Concurrent DASSL: A Second-Generation, DAE Solver Library

Alvin Leung Anthony Skjellum† Geoffrey Fox

Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13244

†NSF Engineering Research Center
Mississippi State University
Mississippi State, MS 39762

Abstract

The goal of the most recent revision of Concurrent DASSL code is to recast it to conform to the Toolbox's object-oriented design philosophy and to enhance its performance. In this report, we will describe in detail improvements in three categories: error handling, performance enhancement and uniform interfaces. Through these improvements, we are able to achieve our goals.

1 Introduction

Concurrent DASSL (Cdassl) implements the well-known algorithms by Petzold incorporated in the Fortran codes DASSL [1] and DASPK [2]. Since the Fortran code DASPK is an functionally enhanced version of DASSL, we will refer to both the Fortran codes DASSL and DASPK as DASPK. Concurrent DASSL dates back to 1989, and we have reworked Concurrent DASSL to reflect improvements in our understanding of needs for concurrent, scalable DAE solvers.

The goals of this revision are to remold Cdassl into an object-oriented library and to enhance its performance. In this report, we will describe three recent important improvements that we have make to achieve our goals.

2 Error Handling

In this version of Cdassl, we have made compromises on error handling to present Cdassl in an object-oriented programming style and to be a more user friendly library. The original DASPK code reports three major categories of runtime error, for proper data storage, for programming error and data consistency and for numerical error.

The DASPK code provides these comprehensive data and storage integrity tests both during initial startup and continuation call. This is necessary because DASPK's data storage is provided by the user through multiple large arrays. These arrays are then used to store DASPK's initial state and the state for continuation calls. The users can determine the state of DASPK by indexing the information stored in the arrays. In addition, the performance of DASPK can be adjusted by modifying certain variables, for example, RTOL and ATOL, where threshold of relative error and absolute error is stored. Consequently, a minor programming mistake can wipe out an important piece of DASPK's state information.

The tests on data and storage integrity are still conducted in the first version of Concurrent DASSL because it requires the user to provide memory storage and the C language does not provide data protection. However, the safety of data in Cdassl is much improved over DASPK because pointers are used to bind symbolic names to the subsets of data. Consequently, the risk of programming error is reduced, since more descriptive names are used instead of index number.

In an object-oriented programming environment, an object is created by a constructor function and is destroyed by a destructor function. In addition, the information within a class structure is protected by access permission. The access permission is set at compile time. Furthermore, the protected data can only be manipulated through member and friend functions. As a result, runtime data storage and sanity checks are no longer necessary.

In the new version of Cdassl, a constructor and a destructor function is provided for the creation of Cdassl data structures. This not only eliminates part of the mentioned tests, but also simplifies the set-up procedure for using Cdassl. Since the risk of programming error is reduced by the use of symbolic names, data integrity checks are eliminated. As a result, the Cdassl's code complexity and sequential overhead is reduced.

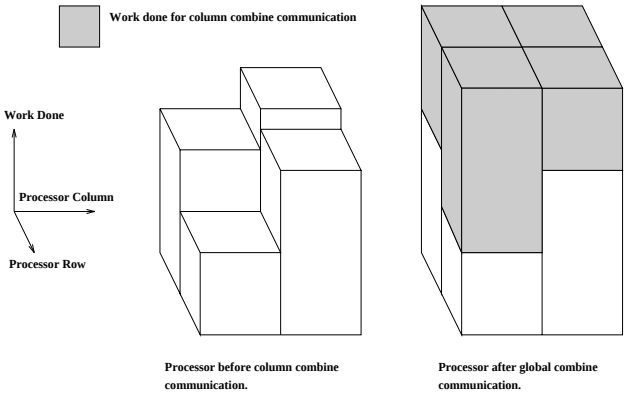

Figure 1:

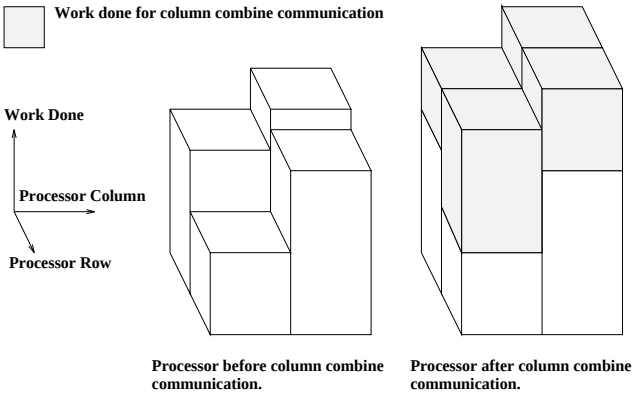

Figure 2:

The issue of numerical error reporting, and clean exit from a concurrent library has not been examined in detail. Each error reporting requires additional communication to the system. Hence, special attention is required to balance the benefit of accurate error reporting and the penalty on performance.

To post a warning and error flag successfully, Cdassl uses a global combine function to provide communication among all processes. However, a global combine communication synchronizes all processes involved as shown in figure 1. In other words, every error reporting call will reduce Cdassl's performance by a certain amount. Consequently, a trade-off has to be made to minimize the cost of the warning and error reporting. This is accomplished by reducing the number of error posting points and by communicating status flags after the existing synchronization points, such as, norm or tolerance computation. During these calculations, column or row combine functions are already used to communicate and to compute results. As a result, all processors has already been row-wise or column-wise synchronized as shown in figure 2, the added synchro-

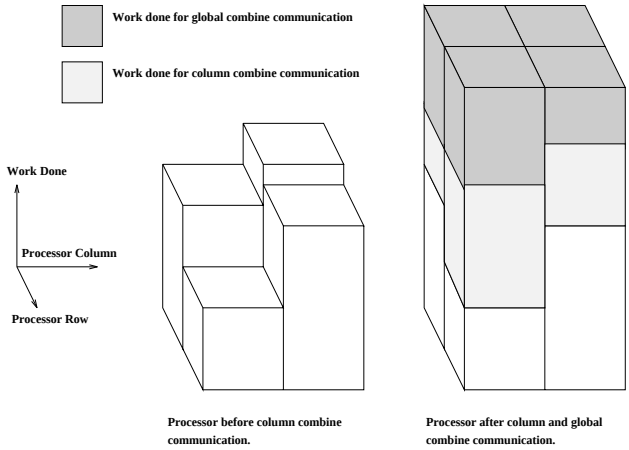

Figure 3:

nization (figure 3) over the processors grid will have reduced impact on Cdassl's performance. In addition, numerical error condition is most likely detected just after the norm or tolerance computation because most of these calculations are used to detect numerical instability of the Cdassl's computation. Furthermore, these calculations happen regularly during the course of Cdassl's operation, so the warning and error status is reported with reasonable reliability compared to the original, sequential code. As a result, Cdassl's runtime warning and error handling provides a reasonably accurate picture of Cdassl's operation without significant performance penalty.

3 Performance Enhancement

In the new version of Cdassl, we have rewritten the code so that it conforms to an object-oriented programming style. This opens the way for Cdassl to tap into high performance numerical subroutines.

The first generation Cdassl code processes concurrent vectors (Cvectors) by using in-line loops over the private data of Cvector. By manipulating Cvector in this fashion, the implementation of Cdassl's function depends on the implementation of Cvector. This implies that Cdassl and Cvector cooperate to get Cdassl's task done. In C programming environment, this is acceptable and highly efficient. However, this approach foregoes the portability of Cdassl because whenever Cvector's implementation changes, Cdassl's code has to change accordingly. Since Cvector is one of the Toolbox's basic data storage classes, Cvector's implementation will be updated to reflect the latest enhancement offered by technology. On

```
double tbx_min_fabs_Cvector(Cvector *X);
double tbx_max_fabs_Cvector(Cvector *X);
```

Figure 4: Two of the Cvector functions most frequently used by Cdassl. Both of these functions use row or column combine communication function.

the other hand, if we change the relationship between Cdassl and Cvector from partnership to master-slave, then Cdassl can be implemented independent from the Cvector classes with the exception of a few minor operations, which we can easily implemented as friend functions of Cvector.

The Second generation Cdassl code uses Cvector's member and friend functions to manipulate a Cvector structure as an object. These member and friend functions are categorized into four types.

1. Cdassl specific friend function
2. Initiation member function
3. Communication intensive friend function
4. Numerically intensive friend function

The first and second categories provide nothing more than Cdassl specific and general purpose initiation operation. The next two function types, however, provide significant performance impact for Cdassl and/or other Toolbox libraries. We will give a brief description for the communication intensive friend function and the numerically intensive friend function.

The communication intensive friend functions, which are shown in figure 4, are newly implemented by using Zipcode row or column combine functions. After the use of these functions, the processor grid is partially synchronized. However, these functions send only short messages. So the special short message communication facility can be used, if it exists. Typically, these functions are used in norm and tolerance calculation in Cdassl.

The numerical computation intensive friend functions, which are shown in figure 5, are implemented by using level 1 Concurrent BLAS (CBLAS) [3]. CBLAS are highly optimized Cvector operators and may exploit special machine hardware, such as vector processors. Consequently, Cdassl will have a significant performance increase, whenever high performance CBLAS are available. On many systems, portable level-1 CBLAS, defined in terms of BLAS, will be sufficient to attain such high performance.

By using these member and friend functions, not only the complexity of maintaining Cdassl code is reduced, but Cdassl is also provided a way to exploit the high performance computation subroutines.

4 Uniform interfaces

In the original Cdassl, the Newton and the linear system solver are tightly integrated into Cdassl's code and data structure. This proves to be the major obstruction when incorporating DASPK's functionalities into Cdassl. As a result, we reorganized and redesigned the Cdassl code before DASPK's functionalities are incorporated.

We first reorganized Cdassl's data structure. In the first version of Cdassl, a multi-layer structure, Cdassl_matrix, is designed and used to keep track of which solver is being used and the solver's specific parameters. In other words, Cdassl_matrix is designed for the sole convenience of resolving function pointers at run time. This is a property that Cdassl inherited from its sequential parent. The complexity of original Cdassl code is high because many pointers are needed to maintain the structure and to dereference the information. The runtime code efficiency also decreases because of the multiple indirect reference. However, this complex data structure design can be avoided, if the user is encouraged to resolve function pointers at compile time.

In most cases, the Cdassl user has a clear idea which solver to use to solve the problem. Hence, we eliminated the Cdassl_matrix structure. We were immediately rewarded by a more clear global view of how Cdassl's data should be organized. Under the new organization, Cdassl is freed from the binding of the linear system's matrix structure. Only the Jacobian computation function needs to manipulate the linear system's matrix structure, but the Jacobian function is now bundled with the Newton solver. As a result, we force Cdassl to conform to the object-oriented design philosophy.

We then redesigned Cdassl's code to eliminate the state machine of Newton and linear system solver. In place of the old Newton and linear system solver, we installed our new uniform calling interfaces. Hence, Cdassl no longer depends on a particular implementation of Newton solver and it has no knowledge of the linear system solver. This increases the flexibility of the Cdassl code tremendously. For example, Cdassl can be easily adapted to use a block-diagonal-bordered direct solver [4] by using the appropriate Jacobian computation function and a Newton solver, which uses

```
void     tbx_mult_Cvector(Cvector *D, Cvector *X);
void     tbx_div_Cvector(Cvector *D, Cvector *X);
void     tbx_cblas_cdscale(int N, double alpha, Cvector *X, int Xs, int Xinc);
Cvector *tbx_cblas_ccdcopy(int N, Cvector *Y, int Ys, int Yinc, Cvector *X, int Xs,
                           int Xinc);
Cvector *tbx_cblas_ccdaxpy(int N, double alpha, Cvector *Y, int Ys, int Yinc, Cvector *X,
                           int Xs, int Xinc);
```

Figure 5: Five of the most frequently used level-1 Cblas functions by Cdassl. Most of these functions do not require any communication.

the direct solver. As a result of these changes, Cdassl has been reorganized and redesigned better to conform to Multicomputer Toolbox's object-oriented design philosophy.

The uniform calling interface [5] consists of a structure, which bundles the information needed by a class of functions, a constructor and a destructor of the structure. All the information which is included in the interface structure, is referenced by a pointer of type void. In addition, no memory or function is associated with the structure during its creation. Therefore, the uniform calling interfaces provide a pure virtual class. If a derived class "public inherit" the uniform calling interface class, then the derived class is provided a uniform interface. Combining this interface class with the derived classes, which conform to the function input and output characteristic, the user is furnished the flexibility for "part-swapping." In figure 6, a outline of changing Cdassl's linear solver form a dense direct solver to a sparse direct solver. This uniform calling interface will also available for DAE and ODE solver.

5 Summary

In this paper, we have described three major types of changes that we made to the Cdassl code. These new changes bring the Cdassl library up to the Multicomputer Toolbox's design standard. In addition, the new code opens the way for Cdassl to exploit the high performance computing library and an easy way for Cdassl users to experiment using different Newton solver and linear solvers to solve their problem. Consequently, the second generation Cdassl library has been significant improved over the first generation both in code design, in the simplicity of use and in the performance.

Acknowledgments

This work has been supported in part by Niagara Mohawk Power Corporation, the New York State Science and Technology Foundation, the NSF under cooperative agreement No. CCR-9120008, and ARPA under contract No. DABT63-91-K-0005.

Special thanks to D. P. Koester and Paul Coddington for proof reading this paper.

References

[1] Linda Petzold, "ddassl.for - differential/algebraic solver," *Computing and Mathematics Research Division, Lawrence Livermore National Laboratory*, June, 1991.

[2] Linda R. Petzold, Peter N. Brown, Alan C. Hindmarsh, and Clement W. Ulrich, "ddaspk.for - differential/algebraic solver," *Computing and Mathematics Research Division, Lawrence Livermore National Laboratory*, Sept., 1990.

[3] Robert D. Falgout, Anthony Skjellum, Steven G. Smith and Charles H. Still, "The Multicomputer Toolbox Approach to Concurrent BLAS and LACS," *Numerical Mathematics Group, Lawrence Livermore National Laboratory*, Feb., 1992.

[4] D. P. Koester, S. Ranka and G. C. Fox, "Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Electrical Power System Applications," *The Scalable Parallel Libraries Conference*, Oct., 1993.

[5] Anthony Skjellum, Alvin P. Leung, Steven G. Smith, Robert D. Falgout, Charles H. Still, Chuck H. Baldwin, "The Multicomputer Toolbox - First-Generation Scalable Libraries," *27th Hawaii International Conference on Systems Sciences*, June, 1993.

Dense matrix solver

```
Matrix_LU_info = new_Clu_info( \ldots );  /* Creates the LU info. */
linear_solver = new_method(Clu_solver, \ldots ); /* bundle the dense solver */
Jacobian_fn = new_method(cdassl_ffdjac, \ldots ); /* bundle the dense Jac. fun.  */
```

Sparse matrix solver

```
Matrix_LU_info = new_Dlu_info( \ldots );  /* Creates the LU info. */
linear_solver = new_method(Dlu_solver, \ldots );/* bundle the sparse solver */
Jacobian_fn = new_method(cdassl_vsfdjac, \ldots ); /* bundle the sparse Jac. fun. */
```

Figure 6: The major code differences between Cdassl using dense and sparse LU solver

[6] Scott Meyers, *Effective C++*, Addison-Wesley, 1992.

[7] Bryan Flamig, *Practical Data Structures in C++*, Wiley, 1993.

[8] James O. Coplien, *Advanced C++*, Addison-Wesley, 1992.