Multiprocessed Parallelism Support in ALDOR on SMPs and Multicores

Marc Moreno Maza Ben Stephenson Stephen M. Watt Yuzhen Xie
Ontario Research Cenre for Computer Algebra
The University *of* Western Ontario
London, Ontario, Canada
{moreno,bdstephe,watt,yxie}@orcca.on.ca

ABSTRACT

We report on a high-level categorical parallel framework, written in the ALDOR language, to support high-performance computer algebra on symmetric multi-processors and multi-core processors. This framework provides functions for dynamic process management and data communication and synchronization via shared memory segments. A simple interface for user-level scheduling is also provided. Packages are developed for serializing and de-serializing high-level ALDOR objects, such as sparse multivariate polynomials, into arrays of machine integers for data transfer. Our benchmark performance results show this framework is practically efficient for coarse-grained parallel symbolic computations.

Categories and Subject Descriptors

G.4 [**Mathematical Computing**]: Mathematical Software— *Parallel and vector implementation*; I.1.3 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation— *Languages and systems*

General Terms

Design, Experimentation, Performance

Keywords

Aldor, Categorical parallelism, Dynamic process management, Multiprocessor parallelism, Shared memory.

1. INTRODUCTION

Throughout the 1980s and 1990s the subject of parallel computer algebra was an active area of research. Many researchers have contributed to this area, both through the invention of parallel algorithms and the development and implementation of parallel systems. An excellent overview of these developments is provided in the *Computer Algebra Handbook* [16]. Over the past decade the area has received less intense attention, but recent developments in widely available computer hardware make the subject now more relevant than ever: Current hardware improvements have focused on increasing the number of computations that can be performed in parallel rather than on increasing clock speed alone. This change in focus has brought multi-core workstations to the desktop, expanding interest in parallel algorithms and revitalizing research in parallel computer algebra.

In this paper we describe a high-level categorical parallel framework in ALDOR [2, 28] that supports high-performance computer algebra. This framework provides multiprocess parallelism support in ALDOR on symmetric multiprocessor (SMP) and multi-core architectures. Our work complements previous work in the area. Gautier and Mannhart previously developed a system known as $\prod^{IT}$ [14, 21], which utilized MPI features in ALDOR to provide computer algebra on *distributed* parallel architectures. Their approach was to develop general facilities and demonstrate their use with sample computer algebra problems. We have come at the problem from the opposite direction: Our design has been motivated by a particular family of challenging problems in the computation of triangular decompositions, and we have tried to generalize for broader applications. From a different perspective, Ashby, Kennedy and O'Boyle have used ALDOR for data-parallel QCD computations [4].

We have chosen ALDOR as our implementation language because it provides both facilities to support high-level mathematical abstraction as well as efficient access to low-level machine control. ALDOR is an extension of the AXIOM computer algebra system which focuses on interoperability with other languages and high-performance computing. The designers of ALDOR and AXIOM overcame many of the challenges associated with providing an environment for implementing the extremely rich relationships that exists among mathematical structures. This was accomplished by providing high-level categorical support which allows for the development of generic algorithms for solving computer algebra problems. Some of the features currently provided by ALDOR are being introduced into Fortress, a language which is presently being developed by Sun. Fortress is specifically designed for both high performance computing and high programmability [10].

Our framework uses ALDOR's low-level access to the machine to make effective use of shared memory multiprocessor and multi-core architectures, providing simple yet powerful supercomputing support for computer algebra. We support the communication of high-level objects between processes without requiring knowledge of low level details, allowing researchers to concentrate on implementing mathematical algorithms.

PASCO'07, July 27–28, 2007, London, Ontario, Canada.

The remainder of this paper describes our framework in detail. It is organized in the following manner. Section 2 introduces our parallel computation framework. Section 3 describes how data is communicated efficiently between the processes and concurrency control. This is followed by a discussion of our serialization tools for high-level ALDOR objects in Section 4. Dynamic process management and user-level scheduling techniques is discussed in Section 5. Benchmark performance results are presented in Section 6. We present our conclusions and future work in Section 7.

2. OVERVIEW OF THE PARALLEL FRAMEWORK

Our goal is to develop a high level, categorical, parallel framework for high-performance computer algebra that effectively exploits the parallel features of modern multi-core and multiprocessor computer architectures. In order to accomplish this goal, we have introduced multiple process parallelism in ALDOR by providing a mechanism for spawning an arbitrary number of new processes dynamically at runtime (within the limits imposed by the operating system). When multiple processors or cores are available, these processes will execute in parallel. Furthermore, we have also implemented the mechanisms necessary to coordinate the execution of these processes and communicate data between them efficiently. A high level description of our framework is presented in the remainder of this section. The technical details of each component are presented in the sections that follow.

The ALDOR programming language is a type-complete, strongly-typed, imperative programming language. It uses a two-level object model consisting of *categories* and *domains*. In many respects these concepts are analogous to *interfaces* and *classes* in Java. ALDOR provides a type system that provides the programmer with the flexibility to build new types by creating new categories and domains, as well as the flexibility to extend existing categories and domains. For example, new categories and domains can be implemented to model algebraic structures (e.g. rings) and their members (e.g. polynomial domains). Pervasive use of dependent types allows static checking of dynamic objects and provides object-oriented features such as parametric polymorphism.

FRISCO (A Framework for Integrated Symbolic/Numeric Computation) was a project funded by the European Commission under the Esprit Reactive LTR Scheme from 1996 to 1999. It resulted in the creation of a library, `BasicMath`, for polynomial arithmetic and a sequential polynomial solver, `triade`, developed in ALDOR. Even today, `triade` out performs three solvers in Maple: `Triangularize`, `RegSer` and `SimSer` [6]. Many of the categories, domains and packages of this sequential implementation (such as polynomial arithmetic, polynomial greatest common divisor and resultant over an arbitrary ring) can be reused or extended for general purpose parallel symbolic computations.

ALDOR source code can be compiled into a variety of formats. These include native operating system executables; native operating system object files that can be linked with each other, or with C or Fortran code to form other applications; portable bytecode libraries; and C or Lisp source code [27]. Aggressive code optimizations produce code that performs comparably to hand-optimized C [26]. This makes it possible to build executables formed from source files written in several languages. Furthermore, by compiling ALDOR code to an object file, it can be linked into many different executables that will be run as independent parallel processes. ALDOR also provides a primitive, `run(...)`, for initiating a new program `P` from within a program `Q`. The `run(...)` primitive is a wrapper for the C *exec(...)* function which launches the target application as a separate process. These features give us the basic functionality necessary for dynamic process management.

Because our parallel workloads will execute as separate processes, it is necessary to establish a mechanism for interprocess communication in ALDOR. We rely on *shared memory segments* from the standard set of UNIX System V interprocess communication tools. A shared memory segment is a block of memory that can be accessed by several processes. Shared memory segments are provided by most flavors of UNIX, providing us with portability across many platforms. The number of shared memory segments available on a system is normally confined to a small value. However, this value can usually be increased. Shared memory segments are commonly regarded as an effective way to communicate large amounts of data efficiently. The shared memory segments are accessed in ALDOR through a newly developed ALDOR domain called `SharedMemorySegment`. Our domain uses interoperability with the C programming language to provide the required functionality.

Our implementation of shared memory segments in ALDOR allows an array of integers to be communicated between processes. High-level ALDOR types can be communicated by converting them into an array of integers and then copying the integers into a shared memory segment. Similarly, the process receiving the data constructs a new instance of the ALDOR high-level data type from the provided array of integers. This serialize/unserialize process is necessary because the data types that represent sparse multivariate polynomials and dense multivariate polynomials are implemented with pointers. Consequently, it is not possible to copy instances of these ALDOR domains directly from one process to another because the pointers will not necessarily be valid in the destination process.

An additional shared memory segment is used to ensure that the serialized data is communicated between processes in a synchronized manner. We will refer to the second shared memory segment, which is a single integer value, as the *tag*. A protocol was developed requiring the destination process to check the tag value before accessing the polynomial data in the data shared memory segment. Our protocol exploits the fact that writing a single integer value to a shared memory segment (or reading a single integer value from a shared memory segment) is an atomic operation when the integer is word aligned. The specification for `shmat()`, the function used to attach to a shared memory segment, can be asked to return a pointer that is rounded down to the nearest multiple of the segment low boundary address multiple [1], giving an address that is guaranteed to be page aligned (and as such, will also be word aligned). Consequently, because the tag value resides at the start of a shared memory segment, it will reside at an address that is word aligned. As such, it will not span multiple cache blocks, and read and write operations will be performed atomically. By following the memory access ordering restrictions of our protocol and exploiting the atomic nature of reads and writes we are able to ensure that our data is accessed in a consistent manner in the presence of parallel execution.

Since parallel applications in computer algebra are generally dynamic and irregular, we must provide scheduling mechanism for processes in ALDOR to achieve load balancing. This parallel framework supports dynamic process management, and it is also feasible for users to apply scheduling techniques.

This section has provided a high level overview of the general concepts used to implement our parallel computer algebra framework. The low-level technical details are discussed in the following sections.

3. DATA COMMUNICATION AND SYNCHRONIZATION

Our framework uses UNIX System V shared memory segments for both data communication and synchronization between parallel processes. We developed a new ALDOR domain, `SharedMemorySegment`, to provide easy access to shared memory segments from ALDOR source code. In order to transfer information from one ALDOR process to another, the information must be serialized into a primitive array of machine integers. The destination process unserializes the data, constructing a new high-level ALDOR datatype, before the data is utilized for other purposes.

Our SharedMemorySegment domain calls C functions in order to gain access to shared memory segments. Prototypes are shown below for the most frequently used functions:

```
key_t ftok(const char *pathname, int proj_id);
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

In our framework, we generally utilize these functions in the following manner:

1. A key is constructed for the shared memory segment using `ftok`, the path to a file, and an integer.
2. The shared memory segment associated with the key is created (or connected to if it has already been created by another process) using `shmget`. The read/write permission of the segment are set by passing an appropriate value for *shmflg*.
3. A pointer to the shared memory is acquired by attaching to the shared memory segment using `shmat()`.
4. Read and write operations are performed on the shared memory by using the pointer returned by `shmat()`.
5. Each process detaches itself from the shared memory using `shmdt`.
6. The shared memory segment is deleted by providing the appropriate command flags to `shmctl()`.

Each shared memory segment is uniquely identified by its key and its size. Any process can access the shared memory segment if it knows both the key values and the size of the segment, provided that the shared memory segment is world readable. Note that it is imperative that applications free each shared memory segment that is allocated, either by invoking `shmctl()` before the application terminates or by using the `ipcrm` utility after the application completes because shared memory segments are not automatically released when a program terminates.

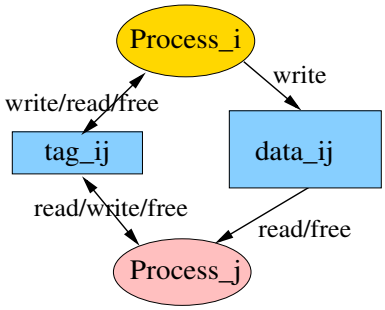

Figure 1: `Process_i` sending data to `Process_j`

In our framework, each ALDOR process is assigned a unique virtual identifier, or *VPID*, when it is created. These VPIDs guide the flow of information between ALDOR processes, serving a similar role to process ranks in MPI's data communication model. Let `Process_i` and `Process_j` be processes with VPIDs i and j respectively. Figure 1 illustrates the strategy used by `Process_i` to send data to `Process_j`.

Our communication protocol uses the `tag_ij` shared memory segment to ensure that the `data_ij` segment is accessed in a safe manner. The keys for both the tag and data segments are created using path names that are unique across our parallel framework. In particular, the key for the tag segment is created using the path */tmp/tag_ij* and the key for the data segment is created using the path */tmp/data_ij*. As a result, we are assured that these shared memory segments will only be accessed by `Process_i` and `Process_j`. Using a separate block of shared memory for each pair of communicating processes eliminates both the need for complex synchronization operations and the bottlenecks that can occur when many processes try to access a shared memory segment at the same time.

The tag segment consists of a single integer. It is used by the sending process, `Process_i`, to inform the receiving process, `Process_j`, of the size of the data that is being transferred[1]. Initially, the value of the shared memory segment is 0, denoting that no data is available to be read by `Process_j`. We do not need to explicitly write this 0 value to the shared memory segment because every byte in a shared memory segment is automatically initialized to zero as part of the allocation process. This initialization behavior is defined in IEEE Standard 1003.1 [1]. Once the polynomial data shared memory segment has been created and the data being transferred has been placed in the data segment by `Process_i`, it changes the value in the tag segment to the size of the data being transferred. `Process_i` is not permitted to change any value in either shared memory segment until it reads a 0 value from the tag segment.

When `Process_j` sees a value greater than zero in the tag segment it is assured that the required information is present in the data shared memory segment. `Process_j` writes the value -1 to the tag to indicate that it is currently accessing the shared memory segment. Once `Process_j` has successfully unserialized the data and freed the data segment, it

[1]The value -2 is used to indicate the transmission of an empty data block.

writes the value 0 to the tag segment, indicating that it is done with that set of data. By freeing the shared memory segments immediately after use, we reduce the overall memory footprint of our framework.

If the value initially read by `Process_j` is -2 then `Process_j` knows that the data block being transmitted is empty. Consequently, it immediately writes a 0 back to the tag acknowledging that there was no data to receive.

Regardless of the tag value read initially, `Process_j` is not permitted to access the data segment after writing 0 to the tag segment until the value of the tag changes to a positive integer. Following this protocol ensures that `Process_j` will always read data that is complete and that `Process_i` will never overwrite data that is still needed by `Process_j`. The tag shared memory segment is released by either `Process_i` or `Process_j` before it terminates. Note that `Process_i` is not permitted to release the tag shared memory unless it reads a 0 value from the tag.

The complete algorithm followed by `Process_i` and `Process_j` is described below:

- `Process_i` **(Sending)**
  1. Create files "*/tmp/data_ij*" and "*/tmp/tag_ij*"
  2. Generate the data segment and tag IPC keys, *data_ij* and *tag_ij* from the integer i and the files "*/tmp/data_ij*" and "*/tmp/tag_ij*" respectively
  3. Create or connect to the tag segment, setting the permission to allow both reads and writes
  4. Repeat until the value of the tag segment is 0
  5. Create the data segment with sufficient size to hold the values being sent to `Process_j`
  6. Write the data to the data shared memory segment
  7. Detach from the data segment
  8. Write the size of the data to the tag segment
- `Process_j` **(Receiving)**
  1. Create files "*/tmp/data_ij*" and "*/tmp/tag_ij*"
  2. Generate the data segment and tag IPC keys, *data_ij* and *tag_ij* from the integer i and the files "*/tmp/data_ij*" and "*/tmp/tag_ij*" respectively
  3. Create or connect to the tag segment, setting the permission to allow both reads and writes
  4. Repeat until the value of the tag segment, t, is greater than 0
  5. Write -1 to *tag_ij*
  6. Detach from the tag segment
  7. Connect to the data segment using key *data_ij*
  8. Read t integers from the data segment
  9. Detach from the data segment
  10. Delete the data segment
  11. Write 0 to the tag segment

We developed an Aldor domain named `SharedMemorySegment` to provide easy access to shared memory segments from Aldor source code. This domain used Aldor's interoperability with C in order to call the shared memory functions described earlier in this section. The domain has methods for creating and connecting to a shared memory segment, reading and writing values to and from the segment, and detaching from and deleting a shared memory segment. While these methods do not provide the full power of shared memory segments, they successfully hide many of the cumbersome details while being sufficient to implement our communication protocol.

The `InterProcessSharedMemoryPackage` domain provides a further level of abstract with the functions *Send(i,j,data)* and *Receive(i,j)*. *Send(i,j,data)* performs all of the operations described previously for `Process_i` while *Receive(i,j)* performs all of the operations described previously for `Process_j`. Using this domain allows the programmer to concentrate on solving computer algebra problems rather than the details of shared memory segments and interprocess communication.

4. SERIALIZATION OF HIGH-LEVEL OBJECTS

On shared memory computer, it would be ideal if we could copy Aldor objects directly between processes. Unfortunately, a direct copy cannot be performed because Aldor objects are represented using pointers, and because each process executes in its own address space. The memory usage within each address space will differ because the specific problem being solved by each process differs. Consequently, a pointer that is valid in one address space will not necessarily reference the correct piece of memory when it is copied to another address space. Consequently, it is necessary to serialize Aldor objects before they can be transferred to another process using shared memory segments. This serialization process converts the high-level object into an array of machine integers before it is placed in the shared memory segment. The destination process uses the array of machine integers to reconstruct the high-level Aldor objects and then performs additional computations using the objects.

We have developed a package named `Serialization` that serializes two polynomial types in the Aldor `BasicMath` library. Presently, serialization is available for `SparseMultivariatePolynomial`, abbreviated SIMPLY, and `DistributedMultivariatePolynomial`, abbreviated DMPOLY. `BasicMath` includes other polynomial representations such as `DenseRecursiveMultivariatePolynomial` and `SpareAlternatedArrayMultivariatePolynomial`. We plan to address the serialization of these polynomial types in the future.

Both SMPOLY and DMPOLY are designed for the efficient representation and manipulation of sparse multivariate polynomials. Solvers have been developed in Aldor, such as `triade` [20] based on the algorithm presented in [23], which are primarily designed for solving large systems with many variables [11]. In `triade`, a polynomial system is solved by way of triangular decomposition. Triangular decomposition requires a recursive vision of multivariate polynomials. Consequently, this solver employs the SMPOLY polynomial domain constructor because the representation is sparse and recursive. Thus a SMPOLY is a univariate polynomial whose coefficients are polynomials themselves. In broad terms, a SMPOLY is represented by a tree. The interior nodes are non-constant polynomials while the leaves are coefficients from the base ring. For example, the polynomial $g = 5x^2y^3 - 8x^2 - 7y^2 + 4$ with a variable order of $x > y$ is viewed as $(4 - 7y^2) + (-8 + 5y^3)x^2$. Figure 2 shows the SMPOLY representation of g.

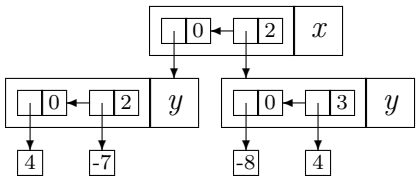

Figure 2: SMPOLY representation of g

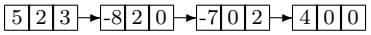

Figure 3: DMPOLY representation of g

A DMPOLY is represented by a list of terms, where each term is an exponent vector together with a coefficient. This data structure is flat. As such, it is more efficient for accessing the monomials in a polynomial, making it a suitable representation for the polynomial arithmetic involved in Gröbner basis computations. Figure 3 illustrates the DMPOLY representation of g.

Our `Serialization` package provides a function, `SerializeDMP()` which converts a DMPOLY into a primitive array of integers by traversing the values in the list of terms. Two functions are provided for converting a SMPOLY into an array of integers. One is called `SerializeSMPbyKronecker()` which turns a SMPOLY into a primitive array of machine integers via a univariate polynomial using Kronecker substitution [13]. A corresponding function is provided named `UnserializeSMPbyKronecker()`. It constructs a SMPOLY from a primitive array of machine integers. While we have successfully used Kronecker substitution here in the context of spare multivariate polynomials, one would expect this representation to be more suitable for representing dense multivariate polynomials. As a result, our work on `SerializeSMPbyKronecker()` represents preliminary work for a future investigation in the serialization of `DenseRecursiveMultivariatePolynomial`.

Another function provided by the serialization package is `SerializeSMPbyDMP`. It converts a SMPOLY into DMPOLY, and then into a primitive array of machine integers. A corresponding function, named `UnserializeSMPbyDMP`, constructs a SMPOLY from a primitive array of machine integers. In addition, we provide functions that convert a list of multivariate polynomials for some list of variables with the ring characteristic into a primitive array of machine integers.

For our example polynomial g, shown previously, the `SerializeSMPbyKronecker()` function will return an array consisting of $\{5, 0, 0, 0, -7, 0, 0, 0, -8, 0, 4\}$. Using `SerializeSMPbyDMP` for g gives $\{5, 2, 3, 8, 2, 0, 7, 0, 2, 4, 0, 0\}$.

5. DYNAMIC PROCESS MANAGEMENT

Since parallel applications in computer algebra are usually dynamic and irregular, dynamic process management adds flexibility and eases dynamic task management. Earlier research has suggested the importance of dynamic process management in computer algebra [25]. In this section, we describe the dynamic process management mechanism used in our parallel framework. In addition, we show that it is convenient to use in user's programs. Using dynamic process management reduces the complexity of data communication and user-level scheduling for load balancing.

We created a new ALDOR function, `Spawn(command, argument)`, which allows an ALDOR program to create a new process. The *command* is the name (with path) of a program to execute as a new process while *argument* is a list of arguments to the command. We implemented our Spawn function using ALDOR's `run()` primitive. Internally, run uses the `system()` function provided as part of the standard C library on most UNIX platforms. As a result, we are able to control how many new processes we spawn, and the order in which the processes are created. Once the new processes are created, their parallel execution is managed by the operating system's scheduler.

We define a **task** to be a program that can be executed independently that performs some function or processes some data. In our framework, it is the the user's responsibility to pass an integer as part of the argument list which is the VPID of the spawned process. This VPID is used to allow the process to communicate with other spawned processes as was discussed previously in Section 3.

This is analogous to what a user must do when using MPI, where a procedure being distributed to a processor is identified by the processor's rank. Using its rank, a procedure can communicate with other procedures executed by other processors.

By building on the `run()` primitive, our `Spawn()` function can be used within a process (say running program A) to launch one or more additional processes that will run other programs independently. What processor each of these processes will execute on depends on the user-level scheduler and the operating system's scheduler.

The key issue that a user needs to pay attention to is the organization of the unique VPIDs within a parallel application. We provide a solution to a general computing model described by a directed acyclic graph (DAG) for two common schemes: "task farming" and "dynamic fully-strict task processing".

The solution for the task farming scheme is simple. A Manger process with VPID 0 starts the main program and spawns worker processes. The manager also sends the data to each worker process. When a worker completes its job, it sends its result back to the manager and then it terminates. If the manager schedules tasks so that the maximum number of worker processes can run in parallel is bounded by *ncpu*, then the manager needs to maintain two integer variables: process counter and VPID counter, abbreviated PC and $VPIDC$ respectively. In addition, a list data structure, $listVPID$, is needed to hold the VPIDs of the active worker processes. PC is initialized to 0, and $VPIDC$ is initialized to 1. Initially $listVPID$ is empty. When a task needs a worker process, the Manager will check if $PC < ncpu$. If the result of the comparison is true then the manager will launch a new worker for the task. The manager will pass the value of $VPIDC$ as the VPID argument to this worker and the manager will send the data needed as well. Then the Manager will add $VPIDC$ to $listVPID$ and increment PC and $VPIDC$. The Manager will repeat this procedure if there are additional tasks and $PC < ncpu$. Otherwise, the Manger will traverses $listVPID$, checking if each worker is done. If a worker has completed, the manager will collect the worker's result, remove its VPID from $listVPID$ and decrement PC. These activities will be repeated until all of the tasks are solved. All data communication between the manager and a worker is achieved by the techniques described in Section 3.

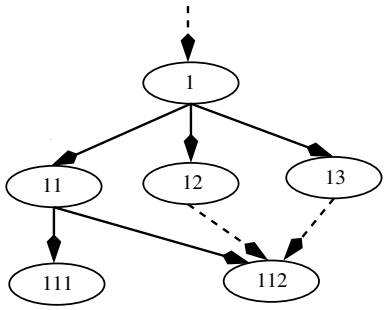

Figure 4: Dynamic fully-strict task processing

The scheduling algorithm in this solution corresponds exactly to the *greedy* scheduling method [17]. A greedy scheduler attempts to do as much work as possible at every step for a given number of processors, P. If there are at least P tasks ready to run, it selects any P of them and runs them. When there are strictly less than P tasks that are ready to run, the greedy scheduler runs them all. Given P processors, a greedy scheduler executes any computation DAG in time: $T_p \leq T_1/P + T_\infty$, where T_p is the minimum running time on P processors, T_1 is the minimum running time on 1 processor, and T_∞ is the critical path length of a DAG. A greedy scheduler is always within a factor of 2 of optimal when ignoring the overhead due to task management and communication/synchronization. It is generally a good scheduler.

A variation based on this scheme is the *task pool with dimension and rank guided dynamic scheduling* designed and implemented in ALDOR for a parallel solver [22]. An implementation of the *greedy* scheduling method has also been realized in this solver for performance evaluation.

Another scheme we provide a solution to is dynamic fully-strict task processing, where tasks are generated dynamically and processed accordingly. In general, this problem can be modeled by a DAG. Part of an example DAG is shown in Figure 4. This type of problem corresponds to a problem known as *fully strict (i.e. well-structured) multithreaded computations*, originally published in [5]. A solution for the organization of unique VPIDs for using multiprocessed parallelism by this framework is illustrated in Figure 4.

In Figure 4, the dotted arrows denote data dependencies, while the solid arrows show when new processes were spawned. The initial process has VPID 1, and starts with the input data. Then the initial process spawns three new processes with VPIDs of 11, 12 and 13 respectively. The following rule is used to generate and assign VPIDs to any new processes it spawns. Let the process' VPID be k. Let the number of new processes it will spawn be n. For $1 \leq i \leq n$, the i^{th} new process is given a VPID of ki, which is obtained by appending i to k. For instance, the first new process is given a VPID of $k1$, the second new process is given a VPID of $k2$, etc. Constructing the VPID as ki guarantees uniqueness within this application because the prefix, k, was unique in its parent's list of spawned processes and i is unique for each process spawned from this process. We note that this solution is akin to the scheme for handling the rank of spawned processes in MPICH2 [3].

The performance of dynamic process management in this parallel framework for coarse grained parallel computations is reported in Section 6.

6. EXPERIMENTAL RESULTS

This parallel framework has been used successfully to implement a parallel symbolic triangular decomposition solver [22]. It utilizes both the dynamic task management and data communication techniques we have described previously to communicate lists of multivariate polynomials to workers which execute in parallel. We describe the performance gains that we have achieved using our framework. In addition, we provide a detailed analysis of the overhead costs introduced such as the cost of process spawning and data communication/serialization for both the Kronecker substitution and `DMPOLY` serialization techniques. Results are presented for several well known examples, allowing the performance and overhead of our parallel framework to be compared with techniques developed by other authors.

Our experimental results were gathered on *Silky*, one of several multiprocessor clusters and SMP systems that make up Canada's Shared Hierarchical Academic Research Computing Network (SHARCNET). Silky is classified by SHARCNET as a mid-range symmetric multiprocessing (SMP) cluster. It is a SGI Altix 3700 Bx2, equipped with 128 Itanium2 processors clocked at 1.6GHz. The cluster has 256GB of main memory with a 6MB cache and runs SUSE Linux Enterprise Server 10 (ia64). Unfortunately Silky is a shared machine accessed by a large number of researchers in areas from computer algebra to computational physics. It is common for the active processes to occupy over 95 percent of the main memory and all 128 processors. As a result, we were unable to acquire sufficient resources on the machine to run large memory examples such as Virasoro.

The parallel solver was developed based on a triangular decomposition algorithm called `Triade` [23]. It makes use of the ALDOR `BasicMath` library for polynomial arithmetic over an arbitrary ring. We compare the performance of our framework with the sequential `triade` solver in ALDOR which uses the same underlying algorithm. Thus, the performance results we present here represent how well we are able to use the parallel resources available to us rather than the difference between two distinct algorithms.

The `Triade` algorithm uses "incremental solving", organizing the computation into a dynamic task tree. A `task` is any pair $[F, T]$ where F is a finite system of equations to solve and T is a solved system. Splitting of a task is based on the `D5 Principle` [9] and case distinction of the form $f = 0$ *or* $f \neq 0$. The parallelization of this algorithm exploits the parallel opportunities created by the task splitting based on the `D5 Principle` combined with modular techniques [8].

Parallelizing the problem in this manner provides a coarse grained division of the work into separate processes. The parallelization is highly dynamic, with the final shape of the task tree being determined only when a solution to the system is achieved. The parallelization is also highly irregular, varying greatly with respect to both the input system of equations and the size of the task represented by each node in the tree. The implementation for this algorithm uses a "task farming" scheme and a cost-guided scheduling algorithm. A manager process preprocesses the input system and distributes intermediate tasks to the worker processes. If only one task needs to be solved then the manager completes the task itself. The manager also processes trivial tasks on its own.

When there is a task to perform, and an available processor, the manager spawns a new worker process. The man-

Sys	Name	n	d	p	Sequential (s)
1	eco6	6	3	105761	4.00
2	eco7	7	3	387799	727.95
3	CNogues2	4	6	155317	476.16
4	CNogues	4	8	513899	2162.40
5	Nooburg4	4	3	7703	4.14
6	UBikker	4	3	7841	866.20
7	Cohn2	4	6	188261	305.24

Table 1: Features of the examples

Sys	CPUs	Kron. (s)	DMP (s)	Kron. Speedup	DMP Speedup
1	5	1.94	1.91	2.1	2.1
2	9	119.44	117.41	6.1	6.2
3	9	207.29	215.28	2.3	2.2
4	9	905.25	1002.56	2.4	2.2
5	9	1.79	1.81	2.3	2.3
6	9	455.21	463.24	1.9	1.8
7	9	96.70	102.55	3.2	3.0

Table 2: Parallel timing on two serializing methods

Sys	Workers (#)	Tags (#)	Read (#int*)	Write (#int)	Total (#int)	Zeros (%)
1	9	9	4131	3586	7717	59
2	24	24	29307	27382	56689	72
3	32	32	57106	55696	112802	73
4	42	42	216000	214217	430217	83
5	14	14	13307	0	13307	72
6	49	49	128983	125162	254145	55
7	44	44	39146	38280	77426	39

Table 3: Dissection of workers' overhead for Kronecker (* One int has 8 bytes)

ager then forwards the data for the task to that worker. The worker will process the task and send the intermediate results back to the manager unless the task is solved. Once the intermediate results or solution are determined the worker process terminates. In our example, each task consists of a list of ALDOR `SMPOLY` polynomials.

Table 1 lists the number of variables, n, and the total degree, d for each of the examples. It also lists the prime number for modular computation and the time required to reach a solution using the sequential solver Triade. Table 2 records the time required to reach a solution in our parallel framework for the two serialization techniques discussed previously. The number of CPUs used and the speedup relative to the sequential algorithm are also reported in this table.

In Table 3 and Table 4 we report additional details about the behavior of the Kronecker and DMPOLY serialization methods respectively. These details include the number of workers, number of tags, number of integers read and written and the percentage of zeros in the integers transferred. In every case, values for reads and writes are reported from the perspective of the worker tasks. Note that each worker process is created dynamically to process a specific task. The worker terminates when it completes its task. Consequently, the total number of workers used is equal to the total number of tasks being executed in parallel.

Table 5 and Table 6 show the time spent spawning workers, synchronizing their execution and communicating data to and from them. The reading time includes both the time required to read the array of machine integers from shared memory and the time required to reconstruct the high-level ALDOR object. Similarly, the writing time includes both the time spent serializing the high level objects and the time spent to copy the serialized data into a shared memory seg-

Sys	Workers (#)	Tags (#)	Read (#int)	Write (#int)	Total (#int)	Zeros (%)
1	9	9	5069	4449	9518	55
2	24	24	36893	35184	72077	57
3	32	32	64106	64106	127304	39
4	42	42	168178	167186	335364	39
5	14	14	12681	0	12681	44
6	49	49	271845	267761	539606	42
7	44	44	104486	103534	208020	40

Table 4: Dissection of workers' overhead for DMPOLY

Sys	Spawns (ms)	Tags (μs)	Read and Unserialize (ms)	Serialize and Write (ms)	Net Work (s)	Overhead (%)
1	358	1067	492	76	3.80	24.4
2	579	3414	1264	184	660.54	0.3
3	773	4887	9682	623	469.48	2.4
4	1695	7221	68737	491	2164.62	3.3
5	452	1940	488	0	3.57	26.4
6	1558	7773	21762	823	871.04	2.8
7	925	6014	2378	369	289.15	1.3

Table 5: Dissection of workers' time for Kronecker (wall time)

Sys	Spawns (ms)	Tags (μs)	Read and Unserialize (ms)	Serialize and Write (ms)	Net Work (s)	Overhead (%)
1	314	1211	347	79	3.71	20.0
2	685	3498	1345	623	611.16	0.4
3	1435	4676	1813	683	474.16	0.8
4	1723	7524	80490	2360	2134.71	3.8
5	552	2224	764	0	3.65	36.2
6	1994	7847	52157	5242	886.59	6.7
7	1110	6673	5881	2063	282.16	3.2

Table 6: Dissection of workers' time for DMPOLY (wall time)

Sys	Per Spawn (ms)	Per Tag (μs)	Read and Unserialize (μs per int)	Serialize and Write (μs per int)
1	40	118	119	21
2	24	142	43	6
3	24	152	169	11
4	40	172	318	2
5	32	138	36	-
6	32	158	168	6
7	21	136	60	9
AVG	30	145	130	9

Table 7: Analysis of workers' overhead for Kronecker

Sys	Per Spawn (ms)	Per Tag (μs)	Read and Unserialize (μs per int)	Serialize and Write (μs per int)
1	35	134	68	17
2	29	146	36	18
3	45	146	180	21
4	41	179	478	14
5	39	158	59	-
6	41	160	192	19
7	26	151	56	20
AVG	37	153	152	18

Table 8: Analysis of workers' overhead for DMPOLY

ment. In addition, we report the total net amount of work performed, which is the time spent by workers excluding the time spent spawning processes and performing synchronization and data communication. The percentage overhead is the ratio of the sum of the time spent on overhead divided by the net amount of work performed. Table 7 and Table 8 go on to show the average cost per worker spawned, per synchronization tag used, per integer read and unserialized and per integer serialized and written for the Kronecker and `DMPOLY` serialization methods respectively.

Examining the total parallel execution time reveals that there is little variation between the Kronecker and DMPOLY serialization techniques. Similar performance was observed because the examples presented here transfer data that is not very sparse, as is indicated in the percent zeros column.

Most of the examples considered in this study show a low amount of parallel overhead. Exceptions to this general pattern are Sys 1 and Sys 5 which are both smaller examples. Even though the level of overhead is low, our results show that this parallel framework is only suitable for coarse grained parallel symbolic computations. The granularity of this parallelization framework is very coarse, as shown by the total number of workers (tasks) that are executed in parallel and the total number of workers used in total. The average cost of spawning a process is approximately 35 milliseconds. Reading one integer and unserializing it to reconstruct the high-level ALDOR object normally takes between 36 and 192 microseconds on average. The cost in Sys 4 is outside of this range, likely due to the relative high degree of the polynomials in the data being transferred. We plan to investigate this phenomenon further in the future.

We also observed that serializing and writing costs are much lower than the reading and unserialization costs. This difference occurs because constructing a new high-level object is expensive, involving numerous memory allocations to represent the object's complex structure. In contrast, serializing the object does not require any memory allocation or deallocation operations. The time complexity of serialization via either Kronecker substitution or DMPOLY is linear. Interestingly, the cost associated with our synchronization tags was small in comparison to the other sources of overhead in our parallel framework.

We also observed that the per integer serialization cost for a SMPOLY via DMPOLY almost doubles compared to using Kronecker substitution. This doubling reveals a drawback in our implementation, which requires that the SMPOLY must be converted to a DMPOLY twice, once to determine the size of the shared memory segment, and then a second time to write the serialized data into the shared memory segment. We expect that it will be easy to remove this inefficiency by improving our implementation.

Finally, we wish to point out that the dynamic nature of our task management technique is particularly advantageous in a shared computing environment such as SHARCNET. While there is overhead associated with spawning and terminating processes, our technique ensures that a process only exists when it has useful work to perform. There will not be any processes left idling, waiting for something to do. This helps ensure that shared computing resources are used effectively. Furthermore, removing idle processes ensures that we do not spend unnecessary time communicating with or synchronizing such processes. Because of this, we have found, both in theory and in practice, that our communication costs do not increase with number of CPUs utilized.

7. CONCLUSION AND FUTURE WORK

We have reported on a high-level categorical parallel framework to support high performance computer algebra on SMPs and multicores. We have used the ALDOR programming language as our implementation vehicle since it has high-level categorical support for generic algorithms in computer algebra, while providing the necessary low-level access for high performance computing.

Our framework provides functions for dynamic process management and synchronized data communication for high-level ALDOR objects such as sparse multivariate polynomials via the shared memory segments. This framework is complementary to $\prod^{IT}$ [21] which targeted distributed architectures.

Throughout the design and implementation of the framework, we have kept the solution of multivariate polynomial systems as a motivating example. Indeed, this framework has been used for the successful implementation of a sophisticated parallel symbolic solver. Our evaluation of the parallel overhead has shown that this parallel framework is efficient for coarse-grained parallel symbolic computations. More experimentation on the granularity of parallelism supported by this framework is work in progress.

We plan to develop a model for threads in ALDOR to support finer grained parallelization, as has been done for SACLIB [19, 24] and KAAPI [18, 15], and support automatic scheduling based on "work stealing" [5, 12]. In this setting, ALDOR's system of parametric types provides opportunities for elegant problem formulation. As a practical matter, it will be necessary to review the ALDOR run-time system to take advantage of threads and modify it in a few places for thread safety. In particular, one current investigation is modification to use "localized tracing" [7], in the garbage collector to allow it to make use of multiple threads.

8. REFERENCES

[1] The Open Group Base Specifications Issue 6. IEEE Std 1003.1, 2004 Edition. http://www.opengroup.org/onlinepubs/009695399/.

[2] aldor.org. *Aldor 1.0.3*. University of Western Ontario, Canada, 2004.

[3] Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[4] T.J. Ashby and A.D. Kennedyand M.F.P. O'Boyle. A modular iterative solver package in a categorical language. In *LNCS vol.47*, pages 123–132, 1993.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *IEEE FOCS94*, 1994.

[6] C. Chen, M. Moreno Maza, W. Pan, and Y. Xie. Verification of polynomial system solvers. In *Proceedings of AWFS 2007*, 2007.

[7] Y. Chicha and S. M. Watt. A localized tracing scheme applied to garbage collection. In *APLAS 2006*, 2006.

[8] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *ISSAC'05*, pages 108–115. ACM Press, 2005.

[9] J. Della Dora, C. Dicrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In *Proc. EUROCAL 85 Vol. 2*, vol. 204 of *Lect. Notes in Comp. Sci.*, pages 289–290. Springer-Verlag, 1985.

[10] Sun Fortress Project Group. The Sun Fortress Project. fortress.sunsource.net, 2007.

[11] M.V. Foursov and M. Moreno Maza. On computer-assisted classification of coupled integrable equations. *J. Symb. Comp.*, 33:647–660, 2002.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN*, 1998.

[13] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[14] T. Gautier and N. Mannhart. Parallelism in aldor – the communication library Piit for parallel, distributed computation. In *ACPC-2, LNCS vol.734*, pages 204–218, 1998.

[15] T. Gautier, J. Roch, and F. Wagner. Fine grained distributed implementation of a language with provable performance. In *Proceedings of ICCS2007/PAPP2007*, May 2007.

[16] J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook*. Springer, 2003.

[17] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(3):416–429, March 1969.

[18] KAAPI group. KAAPI: Kernel for adaptive, asynchronous parallel and interactive programming. http://kaapi.gforge.inria.fr/.

[19] W. W. Küchlin. PARSAC-2: A parallel SAC-2 based on threads. In *AAECC-8, LNCS vol.508*, pages 341–353, 1990.

[20] F. Lemaire, M. Moreno Maza, and Y. Xie. Making a sophisticated symbolic solver available to different communities of users. In *Proceedings of ATCM 2006*, 2006.

[21] N. Mannhart. *Piit: a portable communication library for distributed computer algebra*. PhD thesis, Swiss Federal Institute of Technology, 1997.

[22] M. Moreno Maza and Y. Xie. Component-level parallelization of triangular decompositions. In *Proceedings of PASCO2007*. ACM Press, 2007.

[23] M. Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. Presented at the MEGA-2000 Conference, Bath, England. http://www.csd.uwo.ca/∼moreno.

[24] W. Schreiner and H. Hong. The design of the PACLIB kernel for parallel algebraic computation. In *ACPC-2, LNCS vol.734*, pages 204–218, 1993.

[25] Stephen M. Watt. A system for parallel computer algebra programs. In *LNCS Vol.204*, pages 537–538, 1985.

[26] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. A first report on the $A^{\#}$ compiler. In *ISSAC '94: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, New York, NY, USA, 1994. ACM Press.

[27] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morisson, Jonathan M. Steinbach, and Robert S. Sutor. *AXIOM Library Compiler User Guide*. NAG, The Numerical Algorithms Group Limited, Oxford, United Kingdom, 1st edition, November 1994. AXIOM is a registred trade mark of NAG.

[28] Stephen M. Watt. *Aldor*. In *Computer Algebra Handbook* (J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors), pages 265–270, Springer 2003.