Parallel operations of sparse polynomials on multicores - I. Multiplication and Poisson bracket

Mickaël Gastineau
IMCCE-CNRS UMR8028, Observatoire de Paris, UPMC
Astronomie et Systèmes Dynamiques
77 Avenue Denfert-Rochereau
75014 Paris, France
gastineau@imcce.fr

ABSTRACT

The multiplication of the sparse multivariate polynomials using the recursive representations is revisited to take advantage on the multicore processors. We take care of the memory management and load-balancing in order to obtain linear speedup. The widely used Poisson bracket during the studies of the dynamical systems had been parallelized on these computers. Benchmarks are presented, comparing our implementation to the other computer algebra systems.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algebraic Algorithms

General Terms

Design, Performance

Keywords

Parallel, Sparse, Polynomial, Multiplication, Poisson bracket

1. INTRODUCTION

As many applications, such as celestial mechanics, require to handle large sparse multivariate power series, many specialized or general computer algebra systems had been developed to handle these objects at the time when most of computers had only one processor. But despite multiple cores and multiple processors are now widely available, even in laptop computers, few existing computer algebra systems, such as SDMP [16], TRIP [8] and Piranha [3], take advantage of the presence of these processor-elements to reduce the time of the computation on the multivariate sparse polynomials. The SDMP library performs the multiplication of the sparse polynomials using a heap and divides the work on the multiple processors using a static scheduling. This library stores these objects in a distributed form and could only work with integer coefficients. The computer algebra system TRIP dedicated to celestial mechanics supports several representations for the multivariate sparse polynomials in the computer's memory, such as recursive forms or distributed forms optimized for celestial mechanics. TRIP could handle floating-point numbers (hardware double-precision, multiple precision), integer or rational numbers. The multiplication of polynomials using burst-tries was investigated on multiple processors [7] but it suffers from the merge step of the burst-tries that prevents a good scalability. In the first part, we present an efficient implementation of the multiplication of the multivariate sparse polynomials using the recursive form.

As the operator Poisson bracket on the multivariate sparse polynomials is widely used during the studies of the dynamical systems, Roldan demonstrated that this operator could benefit of the distributed architectures using MPI [18]. But a linear speedup was obtained only on few nodes for the computation on the homogeneous polynomials. In this paper, we present, for the shared memory architectures, a parallel implementation of the Poisson bracket on any sparse polynomials.

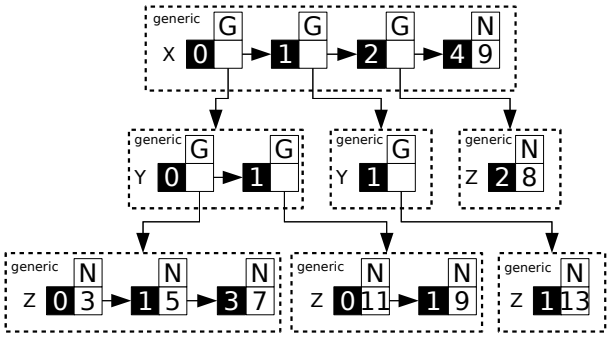
PASCO 2010, 21–23 July 2010, Grenoble, France.

2. DATA AND PARALLEL COMPUTATIONS

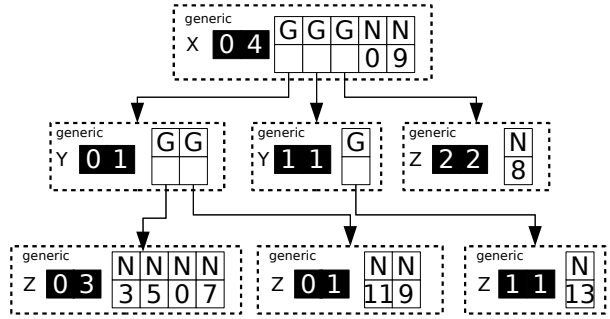
2.1 Polynomials representation

The multivariate polynomials could be represented in the memory using different data structures to keep them in a canonical form [22, 6]. Instead of using a distributed form, the polynomials are stored in a recursive container into the main memory of the computer. The multivariate polynomial in n variables is considered as a polynomial in one variable with coefficients in the polynomial ring in $n-1$ variables. These recursive data structure could be a recursive list or recursive dense vector, such as the containers implemented in the computer algebra system TRIP.

Each element of the recursive singly-linked list contains the exponent and the non-zero coefficients. These coefficients are in the polynomial ring in $n-1$ variables and are also recursive lists. As most problems need not large exponents, the exponents are encoded using hardware integers, e.g. signed 32-bit integers in TRIP. Figure 1(a) shows the representation of a multivariate polynomial as a recursive list which is used in TRIP. If a variable is not present in a term, this variable is missing in the representation (e.g., the variable y is not present in the term $8x^2z^2$ as shown in Fig. 1(a)). The complexity in search/insertion is in $O(\sum_{k=1}^{n} \deg(x_k))$. All polynomials could be represented in this structure.

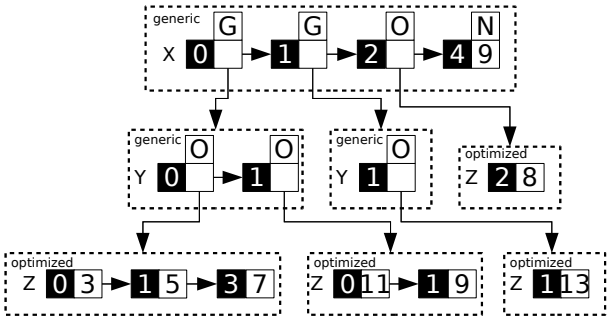


(a) recursive list (*sparse generic*)

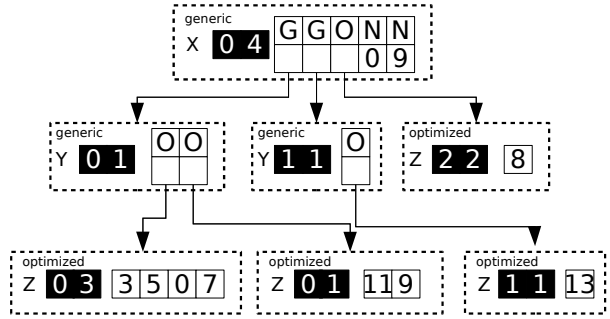


(b) recursive vector (*dense generic*)

Figure 1: representation of the polynomial $P(x, y, z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$ using generic container with the lexicographic order. The type information is G for generic container and N for numerical coefficient.



(a) recursive list (*sparse optimized*)



(b) recursive vector (*dense optimized*)

Figure 2: representation of the polynomial $P(x, y, z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$ using optimized container with the lexicographic order. The type information is G for generic container, O for optimized container and N for numerical coefficient.

The elements stored in the containers could be a polynomial or a numerical coefficient. For example in figures 1(a) and 1(b), the coefficient of x^0 is a polynomial and the coefficient of x^4 is a numerical value. To handle these different data types, the generic container requires additional information to determine the type of stored data. So each coefficient of the list is composed of two fields, type and value. This type field could be hidden if a polymorphic approach is used but, in this case, this type information is still present inside the value of the coefficient. During the computation, these generic containers require additional processing to check the data type of each element and select the appropriate algorithm. TRIP 1.0 encodes all coefficients in a generic container, even in the leaf nodes.

The recursive dense representation is similar to the previous one. The lists of tuple (degree, coefficient) is replaced by a vector of coefficients. All coefficients are stored in the array even the zero coefficients between the minimal and maximal degree. This minimal and maximal degree are kept in the header of the vector. Figure 1(b) shows the representation of a multivariate polynomial as a recursive vector which is used in TRIP. The complexity of the search and insertion algorithm is in $O(n)$ which is very efficient even for large polynomials. But this representation is not optimized for high degrees because the memory footprint of the vector becomes very large in some cases, such as $1 + x^{1000}$. In both representations, each container stores in its header the number of monomials with non-zero number coefficients in the full expanded form of the polynomial represented at that node (e.g., root containers of Fig. 1 and 2 store 8 as number of monomials).

Nevertheless, the leaf nodes contain always only numerical coefficients. To solve this bottleneck in TRIP 1.1, the leaf containers store only a single type of objects. So the type field is removed from the elements of the vector or list in the leaf nodes. This reduces the memory usage proportionately to the number of terms and reduces time consumption to check the data type. Figures 2(a) and 2(b) show the recursive list and vector representations with an optimized container for the leaf nodes. Afterwards, in the following examples, the recursive vector, respectively list, representation with a generic container for the leaf nodes is called *dense generic*, respectively *sparse generic*. The recursive vector, respectively list, representation with an optimized container for the leaf nodes is called *dense optimized*, respectively *sparse optimized*.

2.2 Memory management

As the polynomials stored into these data structures could create many small objects in the main memory, the memory management could become a bottleneck for the scalability on a computer with multiple cores [2]. PARSAC-2 [13] organizes memory in pages but its available pages are

protected by a global lock. To handle these objects, we use two lock-free memory allocators, for fixed-size or any-size objects, based on [19] and [7]. The fixed-size objects, such as elements of the recursive list, do not require a header before the objects. Each thread has its own heap. They always allocate from their heap without locks. The "free" operation is more complex. If the memory was allocated by the same thread, this one released the memory without lock. If the memory was allocated by another thread, the address is pushed into a LIFO list of the distant heap using the lock-free techniques. The access from the cores to the main memory could be non uniform, such as on the Intel Xeon Nehalem processors. To hide the latency of the main memory accesses, the cores of the processors share the cache memory, which could have several megabytes. On these Non-Uniform Memory Architectures (NUMA), the allocators take care of allocating the memory on the same local node.

2.3 Parallel work

Nowadays, desktop computers have between 2 and 8 cores and server computers could have up to 24 cores or more. So the computation on the sparse polynomials could benefit from these multiple cores and could be split between these cores. A static split between them could not be performed because the recursive form could be irregular and unbalanced load occurs. Virtual tasks [20], based on S-threads [12] allow to parallelize algorithms, such as Karatsuba's method. The S-threads is based on the fork-join approach which have a significant overhead. In TRIP, a task stealing model, similar to the *work stealing* model [4], is thus used to balance the load and minimize the overhead. A pool of threads is created at the beginning of the execution of the session. Their number is equal to the number of available cores. At the beginning, these threads are in an idle state. Each parallelized task is divided into small tasks which are pushed into a LIFO queue owned by the thread. If a thread becomes idle, it looks to the queues of the other threads and steals a task if it is available. Our task stealing implementation is similar to the Intel Threading Building Blocks [17] to abstract the decomposition of loops in several tasks.

3. MULTIPLICATION

Some symmetries are present in celestial mechanics, such as d'Alembert relations in the planetary motion [14], and implies that sparse series are manipulated. The degree of these series are low during some computations, such as the computation of the Hamiltonian in the Restricted Three Body Problem [11]. Due to these sparse series and low degrees, the naive (term by term) algorithm of the multiplication is used instead of the fast methods, such as FFT or evaluation/interpolation.

3.1 Recursive dense

The product of 2 recursive dense multivariate polynomials A and B could be done in the same way as for the univariate case.

$$\begin{aligned} A(x_1,...,x_n) &= \sum_i a_i(x_2,...,x_n).x_1^i \\ B(x_1,...,x_n) &= \sum_j b_j(x_2,...,x_n).x_1^j \\ C = A \times B &= \sum_k c_k(x_2,...,x_n).x_1^k \end{aligned}$$

Algorithm 1: FMA(A,B,C). Compute the fused multiplication-addition $C \leftarrow C + A \times B$. A, B and C are multivariate polynomials represented using a recursive sparse or dense structure.

Input: $A = \sum a_i x_a^i$
Input: $B = \sum b_j x_b^j$
Input: $C = \sum c_k x_c^k$
Output: $C = \sum c'_k x'^k_c$

```
// compare order of variables
```
if $x_a < x_b$ **then** `FMAcst` (A,B,C)
else if $x_a > x_b$ **then** `FMAcst` (B,A,C)
else `FMAsame` (A,B,C)

Algorithm 2: FMAcst(A,B,C). Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive dense representations. Assume $x_a < x_b$ or A is a numerical value.

Input: $A = \sum_{i=da_{min}}^{da_{max}} a_i x_a^i$ or A is a numerical value
Input: $B = \sum_{j=db_{min}}^{db_{max}} b_j x_b^j$
n_b = number of monomials in the expanded form of B
Input: $C = \sum_{k=dc_{min}}^{dc_{max}} c_k x_c^k$
Output: $C = \sum_{k=dc'_{min}}^{dc'_{max}} c'_k x_{c'}^k$
Data: $Thres$ thresold integer to perform loop in parallel

```
// Adds a polynomial ∑_{j=db_min}^{db_max} 0 × x_b^j inside C
```
1 $D \leftarrow$ `FindorInsertContainer` $(C, x_b, db_{min}, db_{max})$
2 **for** $j \leftarrow db_{min}$ **to** db_{max}
  do in parallel if $(Thres < n_b)$
3 | `FMA` (a, b_j, d_j)
4 **end**
5 **parallel barrier**
6 Put C in canonical form if $D = 0$

where

$$c_k = \sum_{i+j=k} a_i b_j \tag{1}$$

If the naive algorithm for the univariate case is applied directly to the multivariate case, many data structures will be briefly created in the main memory. Indeed, if the computation of Eq. 1 is performed in two steps : $d \leftarrow a_i \times b_j$ and $c_k \leftarrow c_k + d$, then each computation $a_i b_j$ generates a recursive polynomial d in $x_2, ..., x_n$ and, just after, its content is merged with the current content of c_k. To avoid these unnecessarily data structures, we use a *Fused-Multiply-Add* algorithm for the multiplication of two polynomials.

The main algorithm 1 (FMA) checks the order of the most factorized variable of A and B and selects the appropriate algorithm. If A and B depend on the same main variable, then the algorithm 3 (FMAsame *dense*) is used. In the other cases, the algorithm 2 (FMAcst *dense*) is executed. The first step of this two procedures prepares the addition through the function FindorInsertContainer. This function finds or inserts a container which receives a polynomial depending on x_b by taking care of the order of the variables. If x_b is not present in the recursive structure C, this function inserts a vector depending on the variable x_b. On the other hand, if x_b is present, the corresponding container is resized if neces-

Algorithm 3: FMAsame(A,B,C). *dense.* Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive dense representations. Assume $x_a = x_b$.

Input: $A = \sum_{i=da_{min}}^{da_{max}} a_i x_a^i$
n_a = number of monomials in the expanded form of A
Input: $B = \sum_{j=db_{min}}^{db_{max}} b_j x_b^j$
n_b = number of monomials in the expanded form of B
Input/Output: C polynomial
Data: *Thres* thresold integer to perform loop in parallel

`// Adds a polynomial` $\sum_{j=dab_{min}}^{dab_{max}} 0 \times x_b^j$ `inside C`

1 $dab_{min} \leftarrow da_{min} + db_{min}$
2 $dab_{max} \leftarrow da_{max} + db_{max}$
3 $D \leftarrow$ `FindorInsertContainer` $(C, x_b, dab_{min}, dab_{max})$
`// Computation`
4 **for** $k \leftarrow dab_{min}$ **to** dab_{max}
do in parallel if $(Thres < n_a \times n_b)$
5 **for** $j \in [da_{min}, da_{max}]$ **and** $k - j \in [db_{min}, db_{max}]$ **do**
6 `FMA` (a_j, b_{k-j}, d_k)
7 **end**
8 **end**
9 **parallel barrier**
10 Put C in canonical form if $D = 0$

Algorithm 4: FMAsamelarge(A,B,C). *sparse.* Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive sparse representations. Assume $x_a = x_b$ and $s_a \times s_b$ is large.

Input: $A = \sum_i a_i x_a^i$, s_a = number of elements(A),
n_a = number of monomials in the expanded form of A
Input: $B = \sum_j b_j x_b^j$, s_b = number of elements(B),
n_b = number of monomials in the expanded form of B
Input/Output: C polynomial
Data: *Thres* thresold integer to perform loop in parallel

`// Adds a polynomial` $\sum_j 0 \times x_b^j$ `inside C`

1 $D \leftarrow$ `FindorInsertContainer` (C, x_b)
`// Computation`
2 **foreach** *element* $\{\delta_a, a_{\delta_a}\}$ *in* A **do**
3 **foreach** *element* $\{\delta_b, b_{\delta_b}\}$ *in* B **do**
4 $\delta_d \leftarrow \delta_a + \delta_b$
5 $d_{\delta_d} \leftarrow$ Find/insert an element of degree δ_d in D
6 **do in parallel if** $(Thres < n_a \times n_b)$
7 `FMA` $(a_{\delta_a}, b_{\delta_b}, d_{\delta_d})$
8 **end**
9 **end**
10 **parallel barrier**
11 **end**
12 Put C in canonical form if $D = 0$

sary. For example, if we need to insert a polynomial $\sum_{j=9}^{12} x^j$ inside P (Figure 2(b)), the root container will be resized to the dimension (0,12) and the function returns a reference to this container. Second example, we need to insert $\sum_{j=1}^{5} y^j$ at the location x^4 inside P, the processing moves the numerical value 9 from location x^4 to the location 0 of a new container for y with a dimension (0,5), references this new container at the location of x^4 in the root container and returns the reference to new container. The worst case for this type of algorithm occurs when they are many cancelations, such as in $(1 + x + y)(1 - x - y)$.

The second step of both procedures performs the computation of the d_k term by term. As the d_k could be computed independently [21], the outer loops of FMAsame (line 4) and FMAcst (line 2) could be easily parallelized. Only a synchronization barrier is required after the loop between the threads which process the body loop. This parallelization is done using the task-stealing model. Each different value of the counter loop k corresponds to a task. If the computation of the d_k is shared at each recursive step, the granularity is too fine and the cost of the stealing dominates largely the coefficients' arithmetic and the memory management. To avoid this problem, the coefficients are computed in parallel only if A and B have enough terms. This threshold is based on the product of their number of terms, which is the number of monomials in the full expanded form of the polynomial represented at that node. That is the reason why the number of monomials inside the children vectors is stored at each level of the recursive data structure. If this threshold is too small, performance degradation happens as shown in Fig. 3.

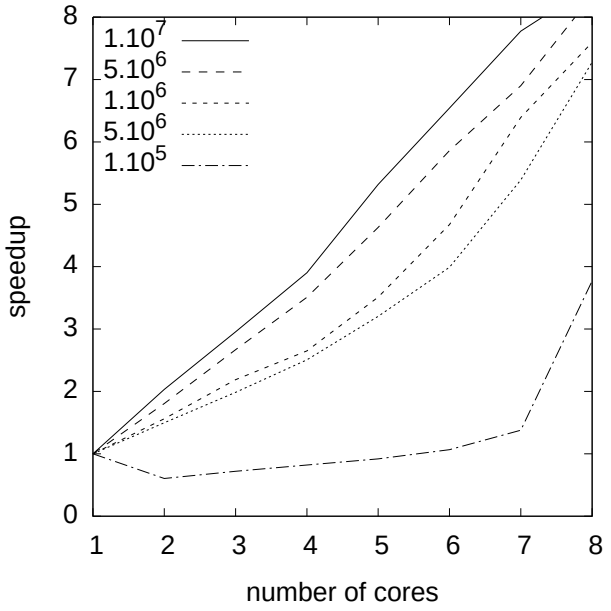


Figure 3: Speedup to compute and expand $f \times g$ with $f = (1 + x + y + z + t)^{30}$ and $g = f + 1$ using the recursive dense representation for several threshold to stop the work stealing.

3.2 Recursive sparse

A similar *Fused-Multiply-Add* algorithm is used to reduce the memory management of the list. The recursive sparse representation uses the same algorithms 1 (FMA) and 2 (FMAcst) as the dense case. As the singly-linked lists are used to store the coefficients, the computation inside the procedure FMAsame cannot be done in the same way as for the dense case, except if the number of elements in the list of

Algorithm 5: FMAsamesmall(A,B,C). *sparse.* Compute the fused multiplication-addition $C \leftarrow C + A \times B$ for the recursive sparse representations. Assume $x_a = x_b$ and $s_a \times s_b$ is small.

Input: $A = \sum_i a_i x_a^i$, s_a = number of elements(A), n_a = number of monomials in the expanded form of A
Input: $B = \sum_j b_j x_b^j$, s_b = number of elements(B), n_b = number of monomials in the expanded form of B
Input/Output: C polynomial
Data: *Thres* thresold integer to perform loop in parallel

```
// Adds a polynomial ∑_j 0 × x_b^j inside C
1  D ← FindorInsertContainer (C, x_b)
2  Copy references of root elements of A in a vector V_a
3  Copy references of root elements of B in a vector V_b
4  Draw up the list of computed degree from V_A and V_b
   inside a vector E (maximal size s_a × s_b).
   // Computation
5  foreach element of degree k in E do
6  |  d_k ← Find/insert an element of degree k in D
7  |  do in parallel if (Thres < n_a × n_b)
8  |  |  forall the i + j = k do
9  |  |  |  FMA (a_i, b_j, d_k)
10 |  |  end
11 |  end
12 end
13 parallel barrier
14 Put C in canonical form if D = 0
```

A and B are small. The algorithm FMAsame for the sparse case just selects the appropriate algorithm 5 (FMAsamesmall) or 4 (FMAsameslarge) depending on the number of terms in A and B.

If the number of elements in the list of A and B are large, the loop (line 5) of FMAsame (*dense*) cannot be done with singly-linked lists. Indeed, this requires many traversals of the polynomial B to get the coefficient b_{k-j}. Therefore, the coefficients d_k are computed using a double loop over the list of A and B in the algorithm 4 (FMAsamelarge). In order to find or insert d_k into the list D, the traversal from the beginning of D is not performed at each computation of $a_i b_j$. Instead, as the result of $a_i b_j$ is stored after $a_i b_{j-1}$, the last position in D is kept for the next search or insertion.

The outer loop (line 2) of the algorithm 4 could not be parallelized because d_k could be accessed at the same time by different iterations, e.g. $a_i b_j$ and $a_{i+1} b_{j-1}$ access to the same location d_{i+j}. However, as the $d_i + a_i b_0$, $d_{i+1} + a_j b_1$, $d_{i+2} + a_j b_2$, ... could be computed independently and writes to a different location in D, the inner FMA statement (line 7) is parallelized. A synchronization barrier is added before the next iteration $(i+1)$ of the outer loop. As for the dense case, the splitting of the work in several parallel tasks is stopped if A and B have not enough terms (number of monomials in the distributed representation). A similar value, as the dense thresold, has been found for the sparse representation.

If the number of elements in the list of A and B are small, which is the case in most of the series used in the perturbation theories, the usage of barrier could be reduced. The following optimization is done in the algorithm 5 (FMAsamesmall). Using the stack frame to avoid memory allocation, the references of the elements of list A and B are thus copied into small vectors. It switches to a similar algorithm as the dense case to compute independently the coefficient d_k. The threshold limit for the size of the list, that is the product of the number of elements of A and B, is fixed to 2000 in order to use not too much stack memory.

Intel Xeon computer	
Processor	2 Intel Xeon X5570 quad-core
Total number of cores	8
Total number of threads	16 (hyper-threading)
L3 Cache Size	8 Mbytes by processor
Memory	32 Gbytes
Operating System	Linux kernel 2.6 - glibc 2.5
Compiler	Intel C++ 10.1 64 bits
Library	GMP 4.2.4

Intel Itanium2 computer	
Processor	4 Intel Itanium2 9040 dual-core
Total number of cores	8
L3 Cache Size	18 Mbytes by processor
Memory	16 Gbytes
Operating System	Linux kernel 2.6 - glibc 2.3
Compiler	Intel C++ 10.1 64 bits
Library	GMP 4.2.4

Table 1: Description of the computers used in the benchmarks

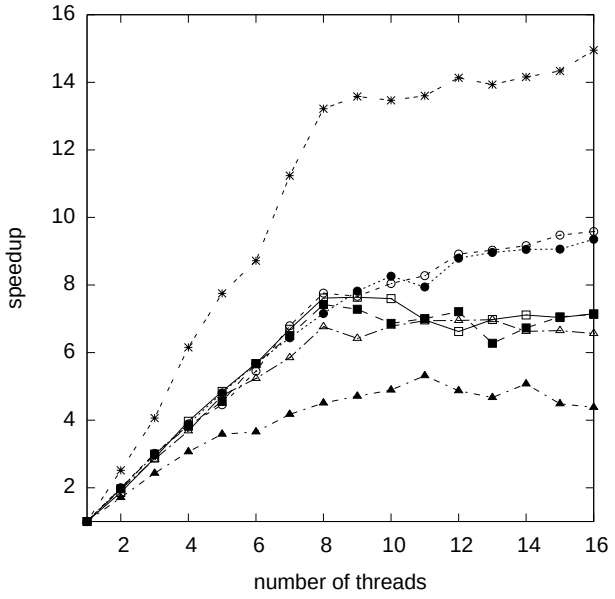
	example 1		*example 2*	
representation	time	memory	time	memory
Maple 13	1943.70	473	2310.23	10152
Singular 3.1.1	720.18	68.55	398.86	3935
SDMP	58.35	40.20	13.27	1291
TRIP 1.0				
vector	71.09	41.75	35.95	2041
list	72.88	52.55	22.68	2321
TRIP 1.1				
generic dense	55.02	41.25	22.47	2023
generic sparse	63.20	52.05	19.64	2304
optimized dense	22.54	31.13	19.63	1477
optimized sparse	29.53	41.92	16.54	1850

Table 2: Sequential execution timing expressed in seconds and memory consumption expressed in Mbytes. The numerical coefficients are integers numbers (GMP or hardware integers). The computations are performed on the Intel Xeon computer.

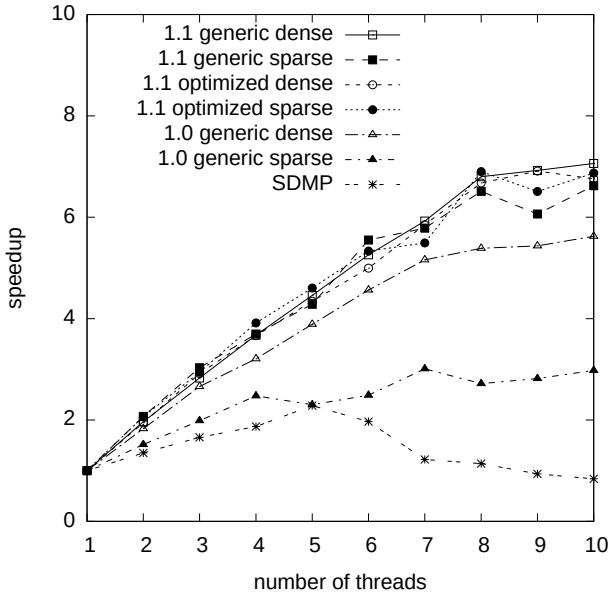
3.3 Benchmarks

The following benchmarks have been selected to test our implementation of the multiplication. These benchmarks are due to Fateman in [6] and Monagan and Pearce in [16].

- Example 1 : $f \times g$ with $f = (1 + x + y + z + t)^{30}$ and $g = f + 1$. f ang g have 46376 terms. The result contains 635376 terms. This example is very dense.
- Example 2 : $f \times g$ with $f = (1+x+y+2z^2+3t^3+5u^5)^{16}$ and $g = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^{16}$. f and g has 20349 terms. The result contains 28398035 terms. This example is very sparse. As shown in [16], a linear speedup is quite difficult to obtain on this example.

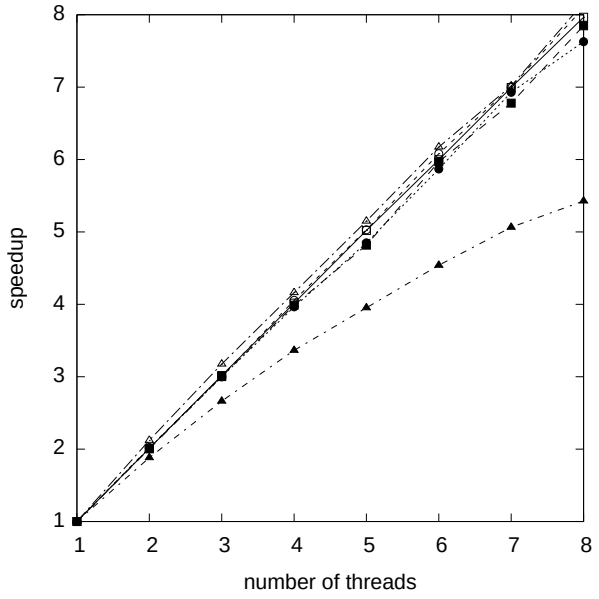


(a) example 1 (dense)

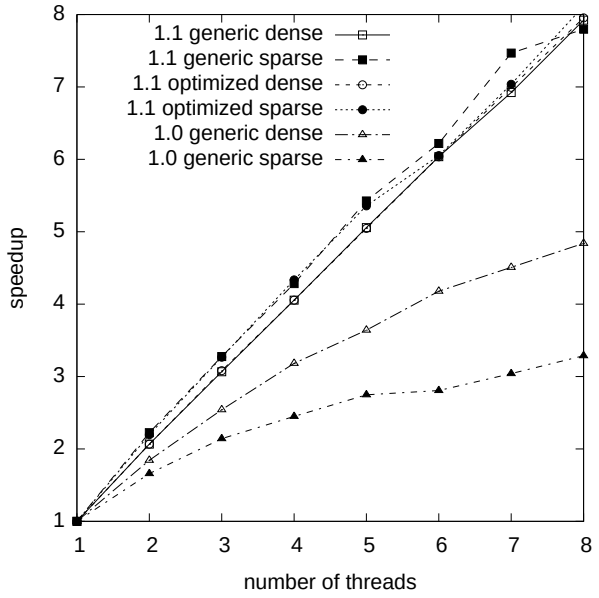


(b) example 2 (sparse)

Figure 4: Speedup to compute the examples on the Intel Xeon computer.



(a) example 1 (dense)



(b) example 2 (sparse)

Figure 5: Speedup to compute the examples on the Intel Itanium2 computer.

Table 1 describes the computer and software used to perform the benchmarks. In the sequential case, our algorithms are implemented in TRIP 1.1 and are compared to the computer algebra systems (Maple and Singular [10]) and to the existing software (SDMP library and TRIP 1.0) optimized for the sparse polynomials on shared memory computers. Table 2 shows the sequential execution timing and memory usage on the previous examples. The SDMP library stores the polynomials in a distributed form with packed exponents. The SDMP package performs the multiplication using a binary heap to sort the terms. TRIP 1.0 uses almost the same algorithm as the version 1.1 but splits the work only on the most factorized variable and does not take in account the number of terms in the children containers. This limitation could produce unbalanced-load if the data structure is very irregular. TRIP 1.0 uses the GNU MP Bignum Library (GMP)[9] to handle the integer or rational numbers. However, SDMP handles integers using hardware registers when these integers are small and comes back to the GNU MP library only when they grow [15]. TRIP 1.1

implements the same improvement for the integers smaller than $2^{63}-1$ on 64-bit computers. This optimization for small integers reduces the computation timing by a factor up to 2 on *example 1* but it has less impact on *example 2*. The specialized container for the leaf nodes of the recursive form strongly improves the computation timings. The SDMP library has better timings and uses less memory on *example 2* because it uses hardware integers up to 192 bits for the accumulation during the intermediate computations.

As shown in Fig. 4(a), the SDMP library has a super linear speedup due to its threads work on their binary heaps which fits in the cache memory. The optimized container in TRIP has a linear speedup up to 8 threads, except for the list container of the version 1.0. The presence of the hyper-threading improves the speedup up to 17% for SDMP and for the optimized containers in TRIP if we increase the number of threads to 16. The figure 5(a) shows the same computation on the computer Itanium with 8 cores without SDMP as this library is not available on the Itanium2 computer.

The SDMP library has only a speedup of 2 with 5 threads on the Xeon processor when it computes *example 2*, as shown in Fig. 4(b). With more threads, the speedup of SDMP decreases. TRIP 1.0 has the same difficulty to scale on this computer and on the Itanium computer, confirming the results founded in [16]. With the improvement of the job dispatching, the vector and list container of TRIP 1.1 have a speedup of about 6.7 with 8 threads. But it does not benefit of the hyper-threading with more threads, as this example requires more memory management. Indeed, this sparse example implies many memory operations with the recursive representations. Figure 5(b) shows that TRIP 1.1 has a linear speedup up to 8 cores on the Itanium2 and takes advantage of the larger cache.

The memory allocator could have a large impact on the execution time and on the memory foot print. Indeed, the speedup of *example 2*, which requires many memory allocations with the recursive form, drops to only 5.44 and the memory footprint is about bigger by half on 8 cores if the operating system allocator is used, as shown in Table 3. Even if the vector representation is used, similar impacts on the execution timings occur due to the resizing step (reallocation) of the vectors.

allocator	threads	time		memory
TRIP	8	8.5	(8.11x)	1851
	4	15.9	(4.33x)	1851
	1	68.7	(1x)	1841
Operating System	8	15.3	(5.44x)	3448
	4	22.1	(3.77x)	3536
	1	83.3	(1x)	2724

Table 3: Execution timing and memory usage on the *example 2* for different memory allocators on the Itanium2 computer using the *optimized sparse* representation. Timings are expressed in seconds and memory consumption expressed in Mbytes.

4. POISSON BRACKET

The equations describing a dynamical system are often expressed using the Hamiltonian formulation [1]. The Hamiltonian form $H(p,q)$, where p are the conjugate momenta of q,

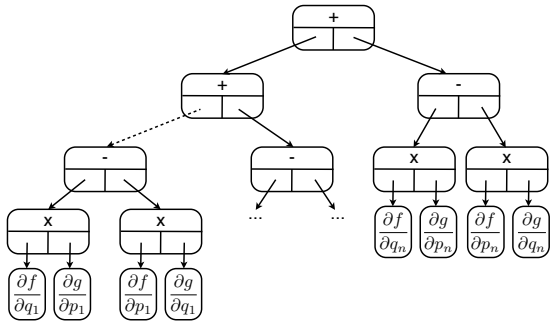


Figure 6: Tasks involved in the parallelization of the Poisson bracket.

is a function that verifies the following differential equations

$$\begin{aligned} \dot{p} &= -\frac{\partial H}{\partial q} \\ \dot{q} &= \frac{\partial H}{\partial p} \end{aligned}$$

The *Poisson bracket* of the two functions $f(p,q)$ and $g(p,q)$ is defined as

$$\{f,g\} = \sum_k \left(\frac{\partial f}{\partial q_k}\frac{\partial g}{\partial p_k} - \frac{\partial f}{\partial p_k}\frac{\partial g}{\partial q_k} \right) \tag{2}$$

This Poisson bracket is an important operator to compute the canonical transformations [5]. Indeed, this operator is intensively used for the computation of the normal forms and for the Lie series.

The summation in the Poisson bracket could be viewed as a summation reduction operation of the computing task $\left(\frac{\partial f}{\partial q_k}\frac{\partial g}{\partial p_k} - \frac{\partial f}{\partial p_k}\frac{\partial g}{\partial q_k}\right)$. The four derivations of this task could be computed independantly followed by the two products. Even if the addition of the two series has a low complexity against the multiplication complexity, the addition becomes a bottleneck in the scalability. So the addition of the polynomials is parallelized. Figure 6 shows the tasks involved in the parallelization of the Poisson bracket.

To test the scalability of the implementation of these algorithms, we select the two following examples. The first example, called *example 3*, is the computation of the poisson bracket of two almost dense polynomials in 6 variables, $f(p_{1..3}, q_{1..3})$ and $g(p_{1..3}, q_{1..3})$ where

$$\begin{aligned} f &= (1 + p_1 + q_1 + p_2 + q_2 + p_3 + q_3)^{12} \\ g &= (1 + p_1^2 + q_1^2 + p_2^2 + q_2^2 + p_3^2 + q_3^2)^{12} \end{aligned}$$

The second example, called *example 4*, is the computation of the Poisson bracket of two almost sparse polynomials in 6 variables, $H_{14}(p_{1..3}, q_{1..3})$ and $G_{14}(p_{1..3}, q_{1..3})$ where H_{14} and G_{14} are the homogeneous polynomials of the degree 14. Instead of taking all coefficients in the monomial sets of the degree 14 (11628 monomials) for these homogeneous polynomials, we set some terms to 0 in order to have a sparse poly-

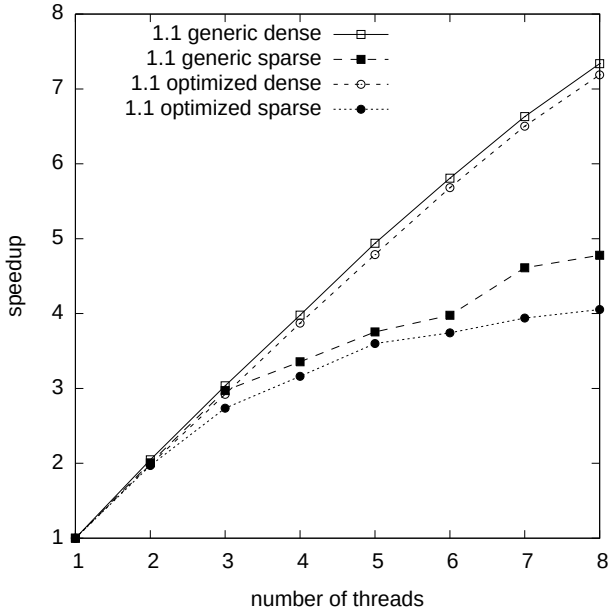


Figure 7: Speedup to compute the Poisson bracket of the example 3 on the Intel Itanium2 computer.

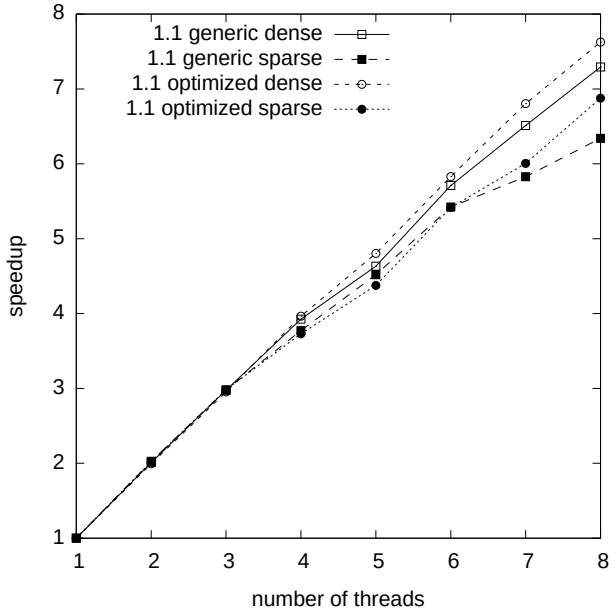


Figure 8: Speedup to compute the Poisson bracket of the example 4 on the Intel Itanium2 computer.

nomial[1]. So, H_{14} has 7722 terms and G_{14} has 5832 terms. The Poisson bracket of H_{14} and G_{14} has 142050 terms. Figures 7 and 8 show the speedup of the Poisson bracket on the Intel Itanium2 computer. The speedups of the recursive vector representations are almost linear on the two examples. On the *example 4*, the recursive list representations have the same behavior but their speedups are only about 5 with 8 threads on the *example 3*. This smaller speedup is due to many cancellations that occur during the addition step which requires more memory management for the elements of the lists. Similar behaviors have been obtained on the Intel Xeon computer.

5. CONCLUSION

The parallelization of the Poisson bracket benefits from the multiple threads but the recursive list representations do not have a linear speedup if many cancellations occur during the addition step. Recursive representations could exploit efficiently the multicore processors and obtain linear speedups for the multiplication of sparse polynomials if a dynamic scheduling, such as work stealing, is used to perform the load-balancing between the cores, even on the NUMA computers.

[1]The coefficients of the monomials $q_1^{d_1}p_1^{\bar{d}_1}q_2^{d_2}p_2^{\bar{d}_2}q_3^{d_3}p_3^{\bar{d}_3}$ are set to 0 if such that

$$\left(\sum_{j=1}^{3} d_j - \bar{d}_j\right) \mod 3 = 0 \qquad \text{for } H_{14}$$

$$\left(\sum_{j=1}^{3} d_j - \bar{d}_j\right) \mod 4 = 0 \qquad \text{for } G_{14}$$

6. REFERENCES

[1] V. I. Arnol'd. *Mathematical methods of classical mechanics*, volume 60 of *Graduate Texts in Mathematics*. Translated from the Russian by K. Vogtmann and A. Weinstein, Springer-Verlag, New York, NY, USA, second edition edition, 1989.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, 2000.

[3] F. Biscani. *Design and implementation of a modern algebraic manipulator for Celestial Mechanics*. PhD thesis, Centro Interdipartimentale Studi e Attivita Spaziali,Universita degli Studi di Padova, Padova, May 2008.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[5] J. R. Cary. Lie transform perturbation theory for hamiltonian systems. *Physics Reports*, 79(2):129–159, 12 1981.

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

[7] M. Gastineau and J. Laskar. Development of trip: Fast sparse multivariate polynomial multiplication using burst tries. *Computational Science –ICCS 2006*, pages 446–453, 2006.

[8] M. Gastineau and J. Laskar. TRIP 1.0. TRIP Reference manual, IMCCE, Paris Observatory, 2009. `http://www.imcce.fr/trip/`.

[9] T. Granlund. GNU multiple precision arithmetic library 4.2.4, September 2008. `http://swox.com/gmp/`.

[10] 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.

[11] A. Jorba. A methodology for the numerical computation of normal forms, centre manifolds and first integrals of hamiltonian systems. *Experiment. Math.*, 8(2):155–195, 1999.

[12] W. Küchlin. The s-threads environment for parallel symbolic computation. *Computer Algebra and Parallelism*, pages 1–18, 1992.

[13] W. Kuechlin. Parsac-2: A parallel sac-2 based on threads. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 341–353, 1991.

[14] J. Laskar. Accurate methods in general planetary theory. *Astronomy Astrophysics*, 144:133–146, Mar. 1985.

[15] M. Monagan and R. Pearce. Sparse polynomial pseudo division using a heap. Submitted to 'Milestones in Computer Algebra', J. Symb. Comput., September 2008.

[16] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 263–270, New York, NY, USA, 2009. ACM.

[17] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[18] P. Roldán. *Analytical and numerical tools for the study of normally hyperbolic invariant manifolds in Hamiltonian systems and their associated dynamics*. PhD thesis, Departament de Matemàtica Aplicada I, ETSEIB-UPC., Avda. Diagonal 647, 08028 Barcelona, Spain, November 2007.

[19] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 84–94, New York, NY, USA, 2006. ACM.

[20] W. Schreiner. Virtual tasks for the paclib kernel. *Parallel Processing: CONPAR 94 —VAPP VI*, pages 533–544, 1994.

[21] P. S. Wang. Parallel polynomial operations on smps: an overview. *J. Symb. Comput.*, 21(4-6):397–410, 1996.

[22] F. Winkler. *Polynomial Algorithms in Computer Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.