Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries

Mickaël Gastineau and Jacques Laskar

Paris Observatory - IMCCE - CNRS UMR8028
77, avenue Denfert Rochereau
75014 Paris
gastineau@imcce.fr, laskar@imcce.fr

Abstract. Flat vector representation of sparse multivariate polynomials is introduced in the computer algebra system TRIP with specific care to the cache memory. Burst tries are considered as an intermediate storage during the sparse multivariate polynomial multiplication by paying attention to the memory allocations. Timing and memory consumption are examined and compared with other recursive representations and other computer algebra systems.

1 Introduction

A large number of celestial mechanics applications, such as the classical expansions of the Kepler problem and several expansions of disturbing functions, requires to handle multivariate generalized power series. We develop a general symbolic and numeric computer algebra system TRIP [1] dedicated to celestial mechanics. It handles generalized power series (1) : Coefficients $C_{j,k}$ can be numerical coefficients (fixed and multi-precision rational number, double and quadruple precision floating-point) or rational functions. Complex expressions of the form $\exp^{\imath\lambda_m}$ are encoded as variables, so negative exponents are permitted.

$$S(X_1,...X_n,\lambda_1,...,\lambda_m) = \sum_{j_1...j_n,k_1...k_m\in\mathbb{ZZ}} C_{j,k}X_1^{j_1}....X_n^{j_n}\exp^{\imath k_1\lambda_1+....+k_m\lambda_m} \tag{1}$$

TRIP is tuned to compute large series with millions of terms depending on a large number of variables. Computation of large series needs fast in-memory data storage and fast algorithms. Computer algebra systems have different internal representations of polynomials and series. Internal representations should depend on the sparsity of polynomials in the studied problem. Some symmetries are present in celestial mechanics problems, such as d'Alembert relations in the planetary motion (see Laskar, [2]). This implies that sparse series appear during computations. Specialized computer algebra systems, called Poisson Series Processors, handle Poisson series as list of objects (e.g., see [3], [4], or [5]). General

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 446–453, 2006.

computer algebra systems represent sparse polynomial as a recursive list or a recursive vector but they always have only one of these types to handle sparse polynomials. TRIP have three internal representations for sparse polynomials, discussed in Section 1.

An efficient multiplication algorithm is critical for most computations on series. Hida [6] demonstrates the effects of polynomials representation and blocking loops on the execution time to perform a sparse polynomial multiplication. We investigate here the usage of burst tries [7] as an intermediate representation during the sparse polynomial multiplication. Memory management for this intermediate representation is investigated in order to reduce memory consumption and improve scalability on SMP hardware. In section 6, we compare performance of polynomials multiplication in TRIP with other computer algebra systems.

2 Series Representations

TRIP supports multiple memory representations of a polynomial in order to handle it efficiently. Polynomials are always stored in an expanded form. In the two first representations described in (Laskar, [8]), multivariate polynomials are stored as a recursive list or a recursive vector in memory. The recursive list is a recursive single linked-list containing only the non-zero coefficients with the associated exponents. The recursive vector stores the minimal and maximal exponents and all coefficients between the minimal and maximal exponents are stored in this array. Depending on the polynomial, the recursive list representation consumes more or less memory than the recursive vector.

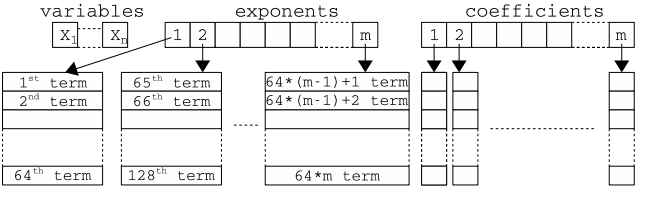

Fig. 1. Flat vector representation of a polynomial with variables $X_1, ...X_n$ containing $64m$ terms. Blocks of exponents are viewed in uncompressed form.

These two previous representations contain many indirections (pointers) to handle large sparse polynomials. Modern computers are less efficient with indirection because load instructions from memory or cache could take long time to complete. In these cases, a processor could stall until the data are available. Hida's experiments [6] to improve cache locality, such as blocking, show a speedup up to 47% on the sparse polynomial multiplication. It shows that flat structure are aware of cache locality.

We introduce a flat vector representation (Fig. 1) in TRIP : Exponents and coefficients of each term are stored in two arrays subdivided in short blocks. Terms

are always sorted on exponents. A short block contains at most 64 terms. Operations on blocks of exponents and on coefficients, when fixed-size coefficients are used, take advantage of cache and SIMD pipelines available on some processors, such as Altivec or SSE extensions. Storing all exponents could consume much more memory than recursive representation. To reduce memory footprint of this representation, we compress blocks of exponents using shift and mask operations when exponents become unused. Compression factor is good because exponents have many times the same values for the most factorized variables in a block. Compression and decompression operations, when polynomial multiplications or additions are performed, show a negligible runtime overhead because it requires only integer arithmetics.

3 Multiplication Using Burst Tries

Multiplication of two sparse polynomials requires to use the naive algorithm of the multiplication because Karatsuba algorithm and FFT methods are only well adapted for dense polynomials. When polynomials are stored in the recursive list or vector, a simple naive recursive algorithm is performed. If the recursive vector or list is viewed as a tree, multiplication consists only as insertion of nodes in a tree. Using the flat vector representation, multiplication requires to sort terms on exponents. Hida [6] uses an hash table to sort terms during multiplication of its flat vectors in order to reduce execution time. To have good performance with hash table, it requires to find an efficient hashing function and the size of the hash table must be sufficiently large in memory. The massively parallel Deprit's algorithm [9] requires much more memory : Their first step needs memory for $2nm$ coefficients to perform the product of two series with n and m terms.

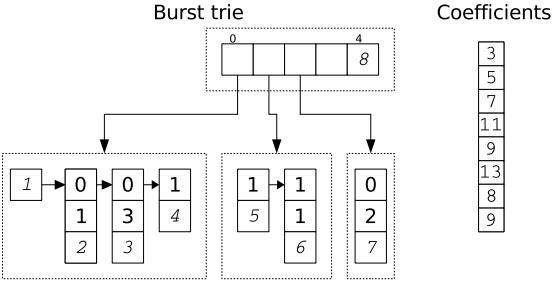

Fig. 2. Burst trie holding $3 + 5z + 7z^3 + 11y + 9zy + 13zyx + 8z^2x^2 + 9x^4$. Here, the burst threshold is set to the value 5 instead of 10. Italic numbers in the burst trie are indices of coefficients in the array of coefficients.

We prefer to find another method to sort terms during multiplication. Exponents in a term could be viewed as a string not of characters but of integers. Efficient data structure to sort strings of characters are binary search trees (BST), splay trees, judy array and tries. Heinz has developed a new data structure, burst

trie [7], to sort large strings in memory. A burst trie consists in an access trie whose leaves are containers. A container, which could be a sorted list or BST, has few data. The access trie can be viewed as a recursive vector whose leaves are list or BST. The trie node contains only the minimal and maximal exponents. The container has all trailing exponents. The trailing zero exponents are not stored in the burst trie. Figure 2 shows an example of burst trie which contains the polynomial $3 + 5z + 7z^3 + 11y + 9zy + 13zyx + 8z^2x^2 + 9x^4$. A burst trie is built during the multiplication and a final step is performed to copy exponents into the vector. When terms are inserted into the burst trie and the number of data in a container reaches a threshold, the container requires to be converted in a trie node with new containers as leaves. This process is called *bursting*. In our implementation, containers are a sorted list and experiments show that bursting must be performed when containers have 10 elements. Coefficients associated with the terms in the burst trie are stored in large blocks and their indexes are stored with the k^{th} exponent if we multiply 2 polynomials of k variables. Figure 3 shows a stable speedup about to 60% over recursive vector when a flat vector representation and multiplication using burst trie as intermediate data structure are used.

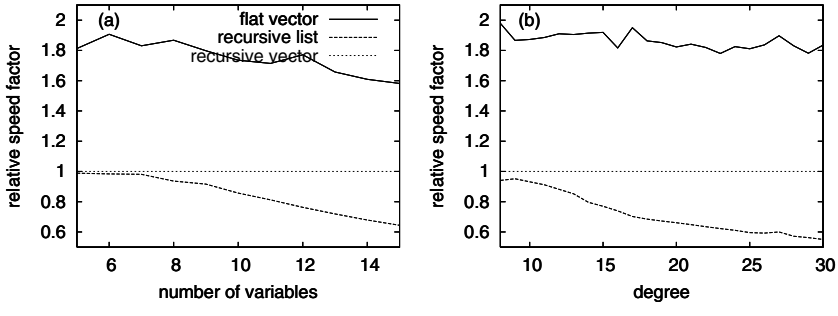

Fig. 3. (a). Relative computation time of $s_1 * (s_1 + 1)$ with $s1 = (1 + x_1 + x_2 + ... + x_p)^{15}$ using different representations versus computation time for the recursive vector. (b). Relative computation time of $s_1 * (s_1 + 1)$ with $s1 = (1 + x + y + z)^p$ using different representations versus computation time for the recursive vector.

4 Parallel Multiplication on SMP Hardware

We decide to parallelize the multiplication operation because it takes most of the time in our computations. Currently, Symmetric Multi-Processing or Multi-Threading Systems are becoming more and more popular and cheaper. TRIP uses threads to implement parallelism in shared memory multiprocessor architectures. All threads within a process share the same address space. Communication between threads is more efficient and easier to use than communication between processes. Threads must take care when they write to shared data : it requires synchronization mechanisms. Thread mechanisms could be implemented using OpenMP API or POSIX Threads API. For POSIX Threads, we

adopt a manager-worker queue model in TRIP : a single thread, the manager assigns work to other threads, the workers. The manager thread uses a dynamic scheduling to split the multiplication operation. All workers are created at the beginning and accept work from a common queue. The number of workers is dynamic : the user can fix the number of workers by setting a global variable in its interactive TRIP session.

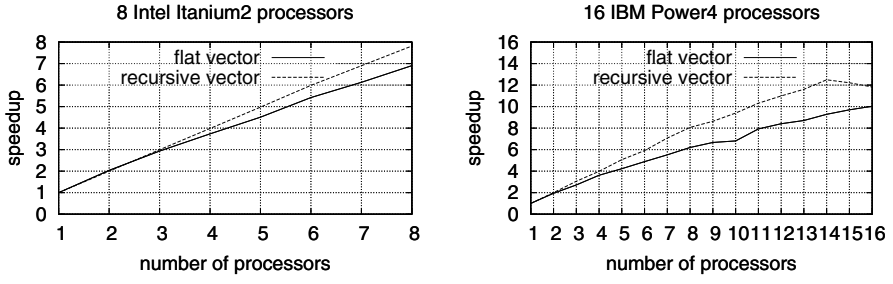

Fig. 4. Speedup of computation $(1+x+3y+5z+7t)^{23} * (1+11x+13y+17z+23t)^{23}$ using different representations with a manager-worker queue model. Computations are performed on a server, equipped with 8 Intel Itanium2 processors (1.5GHz with a 400 MHz system bus) running Linux operating system (kernel 2.6), and another server, equipped with 16 IBM Power4 processors running AIX 5.2 operating system.

Figure 4 shows the scalability of multiplication using the recursive vector and flat vector representations. Recursive vector representation scales linearly but flat vector representation does not scale as well as recursive ones because each thread builds a burst trie and a final step is required to merge all burst tries.

5 Memory Management

Most computer algebra systems use a garbage collector to perform memory operations instead of explicit memory management. TRIP have two objectives for memory management : reduce memory consumption and have efficient performance. Garbage collection generally results in poorer performance or higher space consumption. Garbage collectors require much more memory to be faster than explicit memory management (Zorn, [10] and Hertz, [11]). We decide to use explicit memory management (allocation with system call `malloc` and deallocation with `free`).

The frequency of allocation and the size of allocated objects depend on the representations of series used in TRIP. Flat vectors allocate large blocks of memory but it doesn't happen very often. This representation uses operating system system calls (`malloc` and `free`). A recursive list representation requires many allocations of very small blocks of memory for the elements in the list. These blocks always have the same size. A recursive vector representation often allocates blocks of different sizes but these sizes are smaller than flat vectors ones.

5.1 Memory Management for a Recursive List Representation

Most explicit memory managers add an overhead to each allocated block of memory. For example, Doug Lea's Malloc [12], used in the GNU C library, adds size and status information before the available user space data returned by the system call `malloc`. In addition, it aligns data on 8 bytes even on 32-bit operating systems. So each element in the list of the recursive list representation will have an unnecessary overhead. We designed a custom memory manager for the single threaded version of TRIP, called **rlalloc** in this paper, in order to reduce the overhead in the allocation of elements of lists. The design of an efficient custom memory manager is an hard task as shown in the Berger's review [13]. Our memory manager splits a large memory chunk requested from the operating system in individual blocks of the size of the element of the recursive list representation. The only overhead is a single linked-list which contains the address of the large memory chunk. Free blocks are stored in a single linked-list: The pointer of this list is stored in the user data space. So it doesn't consume extra memory. The free operation sums up to add the freed block in this single linked-list.

Table 1. Computation time and memory consumption of $s_1 * (s_1 + 1)$ with $s1 = (1+x+y+z+t)^{25}$ using different memory managers. Polynomial is stored as a recursive list. Measured time (sum of user CPU time and system CPU time) is expressed in seconds and space used (size of the trip process in memory) in kilobytes.

Machine	time (s)		memory (Kb)	
	system	rlalloc	system	rlalloc
Intel Xeon 3.06Ghz - Linux 32-bits - glibc 2.3.4	108	83	22360	19688
Intel Xeon 3.6Ghz - Linux 64-bits - glibc 2.3.4	87	63	44656	33920
Apple G5 2.0Ghz - Mac os X 10.4 64-bits	225	186	28748	22504

Table 1 presents timing and memory consumption with our custom memory manager and the system memory manager (memory manager located in the C library of the system). Computations are performed on several operating system and hardware with a single cpu to see the effect of the system memory manager on time and memory used. The numerical coefficients in the polynomial $s1$ are double-precision floats in order to reduce computation time in coefficient arithmetic. Our memory manager reduces memory footprint about 25%. We observe increased speed up to 27% with our allocator. Most system memory managers, like "Doug Lea's Malloc" on Linux, use a best-fit algorithm even for small allocated blocks which produces many split and merge operations on their internal chunks. Furthermore, this memory manager is used for fixed-size numerical coefficient, like quadruple-precision float or arbitrary size rational numbers.

5.2 Memory Management for the Burst Tries

When multiplication of two sparse polynomials using burst tries is performed, the containers in burst tries have many small lists. The memory manager described in

the previous paragraph is used to allocate memory for this list in a multi-thread context with no lock mechanism because burst tries are always allocated and freed by the same thread. This provides better performance on SMP hardware.

6 Benchmarks with Other Computer Algebra Systems

Previous benchmarks on sparse polynomial multiplication, such as Fateman's review [14], show that PARI/GP and Singular have very good performance. So we compare the speed and memory consumption of TRIP, PARI/GP [15] and Singular [16] for sparse polynomial multiplication. This benchmark is performed on a single-core processor server to disable parallelization multiplication in TRIP. As the benchmark depends on coefficient arithmetic, the same version (4.1.4) of GNU Multiple Precision Arithmetic Library [17] is used for all computations. The current stable version 2.1.7 of PARI/GP supports only a native kernel for multi-precision kernel, a comparison with the current development version of this library with a GMP kernel is performed in order to have the same multi-precision kernel. Table 2 presents timing results and memory used for computing $s_1 * (s_1 + 1)$ with $s1 = (1 + x + y + z + t)^{20}$ and $s_2 * (s_2 + 1)$ with $s2 = (1/3 * x + 3/5 * y + 5/7 * z + 7/11 * t)^{20}$. Our recursive and flat vector have better execution time than PARI/GP and Singular but the recursive representation consume more memory than Singular in some cases.

Table 2. Comparison of time and memory consumption with other computer algebra systems. Measured time (sum of user and system CPU time) is expressed in seconds and space used in kilobytes. Computations are performed on an Intel Xeon processor with Linux operating system running at 3.06Ghz with a 533 MHz system bus.

Computer algebra systems	$s_1 * (s_1 + 1)$		$s_2 * (s_2 + 1)$	
	time (s)	memory (Kb)	time (s)	memory (Kb)
PARI/GP 2.1.7	82.1	N/A	225.6	N/A
PARI/GP 2.2.11 (GMP kernel)	67.8	N/A	177.6	N/A
Singular 3.0.1	101.5	8383.5	42.8	982.9
TRIP 0.98 (recursive list)	54.6	7138.5	15.5	1370.0
TRIP 0.98 (recursive vector)	42.9	5953.7	16.5	1831.4
TRIP 0.98 (flat vector)	24.1	3609.1	13.3	721.2

7 Conclusion

Flat vector representation of sparse multivariate polynomials improves the execution time and memory usage when multiplications using burst tries are performed. The multiplication using burst tries of sparse polynomials is efficient for a large number of variables and also for a large degree and it scales almost linearly on SMP architectures. Internal representations of sparse polynomials in the computer algebra system TRIP are sufficiently tuned to compute power series and polynomials up to a high degree with a large number of variables.

Acknowledgement. Part of the computations were made at IDRIS-CNRS.

References

1. Gastineau, M., Laskar, J.: TRIP 0.98. Manuel de référence TRIP, IMCCE, Paris Observatory (2005) `http://www.imcce.fr/Equipes/ASD/trip/trip.html`.
2. Laskar, J.: Accurate methods in general planetary theory. Astronomy Astrophysics **144** (1985) 133–146
3. San-Juan, F., Abad, A.: Algebraic and symbolic manipulation of poisson series. Journal of Symbolic Computation **32** (2001) 565–572
4. Ivanova, T.: A New Echeloned Poisson Series Processor (EPSP). Celestial Mechanics and Dynamical Astronomy **80** (2001) 167–176
5. Henrard, J.: A Survey of Poisson Series Processors. Celestial Mechanics and Dynamical Astronomy **45** (1988) 245–253
6. Hida, Y.: Data structures and cache behavior of sparse polynomial multiplication (2002) Class project CS282.
7. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. ACM Trans. Inf. Syst. **20** (2002) 192–223
8. Laskar, J.: Manipulation des séries. In: Modern Methods in Celestial Mechanics, Comptes Rendus de la 13ieme Ecole Printemps d'Astrophysique de Goutelas (France), 24-29 Avril, 1989, Gif-sur-Yvette, Editions Frontieres (1990) 63–88
9. Deprit, T., Deprit, E.: Processing Poisson series in parallel. Journal of Symbolic Computation **10** (1990) 179–201
10. Zorn, B.G.: The measured cost of conservative garbage collection. Software - Practice and Experience **23** (1993) 733–756
11. Hertz, M., Berger, E.D.: Quantifying the performance of garbage collection vs. explicit memory management. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, New York, NY, USA, ACM Press (2005) 313–326
12. Lea, D.: A memory allocator (1998) http://g.oswego.edu/dl/html/malloc.html.
13. Berger, E.D., Zorn, B.G., McKinley, K.S.: Reconsidering custom memory allocation. SIGPLAN Not. **37** (2002) 1–12
14. Fateman, R.: Comparing the speed of programs for sparse polynomial multiplication. SIGSAM Bull. **37** (2003) 4–15
15. The PARI Group Bordeaux: PARI/GP, version `2.1.5`. (2004) available from `http://pari.math.u-bordeaux.fr/`.
16. Greuel, G.M., Pfister, G., Schönemann, H.: SINGULAR 3.0. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern (2005) `http://www.singular.uni-kl.de`.
17. Granlund, T.: GNU multiple precision arithmetic library 4.1.4 (2004) `http://swox.com/gmp/`.