Message Passing with Maple

Justin M. J. Wozniak

September 26, 2003

Abstract

In this report we document a software package under development to allow message passing in the MPI model using the computer algebra system Maple. The new software, called `maplle`, consists of two components, a set of Maple functions and a MPI/C driver. The `maplle` system allows the user to easily parallelize Maple algorithms and use message passing functionality familiar to MPI users in a Maple format.

1 Introduction

A number of algorithms in computer algebra are readily parallelizable. Cooperman [1] describes an important set of these problems, those problems exhibiting the weak sequential dependence property. Examples of these problems include Gröbner basis computation and LLL polynomial factorization. Other algorithms that have been implemented in parallel in a symbolic context are univariate and multivariate polynomial factorization [10], polynomial GCD [6], and the solution of linear systems of equations [5].

To facilitate these computations, a software package was designed to allow message passing in the MPI model using the computer algebra system Maple. The new software provides a set of Maple functions that allow for the implementation of parallel computer algorithms for computer algebra applications. This system allows the user to easily parallelize Maple algorithms and use message passing functionality familiar to MPI users in a Maple format.

The existence of such algorithms has justified the development of parallel computer algebra systems. Sugarbush is a parallel Maple tool that was developed in the early 1990's and based on C-Linda. The C-Linda parallel model is a set of workers, or threads, sharing a read-writable tuple space for exchanging data. This functionality is incorporated in Sugarbush [7].

The Distributed Symbolic Computation tool, often called DSC, is a system to distribute code to computers on a heterogenous network in a variety of languages, including Maple, C, and Lisp. Both the code and the input may be sent over the network, and an output file is returned. The system also has a scheduler which will pick an appropriate processor for each subtask [4].

A system called ||Maple|| is based on the parallel model of Strand. It allows the user to write Strand programs that can access the Maple kernel and library functions [9].

Distributed Maple is a Java based parallel Maple tool that relies on Java sockets to perform the communication. This system also uses independent tasks and a shared data space that may be accessed for process to process communication [8].

In the remainder of this paper, we will discuss how `maplle` was designed to be different from these existing systems, and offer our design considerations. In section 3 we will discuss what functionality is available in the `maplle` implementation, and how the architecture providing that functionality is structured. We then show that the system can be used to obtain a beneficial speedup on a well-known polynomial arithmetic problem in section 4.

2 Design Considerations

Given such an array of existing parallel Maple software, we can now describe where `maplle` fits in. Most importantly, `maplle` uses the MPI model for parallelism. This model is based on a set of identical tasks running the same code, differentiated only by their process ID. All communication is handled by message passing, which is a familiar model for many programmers who are experienced with MPI.

In addition, since `maplle` uses MPI, there is no difference to the programmer where the program is running, be it a TCP/IP network or a high performance shared memory machine. This makes `maplle` highly portable. In addition, `maplle` will gain the full benefit of high performance multiprocessor architectures because the network would not need to be used, it would operate as fast as the underlying MPI system.

For experienced Maple programmers to write quality parallel programs, they must be presented with the appropriate functionality in a familiar interface. The functions should handle all Maple data types and should not require a lot of extra coding, and should require no coding in a language other than Maple.

Another consideration was simplifying the function calls. A typical send function call in MPI has six arguments, in `maplle` this is reduced to two. This was intended to lower the learning curve and enable the code to better fit in the Maple style.

The system is designed to be executed from the command line in a way similar to executing any other MPI program:

```
> mpirun -np N maplle prog.mpl
```

This command will execute the script named *prog.mpl* on N processors. We will not discuss this command in detail in the interest of brevity. For further information concerning the operation of an MPI system consult appropriate

MPI documentation [3].

3 Implementation

The `maplle` implementation consists of two components. The Maple side is a set of functions accessible from Maple which allows for the initialization, communication, and shutdown necessary in the computation. The MPI/C side is the compiled program that uses the MPI system to coordinate communication. The two programs communicate with each other over a Unix pipe.

3.1 `maplle` Functions

The list of accessible `maplle` functions is small and under development, but still useful for a wide class of problems. They are modeled after the MPI functions of the same name but have been simplified. They have shorter argument lists, and they intend to accept any Maple data type. These functions must be read into Maple at the start of execution by the command:

```
read(mpi.mpl);
```

The first function the programmer must call is `MPI_Init()`. This calls the MPI function of the same name and prepares the system for use.

The typical program will next call `MPI_Comm_Rank()` and `MPI_Comm_Size()`. These functions call the MPI functions of the same name and allow the program to detect the number of processors present and the ID of the processor at hand.

At this point, the program is ready to send and receive data from other processors. It does this with the `MPI_Send()` and `MPI_Recv()` commands.

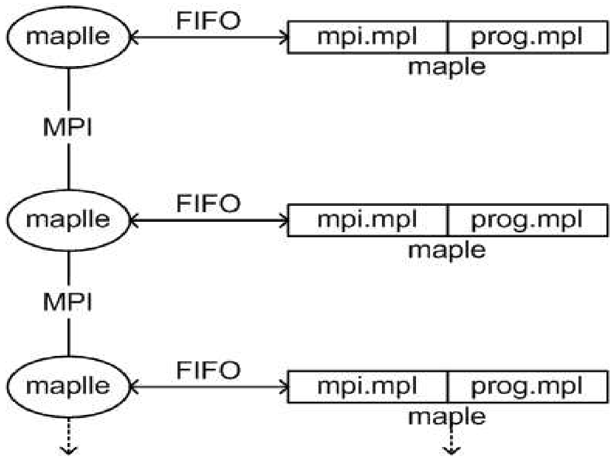
`MPI_Send()` takes two arguments, an integer and a datum. The integer specifies the ID of the processor to whom the data is being sent, and the datum is of Maple type *anything*, which is the information sent.

`MPI_Recv()` takes one argument, an integer which represents the processor from which the datum is to be received. It returns the datum in its appropriate data type.

3.2 `maplle` Internals

From the perspective of the `maplle` system, each function call request is received over the pipe and then processed by making the appropriate call to MPI. Each datum to be sent is sent as an array of type `MPI_CHAR`, encoded in Maple by `convert/string` and decoded by `parse`.

A figure representing the data flow in `maplle` is below.



The figure shows on the right each Maple process, one per node, which is running code from the provided `mpi.mpl` as well as the user program, `prog.mpl` . When the user program makes a call to a communication function in `mpi.mpl`, that function communicates with the corresponding `maplle` process over an automatically created Unix FIFO. The communication then is handed to MPI.

4 Example Application

To demonstrate the usefulness of this system, we perform a well-known polynomial arithmetic computation in parallel. We will then demonstrate the speed up gained from adding processors to the system.

4.1 Fast Polynomial Multiplication

In many computer algebra applications it becomes necessary to multiply two large polynomials together in $\mathbb{Z}[x]$:

$$(a_0 + a_1x + ... + a_{n-1}x^{n-1} + x^n) \times (b_0 + b_1x + ... + b_{n-1}x^{n-1} + x^n) \qquad (1)$$

If both polynomials are of degree n, the computation takes $O(n^2)$ integer multiplications, using the classical algorithm. However, another method exists based on the FFT that runs in $O(n \log n)$ time, and is generally faster than the classical method when $n > 150$ [10]. In this method, each polynomial is evaluated at n points. Each pair of corresponding points is multiplied to get a data set for the product. This data set is interpolated to produce the product in polynomial form.

A summary of this algorithm is below:

1. Evaluate: $O(n^2)$.
2. Multiply: $O(n)$.
3. Interpolate: $O(n^2)$.

However, if we perform the evaluation at Fourier points, the evaluation and interpolation can be computed in $O(n \log n)$ bringing the total run time to $O(n \log n)$ [2].

To compute the FFT in Maple in $\mathbb{Z}$, we use a Fourier transform in a finite field, as documented in [2]. We can find a prime p such that the product is computed in $\mathbb{Z}_p[x]$. To compute the roots of unity use the following algorithm:

- Pick prime $p = O(2m^2)$, with m the size of the maximum coefficient.
- Ensure $n \mid p-1$.
- Factor $p - 1 = q_1 \times q_2 \times ... \times q_k$.
  This isn't too hard.
- Pick a random $\alpha \in \mathbb{Z}_p$.
- If there is no $q_i : a^{\frac{p-1}{q_i}} = 1 \mod p$
  then α is a generator of $\mathbb{Z}_p$.
- About a third ($\frac{3}{\pi^2}$) of the elements in $\mathbb{Z}_p$ are generators.
- Let $\omega = \alpha^{\frac{p-1}{n}} \mod p$, this is a nth root of unity.

Once we have our root of unity, we may perform the FFT in the standard way:

* All computations in $\mathbb{Z}_p$.

1. Get ω, an nth root of unity in $\mathbb{Z}_p$.

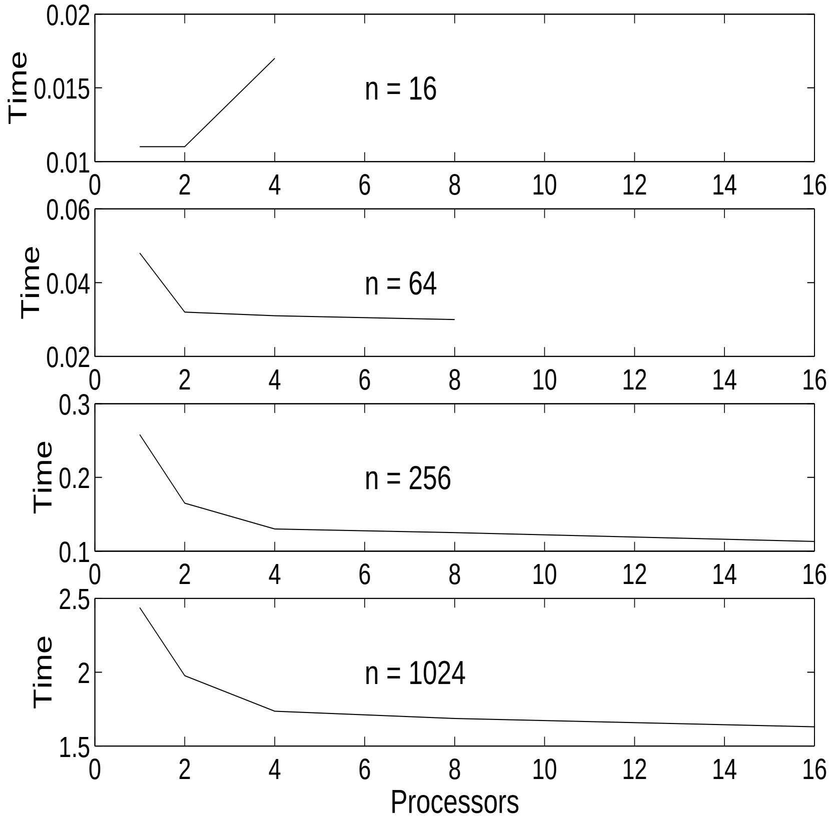
$$\omega^n = 1 \mod p$$

2. Split
3. Recurse
4. Combine

Each recursive call is sent to another processor, so the computation gains considerable speed up.

4.2 Results

The algorithm above was implemented and tested on `flexor`, a 40-processor SGI Origin 3000 at the University of Waterloo. Speedup plots are below:



These plots show the average computation time in seconds of multiplying integer polynomials of degree n. Time is plotted against the processor count to show that increasing the processor count improves the computation speed, but when many processors are added, the system overhead begins to dominate the computation time.

5 Conclusion

In this report, we have shown why `maplle` is a useful tool and how it compares to other parallel computer algebra systems. The architecture and programmers' interface were discussed. We then applied the system to a problem in computer algebra, with a noticeable speed benefit.

More work remains to be done on this project. More of the MPI communication functions remain to be added. Also, different methods of interprocess communication should be explored.

References

[1] Gene Cooperman. STAR/MPI: Binding a parallel library to interactive symbolic algebra systems. In *ISSAC*, 1995.

[2] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, 1992.

[3] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI - 2nd Edition.* MIT Press, 1999.

[4] E. Kaltofen, A. Diaz, and K. C. Chan. A distributed approach to problem solving in Maple. In *Proc. Summer Maple Workshop and Symp.*, 1994.

[5] Erich Kaltofen and Victor Pan. Processor efficient parallel solution of linear systems over an abstract field. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures.* ACM Press, 1991.

[6] Victor Y. Pan. A new approach to parallel computation of polynomial GCD and to related parallel computations over fields and integer rings. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms.* ACM Press, 1996.

[7] Liyuan Qiao. Performance visualization tools for parallelizing computer algebra algorithms. Master's thesis, University of Waterloo, 1995.

[8] RISC-Linz. Distributed Maple - User and Reference Manual, 2001.

[9] Kurt Siegl. Parallelizing algorithms for symbolic computation using Maple. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[10] Paul S. Wang. Parallel polynomial operations on SMPs: an overview. *Journal of Symbolic Computation*, 21, 1996.