J. Symbolic Computation (1996) **11**, 1–25

To appear in 1996 in the Journal of Symbolic Computation.

Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions

SIMON GRAY[†], NORBERT KAJLER[‡], AND PAUL S. WANG[⋆†]

[†] *Department of Mathematics and Computer Science, Kent State University, Kent, OH 44242, USA*

[‡] *Ecole des Mines de Paris, 60 Bd St-Michel, 75006 Paris, France*

[⋆] *Institute for Computational Mathematics, Kent State University, Kent, OH 44242, USA*

(Received April 7, 1995)
(Revised January 8, 1996)

The Multi Project is an ongoing research effort at Kent State University aimed at providing an environment for distributed scientific computing. An integral part of this environment is the Multi Protocol (MP) which is designed to support efficient communication of mathematical data between scientifically-oriented software tools. MP exchanges data in the form of linearized annotated syntax trees. Syntax trees provide a simple, flexible and tool-independent way to represent and exchange data, and annotations provide a powerful and generic expressive facility for transmitting additional information. At a level above the data exchange protocol, dictionaries provide definitions for operators and constants, providing shared semantics across heterogeneous packages. A clear distinction between MP-defined and user-defined entities is enforced. Binary encodings are used for efficiency. Commonly used values and blocks of homogeneous data are further optimized. The protocol is independent of the underlying communication paradigm and can support parallel computation, distributed problem solving environments, and the coupling of tools for specific applications.

1. Introduction

The design of scientific computing systems is moving rapidly away from large monolithic structures and toward interconnected components that are able to run independently (Kajler, 1992a, Dewar, 1994). The power of workstations connected by fast local area networks and the general success of the client-server model are basic reasons for this trend. The distributed approach offers many advantages. Autonomous components can be developed and maintained separately, perhaps even at different locations. They can run on different platforms for convenience, better performance, or to meet license

[†] E-mails: {sgray,kajler,pwang}@mcs.kent.edu.

Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650 and by various grants from University of Nice-Sophia Antipolis, France and RIACA, Kruislaan 419, 1098 VA Amsterdam, The Netherlands.

restrictions. Finally, the components could be reused in different configurations for different applications. The feasibility of this approach depends critically on an efficient data exchange protocol to connect the components. A useful protocol for connecting scientific systems should be simple, efficient, and flexible enough to meet the data transfer needs of most scientific software tools. A simple protocol is easier to understand and use, allowing interfaces to independent packages to be developed more quickly. Special attention should be paid to the problem of efficiently transmitting large mathematical formulas and large amounts of numeric data. Finally, the protocol should be extensible to accommodate the creation of new tools, the evolution of existing tools, and customization of specialized environments. MP is our attempt to address these issues and to contribute to a standard, non-proprietary protocol.

The Multi Project at Kent State is part of an ongoing research effort into the integration of software tools for scientific computing. Among the goals of the project are to design and develop a protocol for efficient communication of mathematical data among scientific computing systems and to explore how such a protocol might be used for distributed/parallel programming, distributed problem solving environments, and to encourage the development of reusable software components, much in the spirit of CAS/PI (Kajler, 1992a). Initial deliberations on Multi started in the late 1980's at Kent and involved Peter Hintenhaus, Michael Rothstein, Paul Wang, and several graduate students. Significant work was begun in 1990 as the authors began to collaborate on the project. The initial design of the Multi Protocol (MP 0.5) was completed in 1993. The first achievement of this research was to provide a protocol for the efficient exchange of mathematical expressions between scientifically-oriented packages (Gray *et al.*, 1994a). The initial design (Gray *et al.*, 1994b) underwent some modifications and additions. A complete set of C routines has been implemented and tested. This new Multi library supports MP 1.0 whose design, implementation, and application are described in the following sections.

2. Tool Integration

Schefström identified three dimensions of tool integration: data, control, and user interface (Schefström, 1989). Together they make it possible for separate components to efficiently cooperate in an open and homogeneous system.

Data integration involves the exchange of data between separate tools, including the definition of a mechanism allowing the tools to agree on the format and meaning of the transmitted data. For homogeneous systems, such as the Maple kernel and its graphing tool, this understanding is relatively simple to arrange. However, for a heterogeneous collection of systems such as Maxima and Matlab, such an agreement must be arranged through a data exchange protocol. Thus, a general-purpose mathematical data protocol should provide a mechanism for the tools to identify the syntax and semantics of the transmitted data. The problem is exacerbated by the desire to support a broad range of tools with diverse data needs, including mathematical expressions, integer and floating-point numbers (fixed and arbitrary precision), as well as character and binary encodings. Specific to the domain of mathematical software is the problem of the efficient representation and exchange of (possibly large) mathematical expressions plus related data.

Control integration concerns the establishment, management, and coordination of inter-tool communications. Tool control issues include how tools are identified, launched, and connected, the dynamic addition and deletion of tools, how they send requests and receive results, and the rules governing how a tool may affect the behavior of others. The particulars of these issues vary with the communication paradigm (e.g., point-to-point, bus-based, parallel) and fall largely within the realm of distributed computing and outside the area of mathematical protocols.

The aim of user interface integration is to provide the user with a logical and consistent style of interaction with the entire system. A homogeneous user interface should hide the implementation details and simplify what the user must know to effectively control a distributed system made from independent pieces. Techniques include uniform look and feel, abstracting similar metaphors, and supporting consistent mental models. Attempts to provide a coherent graphical front end giving access to several computer algebra systems include CaminoReal (Arnon *et al.*, 1988), SUI (Doleh and Wang, 1990), and CAS/PI (Kajler, 1992b). Closely related to these efforts are the use of active structured documents as user interfaces to symbolic computation packages (Quint *et al.*, 1996) and the development of electronic books for teaching Mathematics (Cohen and Meertens, 1996).

As shown by Kajler (1992a, 1992b), the tool integration paradigm can be successfully applied to the design of a distributed computer algebra environment. The resulting architecture is highly flexible and allows packages from different origins to communicate with each other in a seamless way. Still, the efficient exchange of mathematical expressions was not addressed in these preliminary works. Our feeling is that a protocol such as MP should essentially address data integration, with control and user interface integration considered as orthogonal issues, open for other approaches. Ideally, MP should be thought of as the complement of (past, present, and hopefully future) distributed computation environments which lack an efficient way to transmit mathematical data. Indeed, when designing MP we were very careful to make a clear distinction between the scope of the protocol and the problem of tool integration in general. Consequently, we designed MP as a protocol which addresses *only* the representation and exchange of mathematical data, but which can easily fit under existing control technologies such as remote procedure calls for point-to-point communication (Birrell and Nelson, 1984), ToolTalk (SunSoft, Inc., 1991) or Sophtalk (Jacobs *et al.*, 1993) for software bus architectures, and MPI (Gropp *et al.*, 1994) or PVM (Geist *et al.*, 1994) for constructing parallel virtual machines. Here we focus on the data integration aspects by proposing an encoding for the efficient exchange of mathematical expressions and related data.

3. Related Work

Traditionally computer algebra systems were monolithic, stand-alone programs designed to communicate with a user through a specific command language. They did not address interoperability issues such as the exchange of mathematical expressions with other independent programs. More recently, scientific computation systems have adopted the component approach and devised various schemes for inter-component communication. Several notable examples are discussed here.

Since version V, Maple has been composed of a kernel and a set of devices, including a user interface, Iris, and a plotting engine. The kernel and devices can run on remote computers communicating with a proprietary protocol (Leong, 1986). Data can be passed in one of two ways: either as strings suitable to be used as Maple input, or as Directed Acyclic Graphs (DAGs) using Maple's internal data representations. DAGs have two clear advantages: first, they reduce the average amount of data transmitted by sharing common subexpressions; second, using Maple's internal data representation eases data encoding and decoding on Maple's side. Recently, Maple introduced MathEdge (Pintur, 1994), a development toolkit which enables developers to link their applications with the Maple kernel.

Beginning with version 2, Mathematica communicates using MathLink (Wolfram Research, Inc., 1993). MathLink implements a communication protocol and provides a set of procedural interfaces that allow C programs to send and receive data, to call (or be called by) Mathematica, or to allow different instances of Mathematica to communicate with each other. MathLink is fully documented and library routines are provided for advanced users to write their own applications. MathLink's interface exposes Mathematica's representation of expressions, although the Mathematica functions `ToString` and `ToExpression` can be used so that strings are sent instead of the internal structure. Because the details of the communication are hidden, MathLink could be used to transmit DAGs. However, version 2.2 does not have this optimization, nor any direct support for sending supplementary information via annotations.

Independent of MathLink, the commercial package InterCall (Robb, 1992) provides access from Mathematica to the routines in several commercial libraries (e.g., IMSL, NAG, LINPACK) as well as user-created libraries, and allows C, Fortran, and Pascal programs to communicate with Mathematica. The AXIOM-NAG Link (Dewar, 1994) provides a similar connection between AXIOM and the NAG library (NAG Ltd., 1991). The Link uses XDR[†] to encode most data objects and introduces a special mechanism for sending Fortran subroutines as Argument Subprograms. The Link was based on earlier experience with IRENA (Davenport *et al.*, 1991), an interface connecting Reduce with the NAG library.

The POSSO project (Gonzalez-Vega, L. and Recio, T., editors, 1994) is one of the European research projects centered on symbolic computation. It includes the definition of two protocols for exchanging mathematical expressions, XDR-POSSO and ASAP. XDR-POSSO is designed to exchange POSSO data structures using a binary encoding based on the XDR technology. It is strongly tied to the POSSO project and does not include annotations, nor a general extension mechanism to support other kinds of mathematical objects. ASAP (A Simple ASCII Protocol) is more oriented towards portable exchange of mathematical expressions encoded as linearized attributed trees. It provides a basic technology, relying on the user to define the semantics of the expressions exchanged and to provide more optimized encodings when appropriate.

Euromath, another EEC-funded project, defines a Document Type Definition (the Euromath DTD) for SGML[‡], which formally specifies the structure of the mathematical objects to be exchanged (von Sydow, 1992) and provides the GRIF editor for editing them.

Several other (early) works related to the exchange of mathematical data between scientific applications are reported in Arnon (1987). Notable implementations of distributed architectures that provide some exchange of mathematical expressions include Polylith (Purtilo, 1986), CaminoReal (Arnon *et al.*, 1988), SUI (Doleh and Wang, 1990), DSC (Diaz *et al.*, 1991), CAS/PI (Kajler, 1992b), and CC (Dalmas *et al.*, 1994). Realizing the importance of a mathematical data exchange protocol, the Maple group initiated a

[†] XDR, eXternalData Representation, from SUN Microsystems, is a commonly used standard for data representation (Sun Microsystems, Inc., 1990).

[‡] SGML, Standard Generalized Markup Language, is the ISO standard for the exchange of structured documents (I.S.O., 1986).

series of workshops which led to the formation of the *OpenMath* group to develop and standardize such a protocol (Abbott *et al.*, 1995).

4. Goals And Major Design Decisions

Careful consideration of the kinds of tools we want to integrate and the different computing environments in which we envision the tools being used, produced a set of goals and principles which guided the design of MP. The protocol must be:

expressive Scientific computation packages manipulate a wide range of objects from text and symbolic expressions to arrays of floating point numbers. The protocol must be able to represent each with equal ease. Also, there must be a mechanism to convey the "meaning" of symbols. But while the protocol should provide hooks to carry extra information when necessary (e.g., between dissimilar systems), it should not be overly burdensome in situations where the communicating tools already know each other (e.g., a parallel SPMD model, or between a single package and its user interface).

extensible (*open*) The development and inclusion of new and specialized tools should not be precluded by the protocol's design. Tool and interface designers should be able to extend the protocol in useful, but well-defined, ways.

efficient The transmission of data must be efficient. One driving concern is the ability to efficiently transmit large symbolic and/or numeric data such as huge multi-variate polynomials or large vectors and matrices. An important technique is to tune the protocol design in order to make the common case most efficient.

consistent A logically consistent protocol is simpler to understand and to implement. Similar cases should be treated similarly, introduction of special cases must be justified by significant benefits, and there should be a relatively small number of fundamental concepts.

embeddable The protocol should support efficient exchange of mathematical data within a variety of contexts, In particular, it should be embeddable within computational and transport (data delivery) packages without interfering with those packages.

Figure 1 shows how MP can be implemented and used. Application programmers are provided with an interface to MP through a set of language-specific MP libraries. The libraries are based on programming language specifications of the underlying protocol specification. A complete C library has been implemented and is available. We have also developed an experimental GNU Common Lisp (GCL) library. A developer implementing MP in a different language is expected to define a mapping between the language's data types and MP's data types and to provide the routines necessary to convert between the two representations. Furthermore, MP was carefully designed so that efficient routines for encoding and decoding MP data can be written quite easily using byte level instructions and a collection of macro definitions for common types and values (see § 6).

Two important design decisions directly shaped the protocol: first, how best to represent data *and* supply semantic information while maintaining openness, and; second, whether to use a binary or a textual encoding of the data using printable characters only.

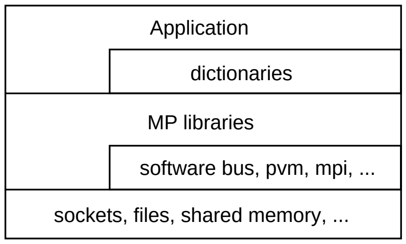
Application
dictionaries
MP libraries
software bus, pvm, mpi, ...
sockets, files, shared memory, ...

Figure 1. The MP programming layers

4.1. SYNTAX AND SEMANTICS

Multi's approach to the first decision was to separate syntactic and semantic issues. At the syntactic level, MP represents structured data (expressions, function calls, matrices, etc.) as *annotated syntax trees*. Each node of this tree is typed, with leaves being any MP *basic type* and internal nodes any MP *operator type*. The syntax tree provides a simple, flexible, and natural representation for mathematical data. The tree can be decorated with annotations that supply extra information. Data is encapsulated in *node packets* or *data packets* and annotations in *annotation packets*. Packet headers contain flags which allow efficient encoding of some essential information about the nature of the packet's content and the way it is encoded.

For nodes carrying symbolic data (a function name for instance), MP requires the sender to attach various flags describing the semantics of the data. These flags ensure that the receiver will always get some minimal *meta-information* about the meaning of each symbol, and the correct way to interpret it. Namely, MP 1.0 requires that each symbol be tagged with 3 flags which explicitly answer the following questions: (1) Does the symbol carry a specific meaning? (2) If so, is this meaning possibly available to the receiver? (3) Is the available meaning MP- or user-defined? Furthermore, MP includes the concept of *dictionaries*. A dictionary is a human readable, off-line document that defines the encoding and semantics of a set of operators and constants. Minimal syntactic information is also provided for operators. Each dictionary has an associated name which can be bound to a node or to a subexpression.

This approach provides both expressiveness and extensibility. It makes it possible to use MP at the syntactic level only (for instance when expressions are exchanged between similar systems), or to provide more complete semantic information when this helps different packages to communicate with each other. In any event, the receiver will always know whether or not it can access the semantics.

4.2. TEXTUAL VERSUS BINARY ENCODING

The second major decision was to use a binary encoding (e.g., 2's complement integers and IEEE floating point numbers for fixnums and the GNU Multiple Precision library (Granlund, 1996) for bignums). A textual encoding (either human-readable such as ASCII or the Universal Character Set Standard (I.S.O., 1993), or a more compact hexadecimal form) may have the advantage of human-readability.[†] But such an encoding works best only with systems whose data is largely textual. The decision to reject a textual encoding was based on three related goals: efficiency, the need to support tools using large amounts of numeric data, and the ability to inexpensively attach flags to many nodes in a tree.

There are two measurable components to efficiency: the *size* of the data block, which affects the time it takes to transmit the data over a network, and the *time* it takes to encode and decode data; that is, to convert between a tool's internal data representation and the protocol's representation. Experience shows that an encoding's effectiveness depends heavily upon the kind of data to be transmitted and also upon the characteristics of the communication medium. More precisely, a binary encoding is far more efficient when transmitting large blocks of numeric data, especially real numbers or large integers. In such cases, a binary encoding reduces both the size of the data to be transmitted and the time necessary to encode and decode the data. On the other hand, expressions composed essentially of symbols and short integers can be encoded more compactly using a textual encoding, while the encoding and decoding times are still comparable to that of a binary encoding. However, one should notice that the conversion time is usually greater than the transmission time, so the advantage of a textual encoding in this case is limited. Furthermore, a binary encoding provides the ability to efficiently attach flags to each node of the tree; a mechanism widely used in MP 1.0. (see § 5.4 and § 6).

As an example of the performance advantage of a binary encoding, consider a simple mesh represented by 2,500 single precision floating point numbers. A binary format requires 10,000 bytes. A human-readable textual encoding representing the values as base 10 strings (e.g., "1.2743", "-1340.002030") requires a minimum of 17,500 bytes when the precision is set to four places (assuming one character for the space between floats, and a minimum six characters per float, including the decimal point). The situation would be far worse for compute servers expecting precision of ten or more places. An alternative to the human-readable textual encoding is to convert the binary representation to a string of hexadecimal digits (e.g., -4 becomes "C0800000" and 17.305 becomes "418A70A4"). Moving between base 2 representations (IEEE and hexadecimal strings) is much more efficient than between binary and a base 10 string representation. Table 1 summarizes the comparison for two Solbourne S4000s (Sun4 clones) connected on an Ethernet subnet. All times are in seconds. The naive algorithm used `sscanf()` and `sprintf()` to do the conversions and certainly could be improved. The hexadecimal algorithm was specially written for these timings and is quite efficient. As the table indicates, both textual encodings require more space than the binary encoding, with the hexadecimal representation requiring twice the space of the binary encoding. This is also reflected in the transmission times. The conversion time dwarfs transmission time for the human-readable textual encoding, as would be expected. The conversion time for the hexadecimal representation is much more tolerable, but is still several times greater than the transmission time.

While the choice between encodings may not be crucial for a distributed problem solving environment, it *is* crucial in parallel applications where issues such as network latency and data conversion time affect both the kind and size of problems for which we can reasonably seek parallel solutions. A binary encoding also proved to be crucial in the

[†] There is nothing in MP 1.0 to preclude writing filters to generate human-readable versions of an MP tree. This is useful for debugging purposes, as is done in the current package. It may also be useful to convert an MP tree into a markup language such as LaTeX (and eventually HTML) for display or inclusion in a document.

Table 1. Transmitting floating-point numbers in text, hexadecimal and binary (in seconds)

Format	# Points	# Bytes	Transmission time	Conversion time to binary	Conversion time from binary	Total time
Textual - human-readable	2,500	17,500	0.0140	2.0325	1.6299	3.6764
Textual - hexadecimal	2,500	20,000	0.0170	0.0430	0.0248	0.0848
Binary - IEEE	2,500	10,000	0.0089	-	-	0.0089

context of interactive visualization of curves and surfaces as soon as real-time interaction was desired (Avitzur *et al.*, 1995).

5. The Design of the Multi Protocol

The Multi protocol focuses on efficient exchange of mathematical data. It was designed with the assumption that it will be embedded in some *other* piece of technology, such as PVM, ToolTalk, Copy & Paste, etc., ..., to transfer mathematical data.

Consequently, MP's primary aim is to provide a well-defined format to effectively encode all kinds of numeric and symbolic data. In addition, libraries to support the protocol in C and Common Lisp have been developed. Based on the format description, alternative implementations can be written, as well as implementations for other languages. Finally, these implementations of MP are expected to be used in various contexts as suggested in § 6.

Thus, MP aims to be a “pluggable” technology that provides efficient mathematical data communication. By concentrating only on data exchange, MP avoids interfering with higher software layers addressing, for example, control integration or typesetting.

5.1. basic and operator mp types

Data in MP is typed. The set of basic types is described below. Structured data are constructed from basic and operator types, and possibly tagged with annotations.

MP 1.0 defines eight families of data types of which seven (integer, real, identifier, constant, string, raw, and meta) are *basic* types which appear as the leaves of a syntax tree, and the eighth is a family of *operator* types for constructing more complex data (functions, expressions, polynomials, matrices, etc.). The types, grouped by families, are explained below. In accordance with the principle of making the common cases most efficient, MP 1.0 makes a distinction between *common* values which are expected to be used frequently and which can be encoded within the node packet header, and *regular* values which appear less frequently and require at least one more field after the packet header (see § 5.2).

Integer types

There are six integer types. The 32-bit signed and unsigned types use the common 2's complement encoding. An arbitrary precision integer type supports values requiring more than 32 bits. The base for arbitrary precision integers is 2^{32} and is adapted from the GNU Multiple Precision library. Finally, three common integer types are available for efficient encoding of small integer values. These are: boolean, 8-bit signed and unsigned integers.

Real types

Machine precision floating point values are represented using the 32-bit and 64-bit standard for normalized floating-point numbers (Institute of Electrical and Electronics Engineers, 1985). An arbitrary precision format supports very large and very small real values. The base for the exponent and mantissa of arbitrary precision reals is 2^{32} and is adapted from the GNU Multiple Precision library.

Identifier types

Identifiers ("variables") composed of characters from the Latin-1 (I.S.O., 1987) character set are transmitted as regular identifiers. Single Latin and Greek character identifiers, which appear frequently in expressions, can be sent very efficiently as common identifiers.

Symbolic Constant types

The regular constant type allows the encoding of *symbolic* constants. Especially common constants such as π or `e` can be encoded in a compact way using the common constant type (see § 5.3 and appendix C).

String type

Strings of up to $2^{32} - 1$ characters in length can be sent. The character set is Latin-1.

Raw type

The raw type supports the exchange of uninterpreted data (e.g., object code, binary file, opaque data). Currently the size limit is $2^{32} - 1$ bytes.

Meta type

The meta type may be thought of as a placeholder for any data type. It is used to provide structuring information and appears only in association with the `type` and `prototype` annotations (see § 5.4.4).

Operator types

The regular operator type supports the transmission of mathematics operators and functions (e.g., *sin*, *integrate*, *divide*) or structured data (e.g., vector, list, polynomial). The common operator type supports a more efficient encoding of frequently used operators (see § 5.3 and appendix B). A separate MP operator (`MP_MpOperator`) is used to carry protocol-specific information. An operator may have 0 to $2^{32} - 1$ operands.

5.2. ANNOTATED TREES

All non-basic data (expressions, data structures, subroutine calls, etc.) are exchanged as linearized annotated syntax trees. This approach has several advantages. It is simple and flexible, as it relies only on a small set of syntactic rules to determine the structure of the tree. It is powerful since there are well-established rules governing the structuring of almost every kind of data we can imagine sending.

Table 2. MP-defined annotations

Annotation type	Valuated	Required
label	y	y
reference	y	y
retrieve	y	y
store	y	y
stored	y	n
prototype	y	y
in-dictionary	y	y
append-dictionary-path	y	y
set-dictionary-for-commons	y	y
comment	y	n
column order	n	n
source	y	n
timing	y	n
type	y	n
unit of measurement	y	n

5.2.1. ANNOTATIONS

Any node of the tree may have zero or more annotations attached to it. Generally, an annotation is a piece of information relevant to a node's data. The decision to attach annotations is made by the sending tool. Annotations may

1 be required for correct decoding of data. For instance, MP uses the `prototype` annotation to specify the format of a special block of data. Without the prototype, the block of data is a meaningless collection of bytes. The same applies to all annotations used to make communication more efficient, including: `label`, `reference`, `store`, `stored`, and `retrieve`. The most critical of them are tagged with the *required* bit set to 1.

2 provide additional interpretive information that may be useful to have, but is not necessarily essential to the data's interpretation. With the `unit of measurement` annotation, for instance, a user can specify that the value sent is in feet, miles, joules, liters, etc.

3 be purely incidental and of no consequence to the correct interpretation of the node's data. For example, `comment ''This solution works best''`.

There are two forms of annotations, *simple* and *valuated*. A simple annotation requires no argument. Valuated annotations take an argument, represented as an annotated tree. MP can support highly structured supplemental information in this way. For example, an MP tree providing structured type information like that used by Axiom would be given as the argument to a `type` annotation. Figure 2 shows the encoding of $7x^2 + a$, tagged with a "type" annotation whose value is `FiniteFieldExtensionByPolynomial(PrimeField 19,` $a^3 + a + 1$`)`.

Type	# Annots	Value	# Operands
`Common-Operator`	1	`+`	2
`TypeAnnotation`			
`Operator`	0	`FiniteFieldExtensionByPolynomial`	2
`Operator`	0	`PrimeField`	1
`Common-Integer`	0	`19`	
`Common-Operator`	0	`+`	3
`Common-Operator`	0	`^`	2
`Common-Identifier`	0	`a`	
`Common-Integer`	0	`3`	
`Common-Identifier`	0	`a`	
`Common-Integer`	0	`1`	
`Common-Operator`	0	`*`	2
`Common-Integer`	0	`7`	
`Common-Operator`	0	`^`	2
`Common-Identifier`	0	`x`	
`Common-Integer`	0	`2`	
`Common-Identifier`	0	`a`	

Figure 2. Providing typing information with the `type` annotation

As was required for nodes, MP requires that some meta-information be enclosed within each annotation to clarify both its meaning and importance. Three flags are defined for this purpose: when set, the *MP-defined* flag indicates that the annotation type is part of the MP specification and that its exact meaning is defined in the MP technical documentation. When this flag is cleared, the annotation is user-defined. The *required* flag indicates that the annotation carries some information essential for properly decoding the data stream and/or understanding the data's meaning. A tool receiving an expression tagged with a required annotation should be able to process the annotation. If it cannot, it must return an error and/or skip the subexpression tagged with this annotation. Conversely, when the flag is cleared, the annotation is said to be supplemental. There should be no harm in disregarding a supplemental annotation. Finally, the *scope* flag determines whether the annotation applies only to the node to which it is attached or to the entire subtree rooted at that node.

Conflicts between contradictory annotations within a node's annotation list are resolved by giving precedence to annotations which come last in the node's list of annotations. Conflicts arising between contradictory annotations within a tree are resolved by giving precedence to the annotation with the most local scope.

5.2.2. NODE, DATA, AND ANNOTATION PACKETS

MP annotated trees are transmitted as a sequence of packets. There are three kinds of packet.

1 A *node packet* is used for each node (interior or leaf) of a tree. Node packets start with a 4-byte packet header whose format is described below. The packet header carries essential information including the node type, number of annotations,

number of children (operands) and the meta-information flags. If the node encodes a common value, then the last byte of the packet header holds that value.

2 A *data packet* is used for efficiency and allows packing a block of homogeneous data (e.g., a vector of floats) without any additional type information.

3 An *annotation packet* contains the annotation type and additional flags. Annotation packets immediately follow the node to which they are attached.

Node packet

A node packet is composed of a packet header followed by zero or more fields, as follows: `NodePacket := NodePacketHeader [# Operands] [# Annotations] [ValueField]` (see also figure 3). The number of fields that may appear after the packet header depends on whether the node carries a "common" or "regular" value, the number of annotations attached to the node, and, for operator types, the number of operands.

The type field (byte 0) of the node packet header contains the information needed to properly identify and recover the rest of the node packet. In addition to identifying the type of the data carried in the node packet, it makes two important distinctions. First, it distinguishes between operator and basic types. The number of operands field is only meaningful for operator types. Second, it distinguishes between common and regular values. Regular values are stored in an additional field in the node packet. Common values are encoded within byte 3 of the packet header.

Byte 1 contains two 4-bit unsigned integers which specify respectively the number of children (in the case of an operator type) and the number of annotations attached to the node. Numbers up to 14 are encoded directly within the 4-bit integer, while a value of 15 indicates that the actual number of children (resp. annotations) follows in an additional 4-byte field, encoded as a 32-bit unsigned integer. This encoding trick proved to be quite effective since for the majority of node packets it allows both fields to be packed in a single byte.

The flags field (byte 2) holds eight 1-bit flags (see table 3), of which three are defined in MP 1.0. These three flags determine whether or not the semantics of an operator, constant, or identifier are known, and if known, where its meaning can be found.

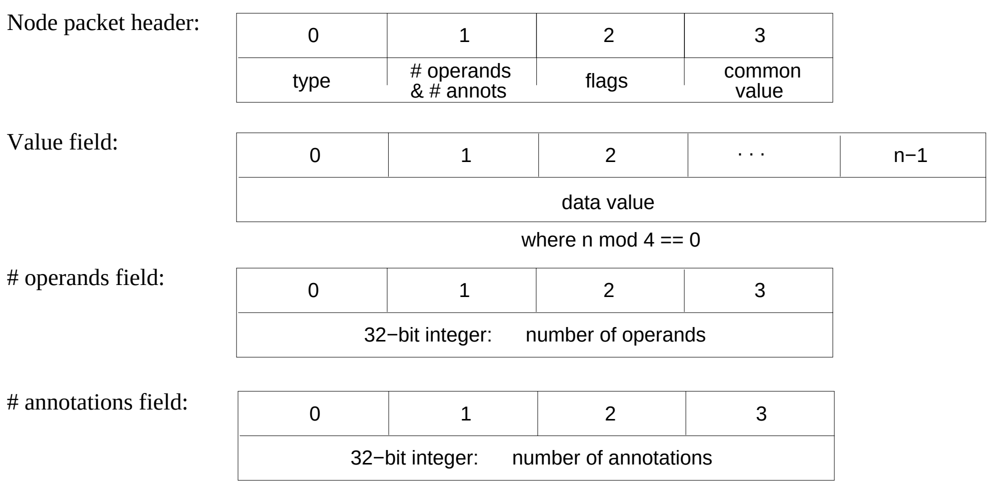

Figure 3. Node packet field formats

Table 3. Node packet flags field

Bit	Meaning
0	has-semantics
1	semantics-available (valid if bit 0 is set)
2	MP-defined (valid if bits 0 and 1 are set)
3 – 7	undefined, reserved for future extensions

When cleared, the `has-semantics` flag indicates that the node's value has no meaning to anyone. In this case, the next two flags are ignored. A sender would clear this flag to indicate, for example, that the identifier x in $f(x)$ has no specific meaning for the sender and that the receiver should not attach any local meaning to it. When the `has-semantics` flag is set, the node's value has a specific meaning and the second flag, `semantics-available`, is valid. If the `semantics-available` flag is cleared, the sender is warning the receiver that the node packet's content has a specific meaning, but that meaning is not to be found in any MP or non-MP dictionary; that is, the meaning is unavailable to the receiver. This could apply, for instance, to the operator f in $f(x)$. When the `semantics-available` flag is set, the third bit, `MP-defined`, is valid. If set, the `MP-defined` flag says that the meaning of the node packet's content is to be found in the MP-defined dictionary for this data type (MP-defined dictionaries are part of the MP format definition). If this flag is cleared, the meaning is to be found in one of the dictionaries associated with the tree by the sender via a dictionary annotation (see § 5.3). The third flag applies only to operators and constants.

Two points should be made about these flags and their location in the packet header. First, MP does not *require* that the receiver do anything with the semantic information provided by the sender. What MP enforces with the three semantic flags is that the *sender* attach some meta-information about the semantics of a node. It is up to the receiver to decide if it can safely perform its task with the semantics available to it. It is entirely possible that some receivers will not need to know all (or any) of the semantics. This would be the case, for example, if the receiver is just to perform some syntactic transformation on the tree. Second, MP-defined objects are defined in the MP 1.0 specification. Hence, when the `MP-defined` flag is set, the semantics of the node's content is known.

The last byte of the packet header is meaningful only when sending common values, otherwise it is not used and is to be ignored by the receiver. For common identifiers (those that are a single byte in length), this byte is a single character. For common constants and common operators, this byte is treated as an 8-bit unsigned integer to be used as an index into a common-dictionary (see § 5.3).

Data packet

A data packet consists of a block of data (with no packet headers) characterized by a regular structure (an array of integers or complex floating point numbers, for instance). Data packets are used for efficiency and appear in conjunction with a `prototype` annotation or a named prototype defined in a dictionary. The simplest application of data packets is for encoding arrays of homogeneous data, but prototypes are sufficiently flexible to handle more complex structures (see § 5.4.4 for an example).

Annotation packet

The format of the annotation packet is shown in figure 4. The annotation type is encoded as an unsigned 2-byte integer. Following the type is a 2-byte flags field providing sixteen 1-bit flags. For the moment, four of these flags are defined: *scope*, *MP-defined*, *required*, and *valuated*.

0	1	2	3
type		flags	

Figure 4. Annotation packet format

Linearization of a complete annotated tree

A linearized version of an annotated tree is formed from a prefix parse of the tree. Node packets are distinguished from annotation packets by their position in the stream of packets. This ordering is well-defined since every tree must begin with a node packet, every node specifies the number of annotation packets associated with it, annotations immediately follow the node with which they are associated, and the operands of operator nodes follow, in order, the last annotation packet related to their parent. Using a BNF notation, this is formalized by: `MP_Tree :== {NodePacket [AnnotationPacket]*}+`

5.3. DICTIONARIES

While syntax trees are a convenient way to transmit structured data between tools, they are not enough to guarantee that the receiving tool will understand the data represented by the tree. There must also be some agreement on the meaning of the operators, constants, and identifiers. To this end, MP supports collections of definitions for mathematical symbols, called *dictionaries*.

The advantage of supporting definitions through dictionaries, as opposed to a single monolithic standard, is twofold. First, specialized tools need not be burdened with understanding operators or constants that are inappropriate for them. Secondly, it allows *other standards* to be independently developed. This is important not only for the ability to support newly developed tools, but also for customization and extensibility.

A dictionary is simply a human-readable document containing an indexed list of symbols together with their meaning. The meaning may be given in a formal or informal way, but needs to be sufficiently precise for a reader to unambiguously understand it. As an example, a dictionary to be used to communicate with Maple (in Maple syntax) could list some or all Maple operators with a sentence to state that the meaning is to be found in the Maple V.3 Reference Manual. Typically, dictionaries are established by type, so there are separate dictionaries for common and regular operators and constants (see appendix B for a sample operator dictionary and appendix C for a sample constant dictionary).

MP-defined dictionaries are part of the MP format specification. They list a subset of most usual mathematical operators and constants. There are MP-defined dictionaries for each of the constant, common constant, operator, and common operator types. The dictionaries for common constants and common operators each may have up to 256 entries, allowing the 8-bit common value to be used as an index into the relevant dictionary. Applications which can send and receive data expressed using *only* MP-defined dictionaries need no additional information to understand the data's "meaning".

Some applications may require more than is found in the MP-defined dictionaries, either because they use a different set of operators, or because they do not want to translate between MP and their native representation, but still wish to take advantage of MP's efficient encoding techniques. In such cases, applications can transmit data encoded using dictionaries defined independently of MP. In general, if a tool knows about a dictionary, it knows about the semantics of the entries in that dictionary. If a tool does not know about the dictionary, it should assume that all node packets which may possibly refer to that dictionary are unknown to it. However it will always be possible to decode the tree at a syntactic level, which may be enough, for example, if the receiving tool is only concerned with archiving expressions or syntax-driven editing.

The following observations on MP nodes and the setting of the semantics flags help make the usage of dictionaries more concrete.

1 Nodes for integers and reals are straightforward to interpret.

2 Identifier or Operator nodes with the `has-semantics` bit cleared are also straightforward to interpret: they are symbolic entities with no specific meaning for either the sender or the receiver, and the receiver should assign none to them.

3 Function nodes such as f or G with `has-semantics` set but `semantics-available` cleared, state explicitly that the node has a meaning for the sender, but this meaning is unavailable to the receiver. These nodes can still be decoded on the syntactic level, but the receiver usually has no way to access the semantics.

4 Nodes like π or $+$, encoded as MP-defined constants and operators (all three bits set), are again straightforward to interpret assuming the receiving application can interpret the MP-defined types as specified in the MP-defined dictionaries. If appropriate, the receiving tool will convert these nodes into its own representation. The dictionary lookup is uniquely identified by the node's type; so, for example, if the type is common-constant, the receiving application would know that the relevant definition is to be found in the MP-common-constant dictionary.

5 Function nodes such as f or G with the `has-semantics` and `semantics-available` bits set and the `MP-defined` bit cleared, indicate that the node has a meaning and this meaning is described in a non-MP dictionary. It is in this fashion that separately developed standards may provide global, system independent meaning.

In the last case (and only in that case), we need an additional mechanism to properly identify the dictionary in which the meaning of the node's content is given. Three annotations are available for this purpose.

First, consider the common constant and common operator types. In both cases, values are encoded in the node packet header as an 8-bit integer to be used as an index into a 256-entry common dictionary. Therefore, there can only be one common dictionary active for each of the two common types. This dictionary is identified with the `set-dictionary-for-common-type` annotation, which takes as its argument a two element list containing a dictionary name, sent as an MP `identifier`, and the MP common type for which it applies. The scope of the annotation is the node to which the annotation is attached and the subtree rooted at that node. The effect of this annotation is that the

Table 4. Encoding comparisons (in bytes)

Expression	Textual (using a *Lispish* syntax)	MP 0.5	MP 1.0 (using *common* types)
$\sin(n\theta)$	17	60	16
$\sin(ax+b)/(1-\cos(cx-b))$	47	208	60
$265252859830848000000/n$	28	44	24
polynomial p40	1,537	2,664	1,228
polynomial p40 (CRE form)	1,006	656	608
array of 500 4-digit integers	2,506	2,040	2,016
array of 500 25-digit integers	13,006	6,040	6,016
array of 500 6-digit floats	4,006	2,040	2,016

meaning of any value of the specified common type in the given subtree is to be found in the specified (non-MP) dictionary.

In contrast, there may be multiple dictionaries associated with each regular type. The names of these dictionaries are given in a *dictionary path*. The `append-dictionary-path` annotation can be used to tag a subexpression with a list consisting of the dictionary name and an MP type. Each path is associated with a regular type (regular constants and operators) and specifies where to look for the meaning of the node's values of that type in the subtree.

Finally, the previous two annotations can be overridden with the `in-dictionary` annotation, which precisely identifies the dictionary to use for a given node.

5.4. OPTIMIZATIONS

MP was designed with efficiency in mind. It uses a binary encoding to efficiently encode numeric data and to compress important additional information in the flags field of both node and annotation packets.

The common types were introduced in MP 1.0 to provide a more compact encoding for commonly used values. Using common types, one-character symbols, small integers, as well as usual constants and operators are encoded within just one byte *inside* the node packet header. This includes for instance: 0, 1, 255, `true`, `false`, a, b, x, α, β, π, e, $\emptyset$, ∞, $+$, $-$, sin, sqrt, list, set, etc. Generally, for expressions containing commonly used operators and single character identifiers, the savings of the new format are significant and on a par with that of a simple textual representation. But keep in mind that through the semantics flags, MP is actually communicating much more information than its textual counterpart.

Table 4 compares the cost in bytes for some sample expressions when encoded in a textual format (using a Lispish syntax), MP 0.5, and MP 1.0. The p40 polynomial first appeared as part of “SIGSAM Problem 7” in a series of practical problems/challenges for SAC systems (Johnson and Graham, 1974). Since then this degree-40 polynomial has been used as a benchmark problem to compare factoring algorithms and implementations. It is given in appendix A. The *Canonical Rational Expression* (CRE) form is a more efficient way to represent and manipulate rational expressions and, as a special case, polynomials. It is a sparse, distributed representation of the polynomial. The coefficients of p40 are bignums and the exponents are fixnums. The MP representation of the CRE form and the 500 element arrays take advantage of prototypes (explained in § 5.4) to pack the data.

Independently of the mechanisms already mentioned, MP includes a collection of optimization techniques for further reducing the amount of data to be transmitted. They are fairly generic and are based on MP-defined annotations. Using the techniques described below is an option generally left to the sending tool.

5.4.1. SUBEXPRESSION SHARING

The first such optimization is common subexpression sharing, supported through the `label` and `reference` annotations. When a subexpression occurs more than once in an expression, the first occurrence may be labeled with subsequent occurrences simply containing a reference to the labeled subexpression. The label is simply an integer created by the sender and each label within the expression must be unique. A node annotated with the reference annotation is treated as a dummy node and is replaced by the referenced node/tree. The scope of the label is retricted to the current expression; labels do not "carry over" to other expressions. Subexpression sharing does not place any requirement on the receiver to store the expression this way. It is intended to increase transmission bandwidth.

5.4.2. REFERENCING PREVIOUSLY TRANSMITTED SUBEXPRESSIONS

The second predefined optimization uses the `store` and `retrieve` annotations, so that two tools can avoid repeated exchanges of identical (sub)expressions. The store annotation takes as its argument a *handle*. A handle is quite similar to a label, but continues to have meaning after the tree in which it is found has been parsed and processed; so the *same* handle may appear as the argument to retrieve annotations in *different* expressions. The sender is responsible for generating a unique handle for each store annotation. The receiving application stores the tree rooted at the current node and associates with it the given handle. These handles can be referenced using the retrieve annotation on a dummy node. The receiver must maintain the handle's uniqueness with respect to the sender as the receiver may be in communication with multiple tools.

5.4.3. LAZY COMMUNICATION

The third predefined optimization uses the `stored` annotation. It is a variant of the store/retrieve mechanism. With it a sending tool can inform the receiver that it has taken the initiative of storing the annotated subexpression. This makes it possible to implement a *lazy* style of communication between applications which is useful in several contexts:

1 A server can return an answer immediately to a client in the form of an empty MP tree with a `stored` annotation to provide a handle to a result yet to be computed.
2 Large expressions can be exchanged incrementally by sending a framework of the expression with many subexpressions annotated as `stored`. The technique is most useful when a receiving application can take immediate advantage of the framework expression. For example, a graphic window can start updating some areas using the framework data. The missing parts can be sent later upon request by the receiver through the use of handles. In CAS/PI (Kajler, 1992b), for instance, the mechanism

is used in connecting remote computer algebra systems and formula editors. Using lazy communication, a CAS sends the editor only those parts of an expression that will be immediately displayed. Unsent subexpressions are displayed as icons. Zooming on such an icon causes the editor to request the missing data from the remote application.

5.4.4. PACKING HOMOGENEOUS DATA

The last predefined optimization addresses the overhead associated with typed data. Each MP node packet starts with a 4-byte header. Frequently, however, a block of data will be sent that is characterized by having a homogeneous format. That is, there is a pattern to the data elements, as in a matrix of arbitrary precision reals or a vector of integers. Unoptimized, each integer of the vector, for instance, would require a complete node packet. For a vector of length 1,000, this would require 4,000 bytes of overhead.

We can take advantage of our knowledge of the pattern by creating a `prototype` annotation specifying the type of the data to be found. The prototype is an abstract syntax tree giving the structure of the data block. Individual nodes of the prototype are either data values or the MP type `meta`, specifying the type of data to be read from the data block. Only the data values corresponding to meta entries in the prototype need be transmitted. Subsequently, the entire collection of data items is placed in a single data packet. Figure 5 illustrates the use of this technique for a vector of 100 complex numbers. The unoptimized encoding (on the left hand side) requires 2,008 bytes while the optimized version (on the right) requires only 820 bytes.

Especially useful prototypes may be named and placed in a dictionary. These *named prototypes* can then be used in operator packets by simply giving the prototype name as the value to the operator and identifying the dictionary in which the named prototype is defined. Combined with compiled routines specific to these prototypes, this is an efficient and flexible way of transmitting structured data.

Type	# annots	Value	# operands
common-op	0	list	100
common-op	0	list	2
real32	0	2.150	
real32	0	1.037	
common-op	0	list	2
real32	0	0.031	
real32	0	3.041	
common-op	0	list	2
real32	0	1.572	
real32	0	0.741	
⋮			

Type	# annots	Value	# operands
common-op	1	list	100
prototype			
common-op	0	list	2
meta	0	real32	
meta	0	real32	
		2.150	
		1.037	
		0.031	
		3.041	
		1.572	
		0.741	
		⋮	

Figure 5. Packing homogeneous data using the prototype annotation, meta type, and data packet

6. Implementation and Applications

Libraries of routines have been written in C and GNU Common Lisp (GCL) to send and receive MP data. These libraries have been used to connect stand-alone tools over a network. This section describes the two libraries and their use with an application.

The C library (MP-C) contains routines to send and receive node, data, and annotation packets, as well as utility routines for error handling, event logging, memory management, and the creation/destruction of links and environments. High level routines read and write complete node packets for each of the MP types. Lower level routines are available to read and write data values (that is, data without the accompanying packet header). The data handling routines take care of converting to and from network byte order. For point-to-point TCP connections, the communicating parties negotiate whether to use big- or little-endian word order, with the default being big-endian (network byte order). Data are sent between applications as "messages", with one or more expressions packed within each message. Messages may be buffered on the sending side until complete and transmitted in their entirety, or they may be transmitted in fragments, allowing parsing on the receiving side to overlap with transmission on the sending side. Utility routines are provided to determine when the end of a message has been reached, for peeking at the next packet header, and for skipping messages. Packages communicate with other packages through an MP link, which is simply a logical endpoint for communication. A package may have several links open at once. Each link maintains a set of buffers which hold data to be delivered to (sending buffers) or data read from (receiving buffers) the underlying data delivery service. Links are created within an MP environment, which maintains global attributes of the session such as the logfile name, the size of the send and receive buffers attached with each link object, whether messages are to be sent as fragments or in their entirety, and so on. Links inherit several important attributes from their environment, but some of these are treated as options which are resettable on a per link basis. One of these options is the ability to enable logging of events (e.g., reading, writing, initializations, etc.) which produces a human-readable log entry for each event type requested. This has been indispensable for debugging purposes. The environment also has some settable options.

For flexibility, the link object is separate from the underlying data delivery service actually responsible for transmitting the data. At link creation time, a particular transport device is bound to the link. The advantage of this approach is that the logical transmission of data is separated from the underlying data delivery system, making it very easy to use MP with sockets, files, shared memory and existing communication technologies such as PVM, MPI, and ToolTalk. In fact, any mechanism that can send uninterpreted data can send MP trees with little or no extra effort. Using a different delivery system simply requires providing suitable interface routines. We have successfully applied this approach in using PVM to send MP trees. This only required writing two small routines to move data between the MP and PVM buffers. Since all the "packing" is done by MP, the PVM `PvmDataRaw` encoding option was used, telling the PVM read and write routines to treat their data arguments as uninterpreted data. No changes to the PVM library were required. An optimization we will explore is using the PVM `PvmDataInPlace` encoding option, which allows PVM to read/write data directly from/to the user's buffers, eliminating the MP-PVM buffer copying. Portability testing of the MP-C library has been done across SUNs, SGIs, RS6000s, HPs, DecStations, and a Pentium running Linux.

Using the C interface provided by GCL, an experimental library of routines (MP-GCL) was written on top of MP-C. MP-GCL has been tested and used with Maxima on Suns. A goal was to make the interface as Lisp-like as possible in its appearance and behavior. For example, the MP stream was converted into a GCL I/O object and an MP package was created to preserve MP name space. As part of a separate project at Kent State, a GCL interface to PVM3 has been built (Li and Wang, 1996). Similarly, a GCL interface to MPI has been written (Cooperman, 1995). These interfaces could use the MP-GCL library for marshaling data. An important side effect of having the MP-GCL library is that it gives us access to Maxima. As a test of the protocol's ability to efficiently parse non-trivial symbolic expressions, a small parser was written to convert between MP trees and the Maxima internal representation. With this parser, we wrote the p40 polynomial (using Maxima internal representations and their equivalent MP representations) to a file and read it back (see also § 5.4 and appendix A). The results are given in table 5. The test was performed on a Sparcstation and I/O was to a local file (NFS was not involved).

Table 5. Maxima-MP p40 timings (in seconds)

Representation	Write time	Read time
native Lisp	0.23	0.35
MP of native Lisp	0.52	1.10
Maxima CRE form	0.15	0.20
MP of Maxima CRE	0.05	0.03

The MP-C and MP-GCL libraries were used together to speedup data exchanges between the graphing package IZIC and Maxima. IZIC (Fournier *et al.*, 1995) is a stand-alone graphing package for plotting curves and surfaces. The meshes are computed by Maxima, packaged in a display object by the Maxima-IZIC interface, MaxIzic (Bachmann, 1994), and transmitted to IZIC for rendering and display. Originally IZIC communicated with remote systems via files using a textual encoding of the data. By sending binary data over a socket instead of textual data via a file, a significant speedup could be achieved as suggested in Avitzur *et al.* (1995).

In order to use MP, we linked IZIC to the MP-C library and extended the command language of IZIC with a series of a new routines for exchanging MP-encoded IZIC meshes. Keywords such as "color" and "domain" introduce values describing some feature of the object. These keywords naturally became operators in an MP tree and the number of

Table 6. Maxima-IZIC I/O timings (in seconds)

Object	IZIC textual encoding			IZIC binary encoding			MP 1.0 binary encoding		
	size	write	read	size	write	read	size	write	read
Line	147	.008	.003	147	.008	.003	332	.003	.002
Cube	2,043	.198	.054	2,076	.075	.010	3,668	.078	.010
Enneper	149,042	11.56	11.54	60,137	.033	.020	60,448	.033	.020

values associated with a keyword became the number of operands. Since IZIC knows the type of data associated with each keyword, those values were sent in data packets. Writing the interface was straightforward. MaxIzic was modified to use the MP-GCL library to construct MP trees. Table 6 gives the sizes of the objects using the different encodings and speedups achieved by using MP 1.0 and its binary encoding over the original textual encoding. All data was sent using sockets between a Sparcstation and an HP 9000/730 connected on a LAN. The tests were done 10 times and an average time was taken. Three objects were used for these tests: a simple line; a cube consisting of 12 lines and 6 surfaces with 4 points per surface; an Enneper surface composed of 7,500 points. The table also gives a comparison to a specially written routine within IZIC that transmitted the object's points as doubles and everything else as text. Doubles were also used in the MP encoding (as required by IZIC). As the table shows, using MP was much faster than the textual encoding, confirming our assumptions about the relative times for conversion and transmission. For small objects, we pay a penalty in size for using a binary encoding, but as the first line of table 6 indicates, we make up for it in speed. The benefits of a binary encoding are more apparent for larger objects. For the Enneper surface, the binary encoding outperforms the textual encoding in both size and speed. Significantly, the MP encoding produces timings as good as or better than the carefully written IZIC routine.

7. Lessons Learned and Future Work

The design changes between MP 0.5 and MP 1.0 are based on our experience using the initial implementation. The four significant lessons learned were that (1) it pays to focus on the most common cases and provide optimized encodings for them; (2) a careful design must take care of implementation issues in order to ensure that not only transmission, but also encoding and decoding can be performed efficiently; (3) a binary encoding is faster and many times more compact than a textual encoding as soon as a large amount of numeric data is involved; (4) with an intelligent design, a binary encoding can be nearly as compact as its textual counterpart, even for small symbolic expressions. We will continue to let our experiences guide further changes to the general design.

An area of intense interest to us is supporting different numeric representations for fixed precision floating point and arbitrary precision numbers. Currently we support a single representation for each of these types. A more flexible and efficient approach is to support multiple representations and to let the communicating parties negotiate which representation to use.

Another promising avenue to explore is to extend the indexing idea used with common dictionaries to the regular operator and regular constant types and to go beyond the 256 entry restriction for common dictionaries imposed by space limitations in the packet header. Indexing leads to very compact encoding of information *and* efficient lookups.

Another issue to consider is using the emerging Universal Character Set Standard (I.S.O., 1993) for encoding strings, identifiers, and constants. Three additional MP types could be added to support the new standard when sending non-Latin-1 characters.

Currently, the dictionaries provided by MP define only the most common operators and constants. Larger, more complete, dictionaries are still to be written. A scheme to construct larger dictionaries based on existing ones (inheritance) should be formalized. Experience and user input will be important in constructing dictionaries.

The protocol's power and flexibility suggest that we should be able to construct new tools from existing tools, or part(s) of existing tools, which previously worked only as separate pieces of software. We continue to evaluate the complexity of MP-tool interfaces, which play a large role in integration. But a long term goal of the Multi Project is to encourage the creation of new tools which have the notions of integration and cooperation with other tools inherent in their design. We hope to use our experience writing MP-tool interfaces to create a set of design guidelines.

We will continue to expand the number of data delivery systems that can carry MP data. Based on our success with the C library for PVM, we will integrate MP with MPI and then do the same with our MP-GCL library and Lisp interfaces for PVM and MPI. Also, the efficiency of the MP-tool interface is important and deserves re-examination.

Finally, it may be time to promote a standard within the computer algebra community for the exchange of mathematical data between applications. Such a standard should be considered within the framework of well-established networking models such as the application and presentation layers of the OSI. To this end, the diverse and complementary efforts made up to now already provide numerous valuable ideas and experience. Merging these efforts should be the next step to achieve.

8. Availability

MP version 1.0 is available via anonymous ftp from `ftp.mcs.kent.edu` from the `/pub/MP` directory. This distribution includes the source code for the MP-C library and a user's guide. Questions, feedback, and requests to be added to a mailing list for announcements about MP can be addressed to any of the authors. More information can be found at `http://SymbolicNet.mcs.kent.edu/areas/protocols/mp.html`.

9. Conclusion

A significant amount of work went into the design and implementation of MP, but it represents only a first step in building a common protocol for distributed scientific computation. The main features of MP include a layered approach, a tool independent format for exchanging mathematical data, efficiency through binary encoding and optimizations, and extensibility through flags, annotations, and dictionaries. Also, the implementation of MP is designed to be readily "pluggable" into complementary technologies and fulfills the need for a mathematical data exchange protocol in various contexts. It is expected that well-designed, efficient, and *standardized* mathematical protocols will be important for the advancement of high-performance scientific computing through parallelism and distribution.

Acknowledgments

The authors would like to thank Olaf Bachmann for his contributions to the MP-GCL library and the Maxima-IZIC interface, and Shi Baw Chng for his contributions to the buffering layer of the MP-C library. We also wish to thank Yaser Doleh, Peter Hintenhaus, and Scott Rhine who contributed to the early design of MP, and Olaf Bachmann, Steve Chapin, Michael Rothstein, Ken Weber, and the reviewers for providing insightful comments on earlier drafts of this paper.

References

Abbott, J., van Leeuwen, A., Strotmann, A. (1995). Objectives of OpenMath. Available from `http://www.can.nl/~abbott/OpenMath/`.

Arnon, D. (1987). Report of the Workshop on Environments for Computational Mathematics. *ACM SIGSAM Bulletin*, **21**(4):42–48.

Arnon, D., Beach, R., McIsaac, K., Waldspurger, C. (1988). CaminoReal: An Interactive Mathematical Notebook. In van Vliet, J. C., editor, *Proc. of EP'88 International Conference on Electronic Publishing, Document Manipulation, and Typography, Nice, France*, pages 1–18. Cambridge University Press. Also available as Technical Report EDL-89-1, Xerox PARC, 1989.

Avitzur, R., Bachmann, O., Kajler, N. (1995). From Honest to Intelligent Plotting. In Levelt, A. H. M., editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'95), Montreal, Canada*, pages 32–41. ACM Press. Also available as RIACA technical report # 5 RIACA, Kruislaan 419, 1098 Amsterdam, Netherlands.

Bachmann, O. (1994). MAXIZIC - A Maxima Interface to IZIC. RIACA technical report 4, RIACA, Kruislaan 419, 1098 Amsterdam, Netherlands.

Birrell, A. D., Nelson, B. J. (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, **2**(1):39–59.

Cohen, A., Meertens, L. (1996). The ACELA Project: Aims and Plans. In Kajler, N., editor, *Human Interaction in Symbolic Computing*, Texts and Monographs in Symbolic Computation. Springer-Verlag. To appear.

Cooperman, G. (1995). STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems. In Levelt, A. H. M., editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'95), Montreal, Canada*, pages 126–132. ACM Press.

Dalmas, S., Gaëtano, M., Sausse, A. (1994). Distributed Computer Algebra: the Central Control Approach. In Hong, H., editor, *Proc. of the 1st Intl. Symp. on Parallel Symbolic Computation (PASCO'94)*, volume 5 of *Lecture Notes Series in Computing*, Hagenberg/Linz, Austria. World Scientific.

Davenport, J., Dewar, M. C., Richardson, M. (1991). Symbolic and Numeric Computation: The IRENA Project. In *Proceedings of the Workshop on Symbolic and Numeric Computing*, pages 1–18, Computing Centre, University of Helsinki, Research Reports 16.

Dewar, M. C. (1994). Manipulating Fortran Code in AXIOM and the AXIOM-NAG Link. In Apiola, H., Laine, M., Valkeila, E., editors, *Proceedings of the Workshop on Symbolic and Numeric Computing*, pages 1–12. University of Helsinki, Finland. Available as Technical Report B10, Rolf Nevanlinna Institute.

Diaz, A., Kaltofen, E., Schmitz, K., Valente, T., Hitz, M., Lobo, A., Smyth, P. (1991). DSC: A System for Distributed Symbolic Computation. In Watt, S. M., editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'91), Bonn, Germany*, pages 323–332. ACM Press.

Doleh, Y., Wang, P. S. (1990). SUI: A System Independent User Interface for an Integrated Scientific Computing Environment. In Watanabe, S., Nagata, M., editors, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'90), Tokyo, Japan*, pages 88–94. Addison-Wesley.

Fournier, R., Kajler, N., Mourrain, B. (1995). Visualization of Mathematical Surfaces: the IZIC Server Approach. *Journal of Symbolic Computation*, **19**(1/2/3):159–173.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. (1994). PVM3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory.

Gonzalez-Vega, L. and Recio, T., editors (1994). The PoSSo NEWSLETTER. Available electronically from `posso.dm.unipi.it`.

Granlund, T. (1996). GNU MP: The GNU Multiple Precision Arithmetic Library, Edition 2.0. Technical report, The Free Software Foundation.

Gray, S., Kajler, N., Wang, P. S. (1994a). MP: A Protocol for Efficient Exchange of Mathematical Expressions. In Giesbrecht, M., editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'94), Oxford, GB*, pages 330–335. ACM Press.

Gray, S., Kajler, N., Wang, P. S. (1994b). Specification of Expression Encoding in the Multi Protocol, Version 0.5. Technical Report ICM-9404-64, Institute for Computational Mathematics, Kent State University.

Gropp, W., Lusk, R., Skjellum, A. (1994). *Using MPI.* MIT Press.

Institute of Electrical and Electronics Engineers (1985). IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard 754-1985.

I.S.O. (1986). Information processing – Text and office systems – Standard Generalized Markup Language (SGML). ISO 8879.

I.S.O. (1987). Information processing – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1. ISO 8859-1.

I.S.O. (1993). Information technology – Universal Multiple – Octet Coded Character Set (UCS)-Part 1: Architecture and Multilingual Plane. ISO/IEC 10646-1.

Jacobs, I., Montagnac, F., Bertot, J., Clément, D., Prunet, V. (1993). The Sophtalk Reference Manual. Rapport Technique 150, INRIA.

Johnson, S., Graham, R. L. (1974). SIGSAM Problem #7. *ACM SIGSAM Bulletin*, page 4.

Kajler, N. (1992a). Building a Computer Algebra Environment by Composition of Collaborative Tools. In Fitch, J. P., editor, *Proc. of DISCO'92, Bath, GB*, volume 721 of *LNCS*, pages 85–94. Springer-Verlag.

Kajler, N. (1992b). CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems. In Wang, P. S., editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'92), Berkeley, USA*, pages 376–386. ACM Press.

Leong, B. (1986). Iris: Design of an User Interface Program for Symbolic Algebra. In Char, B. W., editor, *Proc.of the 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC'86), Waterloo, Canada*, pages 1–6. ACM Press.

Li, L., Wang, P. (1996). The CL-PVM Package. *ACM SIGSAM Bulletin*, **29**(3/4):2–8.

NAG Ltd. (1991). *The NAG Fortran Library Manual – Mark 15.*

Pintur, D. A. (1994). MathEdge: The Application Development Toolkit for Maple. *MapleTech*, **1**(2):31–38.

Purtilo, J. M. (1986). *A Software Interconnection Technology to Support Specification of Computational Environments.* PhD thesis, Dpt. of Computer Science, University of Illinois at Urbana-Champaign.

Quint, V., Vatton, I., Paoli, J. (1996). Active Structured Documents as User Interfaces. In Kajler, N., editor, *Human Interaction in Symbolic Computing*, Texts and Monographs in Symbolic Computation. Springer-Verlag. To appear.

Robb, T. (1992). *InterCall.* Analytica, PO Box 343, Subiaco 6008, Perth WA, Australia.

Schefström, D. (1989). Building a Highly Integrated Development Environment Using Preexisting Parts. In *IFIP 11th World Computer Congress*, San Francisco, USA.

Sun Microsystems, Inc. (1990). *Network Programming Guide (revision A).* Mountain View, CA. Part number 800-3850-10.

SunSoft, Inc. (1991). The ToolTalk Service. Technical report, Sun Microsystems.

von Sydow, B. (1992). The Design of the Euromath System. *Euromath Bulletin*, **1**(1):39–48.

Wolfram Research, Inc. (1993). MathLink Reference Guide (version 2.2). Mathematica Technical Report.

A. Polynomial p40

$$
\begin{aligned}
&1125899906842624Y^{40} + 9007199254740992Y^{39} + \\
&43523068273885184Y^{38} + 96686654500110336Y^{37} + \\
&71892942171668480Y^{36} - 203545990580404224Y^{35} - \\
&3231739551940608Y^{34} + 2967153761027358720Y^{33} + \\
&3933037175129505792Y^{32} - 10392801849559220224Y^{31} - \\
&26535501289876357120Y^{30} + 6097080187006589 3376Y^{29} + \\
&124970981316064444416Y^{28} - 29406195022070965862 4Y^{27} - \\
&377892178261310439424Y^{26} + 1043009160244635893760Y^{25} + \\
&1247931935205212815360Y^{24} - 4589223739355845099520Y^{23} - \\
&5344699760802655109 12Y^{22} + 13715397752064378404864Y^{21} - \\
&13302634037980246573056Y^{20} - 11542780733595462926336Y^{19} + \\
&30185143375271381827584Y^{18} - 11552322281059389603840Y^{17} - \\
&20764788003456939618304Y^{16} + 23716448816180242333696Y^{15} - \\
&14651220988368672604 16Y^{14} - 11834041405374495383552Y^{13} + \\
&7044458729366598924288Y^{12} + 5964534517854644633 60Y^{11} - \\
&2236080014905849481216Y^{10} + 834136964316351980544Y^{9} + \\
&400775346903652253 44Y^{8} - 113182986697447904512Y^{7} + \\
&28989425675169724800Y^{6} + 2379082981480823 04Y^{5} - \\
&1444259673626878112Y^{4} + 274453725912274624Y^{3} - \\
&18357860832301728Y^{2} + 162347279437248Y - \\
&46004560343
\end{aligned}
$$

B. Dictionary example for operators

```
# Index     String Rep        Description            Arity
------------------------------------------------------------------------------
    1           +             Addition               n-ary
    2           -             Subtraction            binary
    9          neg            Negative value         unary
    3           *             Multiplication         n-ary
    4           /             Division               binary
    5          inv            Inverse (1 / x)        unary
    6          pwr            Exponentiation         binary
    7          mod            Modulus                binary
    8          abs            Absolute value         unary
   10         floor           Floor                  unary
   11          ceil           Ceiling                unary
   12         trunc           Truncation             unary
                              floor(x) for x >= 0, ceil(x) for x < 0
   13         round           Round                  unary
                              floor(x + .5) for x >= 0, ceil(x - .5) for x < 0
   14         ln(x)           Natural logarithm      unary
   15         lg(a)           Base-2 logarithm       unary
       . . .
   27          sin            Sine                   unary
   28          cos            Cosine                 unary
        . . .
   47          and            Logical AND            n-ary
   48           or            logical OR             n-ary
   49          not            logical NOT            unary
        . . .
   70         limit           Limit(expr, x->x0)     3-ary
                              arg1: expr, arg2: x, arg3: x0
   71         limitm          Limit, but approach x0 from below
   72         limitp          Limit, but approach x0 from above
   73          diff           Differentiate(expr, x)  binary
                              arg1: expr, arg2: x
```

C. Dictionary example for constants

```
# Index     String Rep         Description
------------------------------------------------------------------------------
    1           Pi             Circumference / diameter of circle
    2       EulerGamma         Euler's constant gamma
    3        infinity          point at infinity
    4        infinityp         positive infinity
    4        infinitym         negative infinity
    5       goldenratio        the golden ratio, (sqrt(5)+1)/2
    6           e              base of natural logarithm
    7           I              the imaginary unit, square root of -1
    8         Catalan          Catalan's constant
       . . .
  # Physical Constants
  191           c              speed of light in a vacuum
  192           h              Planck constant
  193           g              gravitational constant
```