly code to accumulate products of these integers [12]. For multiprecision integers we store a pointer to a GMP structure and call the *mpz_t* routines [6]. The data structure for the polynomial

$$Ax^3 + Bxy^3z + Cxy^2z - 8y^3z - 5$$

in lexicographical order with $x > y > z$ where $A, B$ and $C$ are multiprecision integers is shown below.

**Figure 5: Polynomial Data Structure**



Among the optimizations we made to the single threaded routine, the most important was chaining (see [12]). When a new element is inserted into the heap we do comparisons to determine its position. If we detect equality, rather than enlarge the heap we attach the new element to the existing one to form a chain (a linked list) of like terms. We extract and merge all elements in a chain with one heap operation. Chaining reduces both the size of the heap and the number of monomial comparisons.

Another optimization is that each thread reuses working storage while computing the coefficient of the current term. The single threaded algorithm generates no garbage, and if all the integers are small then the multithreaded algorithm generates no garbage as well. However, when multiprecision coefficients are written to a buffer we copy them into blocks of memory and write pointers instead. In the future we plan to recycle these blocks to reduce the garbage generated.

## 3. BENCHMARKS

We conducted benchmarks using two different quad core processors. The first is an Intel Core i7 920 2.66GHz. This processor is shown in Figure 2. It has a large 8MB shared L3 cache which we thought would speed up memory bound parallel algorithms. We also tested the older Intel Core 2 Q6600 2.4GHz which is commonly found in desktop PCs. This processor resembles Figure 1 except that each pair of cores shares a faster 4MB L2 cache.

The Core i7 computer has 6GB of DDR3 RAM and runs Fedora 10 Linux with the 2.6.27 kernel. The Core 2 machine has 4GB of DDR2 RAM and runs Fedora 9 with the 2.6.26 kernel. Our software (sdmp) was compiled using GCC 4.3.2. These benchmarks use lexicographical order with all of the exponents packed into one 64-bit word.

Although the Core i7 has four cores, each core is able to simultaneously execute two threads, albeit at a slower rate. We disabled this feature because the operating system did not always distribute the load across physical cores, leading to inconsistent times for 2 to 6 threads. In the benchmarks speedup is calculated relative to the sequential routine [12], we did not run the parallel algorithm with only one thread.

## 3.1 Sparsity and Speedup

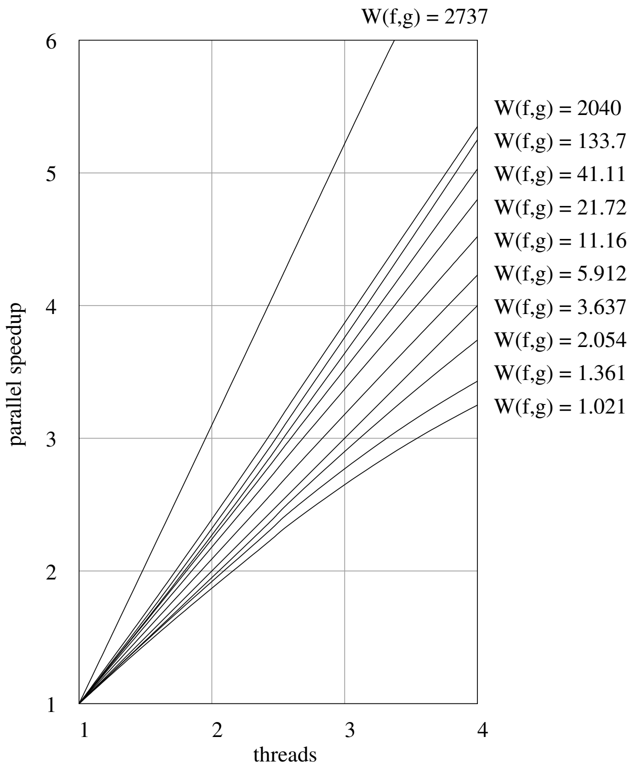
We generated random univariate polynomials with 8192 terms and coefficients chosen from $(-99, 99)$. Starting with $e_0 = 0$, the next exponent $e_{i+1}$ was chosen by adding a random integer to $e_i$. Two polynomials were generated and multiplied modulo 32003. The range of the random integers used to generate the exponents affects the sparsity of the computation. We measure the sparsity of $h = f \times g$ as the *work per term*

$$W(f,g) = (\#f\#g)/\#h$$

that is needed to produce the result (see [12]). For example, $W(f,g) = 1.021$ means $8192^2/1.021 = 65.7$ million terms were produced totalling 1000 MB. In Figure 6 we graph the speedup obtained for different sparsities on the Core i7.

**Figure 6: Sparsity vs. Parallel Speedup over $\mathbb{Z}_p$**

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



As sparsity increases and $W(f,g) \to 1$, communication increases and parallel speedup decreases. This overhead is offset by the extra cache available to the parallel algorithm. Overall the result is very good. We have linear speedup at $W(f,g) = 3.637$ and we never drop below $3x$ on four cores. Our results for $\mathbb{Z}$ are identical except that $W(f,g) = 1.021$ reaches only $3.12x$. For dense problems the heaps collapse and the algorithm is $4.66x$ faster on four cores. Otherwise, problems that generate fewer than 1.85 million terms run at least five times faster using four threads on the Core i7.

The Core i7 has exceptional performance because of its combination of dedicated caches and a large shared cache. The Core 2 Q6600 has 32KB of dedicated cache per core and did not perform as well. We achieved linear speedup at $W(f,g) = 6$, and from $W(f,g) = 11$ to $W(f,g) = 2040$ the speedup varied between $4.2x$ and $4.7x$ with four cores. For $W(f,g) = 2737$ the speedup reached $6.02x$, and on dense problems it reached $4.4x$. On sparse problems the speedup was poor because each pair of cores communicates over the front side bus. For $W(f,g) = 1.021$ the speedup was $1.79x$, $2.70x$, and $2.67x$ with 2, 3, and 4 cores respectively, which suggests that we were limited by the memory bandwidth.

## 3.2 Dense Benchmark

Let $f = (1 + x + y + z + t)^{30}$ and $g = f + 1$. We multiply $h = f \times g$. The polynomials have 46376 terms and 61 bit coefficients and their product has 635376 terms and 128 bit coefficients. The problem is due to Fateman in [3].

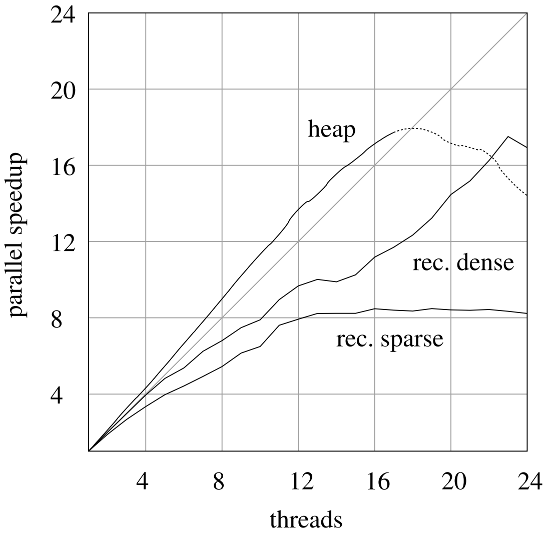
In addition to our software (sdmp), we tested Maple [10], Magma [2], Singular [5], Pari/GP [14], and Trip [4]. Trip is the only other multithreaded program and it supports both recursive sparse and recursive dense polynomials. We used rational coefficients to benchmark Trip. Pari also uses the recursive dense representation but with integer coefficients. Maple, Magma, Singular, and sdmp use sparse distributed representations with integer coefficients.

**Table 1: Fateman benchmark, $W(f,g) = 3332$.**

| | threads | Core i7 | | Core 2 Quad | |
|---|---|---|---|---|---|
| sdmp | 4 | 11.48 s | 6.15x | 14.15 s | 4.25x |
| | 3 | 16.63 s | 4.24x | 19.43 s | 3.10x |
| | 2 | 28.26 s | 2.50x | 28.29 s | 2.13x |
| | 1 | 70.59 s | | 60.25 s | |
| Trip 1.0 beta2 recursive dense | 4 | 23.76 s | 3.89x | 26.86 s | 3.94x |
| | 3 | 31.05 s | 2.97x | 35.65 s | 2.97x |
| | 2 | 46.56 s | 1.98x | 52.98 s | 1.99x |
| | 1 | 92.38 s | | 105.78 s | |
| Trip 1.0 beta2 recursive sparse | 4 | 29.36 s | 3.26x | 31.95 s | 3.38x |
| | 3 | 36.00 s | 2.66x | 39.96 s | 2.71x |
| | 2 | 50.96 s | 1.88x | 56.68 s | 1.91x |
| | 1 | 95.74 s | | 108.15 s | |
| Magma 2.15-8 | 1 | 526.12 s | | | |
| Pari/GP 2.3.3 | 1 | 642.74 s | | 707.61 s | |
| Singular 3-1-0 | 1 | 744.00 s | | 1048.00 s | |
| Maple 13 | 1 | 5849.48 s | | 9343.68 s | |

This result is also very good, but notice how sdmp with one thread is slower on the newer and faster Core i7 CPU. Intel has traded raw sequential performance in the form of a large L2 cache for improved parallel performance with an even larger but slower L3 cache and smaller dedicated L2s. Multiple threads are required to achieve peak performance on this architecture.

**Figure 7: Fateman benchmark in shared memory.**



William Stein kindly offered us the use of a large machine to test the scalability of the algorithms in shared memory. The computer is a four-way Intel "Dunnington" system for which we acknowledge National Science Foundation Grant DMS-0821725. It has four 2.66GHz Xeon X7460 CPUs with six cores each and 16 MB of shared L3 cache. The cores in different CPUs communicate through shared memory.

Figure 7 shows the speedup of the parallel algorithms on the large machine. The heap can use up to 18 cores before the performance drops off due to high latency. Sdmp would normally limit the computation to $\sqrt[3]{46376}/2 = 17$ threads, which takes 3.07 seconds at $17.73x$. Trip's recursive dense algorithm scales nicely but can not maintain linear speedup past five threads. Its best time was 5.416 seconds at $17.51x$. The recursive sparse algorithm scaled to $8.22x$ on 13 cores, taking 11.73 seconds, but beyond that we saw only minimal increases in performance.

## 3.3 Sparse Benchmark

Our final benchmark is a challenge problem where we did not do as well. Let $f = (1 + x + y + 2z^2 + 3t^3 + 5u^5)^{12}$ and $g = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^{12}$. The polynomials have 6188 terms and 37 bit coefficients. Their product has 5821335 terms and 75 bit coefficients.

We used lexicographical order with $x > y > z > t > u$, but the result for graded lexicographical order was similar. The multiplication is quite sparse and the speedup is poor. Based on the sparisty we expected linear speedup.

**Table 2: Sparse benchmark, $W(f, g) = 6.577$.**

| | threads | Core i7 | | Core 2 Quad | |
|---|---|---|---|---|---|
| sdmp | 4 | 0.59 s | 2.64x | 0.71 s | 2.60x |
| | 3 | 0.74 s | 2.11x | 0.88 s | 2.11x |
| | 2 | 1.00 s | 1.56x | 1.22 s | 1.52x |
| | 1 | 1.56 s | | 1.86 s | |
| Trip 1.0 beta2 recursive dense | 4 | 1.49 s | 3.19x | 1.85 s | 3.19x |
| | 3 | 1.87 s | 2.55x | 2.31 s | 2.55x |
| | 2 | 2.59 s | 1.84x | 3.21 s | 1.65x |
| | 1 | 4.76 s | | 5.90 s | |
| Trip 1.0 beta2 recursive sparse | 4 | 1.19 s | 2.50x | 1.62 s | 2.36x |
| | 3 | 1.39 s | 2.15x | 1.85 s | 2.06x |
| | 2 | 1.78 s | 1.68x | 2.32 s | 1.65x |
| | 1 | 2.99 s | | 3.83 s | |
| Magma 2.15-5 | 1 | 15.48 s | | | |
| Pari/GP 2.3.3 | 1 | 59.80 s | | 65.35 s | |
| Singular 3-1-0 | 1 | 26.00 s | | 38.00 s | |
| Maple 13 | 1 | 135.89 s | | 199.86 s | |

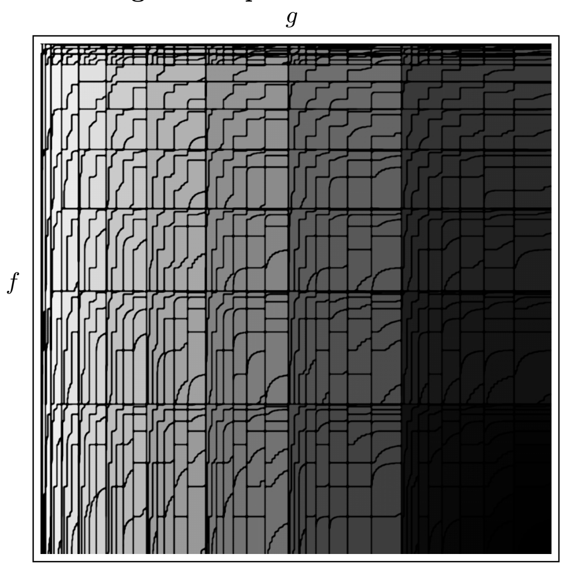
Upon closer examination, we found a highly non-uniform structure in the order of the products merged. In Figure 8 the products $f_i \times g_j$ are colored by rank with respect to $<$. The first product merged is white and the last one is black. Using edge-detection, we identified regions that are merged essentially one after another. They are arranged in vertical strips of blocks, all of which are merged sequentially.

At first we thought that this pattern might be difficult to merge. Bottlenecks can be produced in our algorithm when buffers are filled and emptied unevenly. However, the result is actually due to chaining. We disabled it and the time for one thread on the Core i7 jumped to 3.72 seconds, whereas the time for four threads increased to only one second. The result is much closer to linear speedup.

The sequential algorithm is particularly efficient because the blocks and regions tend to make chains. In the parallel algorithm they are split up and merged in different threads which sharply reduces the effectiveness of the strategy.

We offer this problem as a challenge for other algorithms. If a larger problem is desired one can raise the polynomials to the 16th power instead of the 12th.

**Figure 8: Sparse Benchmark**



## 4. CONCLUSION

Early implementations of parallel algorithms for polynomial multiplication include the work of Wang in [18] and Fitch and Norman [13]. For dense univariate polynomials $f = a_0 + a_1x + ... + a_mx^m$ and $g = b_0 + b_1x + ... + b_nx^n$ Wang used the following formula to multiply $f \times g$:

$$\sum_{k=0}^{m+n} C_k x^k \quad \text{where} \quad C_k = \sum_{i+j=k} a_i b_j.$$

Here, the $C_k$ can be computed *independently* in parallel. This approach could be applied to multivariate $f(x, y, z, ...)$ and $g(x, y, z, ...)$ if they were represented in *recursive form*, that is, as polynomials in $x$ with coefficients $a_i$ and $b_j$ that are polynomials in the other variables. This strategy is used by Trip [4].

However, if the polynomials are dense then FFT based methods are likely to be faster. Xavier and Iyengar's text [20] on parallel algorithms describes how to use an FFT to multiply in parallel and Bini and Pan in [1] give an FFT based parallel division algorithm. In [9], Maza et al. implemented multivariate polynomial multiplication modulo a (dense) triangular set modulo a prime using the FFT. But for sparse polynomials the FFT is not applicable.

In this paper we have taken the fastest known sequential method for multiplying sparse polynomials and parallelized it. We focused on multicore processors because they are in almost every type of computer, and we designed for newer CPUs where shared cache is a standard feature.

Our benchmarks show that substantial speedups may be realized because of the extra cache available to the parallel algorithm. With four cores we often achieve a factor of five speedup, and linear speedup is achieved in all but the most extreme cases of sparsity. We believe that this result will be typical in the multicore era, and that multicore processors, far from being a burden, offer tremendous potential for new parallel program designs. The old free lunch is replaced by a new one, where the price of entry is that we must design software to exploit the memory hierarchy.

## 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 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) (2003) 4–15.

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

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

[6] T. Granlund. The GNU Multiple Precision Arithmetic Library, version 4.2.2. `http://www.gmplib.org/` (2008).

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

[8] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review.*, **25** (1991) 63–74.

[9] X. Li and M. Moreno Maza. Multithreaded parallel implementation of arithmetic operations modulo a triangular set. *Proc. of PASCO '07*, ACM Press, 53–59, 2007.

[10] M. Monagan, K. Geddes, K. Heal, G. Labahn, S. Vorkoetter, J. McCarron, P. DeMarco. *Maple 13 Introductory Programming Guide* Maplesoft, 2009.

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

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

[13] A. Norman, J. Fitch. CABAL: Polynomial and power series algebra on a parallel computer. *Proc. of PASCO '97*, ACM Press, pp. 196–203, 1997.

[14] PARI/GP, version `2.3.4`, Bordeaux, 2008, `http://pari.math.u-bordeaux.fr/`.

[15] D. Reed, R. Kanodia. Synchronization with eventcounts and sequencers. *Comm. of the ACM*, **22** (2) (1979) 115–123.

[16] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. *Proc. of the third annual ACM symposium on Parallel algorithms and architectures.*, (1991), 237–245.

[17] P. Sweazey, A. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. *Proc. of 13th Annual International Symposium on Computer Architecture*, (1986), 414–423.

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

[19] B. Wilkinson, M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, 1999.

[20] C. Xavier, S. Iyengar. *Introduction to Parallel Algorithms* Wiley, 1998. Section 10.5 has an FFT based univariate multiplication.

[21] T. Yan. The Geobucket Data Structure for Polynomials. *J. Symb. Comput.* **25** (1998) 285–293.