Massively Parallel Symbolic Computation

André Deprit
National Institute of Standards and Technology
Gaithersburg, MD 20899

Etienne Deprit
Naval Research Laboratory
Washington, DC 20375

Abstract

A massively parallel processor proves to be a powerful tool for manipulating the very large Poisson series encountered in non-linear dynamics. Exploiting the algebraic structure of Poisson series leads quite naturally to parallel data structures and algorithms for symbolic manipulation. Exercising the parallel symbolic processor on the solution of Kepler's equation reveals the need to reexamine the serial computational methods traditionally applied to problems in dynamics.

1 Introduction

Developing literal, i.e. non-numeric, solutions to problems in non-linear mechanics presents a great challenge. Once the symbolic processor has been tuned to produce expansions to a given order, new problems of physical interest arise which cannot be answered unless the expansions are driven to even higher orders. The three particle Toda problem provides a case in point [8]. While observing in the analytical expansions that the residual perturbation kept breaking a degeneracy, we were forced to stop the calculations at order 20, when the calculations overwhelmed our Lisp workstation. Indeed, in problems of this type, the complexity of the formulas grows in geometric proportion with the order. Hence, the computing time increases dramatically as intermediate steps keep exploding at an accelerated rhythm. Normalization of the three particle Toda system represents only one of many problems which reduce the Lisp machine to a state of constant garbage collection.

The complexity of the non-linear systems encountered in semi-classical quantum mechanics pales in comparison to the the problems faced in celestial mechanics due to the number of variables involved, the length of the input expressions, and the multiplicity of intermediate operations. Geologists invite astronomers to provide a time scale valid over tens of millions of years; the history of the earth's orbital eccentricity could serve the geologist's purpose. Such a long time scale, however, supposes a theory of the solar system—a 9-body problem—in completely literal form, and this theory must be valid to the fourth power in the mass ratios so as to include all major resonances of long period among planets. The task would be enormous, to say the least! Before we attack the planetary problem, we must create algorithms to simplify drastically the mathematical approaches, investigate software techniques designed to speed the code and facilitate its development, and explore the capabilities of new hardware as it becomes available [9].

Along these lines of research, we planned an experiment on a massively parallel processor to determine at what programming cost we could achieve modest increases in speed while performing some of the basic manipulations proper to celestial mechanics. Right from the start, the new computing environment of the parallel processor demanded complete overhaul of the software we have been developing for years on Lisp workstations. We wished to thoroughly explore the possibilities offered by the parallel machine without committing ourselves to a complete redesign of our symbolic processing software. Once satisfied with the new mathematical scheme of canonical simplifications, we will be in position to move our major problems from

1989 ACM 0-89791-325-6/89/0007/0308

serial to parallel computers. Our tentative results appear quite positive. The promise of these preliminary findings emerges quite vividly from Figure 1, where we compare the processing times of three symbolic algebra systems to complete a calculation to increasing orders of complexity.

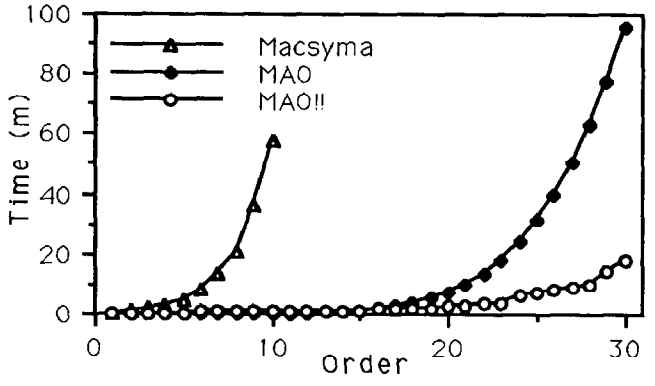

Figure 1: $(r/a)^4 \cos 5f$ with rational coefficients

Several criteria dictated the choice of a sample problem on which to exercise our parallel symbolic processor. First, to avoid entering massive series by hand, we looked for a problem which generates enormous expressions from data involving only a handful of terms. Second, the problem had to be simple enough that it could be ported quickly from system to system. The problem used to test the parallel processor comes from the two-body problem, the basic paradigm of celestial mechanics. The basic difficulty in celestial mechanics stems from the fact that Kepler's equation

$$E - e \sin E = \ell$$

cannot be solved in finite terms to represent the eccentric anomaly E as a function of the mean anomaly ℓ and the eccentricity e. Instead, assuming that e is small enough, E is developed as a Fourier series in ℓ over the algebra of formal power series in e with rational coefficients. From these basic expansions emerge similar series for functions like

$$\frac{r}{a} = (1 - e \cos E)$$

and

$$e \cos f = \frac{a}{r}(1 - e^2)(\cos E - e).$$

For the reader not familiar with the problem, we reproduce below the first few terms in these series.

$$\begin{aligned}
\frac{r}{a} = \quad & (\quad 1 + \tfrac{1}{2}e^2) + \ldots) \\
& + e\cos\ell\,(\quad 1 + \tfrac{3}{8}e^2 - \tfrac{5}{192}e^4 + \ldots) \\
& + e^2\cos 2\ell\,(\quad -\tfrac{1}{2} + \tfrac{1}{3}e^2 - \tfrac{1}{16}e^4 + \ldots) \\
& + e^3\cos 3\ell\,(\quad -\tfrac{3}{8} + \tfrac{45}{128}e^2 + \ldots) \\
& + e^4\cos 4\ell\,(\quad -\tfrac{1}{3} + \tfrac{2}{5}e^2 + \ldots) \\
& + e^5\cos 5\ell\,(\ -\tfrac{125}{384} + \ldots) \\
& + e^6\cos 6\ell\,(\ -\tfrac{27}{80} + \ldots) \\
& + \ldots)
\end{aligned}$$

$$\begin{aligned}
e\cos f = \quad & -e^2 \\
& + e\cos\ell\,(\quad 1 - \tfrac{9}{8}e^2 + \tfrac{25}{192}e^4 + \ldots) \\
& + e^2\cos 2\ell\,(\quad 1 - \tfrac{4}{3}e^2 + \tfrac{3}{8}e^4 + \ldots) \\
& + e^3\cos 3\ell\,(\quad \tfrac{9}{8} - \tfrac{225}{128}e^2 + \ldots) \\
& + e^4\cos 4\ell\,(\quad \tfrac{4}{3} - \tfrac{12}{5}e^2 + \ldots) \\
& + e^5\cos 5\ell\,(\ \tfrac{625}{384} + \ldots) \\
& + e^6\cos 6\ell\,(\ \tfrac{81}{40} + \ldots) \\
& + \ldots)
\end{aligned}$$

In the trial summarized in Figure 1, we measure the time required to obtain the product $h = f^4 g$, where $f = r/a$ and $g = \cos 5f$, when these series are expanded to multiple 30 in ℓ and power 30 in e. In one series of calculations, we use MACSYMA running on a Symbolics 3675 workstation; in the second series, we employ a special purpose processor, Mechanized Algebraic Operations (MAO), designed by Dr. Bruce Miller to run on a Symbolics Lisp workstations. In the third series, we compute the product on a Connection Machine using a package of procedures (MAO!!) we have written in *Lisp. In each run, we time the operations for increasing orders n in e. At order $n = 30$, each factor contains 256 terms. MACSYMA is desperately slow—it took almost 60 minutes to compute the result to order 10. MAO reached order 30 in approximately 95 minutes, while MAO!! did the same five times faster. Appendix A contains the complete timing data for the calculations in Figure 1.

2 Poisson Series Processors

In the two-body problem, as in the overwhelming majority of problems in celestial mechanics, the expressions to be processed take the form of Poisson series [5]. Poisson series may be viewed as formal double

sums

$$\sum_{(\iota,\kappa)\in I\times K} C_{\iota,\kappa} M[\iota]T[\kappa]$$

where ι and κ are vectors of natural integers ($\in \mathbf{Z}$) of dimension m and n, respectively. $M[\iota]$ and $T[\kappa]$ stand for symbolic terms of the type

$$\begin{aligned} M[\iota] &= X_1^{\iota_1} X_2^{\iota_2} \ldots X_m^{\iota_m}, \\ T[\kappa] &= \left\{ \begin{array}{c} \cos \\ \sin \end{array} \right\} (\kappa_1 A_1 + \kappa_2 A_2 + \ldots + \kappa_n A_n). \end{aligned}$$

Poisson series whose coefficients $C_{\iota,\kappa}$ are numbers are said to be *flat*.

A given family of Poisson series forms a commutative algebra over the ring of its coefficients. Poisson series processors differ by the standard representation they adopt; existing processors represent Poisson series as arrays of vectors [1,2,12], as lists [6,11,16], as two-dimensional grids, or even as balanced binary trees. The organization of the memory heap determines the unique representation of a Poisson series as a vector in the free module generated by the products of monomials and trigonometric terms. This standard representation holds strictly not only in the final result of an operation, but at every intermediate step, however ephemeral. To paraphase J. Moser, all current Poisson series processors are as "radical" as one can be in the politics of simplification. Radical representations exist because the processors can afford to micro-manage all basic operations of the algebra. Thus, when performing additions, as well as multiplications and differentiations, similar terms combine as soon as they appear, terms of opposite sign cancel immediately, and angle multiples change sign to preserve normalization rules. In other words, all simplifications occur immediately without intervention by the user.

By contrast, general purpose processors, such as MACSYMA [13], know nothing of these structural constraints. Moreover, MACSYMA does not give the user the possibility of enforcing his own representation standards. So, even if we present the test with g and h in a standard format, for example as Fourier series over the algebra of polynomials in e, MACSYMA simply states that the product h is the expression `(f**4)*g`, and waits for the user to issue commands like *trigexpand* and *trigsimp* to transform the result h into the standard form of a Poisson series. These simplifications are very expensive when applied to general representations that do not reflect the algebraic structure of the problem. Hence, as evidenced in Figure 1, MACSYMA cannot possibly match the performance of a special purpose processor such as MAO.

3 MAO

To use a Poisson series processor, the user must specify parameters characterizing the particular families of Poisson series for the problem at hand: the number of terms in each series, the domain of coefficients, the number and name of the polynomial and angular variables. Ideally, the user would specify these parameters at execution time. Existing processors based on linked-lists dynamically grow and free the series as the number of terms varies. As far as the other parameters, they usually appear as macro variables in the source code; the software package then compiles to produce a set of routines which implements the desired Poisson series algebras. Thus, the software becomes a meta-code which produces an entire family of Poisson series processors. This meta-code, however, is still not general enough, since the parameters of a Poisson series are attached not to a particular series but to the compiled code itself. Only those series whose structures have been foreseen at compile time may be manipulated; once compiled, the code precludes certain mathematical maneuvers requiring the dynamic specification and creation of new families of Poisson series.

The polynomial and angle variables of a Poisson series specify the coordinate map choosen by the mathematician for a particular problem. A change in coordinates often marks a turning point in the calculation and should be accompanied by a corresponding restructuring of the Poisson series. For example, a problem may initially involve terms in $\cos g$ and $\sin g$, a Fourier series in the argument of perigee g. Once recognized that the series exhibits the d'Alembert characteristic, it may be more efficient to recast the Fourier series as a polynomial in $C = \cos g$ and $S = \sin g$. In standard processors, the user must prepare for this eventuality by structuring all Poisson series with angle variable g and polynomial variables C and S. Thus, all advantages of this restructuring have been lost to the increased complexity of the series. Nevertheless, taking advantage of more modern programming contstructs allows us to design a processor which avoids such unnecessary complications. Such a processor must treat a Poisson series as an *object*.

A *flavor* supplies the template characterizing a family of objects. An object embodies both descriptive information detailing the state of the object, as well as procedural information implementing various operations on the object. For instance, multiplying the Poisson series P by the series Q may be accomplished by sending a message to P requesting that it form its product with Q and store the result in a new series object. Any information that P needs about Q may be

obtained by addressing the proper message to Q. The real power of object-oriented programming comes from the ability to construct hierarchies of flavors, whereby the programmer creates his own taxonomy of objects. If object P' belongs to a sub-flavor of P, then P' owns all the descriptive information found in P, and possibly more. P' possesses all the procedural information found in P, and may even handle additional messages. This method of abstract programming proves a powerful tool for preserving the algebraic structure of expressions in a symbolic processor.

The current version of MAO, written in Symbolics Common Lisp, implements the concept of an algebra A over a ring D as an abstract programming object, which in turn refers to another object representing the domain of coefficients. The coefficient domain may in turn represent a new polynomial or Fourier algebra, to any level of recursion. The hierachy of algebras must stop eventually at the field of real or rational numbers. Figure 2 details the Poisson series hierarchy used in the two-body problem. Indeed, with object-oriented programming, this process now becomes dynamic; depending on the state of a calculation, we may introduce new hierarchies of algebras and recast series into this new algebraic structure.

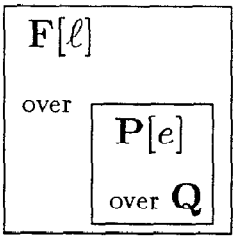

Figure 2: Hierarchy of algebras for two-body problem

MAO has proved a very powerful tool in solving medium sized problems in non-linear dynamics. In spite of all provisions made to ensure speed and efficiency, MAO appears far too slow to handle very large problems. Take the case of the lunar theory. Synchronization of time signals at nanosecond precision requires a lunar theory exact to 1 centimeter, but precision of 20 meters requires series of some 20,000 terms. In search of a faster tool, we turn to the Connection Machine for processing Poisson series.

4 The Connection Machine

The Connection Machine (CM) is a SIMD machine with thousands of processors arranged at the vertices of an N-dimensional hypercube [4,10]. The Naval Research Laboratory has a *quarter* machine with 2^{14} processors. The simple, bit-serial processors each address a local memory of 2^{16} bits. Instructions consist of two one bit operands plus a flag bit, and return a one bit result along with a flag bit. The applications programmer, however, remains safely above the bit level since PARIS (parallel instruction set) provides arithmetic operations on integer and floating-point numbers, communication between processors, and exchange between the front-end computer and the CM [15]. PARIS corresponds to the familiar assembly languages on serial machines.

A CM application runs on a front-end computer which controls the operations of the Connection Machine. The parallel portions of the code consist in sending PARIS instructions and data to the CM, and in retrieving the results from the processors. Since the Connection Machine follows a SIMD architecture, all processors in the hypercube receive the same PARIS instructions. Programs implement flow of control by directing sets of processors to sit out portions of the instruction stream.

Fortunately, higher-level languages have been extended for parallel computation. The front-end computer controlling the CM interprets or compiles the parallel portions of the high-level language into PARIS instructions. MAO!! runs on a Symbolics workstation, and was coded in *Lisp [17,18] (a parallel extension of Common Lisp [19]) along with Symbolics Flavors extensions for object-oriented programming. Due to the pioneering state of the system software, it was often necessary to consult the *Lisp sources—sometimes to clarify the documentation, at other times, to modify a *Lisp primitive to make it work under conditions not foreseen by the designers.

The CM's memory may be pictured as a matrix of 2^N columns, each consisting of 2^{16} bits. Every processor owns a column; a parallel variable (or *pvar* in *Lisp) constitutes a set of contiguous rows. *Lisp and PARIS partition the processor memory into two parts, handling one part as a stack for temporary parallel variables, and another part as a heap for global variables. *Lisp correctly handles the evaluation of nested expressions through the allocation and deallocation of intermediate results on the stack or heap.

For very large series, MAO!! exploits PARIS facilities which multiplex each processor into 2^m virtual processors. The number 2^m is referred to as the virtual processor (VP) ratio. Earlier versions of PARIS forced the ratio to be specified at the outset of an application, with all pvars allocated according to the same VP ratio. However, PARIS now permits dynamic virtualization of individual pvars. Even more, PARIS allows pvars to communicate values among one another, irrespective

of their virtual processor set. This software enhancement proved crucial in coping with the tremendous explosions in the intermediate results when operating on large series.

Simplification and multiplication of series rely critically on the ability to establish various patterns of communication among processors. The CM-2 supplies two communication mechanisms. Given a source pvar S and a target pvar T, how do processors transfer values from S to T? The *router* constitutes a gigantic telephone system, where each processor is identified by its phone number or *send address*. To use the router, the programmer specifies a pvar A of send addresses. For a read, all processors read the value in S at the processor specified in their value of A and set the value into T. For a write, all processors write their value of S into T at the processor whose address is given in A. Note that the mapping A may be an injection, and collisions may occur—a set of processors may either read from or write to a single processor. PARIS provides various functions to combine colliding messages in the course of a write.

In addition to the router, the CM-2 provides the *grid* mechanism, which is less flexible but more efficient. Conceptually, grid communication consists of two elements. First, there is some configuration of the machine as an n-dimensional grid. The processor's coordinate in this Cartesian space forms its *grid address* g. Second, a local pattern of communication consists of a vector of offsets o. For a read operation, each processor reads the value S from the processor at grid address $g + o$ and sets the result in T. For a write operation, each processor writes the value in S into the pvar T at the processor with grid address $g + o$. Note that the vector offset o specifies a bijective mapping between grid addresses, and no collisions are possible during either the read or write operations.

MAO!! uses both communication mechanisms to advantage. Sorting the terms of a series, for instance, requires permuting the terms among the processors. This type of information shuffling calls for the router. Checking if neighboring terms are similar and should be combined, on the other hand, involves a local pattern of communications where each term examines its neighbor. Such an operation fits nicely into the grid mechanism.

5 MAO!!

The Poisson series processor developed on the Connection Machine has been christened MAO!!—the sufix "!!" following the CM programmer's convention for denoting parallelism. MAO!! achieves its gains over MAO by spreading massive Poisson series over thousands of processors. Each processor in the CM holds a single term of a Poisson series. Thus, many series may remain active in the CM, limited only by the number of processors. This distribution of terms provides a simple resource allocation scheme flexible enough to deal with the constant explosion and implosion of partial results so typical of symbolic algebra. During the course of a calculation, MAO!! allocates free processors to hold intermediate terms. During subsequent simplification steps, terms are eliminated by deallocating their representative processors.

Figure 3 details the front-end structure which holds the state of each Poisson series during processing. Each series is identified by a *name*, by which it may be referred to in the front-end. The *tag* identifies which processors hold terms of this series. The front-end structure also keeps track of the *number of terms* in the series. Each time MAO!! retrieves the series from CM memory, the front-end structure keeps the *unpacked terms* and resets the *changed?* flag. The *unsorted?* flag tells the MAO!! routines that the series is not currently in sorted order in the CM. Finally, series must be marked as *killed* when they are deallocated.

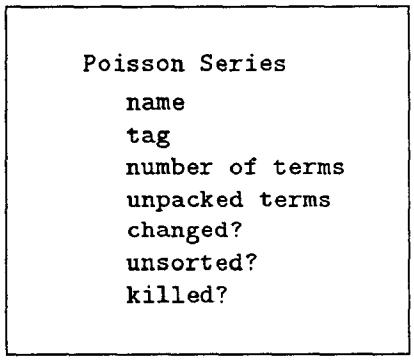

Figure 3: Front-end descriptor for Poisson series

Each term in a Poisson series resides in a single processor. The structure describing such a term is shown in Figure 4. Each term consists of a *coefficient* which may be a real or rational number. The angle variables in each term are packed into a single field, as are the polynomial exponents. The *packed angles* and *packed exponents* fields contain the angle multiples and polynomial exponents, respectively. Packing the angles and exponents reduces storage overhead, and also allows several angle multiples and exponents to be added, subtracted or multiplied by a scalar in a single operation. A *trig flag*, 0 for cosine and 1 for sine, completes the term structure.

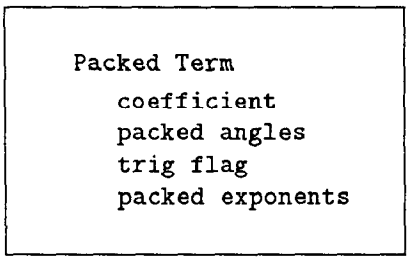

Figure 4: Packed term in Poisson series

From the standpoint of parallelism, algebraic operations fall into two classes. Multiplication by a monomial, partial differentiation and integration typify local operations, requiring only isolated computation in each processor. On the other hand, multiplication and simplification constitute global operations in the sense that processors representing terms of the series must communicate among themselves to produce the final result. Global operations highlight the real power of the CM. Among the global operations, we concentrate on two problems: multiplication and simplification. In both cases, we succeed in introducing a high degree of parallelism. The secret lies in effectively combining general router communications with local communications operations on a grid of terms.

6 Multiplication in Parallel

MAO!! multiplies Poisson series by replicating the factors and forming all partial products simultaneously. The algorithm becomes clear by looking at a simple example. To multiply a second degree polynomial in one variable $a + bx + cx^2$ by a polynomial $A + Bx$, we arrange the machine so that the first six processors on a one-dimensional grid contain the following quantities.

p_0	p_1	p_2	p_3	p_4	p_5
a	b	c	a	b	c
x^0	x^1	x^2	x^0	x^1	x^2
A	A	A	B	B	B
x^0	x^0	x^0	x^1	x^1	x^1

Then, all partial products are computed in parallel so that the one-dimensional grid now holds the quantities below.

p_0	p_1	p_2	p_3	p_4	p_5
aA	bA	cA	aB	bB	cB
x^0	x^1	x^2	x^1	x^2	x^3

There remains to pass the series to the simplification routine and store away the remaining terms. The dynamic virtualization mechanism of the CM makes the multiplication of large series possible. Realize that when multiplying two polynomial series of 256 terms each, the intermediate result will have 2^{16} terms, while our CM provides only 2^{14} physical processors. Nevertheless, the CM may configured as if it had 2^{16} virtual processors, where each physical processor emulates four virtual ones.

The situation becomes slightly more complicated when multiplying Fourier series rather than polynomials. The intermediate product now contains twice as many terms, since each product of sines or cosines produces two new terms. Once again, the flexibility of the virtual processor mechanism and grid addressing comes in handy. Once the two products have been spread out, as in the case for polynomial multiplication, MAO!! merely recopies each product to produce the doubling of terms. MAO!! then applies the standard trigonometric identities to combine the angles and produce coefficients of the correct sign and magnitude. The example below shows the result of multiplying the Poisson series $\cos x + \sin x$ by the series $\cos y + \sin y$, thereby demonstrating all possible combinations.

p_0	p_1	p_2	p_3
$\cos x$	$\sin x$	$\cos x$	$\sin x$
$\cos y$	$\cos y$	$\sin y$	$\sin y$
$\frac{1}{2}$ $\cos(x-y)$	$\frac{1}{2}$ $\sin(x-y)$	$-\frac{1}{2}$ $\sin(x-y)$	$\frac{1}{2}$ $\cos(x-y)$
p_4	p_5	p_6	p_7
$\cos x$	$\sin x$	$\cos x$	$\sin x$
$\cos y$	$\cos y$	$\sin y$	$\sin y$
$\frac{1}{2}$ $\cos(x+y)$	$\frac{1}{2}$ $\sin(x+y)$	$\frac{1}{2}$ $\sin(x+y)$	$-\frac{1}{2}$ $\cos(x+y)$

7 Simplification in Parallel

The most critical operation performed by MAO!! is the simplification of intermediate results. Well-timed, speedy simplification steps help to control the expression swell during calculations. The overall performance of a Poisson series processor depends directly on the efficiency of the simplification routine.

In order to simplify a series, MAO!! first collects all the terms into consecutive processors along a one-dimensional grid. Next, the terms of the series must be sorted. As shown in Figure 5, the packed terms are recast as sorting terms by considering the packed angles, trig flag, and packed exponents as a single unsigned integer. The CM's hypercube architecture provides rapid ranking of the terms in logarithmic time. Sorting the terms—rearranging them according to the

given ranking—requires a global routing step to permute the terms.

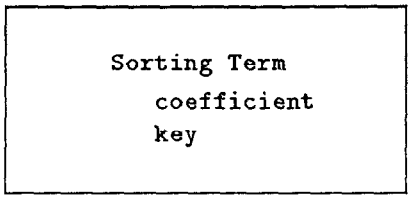

Figure 5: Poisson series term viewed for sorting

Once sorted, the series may be simplified. Since the terms lie along a one-dimensional grid, local communications may be used by each processor to see if it contains a term similar to its neighbor on the right. The processors form a new pvar containing a 1 if their term is similar to their neighbor's, and containing a 0 otherwise. This pvar divides the processors into contiguous *segments* of similar terms. The figure below continues the example of Section 6. The result of the multiplication has been sorted and the processors have determined the segment boundaries.

p_0	p_1	p_2	p_3	p_4	p_5
aA	bA	aB	cA	bB	cB
x^0	x^1	x^1	x^2	x^2	x^3
0	1	0	1	0	1

In the final step of simplification, MAO!! applies a parallel prefix reduction, or *scan* operator, to the pvar of coefficients. For the case of combining like terms, the operator must be addition. After the scan operation, each processor in a segment contains the result of the scan operator applied to all previous processors in the segment, including itself. Thus, the last term in each segment now contains the sum of all its similar terms. Note that the CM performs these parallel prefix reductions in logarithmic time, with the order proportional to the length of the longest segment. Once similar terms are combined, MAO!! shifts the segment pvar one neighbor to the left. As shown in the figure below, those processors now containing a 1 will be kept in the final result; MAO!! sets the coefficients of the remaining processors to zero and later deallocates them.

p_0	p_1	p_2	p_3	p_4	p_5
aA	0	$bA+aB$	0	$cA+bB$	cB
x^0	x^1	x^1	x^2	x^2	x^3
1	0	1	0	1	1

8 Kepler's Equation

For Kepler's equation, as well as for many fundamental problems of non-linear dynamics, parallel architectures force a rexamination of the established methods of solution. A literal solution to Kepler's equation traditionally emerges from two main directions—either by iteration or by application of Laplace's formula.

The iterative method begins with $E_0 = \ell$, and continues with

$$\begin{aligned} E_1 &= \ell + e\sin E_0 = \ell + e\sin\ell + \mathcal{O}(e^2), \\ E_2 &= \ell + e\sin E_1 = \ell + e\sin(\ell + e\sin\ell) \\ &= \ell + e\sin\ell + \frac{1}{2}e^2\sin 2\ell + \mathcal{O}(e^3), \end{aligned}$$

and so on. At step n, the iteration adds to E the finite Fourier series in ℓ that is the coefficient of e^n. The method is easy to implement, providing one has at hand a good procedure for expanding Taylor series.

The iterative method, however, has a serious drawback; it addresses exclusively the problem of solving Kepler's equation. For the more general case of a function $F \equiv F(E,\ell,e)$, one has to resort to straightforward substitution of the solution into F itself developed as a power series

$$F = \sum_{n\geq 0} \frac{e^n}{n!}\frac{d^n F}{de^n}\bigg|_{e=0}$$

Such expansions usually lead to elaborate combinations of Faà de Bruno's partial differential operators for functions of a function defined through an implicit equation. One often replaces such operators with ad hoc compositions of Taylor series.

Laplace's formula addresses precisely the question of building functions of E immediately as power series in e without necessarily solving Kepler's equation in advance. If $G(E)$ is analytical in E around $E = \ell$, then Laplace's formula states that

$$G(E) = G(\ell) + \sum_{j\geq 1}^{\infty} \frac{e^j}{j!}\frac{d^{j-1}}{d\ell^{j-1}}\left[\sin^j\ell\frac{d}{d\ell}G(\ell)\right] \qquad (1)$$

Equation (1) fits quite naturally on a serial computer. Given G, the program proceeds iteratively. From the product

$$G_j = \sin^j\ell\frac{d^j}{d\ell^j}G(\ell),$$

one calculates the product

$$G_{j+1} = \sin\ell G_j$$

at step $j+1$, and then computes successively the first j derivates of G_{j+1}.

Using MACSYMA, we implemented Laplace's formula to produce at low orders in e the coefficients against which we checked the series produced on the CM. It must be noted, however, that as the order increases, Laplace's formula requires more and more derivatives; all derivatives at order j except the last one become garbage as soon as the loop enters the next cycle. Moreover, the work has to be restarted from scratch each time another function requires expansion.

Techniques in computational analysis appeared some time ago [7] which address precisely this defficiency with Laplace's formula. Most recently, these techniques have been adapted to solve implicit equations depending on a small parameter [14]. The basic concept is as follows.

With $y = E - \ell$, Kepler's equation is rewritten as an implicit equation of the form

$$f(y, e; \ell) \equiv y - e \sin(\ell - y) = 0.$$

This equation has a root $y = 0$ at $e = 0$. The implicit equation is solved if one finds a family of transformations

$$\psi : x \mapsto y(x, e; \ell)$$

parametrized by e. The transformation should be such that $y(x, 0; \ell) = x$; furthermore, it should change f into

$$f(y(x, e; \ell)) = x.$$

The latter identity implies in particular that $y(0, e; \ell)$ is the solution of the implicit equation that vanishes at $e = 0$. The transformation ψ will be built as a Lie transformation, i.e. as the solution of a differential equation

$$\frac{dy}{de} = W(y, e; \ell) \qquad (2)$$

for the initial condition $y = x$ at $e = 0$.

In other words, instead of constructing ψ as a power series, one constructs the right hand member W of a differential equation for which ψ is a solution. This sort of indirection has enormous advantages. Indeed, the differential equation 2 defines the Lie operator

$$\mathcal{L} = \frac{\partial}{\partial e} + W \frac{\partial}{\partial e}.$$

Geometrically speaking, $\mathcal{L}$ defines the derivative of any function in the direction determined by the vector field W. In particular, to obtain that the transformation generated by W changes f into x, it is necessary and sufficient to impose that $\mathcal{L}(f) = 0$.

For the case of Kepler's equation, one easily obtains a simple recurrence relation for the terms in W:

$$W_n = \begin{cases} -f_1 & n = 0 \\ -W_{n-1} \dfrac{\partial f_1}{\partial y} & n > 0 \end{cases}$$

In the solution of Kepler's equation, we have pushed this Lie transformation to degree 30 in e. At that order, some of the coefficients, all reductions performed, have numerators and denominators exceeding 40 decimal digits. Indeed, the process proves so efficient that a method we designed specifically for a massively parallel processor ends up underemploying the power of the CM. As evidenced by Figure 6, we must reach order 15 in the calculation of ψ with real coefficients before the CM overtakes the Lisp machine.

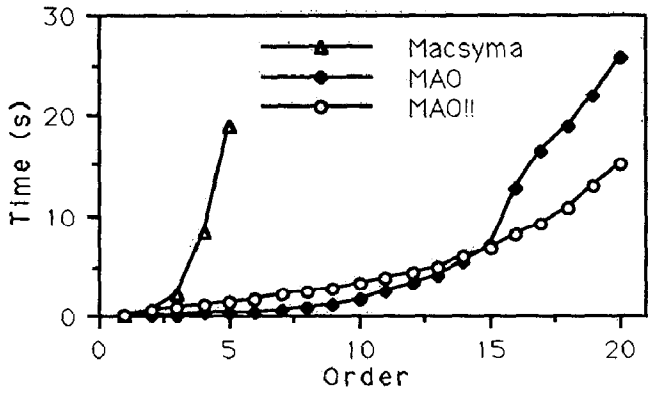

Figure 6: ψ with real coefficients

Having obtained the Lie transformation generating the solution to Kepler's equation, we may compute the expansions of many classical functions in the two-body problem. For any function $F(\psi; e, \ell)$,

$$F = \sum_{j \geq 0} \frac{\epsilon^j}{j!} \mathcal{L}^j(F)\Big|_{\epsilon=0}.$$

These developments were first tabulated by Cayley and his students in the mid-nineteenth century, and many symbolic processors developed by astronomers have been exercised by reproducing Cayley's tables [3]. Since they were computed by hand, the developments were only carried out to low order, far too low to serve the needs of celestial mechanics today. With MAO!!, we were able to extend Cayley's tables for the first time to order 30. Figure 1 shows one such development from Cayley's tables.

9 Conclusions

The trials described in Sections 1 and 8 bring convincing evidence that massively parallel processors present a unique opportunity for computational astronomers. Obviously, problems in celestial mechanics prove far

too large for general-purpose systems such as MACSYMA. Special-purpose systems running on serial machines, like MAO, can handle medium-sized problems. Even implementing MAO on more powerful sequential machines will not be enough; tackling the large problems in celestial mechanics requires the economies of scale that only massively parallel processing can offer. By exploiting the algebraic structure of Poisson series and the natural parallelism in symbolic manipulation, the astronomer working on a Connection Machine may some day push his analytical theories to much higher orders than once thought possible.

Doubtless, MAO!! needs to evolve. The algorithms, as they now stand, require improvement. Recoding the routines in PARIS would eliminate the overhead inherent in *Lisp and produce large gains in speed. Furthermore, MAO!! needs to become more abstract. The underlying concept of Poisson series is too limiting. Ideally, one should be able to treat hierarchies of polynomial or Fourier algebras in order to match precisely the algebraic structure of the problem.

References

[1] Agnese, J.-C. Logiciel MSTN: Manipulation de Séries Trigonométriques. Technical Report CT/DTI/MS/MN/262 (1984), Centre National d'Études Spatiales (CNES).

[2] Broucke, R., and Garthwaite, K. A Programming System for Analytical Series Expansions on a Computer. *Celestial Mechanics* **1** (1969), 271–284.

[3] Cayley, A. Tables of the Developments of Functions in the Theory of Elliptic Motion. *Memoirs of the Royal Astronomical Society* **29**, 191–306. Reprinted in *The Collected Mathematical Papers of Arthur Cayley*, Vol. **3**. Cambridge University Press, 1890, pp. 360–474.

[4] *Connection Machine Model CM-2 Technical Summary.* Thinking Machines Corporation, Cambridge, Mass., 1987.

[5] Danby A. J., Deprit, A., and Rom A. The Symbolic Manipulation of Poisson Series. Boeing Document D1-82-0481 (1965), Boeing Scientific Research Laboratory.

[6] Dasenbrock, R. R. A FORTRAN-Based Program for Computerized Algebraic Manipulation. NRL Report 8611 (1982), U.S. Naval Research Laboratory.

[7] Deprit, A. Canonical Transformations Depending on a Small Parameter. *Celestial Mechanics* **1** (1969), 12–30.

[8] Deprit, A., and Miller, Bruce R. Normalization in the Face of Integrability. *Annals of the New York Academy of Sciences* **536** (1988), 101–126.

[9] Deprit, A., and Miller, Bruce R. Simplify or Perish. To appear in *Proceedings of the International Astronomical Union Colloquium #109* in *Celestial Mechanics*.

[10] Hillis, W. Daniel. *The Connection Machine*. The MIT Press, Cambridge, Mass., 1985.

[11] Jefferys, W. H. A Fortran Based Lisp Processor for Poisson Series. *Celestial Mechanics* **2** (1970), 474–480.

[12] Kusmin, A. V. SASM: Operations on Series. *The Algorithms of Celestial Mechanics* (Institute of Theoretical Astronomy, U.S.S.R. Academy of Sciences, Leningrad) **40** (1982), 54.

[13] *MACSYMA Reference Manual: Version 13.* Symbolics, Inc., Cambridge, Mass., 1988.

[14] Meyer, K.R. Bifurcations and Stability by Lie Transformations. To appear in *Proceedings of the International Astronomical Union Colloquium #109* in *Celestial Mechanics*.

[15] *Paris Reference Manual.* Thinking Machines Corporation, Cambridge, Mass., 1988.

[16] Rom, A. Mechanized Algebraic Operations (MAO). *Celestial Mechanics* **1** (1970), 301–319.

[17] **Lisp Reference Manual. Version 5.0.* Thinking Machines Corporation, Cambridge, Mass., 1988.

[18] **Lisp Release Notes.* Thinking Machines Corporation, Cambridge, Mass., 1988.

[19] Steele Jr., G. L. *Common LISP: The Language.* Digital Press, Burlington, Mass., 1984.

A Sample Timings

Order	Terms	Macsyma (m)	MAO (m)	MAO!! Total (m)	MAO!! Usage (%)
1	3	0.157	0.002	0.284	91.94
2	6	0.645	0.007	0.329	92.24
3	10	1.419	0.014	0.369	92.33
4	15	2.732	0.024	0.417	90.47
5	21	4.521	0.038	0.444	91.33
6	27	7.803	0.063	0.508	90.59
7	33	13.158	0.085	0.548	89.57
8	40	21.142	0.114	0.614	89.02
9	47	35.853	0.150	0.667	88.33
10	55	57.377	0.203	0.745	87.42
11	63		0.264	0.805	86.70
12	72		0.391	0.928	85.87
13	81		0.530	1.013	85.00
14	91		0.742	1.111	84.85
15	101		1.047	1.214	84.55
16	112		1.599	1.358	84.12
17	123		2.318	1.491	83.51
18	135		3.516	1.694	83.49
19	147		5.074	1.848	83.54
20	160		7.258	2.674	86.93
21	173		9.974	3.074	87.66
22	187		13.706	3.677	88.43
23	201		18.242	4.003	88.19
24	216		24.161	6.006	91.30
25	231		31.085	6.874	91.90
26	247		39.895	8.250	92.57
27	263		50.201	9.014	92.56
28	280		62.905	9.824	92.68
29	297		77.400	14.312	94.54
30	315		95.218	17.932	95.31

Table 1: $(r/a)^4 \cos 5f$ with rational coefficients