# MPP: A Framework for Distributed Polynomial Computations*

Olaf Bachmann† Hans Schönemann

Centre for Computer Algebra
Department of Mathematics
University of Kaiserslautern
Kaiserslautern, Germany
{obachman,hannes}@mathematik.uni-kl.de

Simon Gray

Department of Mathematics and Computer Science
Kent State University
Kent, OH 44242, USA
sgray@mcs.kent.edu

**Abstract**

There are obvious advantages to providing communication links between independent software tools, including the ability to do parallel distributed computation, distributed problem solving, and providing more direct access to a wealth of computational resources. The challenge of providing *connectivity* is to produce homogeneity in a heterogeneous environment. We have explored this problem within the context of applications specially designed for polynomial computations. Our solution uses the Multi Protocol (MP) to establish communication links between independent packages, and the ideas of dictionaries, prototypes, and annotations provided in MP. We describe the design of an MP dictionary for polynomial computations (MPP), as well as the implementation of interfaces to SINGULAR, FACTORY, and Mathematica based on this dictionary. Important aspects of the design and implementation include generality, efficiency, and the ability to convey supplemental information. We include a discussion of our experiences and some timings.

## 1 Introduction

Computing in a distributed fashion is coming of age, driven in equal parts by advances in technology and by the desire to simply and efficiently access the functionality of a growing collection of specialized, stand-alone packages from within a single framework. The challenge of providing *connectivity* is to produce homogeneity in a heterogeneous environment, overcoming differences at several levels (machine, application, language, etc.). This is complicated by the desire to also make the connection efficient. We have explored this problem within the context of symbolic computation and, in particular, with respect to systems specially designed for polynomial computations.

*Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650, by the Deutsche Forschungsgemeinschaft, and by the Stiftung Innovation des Landes Rheinland-Pfalz.

†Also: Department of Mathematics and Computer Science, Kent State University Kent, Ohio, email: obachman@mcs.kent.edu.

Appeared in the proceedings of the 1996 International Symposium on Symbolic and Algebraic Computing (ISSAC'96).

Our solution is based on mechanisms found in the Multi Protocol (MP) [9, 10]: dictionaries, prototypes, and annotations. Dictionaries address the problem of application heterogeneity by supplying a standardized representation and semantics for, among other things, operators and data. A prototype is a description of the structure and content of a data object that may appear in a dictionary in combination with an operator and addresses the problem of efficiency. Annotations provide supplemental semantic information that may be useful to the receiver. Within this framework, systems can exchange a polynomial object efficiently and with a shared and well-defined understanding of the object's meaning.

More specifically, we had the following goals:

1. Develop an MP Polynomial Dictionary (MPP) describing the syntax and semantics of polynomial structures and polynomial operations.
2. Provide a collection of library routines which simplify the implementation of interfaces to the MPP encodings.
3. Demonstrate the feasibility of the MPP dictionary and the MPP library by implementing efficient MP link interfaces based on the dictionary to three packages: SINGULAR [12], FACTORY [15], and Mathematica.

### 1.1 Generality and efficiency

There were three concerns that guided our design and implementation. First, the dictionary should be as general as possible. By generality we mean non-system specific and reflecting the needs of most systems. A general dictionary simplifies the design and implementation of interfaces to different systems. Second, the representation of data should be compact and the communication link should be efficient. Efficiency applies to the time it takes to transmit the data, to convert between the transmitted and native encodings, and to parse the expression. Third, the communication protocol should include support for embedding additional useful information within a polynomial expression. This is a concern for systems which are heavily typed and expend some computational effort to generate the information. There is a performance loss if the extra information generated by a system cannot be communicated to the receiver, who must then recompute it. We recognize that in many cases the time to transmit and parse an expression will be dwarfed by the computation time, and that within the context of a distributed problem solving environment, efficiency is not a dominating concern. However, there are circumstances in which efficiency does matter, such as parallel computations and the transmission of especially large data sets, and, certainly, there is no disadvantage to having an efficient link.

There is a well-known and natural tension between generality and efficiency, often resulting in interfaces which are either general *or* efficient. The challenge in designing our dictionary was to make it general enough so that an MPP interface to some system would give it access to the resources of any other system with an MPP interface, yet at the same time be efficient enough so that the *same* interface could be used in situations where efficiency was especially important (e.g. parallel distributed computations).

## 2 The Multi Protocol

The link was built on top of the Multi Protocol (MP). MP's purpose is to support efficient communication of mathematical data among scientific computing systems. MP defines a set of basic types and a mechanism for constructing structured data. Numeric data (fixed and arbitrary precision floats and integers) are transmitted in a binary format (2's complement, IEEE float, etc.). MP transparently handles byte-ordering and floating point representations. Expressions are transmitted between cooperating processes as linearized, annotated syntax trees (preorder traversal) in a sequence of *node packets*, where each node packet contains a node from the expression's syntax tree. The node packet has fields giving the *type* of the data carried in the packet, the number of children (for operators) that follow, the number of annotations, some semantic information, and the data. Annotations efficiently carry additional information which may be supplementary and can be safely ignored by the receiver, or may contain information essential to the proper decoding of the data. MP supports collections of definitions for annotations and mathematical symbols (operators and symbolic constants) in *dictionaries*. Operators and constants which occur frequently have an optimized encoding and are known as "common". MP is independent of the transport medium used to transmit the data and currently works with files, sockets, the Parallel Virtual Machine (PVM) [16], and ToolBus [4], a software bus architecture.

### 2.1 Dictionaries

A dictionary is an off-line, human-readable document composed of three sections (others may be added in the future) containing the definition of operators, constants, and annotations. A definition may be formal or informal, but must be sufficiently precise to be unambiguous.

There is no requirement that each dictionary has all three sections. The first 256 entries in the operator and constant sections are also given integer tags which have an especially efficient encoding within MP (as "common" operators and constants) and are where the most frequently occurring entries should appear.

Each dictionary has a string name which identifies it to users (e.g., "MP Polynomial Dictionary") and a tag number which identifies it to other processes. When sending a dictionary entry, the triple <type, dictionary tag, value id> is enough to uniquely identify the value in a node packet. The type gives the data type (Operator or Symbolic Constant), the dictionary tag is a small integer, and the value id is either a small integer or a string. The tuple <value id, dictionary tag> does the same for annotations.

An implementation of a dictionary with respect to a particular system may simply consist of a table lookup mechanism, mapping between native and dictionary representations, or it may be more complex and include routines for sending and receiving structured data, with accompanying utility routines.

### 2.2 Annotations and prototypes

The sender may attach an *annotation* to any node packet of an MP tree. Annotations are sent in *annotation packets*. MP defines a set of annotations and users may define their own, placing the definitions in a dictionary. Annotations are modified by a set of flags, which, for example, allow the sender to identify an annotation as *required* or *supplemental*. An annotation with the Required flag set indicates that the annotation carries information essential for properly decoding or interpreting the node's data. Supplemental annotations, on the other hand, are just that and may be discarded without fear by the receiver. Another flag indicates whether or not the annotation is *valuated* (takes an argument). The argument is always given as an MP tree.

The prototype annotation, defined within MP, is especially important. In some circumstances we can minimize the overhead incurred by the node packet header. Occasionally, a block of data will be sent that is characterized by having a homogeneous format. A vector of floating point numbers is a good example, but the data could be more structured - a vector of complex or rational numbers, or a polynomial or ideal.

As an optimization, we can take advantage of this pattern by using a `prototype` annotation whose value is an MP tree specifying the structure and type of the data to be transmitted. Individual nodes of this tree are either an operator (for specifying structured data) or one of the MP types `Meta` or `CommonMeta`, which specify the type of data to be found in the data stream. Subsequently, only the data values corresponding to meta entries in the prototype are placed in the data stream. The entire collection of *data* items is placed in a single *data packet*. The receiving side would retrieve the prototype and use it to properly read the (headerless) data from the stream. Figure 1 illustrates a very simple example. For a vector of 1,000 fixed precision integers, the data requires 4,000 bytes and an additional 4,000 bytes for the overhead of the node packet headers. Using the prototype to specify the element type of the vector reduces the total size of the vector's encoding from 8,008 bytes to 4,016 bytes (only 16 bytes of overhead). This may also speedup processing and simplify parsing by allowing blocks of data to be treated as a single data item (and moved efficiently using *memcpy()*), as opposed to treating the elements individually.

It is important to point out that operators appearing within a prototype may have a fixed or a variable number of arguments, providing additional flexibility. If the number of arguments is known at *prototype specification time* (the time at which we create and send the *prototype*), then it can be fixed within the prototype. However, if the number of arguments is not known, or could vary (as in a list of uneven lists), then the number of arguments for the operator in the prototype is not given until *data communication time* (the time at which we actually send the *data*).

A second advantage of prototypes is that supplemental information can be supplied via annotations attached to the nodes *of the prototype*. There is no need, then, to attach those annotations to the actual data, saving both space and processing time. We make extensive use of this mechanism in the transmission of polynomials.

| Type | #Annots | Value | #Args | Remarks |
|---|---|---|---|---|
| COP | 1 | list | 1000 | CommonOperatorPkt |
| AP | | ProtoType | | Prototype Annotation |
| CMP | 0 | IMP_Sint32 | | MP type specification |
| | | 1 | | Beginning of data pkt |
| | | 2 | | |
| | | ... | | |

Figure 1: Encoding of a vector of 1000 fixnums

## 3 A Commented Outline of the MPP dictionary

Here we provide a commented outline of the MPP dictionary and a brief discussion of some of the design decisions. For more complete technical details, the reader is referred to [2].

The goal of this dictionary is to provide a framework for doing distributed polynomial computations using MP. The MPP dictionary introduces operators and annotations, defines their meaning, and explains their syntax.

In describing the operators and annotations defined by the MPP dictionary, we distinguish between (i) the encoding of polynomials, (ii) the encoding of polynomial structures (like ideals, modules, etc.), and (iii) the encoding of polynomial functions (like polynomial arithmetic, factorization of polynomials, Gröbner basis computations of ideals, etc.).

### 3.1 The encoding of polynomials in MPP

Most of the polynomial encodings used in CASs can be identified by one of the following representations:

| representation | applications |
|---|---|
| dense-distributive | Gröbner basis systems: e.g. SINGULAR, PoSSo |
| sparse-distributive | Macaulay 2 front-end ↔ compute-engine communications |
| dense-recursive | earlier version of FACTORY |
| sparse-recursive | polynomial factorization: e.g. Macsyma, FACTORY |
| expression tree | Maple's DAG, Mathematica |

Figures 2 to 5 illustrate the first four of these representations for the polynomial $p(x_3, x_2, x_1) = x_3(x_2(5x_1^2+4)-3) = 5x_3x_2x_1^2 + 4x_3x_2 - 3x_3$.

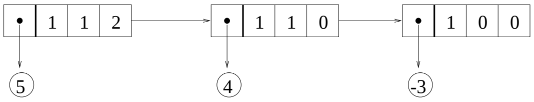


Figure 2: Dense-distributive representation of $p$

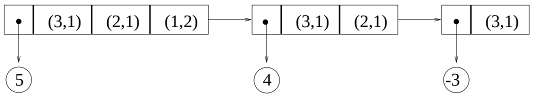


Figure 3: Sparse-distributive representation of $p$

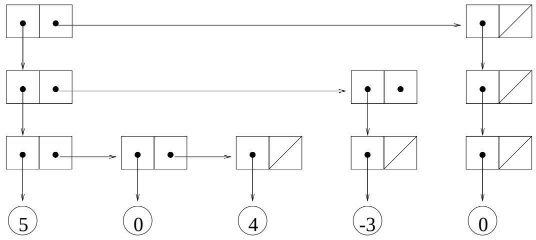


Figure 4: Dense-recursive representation of $p$

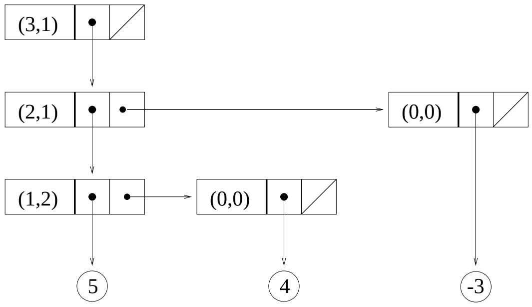


Figure 5: Sparse-recursive representation of $p$

To reach the goal of *connectivity* it would be sufficient to pick *one* of these representations as the standard within MPP. This would require a system to transform polynomials between the standard MPP encoding and its internal encoding each time a polynomial is sent and received. Efficiency-wise, this is certainly acceptable for applications performing more complicated computations on the polynomials (e.g. factoring, Gröbner basis computations, even polynomial arithmetic), since the time and space complexity of transformations between the different polynomial representations is usually dwarfed by the time and space complexity of the actual computations. However, for interprocess communication between homogeneous systems (like those involved in parallel distributed computations), the cost of additional conversions might not only be unacceptable, but also unnecessary. Consequently, to reach our goals of connectivity *and* efficiency, we support five MPP encodings for polynomials based on the dense-distributive, sparse-distributive, dense-recursive, sparse-recursive, and expression tree representations of polynomials.

This approach *minimizes* polynomial transformations: a system can always send polynomials using the MPP encoding which corresponds to its internal encoding. Consequently, the generation of the corresponding MPP encoding incurs little or no additional transformation cost. On the receiving side, a system first checks on the representation of the incoming polynomials. If they are in the representation which corresponds to the system's internal encoding, they can be read in and converted to the system's internal encoding with little or no additional transformation costs. Otherwise, additional polynomial transformations need to be performed prior to the conversion. At first glance, this would seem to require that each system needs to transform polynomials from each of the MPP encodings into its particular internal encoding, and that this makes the implementation of MPP interfaces more complicated. However, a system only needs to implement *one* interface to the MPP encoding which corresponds to its internal encoding: transformations into this encoding are accomplished by routines provided in the MPP library (see section 4.1).

Besides efficiency, there is an additional argument for having multiple representations for polynomials: polynomial transformations might also lead to the loss of certain properties of, or information about, the polynomials. Consider, for example, transformations from a distributive into a recursive representation where the monomials of the distributive polynomial are in a known ordering. Obviously, this ordering property gets lost after transforming the polynomial into a recursive representation.

### 3.2 MPP-polynomials

More formally, we define an MPP-polynomial as follows:

<MPP **Poly**> :: <MPP Number> | <MPP PrototypedPoly> | <MPP TreePoly>

#### 3.2.1 MPP-numbers

MPP-numbers form the leaves of MPP-polynomials. If not further specified by a prototype annotation, MPP-numbers are sent as node packets (i.e. with headers). More precisely, we define

<MPP **Number**> :: <MPP Integer> | <MPP Rational> | <MPP Float> | <`MP_Constant`>

<MPP **Integer**> :: <`MP_Uint32`> | <`MP_Sint32`> | <`MP_ApInt`>

<MPP **Rational**> :: <MPP Integer> | <Cop(`Div`)><MPP Integer> <MPP Integer>

<MPP **Float**> :: <`MP_Real32`> | <`MP_Real64`> | <`MP_ApReal`>

where <Cop(`Div`)> denotes a common operator packet with the (common) operator `Div`, and the prefix `MP_` denotes MP node packets for the respective types.

The different MPP-number types are defined as values for common meta types in the MPP dictionary. We may then refer to these subsets of node packet types within prototype annotations.

#### 3.2.2 MPP-prototyped polynomials

We refer to the encoding of dense-distributive, dense-recursive, sparse-distributive, and sparse-recursive polynomial representations as MPP-prototyped polynomials because they each have a prototype annotation specifying the syntax (and part of the semantics) of the polynomial data. We identify these encodings through the common operators `MPP_DenseDistPoly`, `MPP_DenseRecPoly`, `MPP_SparseDistPoly`, `MPP_SparseRecPoly`.

The semantics of the polynomials is only well-defined if the prototype specifications are of a certain structure. For example, for the dense-distributive encoding, it must be specified (at prototype specification time) that each argument of the `MPP_DenseDistPoly` operator is a list of two elements (i.e. a monomial), where the first element is a type specification (giving the type of the coefficients) and the second element is a fixed-length list of, say, $n$ `IMP_Sint32`s (i.e. there are $n$ variables whose exponents are communicated as a vector of signed 32-bit integers). The prototypes for the other polynomial encodings can be described similarly as can be seen from figures 2 - 5.

It should be noticed that these prototypes completely define the structure of the polynomial data in such a way that no further *node* packets are used in order to communicate the value of the exponents (i.e. all exponents and, if necessary, variable numbers and list lengths are sent without packet headers). This is the key for efficient polynomial communications, since exponent values typically form the majority of the polynomial data.

An obvious alternative to using prototypes is to define the polynomial representations statically (i.e. off-line), as done by PossoXDR, and require that applications supply compiled routines capable of reading and writing these fixed representations. This has the advantage of being quite fast, but at the cost of generality. The advantage of always including a prototype that provides, among other things, syntactic information, is that a system implementing MP does not need to know anything about the specifications in the MPP dictionary and can nevertheless at least parse, display, manipulate, and resend the polynomial data[1]. Furthermore, as the timings in section 5 illustrate, prototypes do not degrade performance. The cost of transmitting the prototype is negligible and with the support of specialized routines which provide high-efficiency parsing for certain representations (like those provided in the MPP library), the reading and writing of polynomials described by the prototype is *as efficient* as a compiled routine for a fixed representation. Prototypes only incur a performance penalty in the case where an application knows nothing of the dictionaries involved and must rely solely on the prototype to correctly read in the data - essentially behaving like an interpreter. In contrast, an application that receives, but does not understand, a statically defined object has no hope of properly reading in the data.

With the prototype specifications at hand, it is straightforward to define the actual interpretation of the polynomial data as is evident from figures 2 to 5.

One component of each encoding's prototype is a specification of the type of the coefficients, which can be prototyped to be a

**MPP-prototyped polynomial** Using this mechanism, we can easily and conveniently build polynomial extensions (or polynomial "towers" as they are called by PoSSo).

**MPP-number** As explained above, MPP-numbers are prototyped using the respective meta types and are sent as *node* packets (i.e. with headers). This is useful for coefficient domains where the elements may have different encodings, depending on their particular value (e.g. rational numbers).

**basic number type specification** This specifies that the type of the coefficients is one of the MP basic number types and the coefficients are sent as *data* packets (i.e. without headers). This is useful for coefficient domains where all elements have the same encoding (like $Z/p$ for small $p$). Notice that in this case the prototype completely specifies the syntax of the polynomial data, allowing it to be sent as a single data packet.

Besides being a syntax specification, a coefficient prototype specification also defines the mathematical coefficient domain of a polynomial.

#### 3.2.3 MPP-expression tree polynomials

As one would expect, an MPP-expression tree polynomial (or tree polynomial) is an MP tree representation for a polynomial constructed from MPP-identifiers, MPP-numbers, and the MP common operators `Add`, `Mult`, and `Pwr`. In order to develop conversion routines from the tree representation into one of the prototyped representations which does *not* require (polynomial) arithmetic operations, it is necessary to further specify the syntax of MPP-tree polynomials. In other words, the syntactic specification of naive polynomials given here describes the (sub) trees of an MP tree which

[1] Our thanks to Dan Grayson for stressing the importance of this point to us.

can be converted into a prototyped polynomial specification by purely syntactic means.

Hence, we define

<MPP **TreePoly**> :: <Term> | <MPP TreePoly> + <Term> | <MPP TreePoly> * <PP>

<**PP**> :: <MPP-identifier> | <MPP-identifier> ^ <MP_Sint32> | <PP> * <PP>

<**Term**> :: <MPP-number> | <PP> | <MPP-number> * <PP>

where <MPP-identifier> is either an `MP_Identifier` or an MP tree specifying an indexed identifier, and the symbols "+, ^, *" are shorthands for the respective MP common operators.

For example, the tree corresponding to `x*(2*y+4)` would be an MPP-tree polynomial, whereas the tree corresponding to `2*x*(y+2)` would be considered as the product of two MPP-tree polynomials (namely, `2*x` and `y+2`).

### 3.3 Polynomial annotations

We refer to annotations attached to MPP-polynomials (or MPP-polynomial type specifications) as polynomial annotations. Within the MPP dictionary, we currently have the following polynomial annotations defined:

| Annotation type | Used to specify |
|---|---|
| `MPP_VarNumAnnot` | number of variables |
| `MPP_VarNamesAnnot` | names of variables |
| `MPP_OrderingAnnot` | monomial orderings |
| `MPP_WeightsAnnot` | weights for orderings |
| `MPP_DefRelsAnnot` | polynomial relations |
| `MPP_IrreducibleAnnot` | irreducibility |
| `MPP_ModuleVectorAnnot` | module vectors |

Polynomial annotations provide additional semantic information. They can be used, for example, to specify properties of the ring over which the polynomials are defined. All polynomial annotations are valuated (except the irreducibility annotation) and supplemental (except the module vector annotation).

#### 3.3.1 VarNumber and VarNames annotations

The argument of the `MPP_VarNumAnnot` is an `MP_Uint32` specifying the number of variables of a polynomial. Although this information is implicitly encoded in the prototype specification or in the polynomial data, it is often advantageous to have that information available before starting to read the data.

The argument of the `MPP_VarNamesAnnot` is either an MPP-identifier or a list of MPP-identifiers. If a polynomial has $n$ variables, say $x_1, ..., x_n$, and $m$ variable names, say $v_1, ..., v_m$ were supplied, then the variable $x_{n-i}$ is assigned the name $v_{m-i}$. Notice that $m$ might be different from $n$ (in fact, it is perfectly valid to specify no variable names at all) and that the variable names are assigned to the actual variables starting at the "end" (for reasons apparent from section 3.3.5).

#### 3.3.2 ordering annotation

This annotation is meaningful only to MPP-distributive polynomial operators and specifies the ordering of the monomials. The meaning of the polynomial data is undefined should the monomials not be in the specified order. We refer to the argument of the `MPP_OrderingAnnot` as an MPP-ordering.

Monomials form a semi-group w.r.t. multiplication. Robbiano ([14]) proved that any semi-group ordering can be defined by a real valued matrix. Hence, from a general mathematical point of view, it would be sufficient to restrict MPP-orderings to matrices. However, not only do most systems to date not support matrix orderings, it is also customary and convenient to refer to commonly used orderings by more compact means.

Hence, we propose an encoding of MPP-orderings which compactly encodes a wide variety of commonly used monomial orderings, and is rich and general enough to express any monomial ordering. For this purpose, we define an MPP-ordering to be either a *simple complete* MPP-ordering or an MPP-product ordering. We further define a *simple* MPP-ordering to be a simple complete MPP-ordering or an *incomplete* MPP-ordering.

A simple MPP-ordering is encoded by an `MP_Uint32` which can itself have an additional valuated annotation of the type `OrderingWeights` whose argument may be either a list or a matrix of `MP_Sint32`s. We currently have the following simple MPP-orderings defined:

| ordering | annotation |
|---|---|
| simple complete orderings | |
| lexicographical | none |
| negative lexicographical | none |
| degree lexicographical | weights (opt) |
| negative degree lexicographical | weights (opt) |
| degree reverse lexicographical | weights (opt) |
| negative degree reverse lexicographical | weights (opt) |
| matrix ordering | matrix |
| incomplete orderings | |
| weight vector | weights |
| increasing module components | none |
| decreasing module components | none |

An MPP-product ordering is encoded as a list whose elements are triples <simple MPP-ordering, $n, m$ >, where $n$ and $m$ are encoded as `MP_Uint32`s and denote the subsets of variables covered by this component of the ordering. There is also the requirement that there exist a sequence of complete MPP-orderings which covers all variables of the polynomial.

This encoding of monomial orderings is very general and, for example, is able to encode the monomial orderings of SINGULAR, Macaulay [3], GB [7], and PoSSo [1].

Since this short treatise on monomial orderings is probably folklore for the Gröbner basis folks, but may be too brief for others, we refer both parties to [2] and [12] for further details.

#### 3.3.3 irreducible annotation

This annotation simply serves as a flag indicating the "irreducibility" of some object. Its precise meaning depends on the type of the object to which it is attached. If given to polynomials, it obviously means that the polynomial is irreducible. If given to an MPP-integer it specifies that the integer is prime. If given to an MPP-rational, it means that the rational number is normalized (i.e. the numerator and denominator have no common divisor). When this annotation is attached to some data which is "reducible" (i.e. does not have the claimed property), the meaning of the data is undefined.

#### 3.3.4 defining relations annotation

The MPP_DefRelsAnnot annotation specifies mathematical equivalence relations. Again, its precise meaning depends on the type of the object to which it is attached. Its argument may be an MPP-integer, a univariate polynomial, or an ideal.

If this annotation has an MPP-integer as its argument, say $p$, and is attached to an MPP-integer (or an MPP-integer type specification), then the integer(s) are considered to be from the mathematical domain $Z/p$.

If this annotation has a univariate irreducible polynomial as its argument, say $f(x)$, and is attached to a polynomial (or a polynomial type specification), then the polynomial(s) are considered to be polynomials from a ring over the algebraic extension with $f(x)$ as its minimal polynomial.

Similarly, if this annotation has an ideal as argument, say $I$, and is attached to a polynomial (or, a polynomial type specification), then the polynomial(s) are considered to represent equivalence class(es) from the residue class ring modulo $I$.

#### 3.3.5 vector module annotation

Polynomial vectors form a free module over a polynomial ring which is generated by canonical base vectors $e_1$ to $e_n$. Hence, an element of this module is a sum of products of a monomial $m_i$ with an $e_i$ and can therefore be considered as a list of pairs $(i, m_i)$ and encoded as "polynomial" with an additional variable which has a special meaning: its first exponent $i$ is considered to be the index of the base vector $e_i$. Furthermore, monomial orderings can be generalized to include this special variable (see the module component orderings).

Hence, if the MPP_ModuleVectorAnnot is attached to an MPP-polynomial, then this "polynomial" encodes an element of a free module where the exponent $i$ of the *first* variable encodes the base vector $e_i$. Since this fundamentally changes the semantics of the data, the "understanding" of this annotation is essential and the annotation Required flag is set.

### 3.4 Encoding of polynomial structures

Experience with systems in the fields of algebra and algebraic geometry (like Macaulay) show that polynomial matrices can be used to express most of the mathematical objects occurring in those fields (e.g. ideals by their generator's (1 x n) matrix, modules by their presentation matrix, etc.). However, there are also stronger-typed systems which do not take such a general approach to polynomial structures, but distinguish between, say ideals, modules, and matrices (for example, Macaulay 2 [11], CoCoA [5], SINGULAR).

In order to support stronger typed systems as well, we follow their approach and provide mechanisms to distinguish polynomial structures by their encoding: different MPP-operators are defined which have MPP-polynomials as arguments and, from an encoding perspective, simply group MPP-polynomials together. However, there is an important semantic requirement for MPP-polynomial structures which distinguishes them from other structures: there must exist a polynomial ring such that all the polynomial arguments of an MPP-polynomial structure are elements of this ring. Otherwise, the semantics of the data are undefined. For example, a list may consist of polynomials from $Z/5[x]$ and $Q[x]$, whereas an ideal may not.

Currently, we have the following operators for polynomial structures defined:

MPP_Vector: The polynomial arguments form a vector over a free module (for another encoding see section 3.3.5).

MPP_Ideal: The polynomial arguments are the generators of an ideal.

MPP_Matrix The number of polynomial arguments of this operator is the total number of elements of the matrix. For row-major (resp. column-major) matrices, the row (column) dimension is given by the valuated annotation NumberOfRows (resp. NumberOfColumns).

MPP_Module The polynomial vector arguments of this operator are the generators of a sub-module of a free module.

It should be noticed that the homogeneity of the polynomial arguments of those structures can be used to efficiently encode them by attaching the appropriate prototype annotation to the structuring operator. This prototype specification then consists of the polynomial operator (such as MPP_DenseDistPoly) together with its annotations. Hence, the headers of the argument MPP-polynomials together with their annotations only need to be communicated *once* with the operator of the polynomial structure, instead of with each polynomial.

Properties of polynomial structures can be communicated using well-defined MPP-annotations. For example:

| annotation | attached to | value | explanation |
|---|---|---|---|
| MPP_IsPrime | MPP_Ideal | none | is a prime ideal |
| MPP_Dim | MPP_Ideal | MP_Uint32 | (Krull-)dimension |
| MPP_IsGB | MPP_Ideal MPP_Module | MPP-ordering | generators form a Gröbner basis w.r.t. ordering |

### 3.5 Polynomial functions

Polynomial functions are encoded using MPP-operators. The mappings between the mathematical functions and MPP-operators is defined by the entries in the operator part of the MPP dictionary. Each entry includes a description of the function, as well as a specification of the number and types of valid arguments.

For example, the operator MPP_Factor represents the factorization of a polynomial and takes one MPP-polynomial as its argument, while MPP_GB represents the standard basis computation and takes two arguments, an MPP-ideal or an MPP-module, and an MPP-ordering.

## 4 MPP Interfaces

### 4.1 The MPP library

The MPP library aids the implementation of interfaces to the MPP dictionary. It includes a general parser for MP trees which reads MP data from an MP link and builds the respective MP trees in memory. This has the advantage that an application can then "walk" through the tree to extract data in any order it wishes and can manipulate the tree before or while parsing it.

The MPP library has special data structures for the different encodings of MPP-polynomials. Based on these data structures, MPP-polynomials can be efficiently and conveniently read in, stored, and manipulated. Routines are provided for converting between the different encodings.

As a result, the library may be used by an application as a "black box", which reads in the data from the MP link and provides MPP-polynomials in the requested encoding, transparently performing any necessary transformations.

### 4.2 Implemented MPP interfaces

The main algorithms in SINGULAR are centered on Gröbner and standard bases computations. SINGULAR uses a dense distributive polynomial representation and is very flexible with regard to monomial orderings. Each polynomial object within SINGULAR belongs to a certain base ring which determines how the object is handled. SINGULAR, which was originally designed as an interactive system, can now also act as a compute server through its MPP interface.

SINGULAR always sends polynomials using the dense distributive encoding and the corresponding ring properties are sent as polynomial annotations (variable names, monomial ordering, coefficient domain, etc.).

Reading MPP dense-distributive polynomials (or polynomial structures containing only polynomials in this representation) is done directly from the MP link, where the annotations and their values are used to constitute a new ring (if they are not compatible with the current global ring). Missing information is recomputed or set to default values. Other MP data is read from the link into the memory representation of MP trees using routines from the MPP library. After the transformation of all polynomial subtrees to the dense-distributive encoding, the tree is converted to SINGULAR data structures.

FACTORY is a general C++ class library of polynomial algorithms for factorization, greatest common divisor, subresultants, etc. It uses a sparse-recursive representation of polynomials internally and naturally uses this representation to send polynomials.

Receiving is done via the utility routines of the MPP library which are used to build an MP tree in memory and transform all polynomials into the MPP sparse-recursive representation. FACTORY does not have the need for a fully specified ring in order to handle polynomial data (e.g. variable names and other properties might be missing) which eased the implementation of the interface. Overall, the design of FACTORY as a general library provided very convenient tools to implement its MPP interface. We expect that this will also apply to implementations of MPP interfaces to other general polynomial libraries (such as Saclib).

Mathematica's data is mainly expression trees which can be sent and received via MathLink (see section 6 for a discussion of MathLink). We used MathLink to send data from Mathematica to a small C program, `mlmp`, which served as a MathLink-MP converter.

The conversion of a MathLink expression tree to an MP tree is done in a straightforward fashion by recursing the tree: each time a MathLink token is received by `mlmp`, a corresponding MP node packet is written to the MP link. Since Mathematica communicates polynomials as general expression trees over MathLink, `mlmp` simply passes them on as such through MP. In other words, polynomials are sent in the MPP-expression tree encoding.

On the receiving side, `mlmp` uses the MPP library simply as a "black box" which reads the data from the MP link and transforms all MPP-prototyped polynomials into the corresponding MPP-expression tree encoding. Afterwards, the data is sent to Mathematica as a corresponding MathLink expression tree using the `mlmp` converter.

The handling of annotations and of the somewhat similar Mathematica concept of "rules" was more complicated. Within MP, a node header provides information about the number of annotations attached to a node and those annotations immediately follow the node header (i.e. before the arguments, if it is an operator), while in Mathematica expressions, rules occur as additional arguments to the operator and can only be detected by preparsing the arguments.

Through its MPP interface, each of these systems can exchange data with any other system implementing the interface providing some of their functionality to each other. This achieved one of our main goals: connectivity.

## 5 Timings

The total time to exchange data between cooperating processes is the sum of the transmission and transformation times. The transmission time depends on the data size and the communication medium. In our context, the transformation time is the time required for MP buffering, the en/decoding of MPP data, and conversions from/to an application's internal data structure. Since this has been the focus of our work, our timings measure transformation costs only, excluding system time devoted to I/O (we used files) and are independent of the communication medium.

The timings in tables 1 and 2 are in seconds and were were taken on an RS/6000 Model 360, running AIX V 3.2.5, with MP 1.1, SINGULAR 0.9.2, FACTORY 1.0, and Mathematica 2.2 for the ideal represented by the 268 elements of the degrevlex Gröbner basis of $(t^3y^7z^4 + 9x^8 + 5t^2x^4y^2 + 2t^2xy^2z^3, 2t^2x^5y^4 + 7txy^6 + 9y^8 + 2t^2x^2yz^3, 3t^4x^2y^3z^2 + 9z^8, 3x^2y^9 + y^9 + 5x^4)$.

| | SINGULAR | FACTORY | MMA |
|---|---|---|---|
| Object copy | 0.70 | 1.93 | - |
| Object size | (3.3) | (9.1) | (9.7) |
| **Sending** | | | |
| ASCII | 8.72 | 12.69 | 71.9 |
| (size) | (1.7) | (1.7) | (1.7) |
| MP "native" | 0.68 | 4.50 | 55.6 |
| (size) | (1.9) | (2.1) | (5.7) |
| **Receiving** | | | |
| ASCII | 70.4 | 70.4 | 115 |
| MP-ExprTree | 40.3 | 40.1 | **89.5** |
| MP-SR | 4.11 | **24.7** | 92.7 |
| MP-DD | **1.13** | 34.6 | 91.9 |

Table 1: Timings for Q[t,x,y,z] (58,962 monomials)

| | SINGULAR | FACTORY | MMA |
|---|---|---|---|
| Object copy | < 0.1 | 0.41 | - |
| Object size | (0.25) | (1.89) | (2.1) |
| **Sending** | | | |
| ASCII | 1.01 | 1.59 | 15.3 |
| (size) | (0.25) | (0.25) | (0.25) |
| MP "native" | 0.11 | 0.80 | 6.77 |
| (size) | (0.29) | (0.43) | (1.2) |
| **Receiving** | | | |
| ASCII | 9.77 | 13.4 | 14.7 |
| MP-ExprTree | 6.38 | 10.0 | **9.31** |
| MP-SR | 0.64 | **3.60** | 10.1 |
| MP-DD | **0.12** | 3.00 | 9.85 |

Table 2: Timings for (Z/32003)[t,x,y,z] (14,782 monomials)

"Object copy" is the time it takes for the system to make an internal copy of the object and can be used as a yardstick against which to compare the other timings.[2] "Object size" is the size (in megabytes) of the internal memory representation of the object. The ASCII format is a straightforward

[2] We were unable to obtain this time for Mathematica.

string representation that includes all the operators. For example, $3x^2 + 1$ would be the string "`3*x^2+1`". The MP "native" format is the MPP-polynomial representation native to the sending application. SINGULAR's native representation is dense-distributive, FACTORY's is sparse-recursive, and Mathematica's is expression tree (recall that the sender always sends using its native representation).

The boldface entries under Receiving give the times to receive polynomials in the system's native representation. In those cases, the data is read in directly from the MP link with no intervening transformations. In the other cases, the data is first read into the MPP data structures, then transformed into the receiver's native MPP encoding before being converted into the system's internal data structures.

As our results illustrate, the best times occur when sending or receiving is done in the application's native encoding. Clearly this is because the transformation overhead is minimized in these cases. Indeed, in the very best case these times are comparable to the "Object copy" times, making our design and implementation appropriate for highly efficient communications like those required by parallel distributed computations. A comparison of the timings for SINGULAR and FACTORY here raises an interesting point. Unlike SINGULAR, FACTORY has no primitive tree manipulation functions for building its data structures (everything is encapsulated by constructors, iterators, etc.) and instead builds them using polynomial arithmetic operations. This is a more expensive method of constructing the internal data structures and accounts for the differences in the ratio between the "Object copy" and receiving times.

When receiving involves transformations between prototyped encodings, the worst-case time complexity of the transformations is $O(n \log n)$, where $n$ is the number of monomials. This is due to the possibility of having to reorder the monomials. However, as the timings verify, the performance penalty is not too great.

When receiving involves transformations between prototyped and expression tree encodings, the worst case time complexity of the transformations is not necessarily greater than that for transformations between prototyped encodings. In practice, however, the additional overhead of the syntactic manipulations lead to a greater cost, as the timings show.

Lastly, the difference between the times for the expression tree and ASCII encodings is mainly attributable to two factors: first, to the cost of string handling operations and, second, to the polynomial arithmetic operations which *must* be used to build the internal data structures.

## 6 Related Work

Currently there are several protocols in use that communicate mathematical expressions.

MathLink [17] transmits expressions as linearized trees. It includes a library of C routines to send and receive terms of the expression. There is no real notion of an annotation built into MathLink, but it would be possible to send additional information via strings embedded within an expression. MathLink has no equivalent to the idea of a dictionary and there is no other support for providing shared semantics, so the interpretation of an expression is left to the receiver. Although MathLink is packaged with Mathematica and has a set of Mathematica-specific routines, it is a general protocol that can be used independently of Mathematica.

ASAP (A Simple ASCII Protocol) [6] is a public domain mathematics protocol that exchanges expressions as linearized, attributed trees. Contrary to its name, binary, fixed precision integers can be transmitted, but most data is transmitted as ASCII (cleartext) strings. Attributes are similar to MP's annotations, but lack flags providing further information about the annotation. Like MathLink, ASAP does not include the idea of a dictionary, relying on the user to define the semantics of the expressions exchanged. Neither MathLink nor ASAP provide specialized encodings for polynomials.

The POlynomial System SOlver (PoSSo) project [8], on the other hand, is primarily devoted to polynomials and includes a protocol, POSSOXDR [1], defining the external representation of the objects (PoSSo data types) manipulated inside PoSSo processes. The encoding is an extension of the eXternal Data Representation (XDR) technology. POSSOXDR does not include annotations, nor a general extension mechanism to support other kinds of mathematical objects. A limited form of a dictionary is available in the sense that it is possible to define new types (described in the Posso Data Description Language) and have them registered with the PoSSo project for general use. However, commands in POSSOXDR are transmitted as data and their meaning is specific to the sender/receiver pair. It was felt that POSSOXDR was inappropriate for our project because it is primarily a collection of data objects in support of transmitting polynomials and is not a general protocol for exchanging mathematical expressions.

These issues are also being considered as part of the OpenMath effort and it is hoped that the lessons learned with MP can contribute to that effort.

## 7 Conclusion and Future Work

The MPP dictionary and library accomplished the goals we set out for them. Perhaps the single greatest barrier to providing connectivity is simplifying the view a software tool has of the "rest of the world". Clearly the more "standards" it must recognize, the larger and more complex the interface must be. Using dictionaries appears to be a realistic way of tackling this problem. It provides a consistent and extensible framework within which heterogeneous systems can exchange data with a shared understanding.

Also, we wanted to maximize efficiency, which is important for high-speed communications, *without* sacrificing the generality that is important to simplify building MPP interfaces to different systems. This goal was mainly accomplished by supporting different polynomial representations and a careful design of their (prototyped) MPP encodings. This resulted in a "view" of polynomials that is general enough to be compatible with most systems and yet efficient enough for high-speed communications between homogeneous systems.

Based on this design, the MPP library eases the implementation of MPP interfaces by providing utility routines for communicating and manipulating MPP-polynomials and other MP data. Supporting multiple polynomial representations complicated the library and the implementation of interfaces, but once the initial library implementation was complete, the creation of additional interfaces was remarkably straightforward. The timings further validate the efficiency aspects of our design.

Still, there is room for improvement and many avenues to explore. We will refine and expand the MPP dictionary and want to initiate an open discussion with a broader audience regarding the dictionary's contents and design. The development of dictionaries in other areas of mathematics is important as well. In general, the full power of the dictionary idea needs to be explored.

The MPP library can be expanded and further optimized. For example, the size of the polynomial expression trees could be greatly reduced by taking advantage of the MP common format and the set of MPP representations should be extended to include straight-line programs in support of "black-box" computations [13].

It is important to extend the number of systems that have an MPP interface: we are especially interested in Saclib, Macsyma, and Maple (but this is problematic for Maple without easy access to its internals).

We are also very interested in exploring what might be called "adaptive communication". Recall that the total time required to exchange data is the sum of the transmission and transformation times. Optimizing for one may have a deleterious effect on the other, and hence on the overall performance. For example, generating an encoding that is more compact may require extra processing time which may completely offset the gains made in transmission time. In the future, we expect it will be possible for the sender to dynamically adapt to its computing environment by choosing an encoding which takes into account the size and nature of the data, the speed of the communication medium, as well as the speed of the processors involved. With respect to exchanging polynomial data using a transmission media at least as fast as a LAN, our experiments have shown that transformation costs usually dominate transmission costs. However, further explorations have to be undertaken to see how best to optimize *overall* performance.

In general, we feel it is important to "think distributively" and to design applications *with connectivity in mind*, remaining aware of the benefits (and difficulties) of the dual goals of generality and efficiency. Projects such as ours are beginning to identify some of the areas where current design paradigms should be rethought.

## 8 Availability

SINGULAR is available in binary format via anonymous ftp from `helios.mathematik.uni-kl.de`. The source for the MP library is available from `ftp.mcs.kent.edu` in /pub/MP. This includes the source and documentation for the MPP library. For more information on SINGULAR and FACTORY see `http://www.mathematik.uni-kl.de/~wwwagag`, and for MP see `http://symbolicnet.mcs.kent.edu/systems/mp.html`.


## Acknowledgments

The authors would like to thank Dan Grayson for his insightful observations on certain aspects of MP and the design of the MPP dictionary, Rüdiger Stobbe for his work on the MPP-FACTORY interface, and Gert-Martin Greuel, Paul Wang, and Norbert Kajler for their comments on earlier drafts of this paper. The authors would also like to thank G.-M. Greuel, G. Pfister, E. Hennig, and R. Sommer for initiating our joint work.


## References


[1] ABBOTT, J., AND TRAVERSO, C. Specification of the POSSO External Data Representation. Tech. rep., Sept. 1995.

[2] BACHMANN, O., AND SCHÖNEMANN, H. A Manual for the MPP Dictionary and the MPP Library. Reports on Computer Algebra 4, Centre for Computer Algebra, Department of Mathematics, University of Kaiserslautern, Jan. 1996.

[3] BAYER, D., AND STILLMAN, M. *Macaulay*: A system for computation in algebraic geometry and commutative algebra, 1992. Available via anonymous ftp from `zariski.harvard.edu`.

[4] BERGSTRA, J., AND KLINT, P. The Discrete Time ToolBus. Technical Report P9502, Programming Research Group, University of Amsterdam, 1995.

[5] CAPANI, A., NIESI, G., AND ROBBIANO, L. CoCoA, 1995. see http://lancelot.dima.unige.it.

[6] DALMAS, S., GAËTANO, M., AND SAUSSE, A. ASAP: a Protocol for Symbolic Computation Systems. Tech. rep., INRIA Technical Report 162, Mar. 1994.

[7] FAUGÈRE, J. *Online documentation of GB*. Available from `http://posso.ibp.fr/Gb.html`.

[8] GONZALEZ-VEGA, L., AND RECIO, T. The PoSSo NEWSLETTER. Available electronically from `posso.dm.unipi.it`, Mar. 1994.

[9] GRAY, S., KAJLER, N., AND WANG, P. S. Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions. *Journal of Symbolic Computing*.

[10] GRAY, S., KAJLER, N., AND WANG, P. S. MP: A Protocol for Efficient Exchange of Mathematical Expressions. In *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'94), Oxford, GB* (July 1994), M. Giesbrecht, Ed., ACM Press, pp. 330–335.

[11] GRAYSON, D., AND STILLMAN, M. *Macaulay 2*, 1996. For further information contact the authors at `dan@math.uiuc.edu` or `mike@math.cornell.edu`.

[12] GREUEL, G.-M., PFISTER, G., AND SCHÖNEMANN, H. *Singular*: A System for Computation in Algebraic Geometry and Singularity Theory. Department of Mathematics, University of Kaiserslautern.

[13] KALTOFEN, E. Greatest Common Divisors of Polynomials Given by Straight-line Programs. *Journal of the ACM 35*, 1 (1986), 231–264.

[14] ROBBIANO, L. Term Orderings on the Polynomial Ring. In *Proceedings of EUROCAL 85, Lecture Notes in Computer Science* **204** (1985), pp. 513–517.

[15] STOBBE, R. FACTORY: a C++ Class Library for Multivariate Polynomial Arithmetic. Reports on Computer Algebra 3, Centre for Computer Algebra, Department of Mathematics, University of Kaiserslautern, Jan. 1996.

[16] SUNDERAM, V. PVM: A Framework for Parallel Distributed Computing. *Concurrency–Practice & Experience 2* (1990), 315–339.

[17] WOLFRAM RESEARCH, INC. MathLink Reference Guide (version 2.2). Mathematica Technical Report, 1993.