

The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models

Jay Larson, Robert Jacob, and Everest Ong

April 27, 2005

*Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Ave., Argonne, IL 60439*

Submitted to *International Journal for High Performance Computing Applications*

Running head:
Model Coupling Toolkit

Jay Larson
(Corresponding Author)
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
630-252-7806
630-252-6104 (fax)
larson@mcs.anl.gov

Robert L. Jacob
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
630-252-2983
630-252-5986 (fax)
jacob@mcs.anl.gov

Everest Ong
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
630-252-6586
630-252-6104 (fax)
eong@mcs.anl.gov

ABSTRACT

Many problems in science and engineering are best simulated as a set of mutually interacting models, resulting in a *coupled* or *multiphysics* model. These models present challenges stemming from their interdisciplinary nature and from their computational and algorithmic complexities. The computational complexity of individual models, combined with the popularity of the distributed-memory parallel programming model used on commodity microprocessor-based clusters, results in a *parallel coupling problem* when building a coupled model. We define and elucidate this problem and how it results in a set of requirements for software capable of simplifying the construction of parallel coupled models. We describe the package we developed—the *Model Coupling Toolkit* (MCT)—to meet these general requirements and the specific requirements of a parallel climate model. We present the MCT programming model with illustrative code examples. We present representative results that measure MCT’s scalability, performance portability, and a proxy for coupling overhead.

1. Introduction

Complex systems comprising numerous, mutually interacting subsystems abound in nature and engineering. These models are commonly called *coupled* or *multiphysics* models. Examples include models of climate (?), space weather (?), reactive flow (?), solid rockets (?) and fluid-structure interaction (?).

A coupled climate model is an excellent example of a multiphysics model, comprising interdependent models that simulate the Earth's atmosphere, ocean, cryosphere, and biosphere. Each of these models requires boundary condition data from other models and, in turn, provides as output boundary condition data for other models in the system. For example, the atmosphere provides to the Earth's surface downward radiative fluxes, momentum fluxes in the form of wind stress, and fresh water flux in the form of precipitation.

Until recently, computer simulation of physical, chemical, biological, and environmental systems has focused on individual subsystems that are part of a greater whole. Study of the Earth's climate system, for example, is undertaken by scientists concentrating in one of the following fields: atmospheric physics, oceanography, sea-ice modeling, and land-surface modeling. These specialists work on models for their respective disciplines, and in the past they ran their subsystem models in isolation, using prescribed data from climatologies, reanalyses, output from the other disciplines' models run off-line, or data computed by using drastically simplified versions of the other subsystems (e.g., use of a mixed-layer ocean to provide ocean surface data for use by an atmospheric GCM).

Parallel coupled models incur a high computational cost from running numerous compute-intensive algorithms to integrate the equations of evolution for each subsystem of the coupled system. In terms of the climate system model as an example, atmosphere and ocean general circulation models (GCMs) are among the most computationally demanding applications in computational science.

The software engineering of a parallel coupled model requires solutions to many challenging problems in the implementation of a large system comprising many mutually interacting and separately developed parts. In this paper, we focus on the data interactions between the models, and consider issues regarding build system and language interoperability beyond the scope of our discussion. Data exchanged among the models typically resides on differing spatial meshes, requiring interpolation between the source and target components' respective grids. The models may also differ in how they discretize time, requiring some scheme to either interpolate or average/accumulate data for translation between the source and target components' time meshes.

The use of distributed-memory parallel programming in each component has led to a *parallel coupling problem*: the challenge of connecting models that not only have different internal data structures but may each have different decompositions of those data structures each on different sets of processors.

Despite these obstacles, parallel coupled models have been created and used successfully (?; ?; ?). Below we discuss in brief typical solutions employed in coupled model development.

In the past, parallel coupled model developers such as those mentioned

above have surmounted the parallel coupling problem by implementing *ad hoc* application-specific solutions. Like language interoperability, the parallel coupling problem is a software barrier amenable to a generic software solution. Parallel coupling at a minimum poses a challenge in parallel data transfer between components—the so-called M-by-N (or $M \times N$) problem (?). A second, often-encountered requirement to implement parallel coupling is the need for distributed intergrid interpolation algorithms. We describe here a general solution to these and other problems in the form of a software library with datatypes and methods for the most commonly encountered problems in building parallel coupled models: the Model Coupling Toolkit (MCT).

In Section ?? we discuss the parallel coupling problem, both in the general case and the specific case of the Community Climate System Model (CCSM). In Section ??, we describe MCT and explain how its kernels support parallel multiphysics coupling. A companion paper (?) gives details on the parallel data transfer capabilities in MCT. In Section ??, we discuss the MCT programming model and present pseudocode to illustrate its usage. A companion paper (?) describes in detail how MCT was employed to create a parallel coupling infrastructure for CCSM. In Section ??, we present performance results. In Section ??, we present our conclusions and chart a possible future course for MCT development.

2. The Parallel Coupling Problem

Versions of the Community Climate System Model (CCSM) through version 2 (?) had a coupler with only shared-memory parallelism. Data was

exchanged with this coupler by single MPI messages between the coupler and the root process of the parallel component models. To accommodate the future development of CCSM, which envisions both increasing horizontal resolution and adding more physics parameterizations, a parallel coupler was needed.

The challenges posed in building a parallel coupler for CCSM led to a general consideration of the software needs of parallel coupled models, which in turn led to the creation of the Model Coupling Toolkit. This general description of the problem is summarized in Section ?? and will be explored further in a future paper. The specific challenges of CCSM are described in Section ??.

2.1. The General Problem of Parallel Coupling

A parallel coupled model is a collection of N component models, or *components*,¹ each of which may employ distributed-memory parallelism. Each component C_i (where $i = 1, \dots, N$) resides on P_i processing elements. Each model may in principle have its own distinct discretization of space and time and, possibly, multiple mesh schemes. Each model shares an interface with one or more of the other $N - 1$ models and requires input field data from and/or supplies output field data to these other models. We define the parallel coupling problem as the transmission and transformation of the various distributed data between the component models comprising a parallel cou-

¹Our use of the word component is from the viewpoint of scientists who develop computer models. Its usage here is related to, but not identical to, the usage of the term in component-based software engineering paradigms such as CORBA (?) or CCA (?).

pled system.

The parallel coupling problem has two major facets: coupled system architecture and parallel data processing required to implement model interaction.

The architectural aspects of the problem fall into two broad categories, and originate either in scientific requirements or software implementation choices.

The coupled model architectural aspects derived from scientific requirements are: (1) *connectivity*, the set all model-model interactions determined by the physics and solution algorithms of the coupled system; (2) *domain overlap*, the simulation space across which two or more models must exchange either driving or boundary condition data, which may be as simple as subsets of physical meshes, or as complex as interactions across spectral space or between Eulerian and Lagrangian models; (3) *coupling cycle*, the period over which all models in the system have exchanged data at least once; (4) *coupling frequency*, the temporal exchange rate for a given model pair; and (5) *tightness* the ratio of the effort (e.g., wall-clock time) by a component spent executing model-model interactions vs. integrating its own equations of evolution.

The coupled model architectural aspects describing software implementation choices are: (1) *component scheduling*, the order of execution for the individual models in the coupled system, which can be *sequential* (Figure ??(a)), *concurrent* (Figures ??(b) and ??(c)), or a combination thereof (e.g., Figure ??(d)); (2) *resource allocation* the number of processors and threads

allocated to each component in the system; (3) *number of executables*, the number of executable images in the coupled system—either single (Figures ??(a), ??(b), and ??(d)) or multiple (Figure ??(c)) executables; and (4) *coupling mechanism*, the way models exchange information, either directly or through an intermediate entity.

In a coupled model, the data exchanged by two components C_i and C_j resides on their overlap domain Ω_{ij} , and in principle each component will have its own discretization of Ω_{ij} . Thus, the parallel data processing challenge in coupled model development comprises the *description* of each component’s distributed mesh and field data on Ω_{ij} , its *transfer* between the processor pools and/or decompositions on which C_i and C_j reside, and its *transformation* for use by the other component.

The data description problem for component interaction comprises four elements: (1) each component’s spatial discretization of the overlap domain, (2) decompositions of the discretizations over the processor pools on which they reside, (3) lists of fields each component either sends or receives and how these fields are bound to their respective distributed discretized domains, and (4) time sampling of field data. The coupled model developer is confronted with a choice for each of these four issues—either work with the native representations used by each component, or impose standard descriptors on which data representations used by the components must be converted.

The data transfer problem involves three basic operations: *handshaking*, *message packing and unpacking*, and *communication*. Handshaking is the process of creating *communications schedulers* from the sending and receiving

components' domain decompositions. The message packing and unpacking process is part of the transfer mechanism and must be interoperable with the coupled system's common field data storage mechanism. Communication is the means of moving the needed data from one model to another.

The data transformation problem comprises two major elements: direct transformation of fields between the source and target components' spatiotemporal meshes, and variable transformations, which includes computation of a needed set of physical quantities for the target component based on a different set of physical quantities from the source component. Direct transformation is the straightforward interpolation in space and/or time between the source and target meshes and is most likely amenable to automation. Variable transformation is problem-specific and thus is best handled by parallel coupled model implementers.

2.2. Parallel Coupling in CCSM

CCSM contained specific examples of the constraints and considerations summarized in the previous section. We will summarize the aspects of the parallel coupling problem raised by CCSM. A more complete discussion is presented in (?).

CCSM imposed no requirements for internal data structures on its components, leaving the separate development groups of the atmosphere and ocean models to make their own choices. As a result, the models that make up CCSM have very different internal parallel data structures, often using Fortran90 derived types and nested derived types instead of arrays; the interface to the parallel coupler must be able to handle these disparate data

types.

Most of the other architecture decisions were imposed: the new coupler had to duplicate the architecture of its predecessor, including a central coupler, and communicate with multiple MPI/OpenMP parallel executables.

The scientific requirements of the simulation determine the coupling cycle and frequency and seldom change. In CCSM, the ocean model and coupler communicate once per simulated day, while communications between the atmosphere and coupler, land and coupler, and sea-ice and coupler occur once per simulated hour. Each model blocks while waiting for data from the coupler, thereby keeping the entire system synchronous in time within the cycle of once per day.

Careful consideration of the data dependencies between the models allows some overlap of communication with computation so that some of the models are integrating simultaneously in the course of one coupling cycle (component scheduling is concurrent execution).

The coupling frequency, cycle and component scheduling in CCSM are considered part of the science of the coupled system, and thus are hard-coded in the model. This eliminated any requirement for the new coupler or MCT to provide methods for selecting arbitrary coupling frequencies for any model-coupler pair. Nevertheless, there is some flexibility in choosing the length of the “fast” frequency—the hourly coupling between the atmosphere, land, and sea ice—in cpl6. And MCT does provide datatypes for time interpolation.

CCSM contained other requirements that the MCT software had to take into account. First, CCSM was already a widely used model with a flexible

and proven architecture; the addition of distributed memory parallelism in the coupler should not alter this system. Second, since the language of choice for all physical models in CCSM is Fortran90, the new coupler software had to have a Fortran interface. Third, the new coupler had to retain the portability of the model. CCSM currently runs on a wide variety of high-performance platforms, from commodity and microprocessor-based machines such as the IBM p690, SGI Altix, and Linux Clusters to vector architectures such as the Cray X1 and the Earth Simulator. CCSM has achieved this portability both by coding within the Fortran90 standard and by limiting its external package dependencies to only MPI and the NetCDF library. Fourth, CCSM is free under an open-source style license; hence, new software added to CCSM must also have no restrictions. Fifth, the new coupler had to allow continuation of the CCSM development philosophy where each component can be developed and used as a standalone executable by the subdiscipline developing the model. The requirements for CCSM's new coupler are described in more detail in ?).

3. The Model Coupling Toolkit

The important aspects of the parallel coupling problem outlined in the preceding section motivate a set of requirements for a software package to support the needs of developers of parallel coupled models. We have considered these requirements in creating a software package called the *Model Coupling Toolkit* (MCT), which can reduce dramatically the developer effort required to construct message-passing parallel coupled models.

Before we describe in detail the elements of MCT, we discuss the main

design decisions we made regarding implementation language, parallelism paradigm, and the reason we built a toolkit rather than a framework. Many of our decisions were motivated by the requirements for a parallel coupler for the CCSM described in Section ???. The choices we have made are intended to balance the interests of supporting the widest possible variety of applications with a fairly small and robust code base capable of achieving high performance on commodity microprocessor-based platforms and vector computers. Where possible, we have labored to provide developers using MCT maximum flexibility to make appropriate architectural choices for their applications.

We chose to implement MCT in Fortran90 because Fortran (meaning f77 and its successors) remains the most widely used programming language in scientific computing and is used in CCSM. We have adhered strictly to the Fortran90 standard because, at present, it is universally supported in commercially available compilers, whereas full support for the Fortran95 standard is less common and support for the emerging Fortran2003 standard is nearly nonexistent. Fortran90 has allowed us to implement MCT in a quasi-object-oriented fashion because it supports or allows emulation of object-oriented features such as encapsulation, data hiding, inheritance, and polymorphism (?, ?). Throughout our discussion of MCT, we will use these object-oriented terms, along with the terms *classes* and *methods* in this context.

The parallelization mechanism we have chosen to support in MCT is message passing using the Message Passing Interface (MPI), specifically version 1 of this standard (?). We chose MPI-1 because it is the most widely used approach for implementing parallelism in high-performance computing and

because most parallel platforms offer a vendor implementation of MPI as part of the overall environment.

We chose to build a toolkit and library in order to allow a maximum of flexibility to users with a minimum of modification to existing source code, a design philosophy also used by CCSM. Calling frameworks such as Earth System Modeling Framework (ESMF) (?) and the Common Component Architecture (CCA) (?) require their users to make substantial structural modifications to their legacy codes. In the case of CCA, one must write wrapper code to create components from each of the system's components, and at present CCA does not offer the variety of ready-to-use components required to solve the parallel coupling problem. ESMF is still under development; and in addition to the similar requirements imposed by CCA on potential users, it is unable to support parallel coupling of multiple executable models.

The MCT consists of nine classes that support parallel coupling. Three of these classes support data description, three support data transfer, and three support data transformation. Figure ?? illustrates MCT's class hierarchy. MCT provides a library of routines that manipulate these objects to perform parallel data transfer and transformation.

3.1. *Utility Layer*

The MCT is built on top of a utility package called *Message Passing Environment Utilities* (MPEU). MPEU was developed by the NASA Data Assimilation office to support their parallel operational data assimilation system, in particular the Physical-space Statistical Analysis System (?). MPEU supports Fortran90, MPI-based parallel codes by providing module-style ac-

cess to MPI, parallel support for `stdout` and `stderr` devices, parallel error handling and application shutdown, and parallel timing facilities including load imbalance metrics. MPEU also extends Fortran by providing some services analogous to the C++ Standard Template Library (?), including `String` and `List` datatypes and a MergeSort facility.

3.2. *Data Description*

The data description approach is based on the desire to represent a wide variety of meshes and domain decompositions with a minimal set of classes. MCT implements separate classes to encapsulate the domain decomposition, field data storage, and mesh descriptions. This choice allows reuse of the domain decomposition descriptor with multiple instantiations of the storage object and with the mesh description. MCT linearizes multidimensional meshes and field arrays, which simplified significantly the implementation of MCT's data model by allowing a one-dimensional representation of all data exchanged in parallel coupling.

3.2.1) DOMAIN DECOMPOSITION

Parallel domain decomposition in MCT is a combination of linearization (?; ?) and explicit strategies. Linearization is mapping from an array element's multiple indices to a single unique index. We employ linearization to yield a single unique index—a global ID number—referencing each element in a global array. The explicit part of the decomposition strategy arises from examination of how the local storage of the linearized array corresponds to the list of global ID numbers for the elements, compressing this index list into

segments of runs of consecutive ID numbers. In our domain decomposition strategy, we relax the requirement of other schemes that each element in the distributed array must reside on one and only one processor. Relaxing this requirement allows *masking* of elements and support for halo points. A point is masked if it resides on no processor. Masking is particularly useful because it allows for compact representation of points relevant only to coupling in situations where a component may organize data for an irregularly shaped overlap domain by embedding it in a larger regular multidimensional mesh. A point is haloed if it resides on more than one processor. MCT supports parallel transfer of data into a decomposition with halo points. It does not support transfer out of a haloed decomposition. MCT's linearized-explicit approach applies to arbitrary decompositions of arrays of any dimensionality.

This linearized-explicit decomposition strategy is embodied by MCT's `GlobalSegMap` class. The `GlobalSegMap` contains a global directory of segments of consecutive global ID numbers and the MPI process on which each resides. It also contains the component ID number for which this decomposition applies. This class has numerous initialization methods supporting the many fashions in which it is used in MCT, including initialization from index data residing only on the component's root process, distributed index data spread across the the component's processor pool, and index data replicated across the processor pool. Also provided are methods for global-to-local and local-to-global index translation, as well as query functions to determine total number of gridpoints stored globally, locally, or on a particular processor, and look-up of process ID on which a particular gridpoint is stored.

3.2.2) FIELD DATA REPRESENTATION

The parallel coupling problem can be viewed as a collection of *pointwise* operations involving multiple data fields. That is, values of multiple fields at gridpoint locations within the overlap domain are sent and received, interpolated, and otherwise transformed. MCT has a single field storage data object, called an *attribute vector*, which is implemented by the `AttrVect` datatype. This datatype is a fundamental type in MCT, forming a basis for other MCT datatypes that encapsulate physical mesh description (the `GeneralGrid`), time accumulation/averaging buffers (the `Accumulator`), and grid transformation data (the `SparseMatrix`) (Figure ??).

The `AttrVect` stores real and integer field data in a pointwise fashion within two two-dimensional arrays. The major index in both arrays is the attribute index, and the minor one the location index. This storage order places field data at a given location adjacent to each other in memory and increases the likelihood they will reside on the same cache line, a feature critical to maximizing on-processor performance on commodity microprocessor-based platforms. This storage order is not modified for vector platforms, which are accommodated by a modest amount of additional code and compiler directives in some of the manipulation methods for this class. Attributes are referenced and accessed by using user-defined character *tokens*. This approach has a number of desirable characteristics. It is *flexible* because the list of fields stored in the `AttrVect` can be set at run time. It allows for easy *extensibility* of application code because the parallel coupled model developer need only add a new field to a given `AttrVect` by adding an additional token

to the list of tokens supplied to its initialization call. Access to attributes based on tokens makes the **AttrVect** *indexable*. This quality makes application source code easier to read (because the tokens can be abbreviations of the physical field names), eliminates the possibility of errors from mistaken user-implemented indexing of field data, and enables automatic cross-indexing of fields shared by two distinct **AttrVects**.

The **AttrVect** has *initialization*, *destruction*, *query*, and *manipulation* methods. The initialization methods create data storage space in the **AttrVect** based on the number of integer and real attributes determined by *lists* of tokens. The numerous query methods return the number of datapoints (or *length*), the numbers of integer and real attributes, the data buffer index of a given real or integer attribute, and lists of real and integer attribute tokens. Manipulation methods for the **AttrVect** include zeroing its attributes, exporting (importing) a given attribute to (from) a one-dimensional array, and copying one or more attributes from one **AttrVect** to another. There are methods for sorting and permuting **AttrVect** entries by using a MergeSort scheme keyed by one or more the attributes of the **AttrVect**. MCT also provides an attribute cross-indexing method for mapping attributes stored in one **AttrVect** onto another. This cross-indexing method is used widely in the MCT's data transformation facilities.

MCT's view of parallel coupling involves communication of data stored in **AttrVect** format. Thus the **AttrVect** has a wide variety of communications methods, including point-to-point send and receive, and collective communications such as broadcast and gather and scatter using the **Glob-**

alSegMap. The MCT also provides global reduction methods analogous to `MPI_AllReduce()`.

3.2.3) PHYSICAL MESH REPRESENTATION

MCT’s linearized description of physical meshes requires a literal listing of each mesh point’s coordinates and geometric attributes. This is encapsulated in the `GeneralGrid` class. The `GeneralGrid` may be employed to store coordinate grids of arbitrary dimension, as well as unstructured grids.

The `GeneralGrid` stores real and integer gridpoint attributes internally in `AttrVect` form and inherits its query, access, and manipulation methods. Grid attributes stored are, at a minimum, coordinates for each gridpoint and one integer attribute—the *global grid point number*, which is a unique identifier for each physical grid location under MCT’s linearization scheme. Examples of real noncoordinate attributes that can be stored in the `GeneralGrid` include grid cell length, cross-sectional area, and volume elements and projections of local directional unit vectors onto Euclidian unit vectors. Commonly used integer attributes that can be stored in the `GeneralGrid` include alternative indexing schemes and indices for defining spatial regions. The `GeneralGrid` allows storage of real and integer grid-masking information as attributes. An *integer mask* can be used to exclude overlap domain grid points at which a component is not generating data (e.g., points on an ocean grid that correspond to large continental land masses). A real mask can be employed to indicate which fraction of a mesh cell is occupied by the component (e.g., fraction of an ocean cell occupied by sea ice).

The `GeneralGrid` is used for storage of length, area, and volume element

sizes in MCT's spatial integration and averaging facilities, described in Section ??, and is also used as a source of mask data in MCT's merging facility, described in Section ??.

3.3. Data Transfer

The MCT solution to the $M \times N$ problem has three stages: *registration* of components, *handshaking* of parallel data connections between components, and *execution* of the transfer. In MCT three classes support $M \times N$ transfers in parallel coupled models: a component registry (`MCTWorld`), communications schedulers for one-way parallel data transfers (the `Router`), and two-way data redistributions (the `Rearranger`).

A detailed discussion of MCT's solution to the data transfer problem is provided in ?) and is summarized here.

For a given grid decomposed over M and N processors, a `GlobalSegMap` can be constructed for each decomposition. Given these two `GlobalSegMaps`, one can build a communication table that lists, for a set of gridpoints in one `GlobalSegMap`, the corresponding locations in the other `GlobalSegMap`. In MCT, this table is stored in a `Router` datatype. For fixed grids, the `Router` is initialized once at startup. The process of exchanging `GlobalSegMaps` and building the `Router` table is MCT's handshaking between two parallel models.

MCT provides a two-sided message-passing model patterned after MPI. Instead of simple arrays and MPI processes ranks, the main arguments are `AttrVects` and `Routers`. These routines, called `MCT_Send` and `MCT_Recv`, transmit field data from the appropriate points of all of the data in the supplied `AttrVect` to the processors listed in the `Router`. In order to lower latency

costs, all the data for a given processor is sent/received in a single message. MCT also provides nonblocking versions of its $M \times N$ communication routines.

The `Router` and `MCT_Send()/MCT_Recv()` routines are for transferring data between models on disjoint sets of processors. The problem of redistributing data within a single pool of processors can in principle require each processor in the pool to both send and receive data. An example of this type of operation is the redistribution of data required for parallel data interpolation (see Section ??). MCT solves this problem by providing the `Rearranger` class to encapsulate the communications schedule for such operations, and the `Rearrange()` method for performing the redistribution.

3.4. *Data Transformation*

In Section ??, we identified the data transformation problem as consisting mostly of interpolation between different resolution grids or averaging in time to compensate for different time steps. Other transformations include spatially averaging outputs from two or more models to form the input for another—merging—and forming the global integral of a quantity. MCT provides classes and methods for all of these needs.

3.4.1) INTERPOLATION

A vital function in parallel coupling is transformation of data from one spatial mesh to another. Often a field value at a given location on a target grid is computed via a transformation that is a linear combination of field values on the source mesh using *interpolation weights*. When combined with MCT's linearization of multidimensional grid-spaces, these transformations

may be implemented as matrix-vector multiplication, and a field \mathbf{x} residing on the source component's mesh is transformed to a field \mathbf{y} on the target component's mesh by using an interpolation matrix \mathbf{T} :

$$\mathbf{y} = \mathbf{T}\mathbf{x}. \tag{1}$$

This approach appears in climate system model coupling software, most notably in CCSM (?; ?), the Ocean Atmosphere Sea Ice Soil (OASIS) model's flux coupler (?), and the Spherical Coordinate Regridding and Interpolation Package (SCRIP) (?). When one considers the typical stencil for these interpolation schemes, the result is an extremely sparse matrix-vector multiply, and this is the approach supported by MCT. MCT provides two types of infrastructure to aid this process: a basic data object for storage of interpolation matrix elements, and an object that encapsulates the complete set of computation and communication operations inherent in parallel sparse matrix-vector multiplication.

In MCT, elements of an interpolation matrix are stored by using the `SparseMatrix` class, which provides storage of nonzero matrix elements in coordinate (COO) format. Vector platforms are supported by an alternate internal storage scheme within the `SparseMatrix` that supports both compressed sparse row (CSR) and compressed sparse column (CSC) formats.

Methods for this class provide support for loading and unloading of matrix elements, counting of nonzero elements and determining sparsity, and sorting elements based on row and column indices. This element-sorting functionality is provided to improve performance of the matrix-vector mul-

tiply operation on commodity microprocessor-based platforms that rely on cache optimization.

For global address spaces (uniprocessor or shared-memory parallel), storage of matrix elements is sufficient to encapsulate the matrix-vector multiplication process. If one wishes to perform *distributed-memory parallel* matrix-vector multiplication, however, one must consider *communication*.

Three message-passing parallel strategies exist for computing (??). The first two decompose the problem according to the domain decomposition of \mathbf{y} or \mathbf{x} and are described by (?). The third method employs a user-defined decomposition of the elements of \mathbf{T} , which can be used to correct load imbalances in the compute part of the calculation, for example, decomposition of (??) using graph partitioning. In this scheme, the decomposition of the elements of \mathbf{T} determines two intermediate distributed vectors \mathbf{x}' and \mathbf{y}' , which allow an embarrassingly parallel calculation $\mathbf{y}' = \mathbf{T}\mathbf{x}'$. Communications occur first to assemble \mathbf{x}' from \mathbf{x} and, after the computation, to reduce the partial sums in \mathbf{y}' to the final result \mathbf{y} .

The entire parallel matrix-vector multiplication process is encapsulated in MCT's `SparseMatrixPlus` class, which contains both storage of distributed nonzero matrix elements in `SparseMatrix` format and instances of `Rearranger` communications schedulers needed to complete the parallel multiplication process (Figure ??).

The matrix-vector multiplication routines in MCT implement the solution of (??) by representing \mathbf{T} in either `SparseMatrix` or `SparseMatrixPlus` form and the vectors \mathbf{x} and \mathbf{y} in `AttrVect` form, allowing pointwise interpolation of

multiple data fields and automatic matching of attributes stored in \mathbf{y} with their corresponding attributes in \mathbf{x} . Vector platforms are supported by an additional matrix-vector multiplication function tuned to work with the CSC and CSR element tables stored in the `SparseMatrix`.

3.4.2) SPATIAL INTEGRALS AND AVERAGES

Conserving fluxes and maintaining constancy of spatial integrals and averages across the overlap domain are often desired in parallel coupled models. Interpolation between two different spatial discretizations of the overlap domain can, in principle, alter these results. MCT has routines to compute spatial integrals and averages. These functions allow the user to compute with ease global integrals and averages to test for and enforce conservation, as well as global diagnostics.

In MCT, the discrete versions of the spatial integral I and average $\bar{\Phi}$ of a field $\Phi(\mathbf{x})$ over domain Ω_{ij} are implemented as

$$I = \sum_{n=1}^N \Phi_n \Delta\Omega_n \quad (2)$$

and

$$\bar{\Phi} = \frac{\sum_{n=1}^N \Phi_n \Delta\Omega_n}{\sum_{n=1}^N \Delta\Omega_n}, \quad (3)$$

where N is the number of physical locations, Φ_n is the value of the field Φ at location \mathbf{x}_n , and $\Delta\Omega_n$ is the spatial weight (length element, cross-sectional area element, volume element, etc.) at location \mathbf{x}_n . MCT functions for computing these integrals take field data packaged in `AttrVect` form and thus are

capable of computing the same spatial integral for numerous fields simultaneously.

MCT functions for spatial integration and averaging also support *masked* integrals and averages. MCT recognizes both *integer* and *real* masks, and allows multiple masks to be used simultaneously. An integer mask M is a vector of integers (one corresponding to each physical location) with each element having value either zero or one. Integer masks are used to include or exclude data from averages or integrals. Masked integrals and averages are represented in the MCT by

$$I = \sum_{n=1}^N \prod_{j=1}^J M_n^j \prod_{k=1}^K F_n^k \Phi_n \Delta\Omega_n \quad (4)$$

and

$$\bar{\Phi} = \frac{\sum_{n=1}^N \left(\prod_{j=1}^J M_n^j \right) \left(\prod_{k=1}^K F_n^k \right) \Phi_n \Delta\Omega_n}{\sum_{i=1}^N \left(\prod_{j=1}^J M_n^j \right) \left(\prod_{k=1}^K F_n^k \right) \Delta\Omega_n}. \quad (5)$$

In (??) and (??), there are J (K) integer (real) masks, with M_n^j (F_n^k) representing the value of the j th integer (k th real) mask at grid location x_n .

MCT also provides *paired integral* and *paired average* facilities that allow simultaneous computation of the quantities defined in (??) and (??) on both the source and target discretizations to minimize global sum latency costs.

3.4.3) TIME SYNCHRONIZATION OF DATA

In addition to the spatial interpolation of field data, coupled models often also require *temporal* transformation of data between source and target

components' time meshes. Strategies for temporal transformation use either *instantaneous* or *accumulated* field values. If the time part of the parallel coupling problem is solved by exchanging instantaneous values, one or more instantiations of MCT's **AttrVect** class are sufficient to construct a solution. Coupling that uses accumulated data for time transformation requires *accumulation registers* for time summation or averaging of field data and the means to accumulate instantaneous field data values into these registers. MCT provides accumulation registers in its **Accumulator** class and routines for time accumulation of **AttrVect** field data into these registers.

The **Accumulator** stores real and integer field attributes internally in **AttrVect** form and inherits its query, access, and manipulation methods (Figure ??). In addition to accumulated field data, the **Accumulator** stores the length of the *accumulation cycle*, which is defined in terms of the number of time steps over which accumulation occurs, the number of time steps completed in the accumulation process, and the specific accumulation *action*. Currently, two options exist: time averaging and time summation. MCT provides a library routine `accumulate()`, which takes an **AttrVect** and accumulates any of its attributes that match the attributes of the **Accumulator**. This process is handled automatically by MCT's attribute cross-indexing facility. Currently, the **Accumulator** is designed to work with fixed timestep sizes, but this restriction still allows it to support many applications.

3.4.4) MERGING DATA FROM MULTIPLE COMPONENTS

The need to merge field data from multiple components arises when a component's overlap domains with two or more other components intersect

and the source components are providing one or more identical fields on the intersection domain. The merge occurs once the shared fields are interpolated onto the same discretization of the intersection domain. The merge is a weighted average of the field values at each mesh point on this domain.

MCT offers a **Merge** facility in the form of library routines that allow merging of data from up to four components for use by a fifth component. These routines work on the assumption that data is represented in **AttrVect** form and that each of these input arguments and the resulting merged **AttrVect** share the same domain decomposition, making the **Merge** operation embarrassingly parallel. Attributes of the input **AttrVects** are cross-indexed with those of the merge result **AttrVect** and are merged automatically.

MCT supports use of integer and real masks to weight data for the merge operation. To see this, consider the example of a merge of one field from two components for use by a third. Let the vectors \mathbf{a} and \mathbf{b} represent this field from components A and B that have been interpolated onto the physical grid of another component C . The merge operation combines the data from A and B , resulting in a vector \mathbf{c} , which represents the merged data on the grid of component C . This merge process is an element-by-element masked weighted average:

$$c_i = \frac{\prod_{j=1}^J M_i^j \prod_{k=1}^K F_i^k a_i + \prod_{p=1}^P N_i^p \prod_{q=1}^Q G_i^q b_i}{\prod_{j=1}^J M_i^j \prod_{k=1}^K F_i^k + \prod_{p=1}^P N_i^p \prod_{q=1}^Q G_i^q}. \quad (6)$$

In (??), data from component A has J integer masks ($\mathbf{M}^j, j = 1, \dots, J$) and K real masks ($\mathbf{F}^k, k = 1, \dots, K$), while data from component B has P integer masks ($\mathbf{N}^j, j = 1, \dots, P$) and Q real masks ($\mathbf{G}^k, k = 1, \dots, Q$). These masks

are optional and can be provided to the merge facility either in array form or as attributes in an MCT `GeneralGrid`.

4. Programming Model

The MCT programming model is based on its Fortran API and has three elements: access to MCT datatypes and routines through Fortran module use, declaration of variables of the class datatypes defined in Section ??, and invocation of MCT library routines to accomplish parallel data transfer and transformation. In this approach, the user writes the top-level control program(s) for the application, and any individual subroutines that are used to implement the components of the parallel coupled model. In this section, we provide examples of how some of the class datatypes and routines defined Section ?? are used.

Initialization of an MCT-based parallel coupled model occurs in two stages. The first stage is the initialization of MPI and the subsequent partitioning of `MPI_COMM_WORLD` into the set of communicators for each distinct processor pool. This communicator partitioning can be performed either through user-supplied calls to `MPI_COMM_SPLIT()` or through use of a communicator partitioning tool such as the Multi-Program Handshaking (MPH) utility (?). The next stage is the establishment of which component IDs are bound to the various processor pools. Examples of the various types of parallel coupling configurations MCT supports can be found in Figure ??. `MCTWorld_init()` is called to create the `MCTWorld` component registry.

```
call MCTWorld_init(nComponents, WorldComm, MyComm, CompID)
```

The supplied arguments are the total number of components in the parallel coupled model (`nComponents`), the global communicator for the overall model (`WorldComm`), the communicator for the pool on which a given processor resides (`MyComm`), and the component ID number(s) that execute on a given processor (`CompID`). For the configurations shown in Figures ??(b) and ??(c), and processors on `comm1` in Figure ??(d), there is one component per processor pool and the value of `CompID` is a scalar. The sequential configurations shown in Figure ??(a) and on `comm2` in Figure ??(d) have multiple components executing on a given processor, and the value of `CompID` is an array.

After the `MCTWorld` has been initialized, the user can create parallel coupling connections between components. Data exchanged in coupling is described by declaring variables of MCT’s descriptor datatypes—the `GlobalSegMap`, `GeneralGrid`, and `AttrVect`. For example, consider the case of a model that has one incoming and one outgoing data connection, with data supplied on the same physical grid and decompositions, but with different field storage data structures.

```

type(GlobalSegMap) :: MyDecomp
type(GeneralGrid)  :: MyGrid
type(AttrVect)     :: InputAV, OutputAV

```

The user must describe the discretization of the overlap domain and its domain decomposition. MCT’s linearized view of data and domain decomposition requires the user to create an element-numbering scheme for multidimensional arrays describing grid and field data. This numbering scheme

maps multiple indices to a single global ID index for each point in the domain. The scheme is then used to relate data in a multidimensional field array to a one-dimensional array suitable for description as an `AttrVect` and to map points from the corresponding multidimensional spatial grid to a `GeneralGrid`. Domain decomposition is then a set of segments of runs of consecutive global ID numbers. Each segment has a global starting index, a length, and the processor ID where it resides. These segment data are used to create the `GlobalSegMap`.

```
call GlobalSegMap_init(MyDecomp, starts, lengths, root, &  
                      MyComm, MyCompID)
```

In this call, `MyDecomp` is the `GlobalSegMap` created, `starts` and `lengths` are arrays containing local segment start and length values, `root` is the root for the communicator `MyComm` on which the decomposition exists, and `MyCompID` is the MCT component ID for this model. The call is a collective operation, and the result is a domain decomposition descriptor containing all the information needed to locate a given element and to perform global-to-local and local-to-global index translation.

Physical meshes are described by supplying the dimensionality of the mesh, coordinate names, gridpoint coordinate values, and associated weights such as length elements, cell cross-sectional areas, and cell volumes. For example, a `GeneralGrid` capable of describing Euclidean 3-space can be created as follows.

```

call GeneralGrid_init(MyGrid, 'x:y:z', &
                        WeightChars='dx:dy:dz:Axy:Axz:Ayz:V', &
                        LocalLength)

```

In this call, the tokens referencing individual coordinates x , y , and z are supplied in the second argument as a list, and the cell length, area, and volume elements are referenced by tokens given as a list in the third argument. The number of grid points residing on the local processor is defined by the argument `LocalLength`. This call creates a `GeneralGrid`—the argument `MyGrid`—that has three dimensions, and will allocate sufficient space to store gridpoint coordinates and grid cell weights. The coordinate and weight information is then loaded into `MyGrid` by the user.

Field data are stored as attributes in an `AttrVect` whose length is the same as the argument `LocalLength` supplied above in the creation of `MyGrid`. For example, suppose we wish to store two integer fields and three real fields in the `AttrVect` `OutputAV`.

```

call AttrVect_init(OutputAV, 'if1:if2', 'rf1:rf2:rf3', &
                    LocalLength)

```

In this call, the second and third arguments are lists of tokens used to identify integer and real attributes, respectively. The result of this operation is the `AttrVect` `OutputAV`, which is capable of storing two integer and three real attributes for `LocalLength` points. The actual field data is then moved into `OutputAV` by the user.

Once all the data description structures are initialized, one can perform parallel data transfer and transformation.

Parallel data transfer between concurrently executing components is described by using the `Router` data type. A `Router` is created by a call by each of the communicating components to `Router_init()`.

```
call Router_init(RemoteCompID, MyDecomp, MyComm, Route)
```

The first argument in this call is the MCT ID for the remote component participating in the data transfer. `MyDecomp` is a `GlobalSegMap` describing the local domain decomposition across the communicator `MyComm`, and the result is the `Router` object `Route`. Each of the pair of communicating components creates its own `Router` used to schedule the send (receive) operations for the transfer.

Once a `Router` has been constructed by each of the two components participating in an $M \times N$ transfer, data is exchanged through calls to MCT's parallel intercomponent communications routines as shown below. For the source component residing on M processors, `MCT_Send()` is called.

```
call MCT_Send(Model1_AttributeVector, Model1_Router)
```

The target component residing on N processors calls `MCT_Recv()`.

```
call MCT_Recv(Model2_AttributeVector, Model2_Router)
```

In each of these calls, the first argument is the `AttrVect` in which field data to be sent or received is stored, and the second argument is a `Router` that schedules the send or receive operations and the points on the domain for which data is being communicated.

5. Performance

The performance-sensitive parts of the parallel coupling process are data transfer and transformation. Parallel data transfer is a message-passing parallel process in which communications are the dominant cost. Data transformation algorithms by contrast have varying sensitivities to communications costs. MCT's `Accumulate` and `Merge` operations are embarrassingly parallel, and their performance is sensitive only to load imbalances resulting from disparities in the number of gridpoints assigned to each processor and to single-processor performance issues such as cache usage. The spatial integral and average routines have the same sensitivity to load balance as the accumulation and merge operations and are sensitive to the performance of the implementation of the `MPI_AllReduce()` used to perform the global sum. MCT's parallel interpolation routines are a combination of computation and communication and, as such, are the most interesting from a performance viewpoint.

A thorough analysis of the performance of MCT's `Router` initialization, parallel data transfer, and parallel interpolation facilities can be found in (?). This analysis has shown that MCT's parallel communication and interpolation routines scale well up to processor pool sizes likely to be used by parallel coupled models. (?) provide information on the performance of the MCT-based parallel data transfer and interpolation schemes in CCSM3.

Here, we present results for two other measures of MCT's performance as parallel coupling infrastructure—scaling of MCT's parallel interpolation facility to larger processor pools for a very large problem size, and overall

model throughput for CCSM at a typical resolution.

Performance results presented in this section were obtained with four different platforms: an IBM p690 (Bluesky) located at the National Center for Atmospheric Research, an HP Alpha Cluster (Lemieux) located at the Pittsburgh Supercomputing Center, a Linux cluster (Jazz) located at Argonne National Laboratory, and the Earth Simulator. Bluesky is an IBM p690 with 1,600 processors and an IBM Colony switch. Processors on Bluesky are 1.3 GHz Power4 processors, each of which has 2 GB of memory. Processors are grouped into shared-memory nodes, which can have either 8 or 32 processors, called 8-way and 32-way, respectively. Lemieux is an HP Alpha Cluster with 750 HP/Compaq Alphaserwer ES45 nodes that are connected by a Quadrics switch. Each node has four 1 GHz processors and 4 GB of shared memory. Jazz is a Linux cluster comprising 350 nodes connected by a Myrinet 2000 switch. Each node has one 2.4 GHz Intel Pentium Xeon processor and either 1 GB or 2 GB of memory. The Earth Simulator is a cluster of 640 nodes connected by a full crossbar switch. Each node has eight 500 MHz NEC SX-6 vector processors and 16 GB of shared memory. For all performance studies, we used the vendor Fortran compiler and MPI implementation. On Jazz, we used the Intel Fortran compiler and MPICH.

Figure ?? shows performance for a very high-resolution version of the MCT atmosphere-to-ocean interpolation benchmark. We present only this benchmark because it is representative of the data motion costs involved in both atmosphere-to-ocean and ocean-to-atmosphere interpolation. A more detailed discussion of differences between these benchmarks is presented in

?). The atmosphere data for this case reside on the CAM gaussian T340 grid, and the ocean grid is the POP 0.1° grid. This combination of resolutions may be used in future climate models. A set of 12 fields is interpolated hourly for ten model days (240 calls to the MCT interpolation routine).

Across all platforms tested, this benchmark shows good scaling to a processor pool size larger than what would normally be used for a centralized coupler in a climate model.

To application scientists and engineers, the ideal metric for assessing overhead imposed by a coupling mechanism is the amount of time components are forced to be idle while awaiting data from the coupler. In a concurrently scheduled model such as CCSM, it is difficult to measure this quantity or distinguish it from idling of a given component due to a sequential data dependency with another non-coupler component. Performance studies of CCSM indicate that the MCT-based CPL6 does not impose measurable overhead of this type. A possible explanation for this can be found by comparing scalability of the components' *throughput*—the amount of simulation achieved per unit of wall-clock time. Figure ?? shows throughput expressed in model years per wall-clock day on the Earth Simulator for set of component models either identical or similar to those in CCSM 3.0. The atmosphere model results are for CAM 2.02 with T85 horizontal resolution with 26 vertical layers. The ocean results are for POP 1.4.3 with a displaced pole grid with resolution of 1° in the longitudinal direction and finer variable latitude resolution that is approximately 0.3° at the equator. The land model is version 2.1 of the Community Land Model with T85 resolution. The sea ice model is version

3.1 the Los Alamos CICE model (as opposed to CCSM’s CSIM sea ice model) and uses the same horizontal grid as POP for these measurements. Note that the system’s MCT-based CPL6 coupler has throughput that scales dramatically better than the system’s other component models, demonstrating the efficiency of MCT’s services in this overall coupling mechanism.

6. Conclusions and Future Work

Parallel coupled models are now the state of the art in computational science and engineering. These models present numerous software engineering and algorithmic challenges, and the advent of distributed-memory parallelism has created a new challenge—the parallel coupling problem. We have described many facets of this problem and the requirements they impose on parallel-coupled-model developers.

We have described a new software package that aids coupled model development, the Model Coupling Toolkit, version 2.0. MCT offers a Fortran-based object model and a collection of library routines that dramatically reduce the effort required to couple separately developed message-passing-parallel component models into a single parallel coupled model. The MCT object model offers conceptual ease of use in creating coupling code. MCT’s library routines automate the most complex parts of the coupling process. This combination allows parallel coupled model developers to concentrate attention on the high-level issues in coupled model development, namely, what to couple, when to couple it, and what scientific issues to consider. We have demonstrated how MCT may be employed to accomplish fairly complex coupling operations with relatively little effort, with a reduced level of additional

introduced source code, and with great flexibility, while still ensuring high performance.

MCT is currently used in production by two major applications to couple geophysical codes: the CCSM flux coupler and the WRF coupling API. The CCSM coupler has been described extensively in a companion paper (?), and is part of version 3.0 of CCSM. An MCT-based version of the WRF coupling API exists and can be downloaded from the MCT Web site. This API has been used to create a wide variety of parallel couplings that support sequential and concurrent scheduling, multiple executables, and computational grid paradigms (Dan Schaffer, personal communication).

The version of MCT described in this paper reflects our goal of addressing the specific requirements of CCSM with the resources we had at our disposal. Future development of MCT will address its current limitations: a Fortran-based interface, a one-dimensional data model, and a purely distributed-memory parallelism paradigm.

Work is under way to export MCT's capabilities to other programming languages. We are using the Babel language (?) interoperability tool to create a limited set of prototype bindings for the C++ and Python programming languages. This work will be extended to provide access to MCT's classes and a core set of coupling services that will eventually be available to these and other programming languages.

The one-dimensional data model has great advantages in terms of universality but does impose on MCT users some effort in defining linearization schemes to describe multidimensional data in an MCT context. To address

this issue, we will first offer facilities to map multidimensional data onto MCT’s linear model and will then include additional classes for domain decomposition, data storage, and data transfer to offer users an easier-to-use data model.

The parallelism paradigm used in MCT will be expanded to support *hybrid* parallelism, specifically a combination of MPI for distributed-memory parallelism with OpenMP for shared-memory parallelism. This will have a direct effect on improving memory copies into and out of `AttrVects` and the compute-intensive services in MCT, specifically, the data transformation operations of interpolation, time averaging and accumulation, and MCT’s `Merge` facility. The enhancement will improve MCT’s scalability for data transformation operations and will provide users greater flexibility for placing MCT transformation operations in the coupled system and using all available computational resources.

Acknowledgements

We thank the many people who have offered advice throughout the design and development stages of MCT, specifically Anthony Craig, Brian Kauffman, Maria Vertenstein, Tom Bettge, and John Michalakes of the National Center for Atmospheric Research, and Ian Foster of Argonne National Laboratory. We gratefully acknowledge source code contributions from Jing Guo of the NASA Global Modeling and Assimilation office, who designed MPEU and the original version of the `AttrVect` class; and Jace Mogill and Celeste Cory of Cray Incorporated, who improved dramatically the performance of MCT’s parallel data transfer handshaking facility. MCT’s port to the Earth

Simulator and prototype vectorized interpolation facility were developed by Clifford Chen of Fujitsu America, Yoshi Yoshikatsu of Japan's Central Research Institute of Electrical Power Industry (CRIEPI), and Junichiro Ueno, Hidemi Komatsu, and Shin-ichi Ichikawa of the Computational Science and Engineering Center of Fujitsu, Limited, Japan. We thank Yoshi Yoshikatsu for MCT timings and CCSM throughput measurements on the Earth Simulator. We thank two anonymous referees whose helpful suggestions have improved this paper.

This work was supported by the US Department of Energy under the Accelerated Climate Prediction Initiative *Avant Garde* project and the Climate Change Prediction Program, which is part of the DOE Scientific Discovery through Advanced Computing (SciDAC) initiative under contract number W-31-109-ENG-38.

Figure 1: Basic types of communicator and component layouts for parallel coupled models: (a) sequential coupling, (b) concurrent coupling with a single executable, (c) concurrent coupling with multiple executables, (d) combination of concurrent and sequential coupling. The horizontal axis corresponds to system resources (e.g., MPI processes), the vertical axis time, and directed arrows denote caller/callee relationships.

Figure 2: MCT class hierarchy. Downward directed arrows point from child to parent class.

Figure 3: Timings of the MCT atmosphere-to-ocean interpolation benchmark on a variety of platforms.

Figure 4: Throughput of CCSM on the Earth Simulator.