Advanced Cluster Systems

SET™ v1.0

Supercomputing Engine Technology™

Reference Manual

U.S. Patents 8082289, 8140612, & 8402083,
Japan Patent 4995902, and Patents Pending

Table of Contents

Introduction

The Problem

Since the 1980s, HPC (High Performance Computing) explored and established the techniques needed to wield parallel computation—computing using many processors at the same time—to solve ever larger computational problems. Although HPC has reaped benefits for its industry and the most technically advanced parts of academia, the sciences, and government, it has failed, after many attempts, to carry its benefits to the mainstream software industry and non-expert programmers. A UC Berkeley publication stated: "Experience teaching parallelism suggests that not every programmer is able to understand the nitty gritty of concurrent software and parallel hardware." (Krste Asanovic et al. - "A View of the Parallel Computing Landscape" article, UC Berkeley ParLab, October 2009).

This separation mattered little before the 21st century, because mainstream computers had only a single processor in them, but today virtually all new computers have multiple computing cores. With the advent of multicores in desktop and laptop computing, tablets and mobile phones, a gap between the theoretical computing capability of computing hardware and the performance achieved by its software is growing exponentially. By 2012, it is well known that the typical industry software writer cannot efficiently use computational power beyond one processor's core.

The software industry cannot ignore this situation much longer. Desktop hardware has 24 virtual cores, and CPU manufacturers like Intel and AMD are experimenting with 80 cores per chip and more. Likewise, hardware manufacturers in HPC would like to grow their audience beyond their traditional HPC industry, so they need a way to make it practical for more software to run on their hardware.

The Solution - ACS'S SET™ (Supercomputing Engine Technology™)

SET provides a way for software writers to continue "thinking" in serial (i.e., continue writing for a single processor), yet with a minimum of glue code (i.e., code that does not change the program's functionality), utilize parallel computing in the most successful way found in HPC - Distributed Memory MPI. Application writers with a well-organized, modular code already take advantage of the clear organization of serial implementations of computations that is vital not only to code maintenance but also to any parallel computing technique. Because the application writers understand the nature of their code, particularly its data management needs, those writers are best suited to determine which parts of SET to apply to their code, to enable a full-fledged parallelized application.

Parallel Computing

Parallel computation occurs when multiple processing units are working at the same time. This general concept is divided into three categories:

Categories of Parallel Computing

Type of Parallel Computing	Characteristic architecture	Communication pattern	Possible applications
Independent computing	Many processing machines operating without interaction	No communication	single-processor problems
Distributed computing	One or a few centralized servers directing many slave machines	Master-slave	above plus brute-force problem space sampling
Cluster computing & supercomputing	Interdependent computing nodes coordinating work and data	Peer-to-peer and collective	above plus large problems with strong interdependence

The capabilities of each category of parallel computing build on those of the one before. Many personal computers being used in an office are implementing the independent computing category. Grid computing today typically implements a distributed computing model. Famous examples include SETI@home, Folding@home, and brute force RSA key breaking, where many client machines receive instructions from a central server or servers and report to that server whether they found the answer.

In the 1980s, scientists developed techniques to solve the largest, most complex problems they could conceive using parallel computing. Through much work, development, and trial and error, they also found that communication between different computing processors was required to address such problems. In 1994, the Message-Passing Interface (MPI) standard was established as a supercomputing platform-independent programming interface that could enable any complex problem to be performed on a parallel computer. Today, usage of MPI has since grown so that MPI has become the de-facto supported standard in clusters and supercomputers of all types, while almost all known problem types have been parallelized using MPI. In turn, SET™ adopts the parallel computing paradigm of MPI and its other features that made MPI so successful.

SET/Pooch

SET/Pooch is the clustering component of SET that makes it easy to build your cluster, run parallel computations on it, and keep it running. Pooch keeps its footprint small, not only on the hardware, but also on the operating system, by requiring the most basic features to be in place: a working operating system with an active TCP/IP connection to other compute nodes running SET/Pooch. It does not need special libraries installed in the OS to run. This has direct benefit to users and administrators because SET/Pooch will continue to operate given a variety of different variations in operating system and configuration. SET/Pooch solves the “cluster problem” of parallel computing: how to get your cluster working and keep your cluster running and working.

The SET™ Approach

SET helps solve the “application problem” of parallel computing and is the result of the culmination of experience writing parallel code for a variety of projects. The practical success of the distributed-memory message-passing paradigm of MPI made it clear that a similar paradigm was needed. The problem with MPI, as well as some of its alternatives, was that it intimidated too many because it expected too much of software writers.

Like the benefits of high-level programming languages over assembly language, SET offers a high-level (or simpler) language for common parallel computing scenarios and message-passing patterns. We do not mean that we simply pare down MPI. We mean that SET allows the software writer to focus on his or her application code while SET takes care of as much of the parallelism and other parallel computing issues as possible.

To do so, it becomes necessary for SET to understand more about the structure of the data being processed, particularly arrays. Most parallel computing applications processor substantial amounts of data, usually structured into many arrays. MPI, OpenMP, and most other parallel computing approaches make little attempt to maintain information on the organization of application data. The application must inform the API of these other approaches at the moment that API is invoked. SET, on the other hand, offers an API to create array structures of sufficient complexity in service to the application. Application code can process that data at will, but when any of that data needs to be rearranged or reorganized, particularly between processing elements, SET takes care of that communication and data rearrangement.

SET offers an API organized into several major sections. For most applications, the most important of these are:

- Array and Data Management - Applications can allocate and access arrays via SET™
- Application Execution - Application code to be run in parallel
- Collective - Common message passing patterns across all processing elements
- Divide and Conquer - Independent case-by-case execution of application code
- Guard-Cell Management - Maintenance of edges of partitioned data sets
- Element Management - Migration of data elements between processing elements

Other situations exist where more delicate control is needed or other application-specific situations:

- Communicator - Defining a subsets of execution and communication
- Matrix and Vector Manipulation - Basic linear algebra across the cluster
- Fourier Transform - Two- and three-dimensional Fourier transforms
- Parallel Disk I/O - Collating data for reading and writing or reading and writing scratch data in parallel
- Low-Level Message-Passing - For the experts, sending and receiving individual messages

We would much rather promote using high-level routines rather than direct message passing as the best approach for software writers new to parallel computing. Nevertheless we support both needs.

SET™ Execution Structure

Front End and Back End

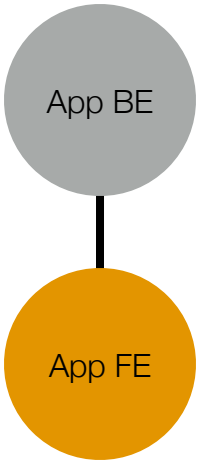
The SET™ method for parallel computing requires organizing the application into a Front End (FE) and Back End (BE). Since modern software practices mandate modular programming, where the software can be broken down into components that are theoretically interchangeable with other components with identical input and output, it should be straightforward to implement the separation of these software components of any modern application.

Front End

Front End application code is the part of the application that controls the overall execution of the application and manages input to and output from the application at a high level. This input and output includes an interface to the user, at a command-line or a GUI, as well as input and output files. The essential feature of the FE is that it directs the highest level of order of execution of the overall application. This could be the main() of a simple C code or the main event loop of a GUI application.

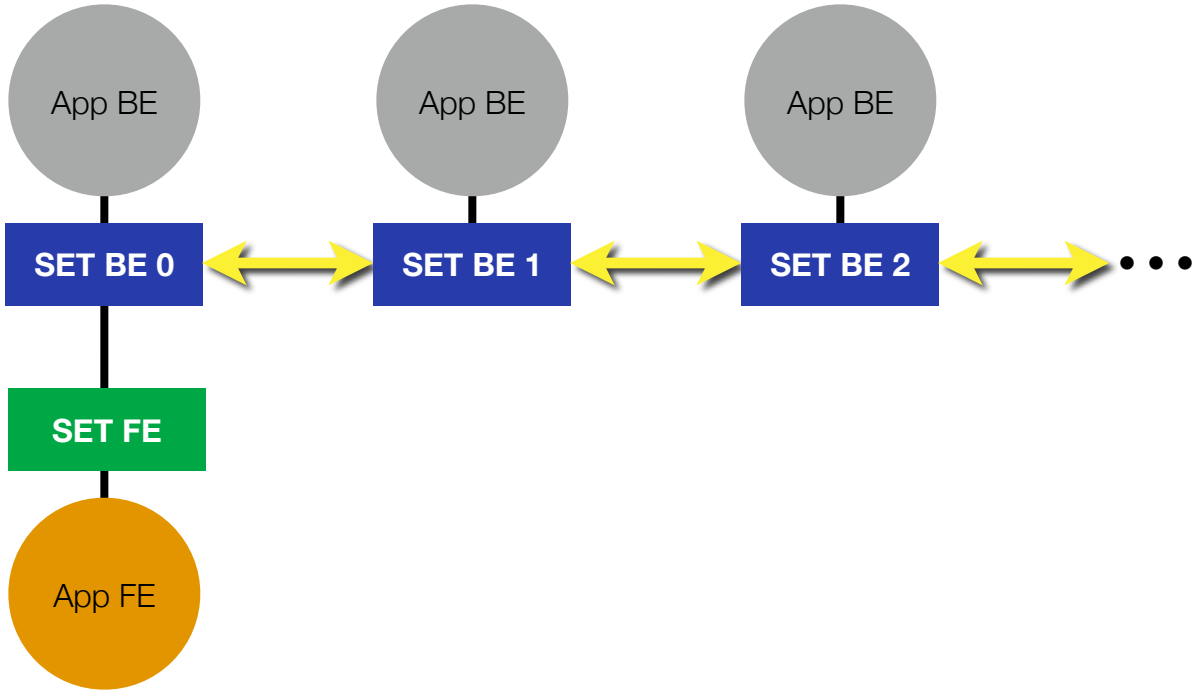
Back End

Back End application code is the part that performs the actual low-level work, including the level above the machine instructions that does the fundamental work of the application. In practice, the application's BE code includes the subroutines or functions that perform these basic application processes or algorithms.



***Application block diagram**. A well-organized (modular) application can be arranged into a Front End and a Back End. If the application is being controlled by a user, the FE would "face" the user.*

When executing on a single-processor, the FE code merely calls the BE code directly. A simple case is a main() (the FE) has an iteration loop, in which many other subroutines (the BE) are called. SET™ penetrates between the FE and BE of the application to provide parallelism. Because the application BE code should simply carry out the directive of the application FE code, it is therefore possible for that application BE code to execute on many processors simultaneously and (for that moment) independently.



***SET™ block diagram**. The organization of SET™ and application code into a Front End and many running instances of the Back End. The blue SET™ BE's and yellow arrows represent the Multiprocessor Parallelization Layer™ which in fact supports all-to-all communication (drawn as nearest-neighbor to simplify the illustration).*

The application FE, after directing SET™ to launch the BE code on the parallel computing system, will then direct, via SET™, execution of the application BE code and needed data communication between BEs. As in the application before SET™, the FE needs to provide parameters and input to the BE to operate, but again SET™ facilitates those actions as well.

Using SET™

Setting Up the Supercomputing Engine Technology

SET compiles with the application as a collection of libraries, and therefore does not require installation into the operating system. Simply copy the SET directory and its contents to a convenient, and writable, location on your hard drive. Pooch and SET write temporary files to /tmp/set in the course of normal operation. The libraries and header files needed to compile with the application are:

Application Front End

- setFE.h - SET Front-End header file
- setfe.a - SET Front-End library
- SETConstants.h - SET constants used by both FE and BE

Application Back End

- setBE.h - SET Back-End header file
- setbe.a - SET Back-End library
- SETConstants.h - SET constants used by both FE and BE
- fourier.a - SET Back-End Fourier transform dispatch
- libfftw3.a, libfftw3f.a, libfftw3l.a - (optional) Open Source FFTW, implementing the basic Fourier transform functions

We supply the source code the fourier.a in an accompanying Fourier source code folder, so that FFTW may be substituted for another preferred implementation. To remove the use of Fourier transforms entirely, make the fourier.a can be made a null dispatch. Because FFTW is Open Source and available for public, it is supplied unchanged.

Compiling with the Supercomputing Engine Technology

Compiling with SET is simply a matter of compiling the libraries with the application code. The most instructive examples are in the example code, particularly SETadder, SETpascalstriangle, and SETlife, which use, without moving them, the libraries and header files in a dedicated “SET” folder. Type “make” in either of those example code directories.

Nonetheless the basic template is:

Front End Compile

cc -o foofe foofe.c setfe.a •••

Back-End Compile

cc -o foobe foobe.c setbe.a fourier.a libfftw3f.a libfftw3.a libfftw3l.a •••

Where ••• represents addition operating-system-specific and context-specific libraries, include paths, or compiler options. Please see the code examples and their makefiles (makefileosx and makefilelinux) for Mac OS X- and Linux-specific details.

Compiling and Running on Classic Linux Clusters

To run on classic Linux clusters, such as those at major computing centers, the compile and run sequence is different because such clusters have their own MPI communication libraries and a specific launch sequence. Instead of setbe.a, the Back End uses these files:

- setbeC.a - SET Back-End library for classic Linux clusters
- SETMessagePassing.c - Message passing glue code to classic Linux clusters
- SETMessagePassing.h - Message passing glue code header for classic Linux clusters

This approach allows SET to use the native message-passing libraries of the classic Linux cluster. We provide glue code for use with a conventional MPI in SETMessagePassing.c. We provide a makefile to use mpicc to compile the application Back End together. SETMessagePassing.c contains routines that are basic wrappers over the classic MPI_Isend, MPI_Irecv, and MPI_Test, but not all such clusters are alike, so we expose SETMessagePassing and the makefile for needed customization. For example, shared memory systems can recreate MPI_Isend and MPI_Irecv with memory copies, and some clusters use a compile tool other than mpicc.

When running in this scenario, the computing job needs to specify the Back End executable as the primary executable and also supply Front End executable as an accompanying file in the same directory. For example, a cluster using mpirun would use something like:

mpirun -np 8 adderbe adderfe

because mpirun uses the first file as the executable and merely makes later files available to the first executable. At run time SET detects this scenario and adjusts, but only if the SETBEDidFinishLaunching() calls SETBEInitializationParameters() with a SETBEInitializeStruct specifying the name of the Front End executable. (Some applications may need to control their command-line arguments.)

Running with the Supercomputing Engine Technology

Once the code compiles and the application FE runs (simply run it, such as "./adderfe") and invokes SETInitialize with the path to the BE executable and the appropriate parameters, SET then launches the BE components on the cluster and connects them for runtime operation. Make sure your application FE calls SETFinalize to properly shut down all BE modules. In the primary implementation, SET uses Pooch to run on the cluster, whose installation is quite simple.

Note: Use the same procedure on a single Mac or single Linux PC; each of these is a one-node cluster.

Installing SET/Pooch

Macintosh

SET/Pooch is very easy to install. Just double-click the Pooch Installer on each node where the application can be run. For command-line installation, use the poochclinstaller.tar.gz tar ball. More information at: http://daugerresearch.com/pooch/

64-bit Linux

SET/Pooch on Linux begins with copying the poochinstaller.tar.gz file to the node running 64-bit Linux. The steps and commands are:

1. Copy the poochinstaller.tar.gz file to the Linux machine (e.g., using scp or sftp)

2. Log in to the Linux machine (e.g., ssh) to issue the following commands at the directory of the poochinstaller.tar.gz:
3. tar -xzf poochinstaller.tar.gz
4. cd poochinstaller
5. su
6. - enter your password for root access -
7. ./install
8. exit
9. logout

Pooch should then be installed on this Linux machine and running. These are located in sbin/ and bin/ of /var/pooch/.

See http://daugerresearch.com/pooch/ for further details.

Examples

Model-View-Controller and SET

Modern applications are modular, and most of these follow a widely-used paradigm called “Model-View-Controller” (MVC). In many cases, even if the software writer of an application was not aware of the terms Model, View, or Controller, one can recognize, in hindsight, this paradigm applied in that application’s code. The View is the part of the code that shapes the overall execution of the code and expresses that execution to the user, as input and/or output. The Model is the code or collection of modules or functions that actually manipulates the data at a low level. The Controller connects or bridges the Model and View.

To adapt a MVC structured application to SET™, the Model would reside in the Back-End, the View in the Front-End, while the Controller would be split between Back-End and Front-End and adapted to work with SET™. In virtually all practical cases, the Model and View portions of the code are unchanged, and the code to add to the Controller to work with SET™ is a small fraction of the overall project.

Template to Enable Applications for SET

The code examples included with SET™ illustrate the simplest possible structure to organizing a parallel application for running under SET™. It consists of two portions of code:

- Front-End source code containing:
  - the application’s Front-End or View code, which calls Controller code
  - modified Controller code consisting largely of calls to SETInitialize(), SETFinalize(), and other SET™ calls
- Back-End source code containing:
  - the application’s Back-End or Model code
  - glue code or modified Controller code which call unmodified application Back-End or Model code
  - required SET™ functions SETBEDidFinishLaunching() to register the glue code with SET™ and set up application-specific services, if needed, and SETBEWillTerminate() to tear down application-specific services, if needed

The included code examples are, by design, simple enough to implement in just two .c files, but in principle many files of source can be linked either on the Front-End, Back-End, or both by modifying the makefile.

That is the overall structure of an application enabled for parallel computing using SET™. The overall control of the SET™-enabled application remains in its Front-End or View, and the application-specific low-level code that performs the “real work” of the code resides in the Back-End or Model, but the modified Controller interfaces the two parts with the help of the SET™ API. Below are examples meant to be illustrative of how to parallelize an application using SET™.

Seeing the Pattern

The first step towards parallelizing a code is to recognize what kind of pattern of data movement and computation is occurring in that code. The following code examples give three illustrative examples of moving a serial code to be parallelized with SET™. Once the application writer identifies what part of SET™ corresponds to their code’s data movement and execution needs, it is a simple matter to connect these components together. The following examples are meant to be illustrative of how to parallelize using SET™, and the reader is invited to examine the example code distributed with SET™ alongside the description below.

Adder

SETadder: The adder.c example is a simple divide-and-conquer application. A function is called many times over a range of integers and returns a unique result in each instance. Adder collects those results, saves them to a file, and adds them together. adder.c has three functions:

- kernelroutine() - the function that performs the work
- computeloop() - calls kernelroutine() over a range of integers and manages the data
- main() - calls computeloop() and saves its output to a file

To parallelize with SET™, we identify that main() is FE code, and kernelroutine() is BE code, so clearly these two will reside, without modification, in adderfe.c and adderbe.c, respectively.

computeloop() serves to interface between the FE and BE, so in the serial code it is the glue code between the application's FE and BE. Its purpose is split between the adderfe.c and adderbe.c.

In adderbe.c, we simply need to inform the SET BE about kernelroutine(), so we add to SETBEDidFinishLaunching() merely the line:

```
out=SETBERegisterProcedure(set, "kernelcaller", kernelcaller);
```

In adderfe.c, computeloop() is altered perform the following tasks:

1. Create the SET environment while identifying the BE executable via SETInitialize()
2. Use SETFunctionToArray() to call kernelcaller() many times and generate a distributed array

```
err=SETFunctionToArray(set, "array", sizeof(double), setDoubleDataType,
   "kernelcaller", HighestIndex);
```

3. Use SETGather() to gather that generated data together

```
err=SETGather(set, "gatheredArray", "array", 0);
```

4. Use SETGetBEtoFE() to bring the gathered data to the FE

```
err=SETGetBEtoFE(set, &length, &myArray, "gatheredArray", 0);
```

5. Perform the addition, shut down the SET environment, and return the array to its caller.

This example illustrates perhaps the simplest use of SET, to parallelize a computation with many independent results and collect that output. It is possible that specific segments of an application will fit this divide-and-conquer paradigm.

Pascal's Triangle

SETpascalstriangle: The pascalstriangle.c example shows how to use SET to support the simplest dependency between processing elements during a computation. The classic Pascal's triangle calculation has the new element of the next row be the sum of the nearest two elements of the last row. It generates what is known as a binomial sequence, and a side-effect of the pattern that forms is a fractal known as a Sierpinski gasket. pascalstriangle.c has these functions:

- addright(), addleft(), and iteration() - the functions that perform the binomial work
- computeloop() - allocates and initializes data, calls iteration() repeatedly until the calculation is complete

- main() - calls computeloop() and saves its output to a file

To parallelize with SET™, we identify that main() is FE code, so main() is copied to pascalstrianglefe.c, while addright(), addleft(), and iteration() is BE code, so these will reside, without modification, in pascalstrianglebe.c.

computeloop() prepares the data for computation and manages the computation besides being the interface between the FE and BE, so again it is split between the FE and BE.

In pascalstrianglebe.c, we add two routines:

- initialize() - seeds the data set with the initial “1” in the correct location
- iterate() - serves as the “glue” between the SET array and the unmodified iteration() function

and inform the SET BE of these two functions in SETBEDidFinishLaunching().

In pascalstrianglefe.c, computeloop() is altered:

1. Create the SET environment while identifying the BE executable via SETInitialize()
2. Use SETNewArray() to create an array of the correct dimension and properties, particularly set1GuardCellLayer so that SET knows it has one guard cell.

```
err=SETNewArray(set, "array", sizeof(myType), setInt32DataType|
   set1GuardCellLayer, 1, (binomalExponent+nproc+1)/nproc);
```

3. Use SETCall() to call initialize to seed the array

```
err=SETCall(set, "initialize(array)");
```

4. In a loop, alternate between calling SETEdgeCell() to maintain the edges of the array and SETCall() to perform the calculation

```
err=SETEdgeCell(set, "array");
err=SETCall(set, "iterate(array)");
```

5. Use SETGather() to gather that generated data together

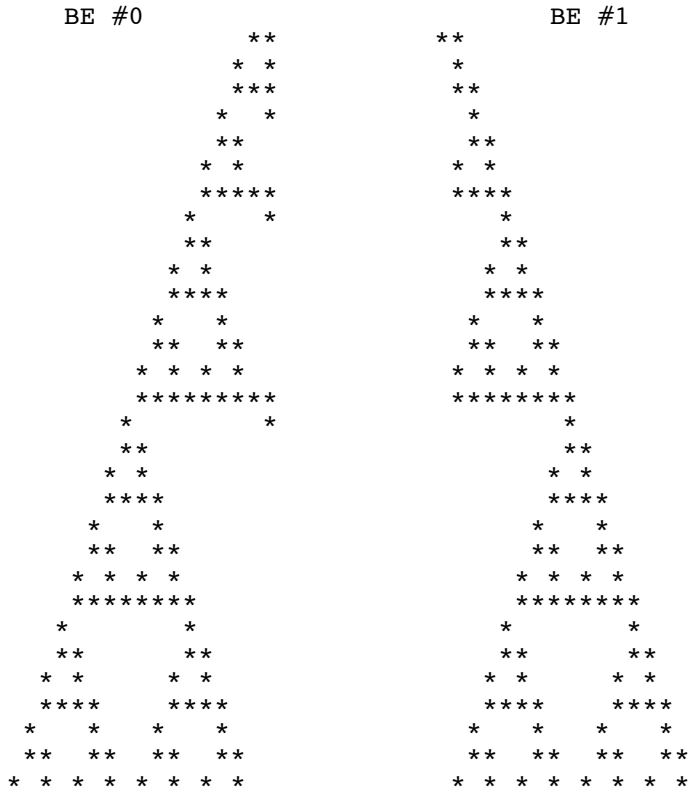
```
err=SETGather(set, "gatheredArray", "array", 0);
```

6. Use SETGetBEtoFE() to bring the gathered data to the FE

```
err=SETGetBEtoFE(set, &length, &myArray, "gatheredArray", 0);
```

7. Shut down the SET environment and return the array to its caller.

Running this code on two computing cores produces this output for each core:



Note that the two BE's have printed their half of a Sierpinski gasket. That would not have been possible if the BE's could not communicate.

This example illustrates nearest-neighbor message-passing in SET, necessary to support each step of the calculation. The iterate() function performs the calculation, but the SETEdgeCell() call maintains the boundaries between partitions of the calculation. This illustrates an interdependency that can occur between portions of a problem run on many processing elements.

Conway's Game of Life

SETlife: The life.c example shows how to use SET to support the nearest-neighbor dependency for a two-dimensional problem between processing elements during a computation. This classic computation of cellular automata created by John Horton Conway has the life of one cell at the next time step be dependent its life before and its immediate neighbors in a square grid. life.c has these functions:

- propagatelife() - the function that performs the life calculation
- maintainboundaryconditions() - maintains periodic boundary conditions in the space of the problem
- fprintresult() - prints the status of life on the grid
- addmaterial() - adds life material, to be called if a region becomes too static
- throwoutrand() - initializes the pseudorandom number generator

- performcalculation() - allocates and initializes data, calls maintainboundaryconditions(), propagatelife(), and fprintresult() repeatedly until the calculation is complete while monitoring the condition of the space if addmaterial is needed()
- main() - calls computeloop() and saves its output to a file

To parallelize with SET™, we identify that main() is FE code, so main() is copied to lifefe.c, while propagatelife(), maintainboundaryconditions(), fprintresult(), addmaterial(), and throwoutrand() is BE code, so these will reside, without modification, in lifebe.c. Also a life.h header file is added to declare the Byte data type in both FE and BE.

performcalculation() prepares the data for computation and manages the computation besides being the interface between the FE and BE, so it is split between the FE and BE.

In lifebe.c, we add four routines:

- preparerand() - calls throwoutrand() with an amount unique to each BE so that the pseudorandom numbers are relatively unique to each BE
- initialize() - seeds the cells set with initial random life and creates internal indicies
- maintainBCs() - serves as the “glue” between SET unmodified maintainboundaryconditions() function
- setpropagatelife() - serves as the “glue” between SET unmodified propagatelife() function

and inform the SET BE of these functions in SETBEDidFinishLaunching().

In lifefe.c, performcalculation() is altered:

1. Create the SET environment while identifying the BE executable via SETInitialize()
2. Use SETCall() to call initialize to initialize the pseudorandom number generators on all BE’s

```
err=SETCall(set, "preparerand");
```

3. Use SETNewArray() to create two arrays, as double buffers, of the correct dimension and properties, particularly set1GuardCellLayer so that SET knows it has one guard cell. Each array is called with this prototype:

```
err=SETNewArray(set, "lifeArrayA", sizeof(Byte), setInt8DataType|
   set1GuardCellLayer, 2, RowDimension+2, ColumnDimension);
```

4. Use SETCall() to call initialize the arrays

```
err=SETCall(set, "initialize(lifeArrayA, lifeArrayB)");
```

5. Here the cycle of life begins. Each iteration, the double buffers are alternated, so that A copies to B, then vice-versa. The names of these arrays are merely alternated at each time step.
6. Use SETCall() to maintain the boundary conditions in the direction orthogonal to the axis of parallelization.

```
sprintf(command, "maintainBCs(%s)", lastArray);
err=SETCall(set, command);
```

7. Use SETEdgeCell() to maintain the boundary conditions between BE’s

```
err=SETEdgeCell(set, lastArray);
```

8. Use SETCall() to propagate life from the previous array to the next array

```
sprintf(command, "propagatelife(%s, %s)", nextArray, lastArray);
err=SETCall(set, command);
```

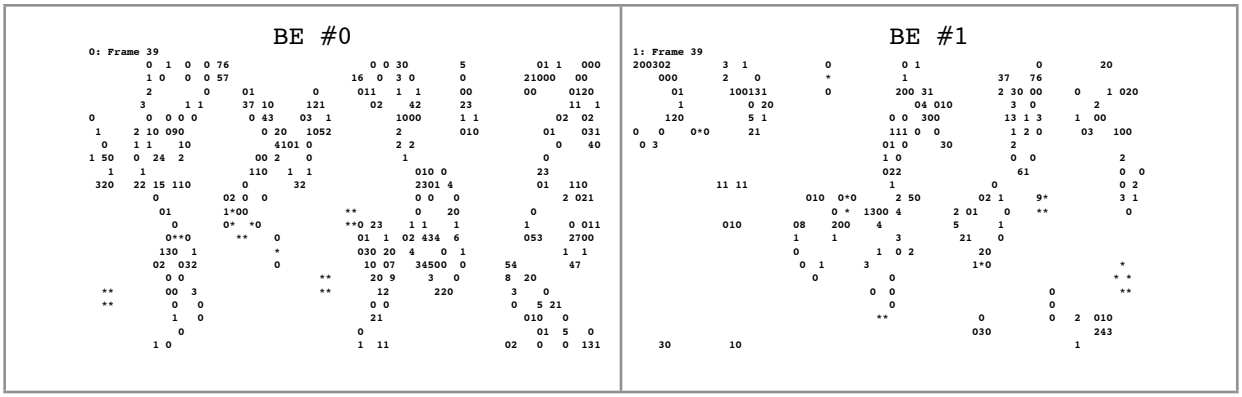
9. Use SETCall() to print life on each processor, but inside it also checks for static conditions and adds material if needed

```
sprintf(command, "printlife(%s)", nextArray);
err=SETCall(set, command);
```

10. After some boilerplate iteration code, the cycle of life repeats

11. Shut down the SET environment and return the array to its caller.

Running this code on two computing cores produces output showing the evolution of life on each core. For example, at this frame:



Note that the two BE's have printed their half of the calculation and that the left edge of one BE's space interfaces with the right edge of the other. The SETEdgeCell call maintains those edge conditions in between computation.

This example illustrates nearest-neighbor interaction on a two-dimension problem in SET as needed to support each step of the calculation. The iterate() function performs the calculation, but the SETEdgeCell() call maintains the boundaries between partitions of the calculation. This is a more visual example that illustrates an interdependency that can occur between portions of a problem run on many processing elements. This kind of partitioning also scales well because the amount of computation time is proportional to the volume of the problem, but the communication time is proportional only to the "surface area" of each problem, so as the problem grows larger, the communications time becomes small relative to the computation time.

Function Reference

Calling the Supercomputing Engine Technology

The SET calls are divided into several general categories, depending on type. Some regard code execution (such as Application Execution and Divide and Conquer) and some regard data movement (such as Collective and Element Management). We encourage users of SET to work with the high-level calls rather than build from low-level message passing.

The format of the reference below begins with the name, a short description, the function prototype, and then a detailed description of its features. Note that some functions, constants, and data structures are available from only:

- the Front End, identified with FE or F or
- the Back End, identified with BE or B or
- both, identified with “F and B”.

The end of the short description labels these as such.

All data types conform to the conventions defined in <stdint.h> for compatibility and consistency with both 32-bit and 64-bit computing. Generally integers in structures are defined by explicit sizes (int32_t, uint32_t). Integers outside structures, such as in function return values, can be int. Buffer lengths are generally defined as long to support both 32-bit and 64-bit buffer lengths.

SET Execution and Environment

To create the SET runtime environment and beginning running application BE code via SET, the application FE needs to open the SET FE with SETInitialize. SETFinalize performs the opposite effect. The following routines identify the how to create, edit, and destroy the SET environment.

SETInitialize

Start up the SET environment - F

```
SETRef SETInitialize(SETInitializePtr parameters);
```

Parameters

parameters Pointer to a SETInitializeStruct (defined below) containing parameters specifying the path to the BE executable and other parameters of the execution environment.

Return Value

A new opaque SETRef object on success or NULL on failure.

Discussion

Launches the specified BE executable on the system on the nodes as specified in the SETInitializeStruct and prepares the communication network between BE’s and FE. Upon success, it returns the newly allocated SETRef object. The contents of the given SETInitializeStruct may be modified on return. It is okay if both SETInitialize() and SETBEInitializationParameters() (below) is present in the application code. If SETBEInitializationParameters()

(below) is called first, as it would in a classic Linux cluster, SETInitialize() detects this condition and does not launch additional BE executables.

Declared In

setFE.h

SETFinalize

Stop and deallocate the SET environment - F

```
int SETFinalize(SETRef set);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

Return Value

An int with the value of 0 on success or an error code on failure.

Discussion

Tears down the SET execution environment, including notifying all running BE code that began due to the corresponding SETInitialize call to quit. The SETRef object is deallocated and its value should no longer be used.

Declared In

setFE.h

SETSetParameter

Set a parameter about the SET environment's execution at run time - F

```
int SETSetParameter(SETRef set, uint32_t selector, void *parameter);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

selector Which parameter of the SET environment to change

parameter Pointer to an object containing the new value

Return Value

An int with the value of 0 on success or an error code on failure. If set, selector, or parameter are invalid, parameterErr is returned.

Discussion

Changes the parameter chosen by selector to the new target value, if possible.

Declared In

setFE.h

SETGetParameter

Get the value of a parameter of the SET environment - F

```
int SETGetParameter(SETRef set, uint32_t selector, void *parameter);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

selector Which parameter of the SET environment to access

parameter Pointer to an object that receives the value

Return Value

An int with the value of 0 on success or an error code on failure. If set, selector, or parameter are invalid, parameterErr is returned.

Discussion

Accesses the parameter chosen by selector and writes it to the pointer given in parameter.

Declared In

setFE.h

SETBEInitializationParameters

Supply parameters to start up the Front End and SET environment, if needed - B

```
int SETBEInitializationParameters(SETRef set, SETBEInitializePtr parameters)
```

Parameters

set An opaque SETRef object previously given via SETBEDidFinishLaunching().

parameters Pointer to an object that with specific parameters, such as the name of the Front End executable

Return Value

An int with the value of 0 on success or an error code on failure. If set or parameters are invalid, parameterErr is returned.

Discussion

This routine is needed only in a classic Linux cluster scenario, but it is okay to call from SETBEDidFinishLaunching() in any cluster launch scenario and SET will take appropriate action. This routine's most important purpose is to supply the name of the Front End executable to the SET BE #0 via a SETBEInitializeStruct. The application Front End still needs to call SETInitialize(). If SET detects that it has no Front End running, SET BE #0 will launch the Front End executable specified here and connect with it so that the complete SET process can occur. The application code can be written without regard to these details and will operate in either a Pooch cluster or a classic Linux cluster.

Declared In

setBE.h

Array and Data Management

The application can ask SET to create arbitrarily-sized arrays. While the FE is responsible for management of these arrays, data in these arrays are primarily accessible to the application BE code. The FE can then direct SET to process or reorganize these arrays as needed to support parallel computation. Also, the use of the word “array” is a generalization. Some unchanging data makes sense to define in a large single-element array.

SETNewArray

Create and allocate a new array of elements across the cluster - F and B

```
int SETNewArray(SETRef set, char *name, int elementSize, int32_t flags, int
      dimensionCount, int dimension0, ...);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

name A string containing the name of the new array.

elementSize The size, in bytes, of the each element of the array.

flags A bit mask specifying other characteristics of the array and its elements.

dimensionCount The number of dimensions of the array. Must be greater than zero.

dimension0, ... A list of the dimensions of the array, starting with the lowest dimension (lowest stride between elements).

Return Value

An int with the value of 0 on success or an error code on failure. If set, name, or elementSize are invalid, parameterErr is returned.

Discussion

Create and allocate a new array across the cluster. Clears array to 0 upon creation. Calling from the BE has only a local effect.

Declared In

setFE.h, setBE.h

SETDisposeArray

Deallocate an array, across the cluster if from the FE, only locally if from a BE - F and B

```
int SETDisposeArray(SETRef set, char *name);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

name A string containing the name of the array to be deallocated

Return Value

An int with the value of 0 on success or an error code on failure. If set or name are invalid, parameterErr is returned.

Discussion

Deallocate an array, across the cluster if from the FE, only locally if from a BE. As a safety feature, calls by BE on arrays created by FE are rejected and vice-versa.

Declared In

setFE.h, setBE.h

SETCopyArray

Copies an array, its parameters and its contents - F and B

```
int SETCopyArray(SETRef set, char *target, char *source);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

target A string containing the name of the new array or array to be replaced.

source A string containing the name of the array to be copied.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Copies a source array to an existing target array across the cluster if from the FE, only locally if from the BE

Declared In

setFE.h, setBE.h

SETNewSimpleArray

Copies an array, its parameters and its contents - F and B

```
int SETNewSimpleArray(SETRef set, char *name, int32_t flags, long
      inputLength, void *input);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

name A string containing the name of the new array.

flags A bit mask specifying other characteristics of the array and its elements.

inputLength Length in bytes of the data pointed by input.

input Data provided by caller to copy to the new array.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Create and allocate a simple array on the cluster, duplicating FE input data for each processor if from the FE, only locally from the BE. When calling from the FE, this call is largely the same as calling SETNewArray then using SETPutFEtoBE on every target BE to replicate the same source data on every BE.

Declared In

setFE.h, setBE.h

SETNewSimpleArray

Copies an array, its parameters and its contents - F and B

```
int SETNewSimpleArray(SETRef set, char *name, int32_t flags, long
      inputLength, void *input);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

name A string containing the name of the new array.

flags A bit mask specifying other characteristics of the array and its elements.

inputLength Length in bytes of the data pointed by input.

input Data provided by caller to copy to the new array.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Create and allocate a simple array on the cluster, duplicating FE input data for each processor if from the FE, only locally from the BE. When calling from the FE, this call is largely the same as calling SETNewArray then using SETPutFEtoBE on every target BE to replicate the same source data on every BE. This is a convenient way to provide application-specific parameter data to every BE.

Declared In

setFE.h, setBE.h

SETPutFEtoBE

Puts data from the FE into an array of one BE - F

```
int SETPutFEtoBE(SETRef set, char *name, int target, long inputLength, void
      *input);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

name A string containing the name of the existing array to write to.

target The processor ID of the BE to access in the current communicator.

inputLength Length in bytes of the data pointed by input.

input Data provided by caller to copy to the targeted array.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Puts data from the FE into an array of one BE whose processor ID in the current communicator is target.

Declared In

setFE.h

SETGetBEtoFE

Gets data from the array of one BE, copying it in the FE - F

```
int SETGetBEtoFE(SETRef set, long *outputLength, void **output, char *name,
      int target);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

outputLength Pointer to a long which will hold the size in bytes of the returned buffer

output Pointer to a void* where the pointer to the returned buffer will be placed. Note that the caller is responsible for calling free() on the returned buffer when finished with it.

name String holding the name of the array to be accessed.

target The processor ID of the BE to access in the current communicator.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Gets data from the array of one BE whose processor ID in the current communicator is target, copying it to a buffer in the FE. The pointer to that buffer is placed into the void * pointed to by output, and the length of the

buffer is in the long pointed to by outputLength. The caller is responsible for calling free() on the output buffer when finished.

Declared In

setFE.h

SETGetLocalArrayAccess

Get parameter structure for accessing an array - B

```
int SETGetLocalArrayAccess(SETRef set, char *array, SETAccessorPtr *access);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array Pointer to a string identifying the array

access Pointer to a SETAccessorPtr containing all needed parameters to access the array.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Returns parameters for accessing an array (or its subarray) identified by the string array. It returns a SETAccessorPtr that can be used in other accessors such as ArrayAccess() to read or write elements of the array. Caution: Do not deallocate or free the returned SETAccessorPtr or modify its fields; that is the responsibility of SET.

Declared In

setBE.h

ArrayAccess

Get parameter structure for accessing an array - B

```
ArrayAccess(a, x, type)
```

Parameters

a A SETAccessorPtr returned via SETGetLocalArrayAccess or SETCall.

x Index of the array element.

type Data type of the element.

Discussion

This is the primary recommended way to access elements of the arrays managed by SET. Elements can be read (y = ArrayAccess(a, x, type)) or written (ArrayAccess(a, x, type) = y). Multidimensional arrays are accessed

using “LinearFrom#”. For example, a 3-dimensional array of double’s can be accessed using ArrayAccess(array, LinearFrom3(array, x, y, z), double). Uses a #define for speed.

Declared In

setBE.h

SETResizeLocalArray

Get parameter structure for accessing an array - B

```
int SETResizeLocalArray(SETRef set, char *array, int32_t flags, int
      dimensionCount, int dimension0, ...);
```

Parameters

set An opaque SETRef object.

array Pointer to a string identifying the array

flags A mask of flags indicating new properties of the resized array.

dimensionCount The number of dimensions of the new array. Must be greater than zero.

dimension0, ... A list of the dimensions of the new array, starting with the lowest dimension (lowest stride between elements).

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Resizes the local array in any particular dimension. Resize is centered around the (0,0,...) origin by default. Smaller amounts destroys data, newly created regions are set to 0.

Declared In

setBE.h

Application Execution

The application FE can call SET to execute any code of the application BE registered with SET. Below are the calls to enable this execution of application code.

SETCall

Gets data from the array of one BE, copying it in the FE - F

```
int SETCall(SETRef set, char *command);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

command A string of the form foo(array1, array2, ...) specifying the registered function and the arrays it operates on to be executed on all BEs in the communicator.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Calls a registered BE procedure (via SETBERegisterProcedure) by name with the arrays given in the string. command is a string of the form foo(array1, array2, ..). If the procedure or any array is not found, a non-fatal error is returned and the effect is a no-operation. The routine in the BE is of the form foo(SETRef, SETAccessorPtr, SETAccessorPtr, ...) (see SETFunctionPtr) and must be registered by each BE using SETBERegisterProcedure prior to being called. It is also possible for such functions to access other named SET arrays not provided by the FE using SETGetLocalArrayAccess, if that BE knows its name.

Declared In

setFE.h

SETBERegisterProcedure

Associate an application-defined function with a string in the SET BE - B

```
int SETBERegisterProcedure(SETRef set, char *name, void *procPtr);
```

Parameters

set An opaque SETRef object given by a SET routine such as SETBEDidFinishLaunching.

name String identifying the function given in procPtr. Cannot be longer than MaxSETNameLength bytes.

procPtr Pointer to a given function to be associated with the name string in the BE.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Allows the BE to register its routine with SET, naming each by a string, so that it can be called by the FE via SETCall. Typically SETBERegisterProcedure is called during execution of SETBEDidFinishLaunching, but in principle it could be called from any application-defined code running in the BE that has access to the SETRef object.

Declared In

setBE.h

SETBEDeregisterProcedure

Disassociate an application-defined function and its string in the SET BE - B

```
int SETBEDeregisterProcedure(SETRef set, char *name);
```

Parameters

set An opaque SETRef object given by a SET routine such as SETBEDidFinishLaunching.

name String identifying the function

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Allows the BE to deregister a routine, identified by name, from SET. It is not required to be called before program termination, at which time all functions will deregistered automatically.

Declared In

setBE.h

SETBEDidFinishLaunching

Callback routine in the application BE called by SET signaling processing will soon begin - B

```
int SETBEDidFinishLaunching(SETRef set);
```

Parameters

set A given opaque SETRef object.

Return Value

An int with the value of 0 on success or an error code on failure.

Discussion

REQUIRED SET requires SETBEDidFinishLaunching to be declared in the application BE. SET calls this routine, residing in the BE, after the parallel environment is set up and before computations begin to allow the BE to initialize, set up connections, allocate buffers, or other operations appropriate to do on first execution. This is also the primary opportunity to register BE routines with SET using SETBERegisterProcedure.

Declared In

setBE.h

SETBEWillTerminate

Callback routine in the application BE called by SET signaling execution is ending - B

```
int SETBEWillTerminate(SETRef set);
```

Parameters

set A given opaque SETRef object.

Return Value

An int with the value of 0 on success or an error code on failure.

Discussion

REQUIRED SET requires SETBEWillTerminate to be declared in the application BE. SET calls this routine residing in the BE to allow the BE to close connections, deallocate buffers, or other operations appropriate when execution ends.

Declared In

setBE.h

Collective

Collective calls provide commonly mechanisms to send expressions between groups of processing BEs.

SETBroadcast

Broadcasts an array from the root BE to all others - F

```
int SETBroadcast(SETRef set, char *destinationArray, char *sourceArray, int
      root);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

destinationArray String naming the array that will hold the broadcasted data. If it already exists, it will be overwritten.

sourceArray String naming the source array on the root processor.

root The processor ID of the BE in the current communicator whose sourceArray is to be broadcast.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Performs a broadcast of sourceArray from the root processor BE to all other BEs in the default communicator world. Data is expected to be supplied by the root processor BE, then it returns the incoming expression to destinationArray to all processor BEs.

Declared In

setFE.h

SETGather

Copies data from the all other processors to the root processor - F

```
int SETGather(SETRef set, char *destinationArray, char *sendArray, int root);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

destinationArray String naming the array that will hold the gathered data. If it already exists, it will be overwritten.

sourceArray String naming the source array on the each processor.

root The processor ID of the BE in the current communicator who holds the result of the gather.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

All processors (including root) in the default communicator send their data in sendArray to the root processor, which produces an concatenated array with this data, in the order according to the default communicator, and returns the result in the array named by destinationArray. On processors that are not root, destinationArray is ignored.

Declared In

setFE.h

SETAllgather

Copies data from the all processors into one array, which is copied to all processors - F

```
int SETAllgather(SETRef set, char *destinationArray, char *sendArray);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

destinationArray String naming the array that will hold the gathered data. If it already exists, it will be overwritten.

sourceArray String naming the source array on the each processor.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

All processors in the default communicator send their sendArray, which are then concatenated into one array in the order according to the default communicator. All processors in the communicator are returned the array named by destinationArray.

Declared In

setFE.h

SETScatter

Distributes data from the root processor to the all other processors - F

```
int SETScatter(SETRef set, char *destinationArray, char *sendArray, int
      root);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

destinationArray String naming the array that will hold the portion of scattered data. If it already exists, it will be overwritten.

sendArray String naming the source array on the root processor BE.

root The processor ID of the BE in the current communicator who holds the result of the gather.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Processor BE root partitions the list in sendArray into equal parts (if possible) and places each piece in the array named by destinationArray on all the processors (including root) in the default communicator, according the order and size of that communicator.

Declared In

setFE.h

SETAlltoall

Redistributes data in order from all processors to all processors - F

```
int SETAlltoall(SETRef set, char *destinationArray, char *sendArray);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

destinationArray String naming the array that will hold each processor's portion of data. If it already exists, it will be overwritten.

sendArray String naming the source array on the each processor BE.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Each processor BE sends equal parts of sendArray to all other processor BEs in the default communicator world, which each collects from all other processors and organizes into the order according to that communicator. The resulting data is placed in the array named by destinationArray.

Declared In

setFE.h

SETReduce

Reduces data of an array from all processors, saving the result in the root processor - F

```
int SETReduce(SETRef set, char *targetArray, int operation, int root);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

targetArray String naming the array that holds the source data and will hold the reduced data.

operation Integer specifying mathematical operation to perform on the data. Can be SETReduce_Sum, SETReduce_Max, or SETReduce_Min.

root The processor ID of the BE in the current communicator who holds the result of the data reduction.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Performs a collective reduction operation between arrays on all processors in the default communicator world for every element in the array in targetArray returning the resulting list in targetArray on the processor with the ID root. The data type of the array must be an integer or floating-point number as specified via the flags argument when the array was created.

Declared In

setFE.h

SETAllreduce

Reduces data of an array from all processors, copying results to all processors - F

```
int SETAllreduce(SETRef set, char *targetArray, int operation);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

targetArray String naming the array that holds the source data and will hold the reduced data.

operation Integer specifying mathematical operation to perform on the data. Can be SETReduce_Sum, SETReduce_Max, or SETReduce_Min.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Performs a collective reduction operation between arrays on all processors in the default communicator world for every element in the array in targetArray returning the resulting list in targetArray on all processors. The data type of the array must be an integer or floating-point number as specified via the flags argument when the array was created.

Declared In

setFE.h

SETReduceScatter

Reduces data of an array from all processors, redistributing results to all processors - F

```
int SETReduceScatter(SETRef set, char *targetArray, int operation);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

targetArray String naming the array that holds the source data to be reduced and will hold this processor BE's portion of the reduced data.

operation Integer specifying mathematical operation to perform on the data. Can be SETReduce_Sum, SETReduce_Max, or SETReduce_Min.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Performs a collective reduction operation between arrays on all processors in the default communicator world for every element in the array in targetArray, partitioning the resulting data into pieces in each processor's targetArray. The data type of the array must be an integer or floating-point number as specified via the flags argument when the array was created.

Declared In

setFE.h

Divide and Conquer

Asynchronous calls make it possible for the kernel to do work while communications is proceeding simultaneously. It is also possible that another node may not be able to send or receive data yet, allowing one kernel to continue working while waiting.

SETFunctionToArray

Evaluates many iterations of a function and saves its results to a distributed array - F

```
int SETFunctionToArray(SETRef set, char *array, int elementSize, int32_t
      flags, char *functionName, int count);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the new 1-dimensional array to be filled with results of the called function.

elementSize Size of the element generated by the function.

functionName String naming the function registered by the BE, conforming to SETIterationFunctionPtr, that will be repeatedly called to fill the array.

count Global count specifying the range of integers to call the given function.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Evaluates the function foo(SETRef, int, void *) (see SETIterationFunctionPtr) from 0 to count-1, but across the cluster, and returns these results in a new array with the given name. Results computed on a BE are saved to that BE's array.

Declared In

setFE.h

SETLoadBalanceFunctionToArray

Evaluates many iterations of a function with load balancing saves its results to a distributed array - F

```
int SETLoadBalanceFunctionToArray(SETRef set, char *array, int elementSize,
      int32_t flags, char *functionName, int count);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the new 1-dimensional array to be filled with results of the called function.

elementSize Size of the element generated by the function.

functionName String naming the function registered by the BE, conforming to SETIterationFunctionPtr, that will be repeatedly called to fill the array.

count Global count specifying the range of integers to call the given function.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Like SETFunctionToArray, evaluates the function foo(SETRef, int, void *) (see SETIterationFunctionPtr) from 0 to count-1, but across the cluster, and returns these results in a new array with the given name. Results computed on a BE are saved to that BE's array. The difference is that it performs the functions with load balancing, so that it is less sensitive to different function calls that take vastly different times to execute.

Declared In

setFE.h

SETGenerateArray

Evaluates many iterations of a function in a specific range and saves its results to a distributed array - F

```
int SETGenerateArray(SETRef set, char *array, int elementSize, int32_t flags,
      char *functionName, int loopStart, int loopEnd);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the new 1-dimensional array to be filled with results of the called function.

elementSize Size of the element generated by the function.

functionName String naming the function registered by the BE, conforming to SETIterationFunctionPtr, that will be repeatedly called to fill the array.

loopStart Integer specifying the lowest integer to call the given function.

loopEnd Integer specifying the highest integer to call the given function.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Evaluates the named function conforming to SETIterationFunctionPtr from loopStart to loopEnd inclusive across the cluster, rather than on just one processor, returning the locally evaluated portion in an array.

Declared In

setFE.h

SETFunctionCall

Evaluates a function on elements an input array producing output in an output array - F

```
int SETFunctionCall(SETRef set, char *output, char *functionName, char
      *input);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

output String naming the new array, whose dimensions and parameters match that of the input, to be filled with results of the called function. If no output is specified, input will be overwritten.

functionName String naming the function registered by the BE, conforming to SETIterationFunctionPtr, that will be repeatedly called to fill the array.

input String naming the input array whose elements will be made available to the function. If no output is specified, output of the function will overwrite elements of input.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Evaluates the function conforming to SETInOutFunctionPtr on that BE's subset of inputs scattered across the cluster in inputs and return an equal-sized array. If no new output is provided, input is overwritten with a new array.

Declared In

setFE.h

Guard-Cell Management

Typically the space of a problem is divided into partitions. Often, however, neighboring edges (but only the edges) of each partition must interact, so a "guard cell" is inserted on both edges as a substitute for the neighboring data. Thus the space a processor sees is two elements wider than the actual space for which the processor is responsible. EdgeCell helps maintain these guard cells.

SETEdgeCell

Updates the edge cells of the given array with copies of the edge of that array in the neighboring processors - F

```
int SETEdgeCell(SETRef set, char *array);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the existing array whose edges between processor will be "stitched". This array may or may not have guard cell metadata.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Copies the second element of array to the last element of the left processor and the second-to-last element of list to the first element of the right processor while simultaneously receiving the same from its neighbors. If the array has more than one dimension, entire layers (the subarray where the last dimension of the array is ignored) will be copied rather than just one element. If guard cell layer flags (set1GuardCellLayer, set2GuardCellLayers) were set when the array was created, these extra guard cell layers will be written with neighboring edge data instead.

Declared In

setFE.h

Matrix and Vector Manipulation

Matrices are partitioned and stored in processors across the cluster. These calls manipulate, generate, and process these matrices in common ways.

SETTranspose

Transposes a matrix across the cluster - F

```
int SETTranspose(SETRef set, char *matrix);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

matrix String naming the existing matrix to be transposed.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Transposes matrix that is in fact represented across the cluster, rather than on just one processor. It returns the portion of the transposed matrix meant for that processor BE.

Declared In

setFE.h

SETMatrixProduct

Multiples two matrices across the cluster - F

```
int SETMatrixProduct(SETRef set, char *matrix1, char *matrix2);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

matrix1 String naming the first matrix of the product and the location of the output.

matrix2 String naming the second matrix of the product.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Evaluates the product of matrix and matrix (or vector), where these matrices are represented across the cluster. matrix1 is overwritten the results.

Declared In

setFE.h

SETGetMatrixDimensions

Gets the dimensions of a matrix across the cluster - F

```
int *SETGetMatrixDimensions(SETRef set, char *matrix);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

matrix String naming the matrix to be measured.

Return Value

A pointer to int's on success or NULL on failure.

Discussion

Measures the dimensions of an existing matrix is represented across the cluster, rather than on just one processor. It returns a list of each dimension in a new buffer of ints. The caller is responsible for calling free() on the buffer.

Declared In

setFE.h

SETTrace

Computes the trace of a matrix across the cluster - F

```
MatrixElementType SETTrace(SETRef set, char *matrix);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

matrix String naming the matrix to be traced.

Return Value

A floating-point value of type MatrixElementType.

Discussion

Calculates and returns the trace of a matrix represented across the cluster.

Declared In

setFE.h

SETMatrixIdentity

Creates a new identity matrix across the cluster - F

```
int SETMatrixIdentity(SETRef set, char *newMatrix, int rank);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

newMatrix String naming the new matrix to be created.

rank Integer specifying the size of the new matrix to be created.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Generates a new identity matrix represented across the cluster. It returns the portion of the new matrix to each processor BE in a new matrix named by newMatrix.

Declared In

setFE.h

SETOuterProduct

Calculates a matrix across the cluster from the outer product of two vectors - F

```
int SETOuterProduct(SETRef set, char *newMatrix, int whichOperator, char
      *vector1, char *vector2);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

newMatrix String naming the new matrix to be created.

whichOperator Integer specifying the operator to be used for the outer product.

vector1 String identifying the first vector of the outer product.

vector2 String identifying the second vector of the outer product.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Computes the outer product, with the given operator in whichOperator, of the two given vectors. The answer becomes a matrix represented across the cluster storing the portion of the new matrix in each processor BE.

Declared In

setFE.h

SETMatrixInverse

Calculates the inverse of a matrix across the cluster - F

```
int SETMatrixInverse(SETRef set, char *newMatrix, char *matrix);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

newMatrix String naming the new matrix to be created.

matrix String identifying the matrix to be inverted.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Computes the inverse of the given matrix and returns its answer as a matrix is represented across the cluster. If no new name is given, input is overwritten.

Declared In

setFE.h

Element Management

Besides the obvious divide-and-conquer approach, a list of elements can also be partitioned in arbitrary ways. This is useful if elements need to be organized or sorted onto multiple processors. For example particles of a system may drift out of the space of one processor into another, so their data would need to be redistributed periodically. The following calls support this behavior.

SETElementManageSwitched

Redistributes as needed the elements of an array across the cluster - F

```
int SETElementManageSwitched(SETRef set, char *array, char
      *switchFunctionName);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the existing array whose elements may be redistributed across the cluster.

switchFunctionName String identifying the function that reports where an array element should go.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Selects which elements of array will be sent to which processor BEs according to the named switch function conforming to SETElementManageSwitcher evaluated on each element of array. The switch function should return the ID number of the processor that element should be sent. This call operates on the input array and returns its output in the same array.

Declared In

setFE.h

SETElementManage

Redistributes as needed the elements with assumed format of an array across the cluster - F

```
int SETElementManage(SETRef set, char *array);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the existing array whose elements may be redistributed across the cluster.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Selects which elements of the array to be sent to which processors. This version without the switch function uses the first int32_t of each element of array for sorting. This call operates on the input array and returns its output in the same array.

Declared In

setFE.h

Fourier Transform

Fourier transforms of very large arrays can be difficult to manage, not the least of which is the memory requirements. Parallelizing the Fourier transform makes it possible to make use of all the memory available on the entire cluster, making it possible to manipulate problem sizes that no one processor could possibly do alone.

SETFourier

Computes the Fourier transform of an array across the cluster - F

```
int SETFourier(SETRef set, char *output, char *input, int forwards);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

output String naming the array which holds the output.

input String naming the existing array to be transformed across the cluster.

forwards Integer identifying a forward transform (1) or backwards transform (0).

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Computes the Fourier transform of the given two- or three-dimensional array represented across the cluster, like for matrices, above. It returns the portion of the Fourier-transformed array meant for that processor. forwards specifies either a forward- or backward-transform.

Declared In

setFE.h

Parallel Disk I/O

Data needs to be read in and out of the cluster, but in such a way that the data is distributed across the cluster evenly. These calls perform these actions.

SETWriteToLocal

Writes array data local to each processor across the cluster - F

```
int SETWriteToLocal(SETRef set, char *array, char *filename, int format);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the array which holds the source data.

filename String naming the new file to be written across the cluster.

format Integer identifying format of the written file.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Writes the contents of array into the file with the name filename on the local drive on each processor BE.

Declared In

setFE.h

SETWriteToRoot

Writes array data from each processor across the cluster to root - F

```
int SETWriteToRoot(SETRef set, char *array, char *filename, int format, int
      root);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

array String naming the array which holds the source data.

filename String naming the new file to be written at root.

format Integer identifying format of the written file.

root Integer identifying the processor BE of the current communicator who writes the file.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Writes the contents of array from each processor BE into the file with the name filename in order on processor root.

Declared In

setFE.h

SETReadFromLocal

Reads array data local to each processor across the cluster - F

```
int SETReadFromLocal(SETRef set, char *newArray, char *filename, int format);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

newArray String naming the new array which holds the file data.

filename String naming the new file to be read across the cluster.

format Integer identifying format of the file.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Reads data into newArray from the file with the name filename located at the local processor BE across the cluster.

Declared In

setFE.h

SETReadFromRoot

Reads array data local to root and distributes that data to each processor across the cluster - F

```
int SETReadFromRoot(SETRef set, char *newArray, char *filename, int format,
      int root);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

newArray String naming the new array to holds the file data.

filename String naming the file to be read at root.

format Integer identifying format of the file.

root Integer identifying the processor BE of the current communicator who reads the file.

Return Value

An int returning 0 on success or an error code on failure.

Discussion

Reads data into newArray from the file with the name filename on processor root and partitions that source data into each processor on the cluster.

Declared In

setFE.h

Communicator

Parallel operations need some way to identify and distinguish different processing elements in use. SET adopts the practice of assigning each a unique integer (IdProc) starting with 0 up to the number of processors (NProc). This identifying number (IdProc), with a knowledge of the total count (NProc), makes it possible to programmatically divide any measurable entity. Communicators organizes groups of nodes into application-defined subsets.

SETGlobalComm

The communicator world of the entire cluster - F and B

```
int SETGlobalComm(void);
```

Return Value

An int with the value of 0 on success or an error code on failure.

Discussion

The communicator world of the entire cluster (see other Communicator calls, below). Value is always zero.

Declared In

setFE.h, setBE.h

SETSetComm

Sets the default communicator world for subsequent calls - F

```
int SETSetComm(SETRef set, int newComm);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

newComm Integer specifying the communicator world for subsequent SET calls.

Return Value

An int with the value of 0 on success or an error code on failure. If a given input is invalid, parameterErr is returned.

Discussion

Sets the default communicator world for subsequent calls.

Declared In

setFE.h

SETGetComm

Gets the current communicator world - F

```
int SETGetComm(SETRef set);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

Return Value

An int identifying the current communicator (always nonnegative) on success or an error code (negative) on failure.

Discussion

Gets the current communicator world.

Declared In

setFE.h

SETCommSize

Gets the processor count of the specified communicator world - F

```
int SETCommSize(SETRef set, int comm);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

comm Integer specifying the communicator world to measure.

Return Value

An int with the size of the current communicator (always positive) on success or an error code (negative) on failure.

Discussion

Gets the processor count of the communicator world comm.

Declared In

setFE.h

SETGetIdProc

Gets the current processor ID - B

```
int SETGetIdProc(SETRef set);
```

Parameters

set An opaque SETRef object.

Return Value

An int with the identification number of the current processor (always nonnegative) on success or an error code (negative) on failure.

Discussion

Gets the identification number of the current processor of the current communicator.

Declared In

setBE.h

SETGetNProc

Gets the processor count of the current communicator world - B

```
int SETGetNProc(SETRef set);
```

Parameters

set An opaque SETRef object.

Return Value

An int with the size of the current communicator (always positive) on success or an error code (negative) on failure.

Discussion

Gets the number of processors in the current communicator world.

Declared In

setBE.h

SETCommMap

Returns the processor id in the global communicator of the specified processor of another communicator world - B

```
int SETCommMap(SETRef set, int comm, int target);
```

Parameters

set An opaque SETRef object.

comm Integer with the communicator comm.

target Processor ID in the communicator comm.

Return Value

An int with the processor ID of the global communicator (always nonnegative) on success or an error code (negative) on failure.

Discussion

Returns the mapping of the communicator comm to the processor indexed according to the global communicator. The target argument returns the ID of the processor in the global communicator whose ID is target in the communicator comm.

Declared In

setBE.h

SETCommDup

Duplicates the specified communicator world - F

```
int SETCommDup(SETRef set, int comm);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

comm Integer specifying the communicator world to duplicate.

Return Value

An int with the new communicator (always nonnegative) on success or an error code (negative) on failure.

Discussion

Duplicates the communicator world comm and returns the integer specifying this new communicator world.

Declared In

setFE.h

SETCommSingle

Returns a new communicator containing only one rank of the communicator comm - F

```
int SETCommSingle(SETRef set, int comm, int rank);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

comm Integer specifying the communicator world to duplicate.

rank Integer specifying the rank of the processor BE of the given communicator world to include.

Return Value

An int identifying the new communicator (always nonnegative) on success or an error code (negative) on failure.

Discussion

Creates a new communicator world using only BE with the processor ID rank of the communicator world comm and returns the integer identifying this new communicator world.

Declared In

setFE.h

SETCommSplit

Creates a new communicator calculated given color and key functions - F

```
int SETCommSplit(SETRef set, int comm, char *colorFunctionName, char
      *keyFunctionName);
```

Parameters

set An opaque SETRef object previously returned from SETInitialize.

comm Integer specifying the initial communicator world.

colorFunctionName String naming the color function registered by the BE.

keyFunctionName String naming the key function registered by the BE.

Return Value

An int identifying the new communicator (always nonnegative) on success or an error code (negative) on failure.

Discussion

Creates a new communicator, starting with the communicator comm, into several disjoint subsets each identified by color. The sort order within each subset is first by key, second according to the ordering in the previous communicator. Processors not meant to participate in any new communicator indicates this by passing -1. The corresponding communicator is returned to each calling processor. key and color functions on the BE must conform to SETFunctionPtr.

Declared In

setFE.h

Low-Level Message Passing

Simply sending expressions from one processor BE to another is possible with these most basic MPI calls. What is unique about message passing in SET, in contrast to a traditional implementation like MPI, is that the messages are entire named arrays, passed from one BE to another, rather than arbitrary data buffers.

One BE must call to send an expression while the other calls a corresponding routine to receive the sent expression. Because it is possible that the receiver has not yet called a Receive even if the message has left the sending node, completion of Send is not a confirmation that it has been received. Note: We advise to use the earlier, high-level communication routines when possible, rather than these low-level calls, but we provide the low-level calls just in case the application requires it.

SETSend

Sends an array to a target processor BE - B

```
int SETSend(SETRef set, char *array, int target);
```

Parameters

set An opaque SETRef object.

array A string containing the name of the array.

target Integer holding the processor ID of the target BE.

Return Value

An int with the value of 0 on success or an error code on failure. If set, array, or target are invalid, parameterErr is returned.

Discussion

Sends a named array to a processor with the ID target in the default communicator, waiting until that array has been sent. Must be balanced by a SETReceive or SETAsynchronousReceive on the target processor.

Declared In

setBE.h

SETReceive

Receives an array from a target processor BE - B

```
int SETReceive(SETRef set, char *array, int target);
```

Parameters

set An opaque SETRef object.

array A string identifying the name of the array when received.

target Integer holding the processor ID of the sending BE.

Return Value

An int with the value of 0 on success or an error code on failure. If set, array, or target are invalid, parameterErr is returned.

Discussion

Receives a named array from a processor with the ID target in the default communicator world, waiting until the array has arrived, then associating that array with the name given in the array string. If array exists prior to this call, it is overwritten. This call must be balanced by a SETSend or SETAsynchronousSend on the target processor.

Declared In

setBE.h

SETSendRecv

Receives an array from a target processor BE - B

```
int SETSendRecv(SETRef set, char *sendArray, char *receiveArray, int dest,
      int source);
```

Parameters

set An opaque SETRef object.

sendArray A string identifying the name of the array to send.

receiveArray A string identifying the name of the array when received.

dest Integer holding the processor ID of the BE where sendArray is to be sent.

source Integer holding the processor ID of the other sending BE.

Return Value

An int with the value of 0 on success or an error code on failure. If set or the other parameters are invalid, parameterErr is returned.

Discussion

Simultaneously sends the expression sendArray to the processor BE with the ID target and receives receiveArray from the processor BE with the ID source in the default communicator world, waiting until both operations have returned. Must be balanced with appropriate sends and receives on the other processor BEs.

Declared In

setBE.h

SETAsynchronousSend

Sends an array to a target processor BE asynchronously - B

```
int SETAsynchronousSend(SETRef set, char *sendArray, int target,
      SETRequestRef *req);
```

Parameters

set An opaque SETRef object.

sendArray A string containing the name of the array to send.

target Integer holding the processor ID of the target BE.

req Pointer to a SETRequestRef for later use by SETTest or SETWait.

Return Value

An int with the value of 0 on success or an error code on failure. If set, sendArray, or target are invalid, parameterErr is returned.

Discussion

Sends the named sendArray to a processor with the ID target in the current communicator, returning immediately. Caution: It must be balanced with calls to SETTest(req) until SETTest(req) returns noErr and balanced by a SETReceive or SETAsynchronousReceive on the target processor.

Declared In

setBE.h

SETAsynchronousReceive

Receives an array from a target processor BE asynchronously - B

```
int SETAsynchronousReceive(SETRef set, char *receiveArray, int target,
      SETRequestRef *req);
```

Parameters

set An opaque SETRef object.

array A string identifying the name of the array when received.

target Integer holding the processor ID of the sending BE.

req Pointer to a SETRequestRef for later use by SETTest or SETWait.

Return Value

An int with the value of 0 on success or an error code on failure. If set, receiveArray, or target are invalid, parameterErr is returned.

Discussion

Receives a named receiveArray from a processor BE with the ID target in the current communicator, returning immediately. Once the array has arrived, SET associates that array with the name given in the receiveArray string. If array exists prior to this call, it is overwritten. Caution: It must be balanced with calls to SETTest(req) until SETTest(req) returns noErr and balanced by a SETSend or SETAsynchronousSend on the target processor. Very important: Received array is not safe to access until SET(req) returns noErr.

Declared In

setBE.h

SETTest

Completes asynchronous behavior of SETAsynchronousSend and SETAsynchronousReceive - B

```
int SETTest(SETRequestRef *req);
```

Parameters

req Pointer to a SETRequestRef provided by SETAsynchronousSend or SETAsynchronousReceive.

Return Value

An int with the value of 0 on completion of the asynchronous message passing call or an error code. If req is invalid, parameterErr is returned. If the message has not been completed, the non-fatal code tryAgainErr is returned.

Discussion

Completes asynchronous behavior of SETAsynchronousSend and SETAsynchronousReceive, returning immediately. If the message operation is not yet complete, tryAgainErr is returned. If the message completes, noErr is returned, and the req object is no longer valid.

Declared In

setBE.h

SETWait

Completes asynchronous behavior of SETAsynchronousSend and SETAsynchronousReceive - B

```
int SETWait(SETRequestRef *req);
```

Parameters

req Pointer to a SETRequestRef provided by SETAsynchronousSend or SETAsynchronousReceive.

Return Value

An int with the value of 0 on completion of the asynchronous message passing call or an error code. If req is invalid, parameterErr is returned.

Discussion

Completes asynchronous behavior of SETAsynchronousSend and SETAsynchronousReceive, not returning until the message completes or an error occurs. If the message completes, noErr is returned, and the req object is no longer valid.

Declared In

setBE.h

SETWaitall

Completes behavior of a series of asynchronous message requests - B

```
int SETWaitall(SETRequestRef *reqList, int count);
```

Parameters

reqList Pointer to a list of SETRequestRef provided by SETAsynchronousSend or SETAsynchronousReceive.

count Count of the list pointed to by reqList.

Return Value

An int with the value of 0 on completion of the asynchronous message passing calls or an error code. If reqList is invalid, parameterErr is returned.

Discussion

Completes asynchronous behavior of SETAsynchronousSend and SETAsynchronousReceive, not returning until all messages complete or an error occurs. If these messages complete, noErr is returned, and all SETRequestRef objects are no longer valid.

Declared In

setBE.h

SETWaitany

Completes behavior of a series of asynchronous message requests - B

```
int SETWaitany(SETRequestRef *reqList, int count, int *which);
```

Parameters

reqList Pointer to a list of SETRequestRef provided by SETAsynchronousSend or SETAsynchronousReceive.

count Count of the list pointed to by reqList.

which Pointer to an integer that will indicate which SETRequestRef completed.

Return Value

An int with the value of 0 on completion of an asynchronous message passing call or an error code. If reqList or which is invalid, parameterErr is returned.

Discussion

Completes asynchronous behavior of SETAsynchronousSend and SETAsynchronousReceive, not returning until one message completes or an error occurs. If a message completes, noErr is returned, and that SETRequestRef objects is no longer valid. The index of the SETRequestRef in reqList that completed is returned in the int pointed to by which.

Declared In

setBE.h

Data Structures

SET Data Structures

To support operation of the SET functions, specific data types were necessary. Where possible, data internal to SET is protected with an opaque data structure. We strongly discourage other writers to query inside SET's internal data structures or depend on undocumented features found there, as they are subject to change in a future version of SET. SETAccessorStruct was necessary to expose in order to support high-speed data access, but likewise we strongly recommend not changing the metadata contents directly, only the application data. By contrast, data structures provided by the application to SET should be edited by the application.

SETRef

Object returned by SET upon initialization or call backs - F and B

```
typedef struct SETOpaqueStruct {
      int32_t version;
      /* this is the structure visible to the outside world */
      } SETOpaqueStruct, *SETRef;
```

Fields

version Version of SET being executed.

Discussion

SETRef is an opaque object returned by SETInitialize in the FE or by callbacks in the BE and is needed by SET to complete operations or requests.

Declared In

setFE.h, setBE.h

SETInitializeStruct

Structure used when calling SETInitialize to set up the SET environment - F

```
typedef struct SETInitializeStruct {
      char *bePath; /* points to user-allocated string with path to BE
      executable */
      int32_t flags;
      uint32_t selector;
      union {
            struct {
                  int32_t numberNodes;
                  int32_t numberTasksPerNode;
```

```
            } nodesTasks;
      } u;
} SETInitializeStruct, *SETInitializePtr;
```

Fields

bePath String containing full path name to the compiled BE executable.

flags Integer containing flags guiding the behavior of SET. For example SETInitializeVerboseFlag has SET produce diagnostic data to standard output.

selector Integer executable specifying launch mode and which member of the following union has meaning. Only 0 is allowed in SET version 1.

numberNodes Integer specifying the number of nodes on which to launch the BE's. Leaving this value at 0 or 1 leads to the default behavior of launching on just the local node.

numberTasksPerNode Integer specifying the number of BE tasks to launch per node. Leaving this value at 0, the default, means the number of tasks will be the number of processors reported by the operating system, which is usually the number of "virtual cores", on each node.

Discussion

SETInitializeStruct is a data structure passed to SETInitialize in the FE to specify initial behavior such as the number of BE's to launch and how to launch them as well as ongoing behavior such as diagnostic output.

Declared In

setFE.h

SETAccessorStruct

Structure used when calling SETInitialize to set up the SET environment - B

```
typedef struct SETAccessorStruct {
      void *p;
      int32_t flags;
      int32_t elementSize;
      int32_t dimensionCount;
      int32_t dimension[4]; /* sized as needed */
      /* raw array data, 16-byte aligned, follows */
} SETAccessorStruct, *SETAccessorPtr;
```

Fields

p Pointer to the array data.

flags Bit mask specifying metadata about the array.

elementSize Size in bytes of an indivisible element of the array.

dimensionCount Integer specifying the number of dimensions of the array.

dimensions[] Array of integers describing the size of each dimension of the array, starting with the lowest stride (one element per array index).

Discussion

The fields of SETAccessorStruct are not usually addressed directly, but the SETAccessorPtr is used in a series of #define macros to access the array. These macros include ArrayAccess, ElementAddress, LinearFrom1, etc. It is necessary to expose the contents of this structure for the macros to operate with high speed. The application should not modify this structure's contents in any way, only the payload array data.

Declared In

setBE.h

SETFunctionPtr

Function pointer for execution of SETCall - B

```
typedef int (*SETFunctionPtr)(SETRef set, SETAccessorPtr a, ...);
```

Parameters

set An opaque SETRef object for this BE.

a, ... A series of SETAccessorPtr.

Return Value

An int with the value of 0 on success or an error code.

Discussion

FE execution of SETCall has the BE execute the requested command. SET parses the command for a function, registered by the BE via SETBERegisterProcedure, by the given function name as well as any optional arrays, each by name. The function must conform to the above prototype, however, this prototype allows one or more arrays on the argument list. The function can use the provided SETRef and SETAccessorPtr to access local arrays are perform other functions.

Declared In

setBE.h

SETIterationFunctionPtr

Function pointer for execution of "Divide and Conquer" SET routines - B

```
typedef int (*SETIterationFunctionPtr)(SETRef set, int32_t which, void*
      output);
```

Parameters

set An opaque SETRef object for this BE.

which Integer specifying which iteration to perform.

output Pointer to the target data.

Return Value

An int with the value of 0 on success or an error code.

Discussion

FE execution of SETFunctionToArray, SETLoadBalanceFunctionToArray, and SETGenerateArray has the BE search for an function, registered by the BE via SETBERegisterProcedure, that conforms to this function prototype. The function will be called many times, once for each instance of which, but not necessarily in order, and will be expected to execute and write its output to the output pointer. The FE has control over the size of the output data element via the array it defines. If needed, this function can access other arrays allocated via SET using the given SETRef object.

Declared In

setBE.h

SETInOutFunctionPtr

Function pointer for execution of SETFunctionCall - B

```
typedef int (*SETInOutFunctionPtr)(SETRef set, void* input, void* output);
```

Parameters

set An opaque SETRef object for this BE.

input Pointer to the source data.

output Pointer to the target data.

Return Value

An int with the value of 0 on success or an error code.

Discussion

FE execution of SETFunctionCall has the BE search for an function, registered by the BE via SETBERegisterProcedure, that conforms to this function prototype. The function will be called many times, once for each instances of input, but not necessarily in order, and will be expected to execute and write its output to the output pointer. The FE has control over the size of the input and output data elements via the arrays it identifies. If needed, this function can access other arrays allocated via SET using the given SETRef object.

Declared In

setBE.h

SETElementManageSwitcher

Function pointer for to supply target data for execution of SETElementManageSwitched - B

```
typedef int (*SETElementManageSwitcher)(SETRef set, void* input);
```

Parameters

set An opaque SETRef object for this BE.

input Pointer to the source data.

Return Value

An int with the processor ID identifying the destination BE of the data element at the input pointer. If this data is to stay in this BE, return SETGetIdProc(set).

Discussion

FE execution of SETElementManageSwitched has the BE search for an function, registered by the BE via SETBERegisterProcedure, that conforms to this function prototype. The function will be called many times, at least once for every element of the target array, but not necessarily in order. The FE has control over the size of the input data element via the array it identifies. If needed, this function can access other arrays allocated via SET using the given SETRef object. The function should return the processor ID of the BE where this data element should go. Data that stays in the BE should return SETGetIdProc(set).

Declared In

setBE.h

SETRequestRef

Object returned when initiating asynchronous message passing - B

```
typedef struct SETOpaqueRequestStruct {
      uint32_t which;
      /* this is the structure visible to the outside world */
} SETOpaqueRequestStruct, *SETRequestRef;
```

Fields

which Integer indicating data type.

Discussion

Returned via SETAsynchronousSend and SETAsynchronousReceive, the SETRequestRef is later given to SETTest or SETWait for completion of the message passing. Once SETTest or SETWait returns 0 to indicate the operation has completed, the resulting SETRequestRef is invalid. The application should not modify the contents of this structure.

Declared In

setBE.h

SETBEInitializeStruct

Structure used when calling SETBEInitializationParameters to help set up the SET environment - B

```
typedef struct SETBEInitializeStruct {
      char *fePath; /* points to user-allocated string with path to FE
      executable; pointer is not retained by SET */
      int32_t flags;
      uint32_t unused[4]; /* reserved for future expansion */
} SETBEInitializeStruct, *SETBEInitializePtr;
```

Fields

fePath String containing full path name to the compiled FE executable.

flags Bit mask specifying other initialization parameters.

unused For future expansion

Discussion

Used to provide data to SETBEInitializationParameters(), most importantly the path to the Front End executable. Action on this data may or may not be taken, depending on the cluster launching scenario. In a classic Linux cluster, the BE is launched first, so this structure provides a while for SET to launch the FE.

Declared In

setBE.h

Constants and Definitions

SET error codes

Error codes in SET are generally negative numbers, while 0 has special meaning as no error. - F and B

```
enum {
      noError           =     (0),
      tryAgainErr       =     (-1),
      notFoundErr       =     (-43),
      busyErr           =     (-47),
      duplicateNameErr  =     (-48),
      parameterErr      =     (-50),
      memoryFullErr     =     (-108),
      channelClosedErr  =     (-916),
      timeOutErr        =     (-3259)
};
```

SET array flag constants and definitions

This flags are to be passed in the flags parameter when creating a SET array. Although it is such situations do not need to identify the array data as integer or floating-point types, it is necessary for operations such as reduction. This metadata is also stored in the flags field of SETAccessorStruct. - F and B

```
#define setDataTypeMask                   0x000000ffL
#define SETArrayFlagsDataTypeIs(flags,type)
            ((type)==(setDataTypeMask&(flags)))
#define SETArrayDataTypeIs(array,type)
            SETArrayFlagsDataTypeIs((array)->flags,type)
/* noDataType is for */
#define     setNoDataType                           (0)
/* Integer data types */
#define     setInt8DataType                         (1L)
#define     setInt16DataType                  (2L)
#define     setInt32DataType                  (3L)
#define     setInt64DataType                  (4L)
/* Floating-point data types */
#define     setFloatDataType                  (9L)
#define     setDoubleDataType                 (10L)
#define     setLongDoubleDataType             (11L)
```

```
#define setGuardCellLayersMask            0x00000f00L
#define SETArrayFlagsHasGuardCell(flags)
            (setGuardCellLayersMask&(flags))
#define SETArrayFlagsGuardCellIs(flags,which)
            ((which)==SETArrayFlagsHasGuardCell(flags))
#define SETArrayHasGuardCell(array)
            SETArrayFlagsHasGuardCell((array)->flags)
#define SETArrayGuardCellIs(array,which)
            SETArrayFlagsGuardCellIs((array)->flags,which)
#define set1GuardCellLayer                0x00000100L
#define set2GuardCellLayers               0x00000200L

#define SETArrayFlagsReserved             0xfffff000L
```

SET array access definitions

Generally the application will use ArrayAccess or ElementAddress on the given SETAccessorPtr. To access more than one dimension, these can be used in tandem with the “LinearFrom#” series of definitions. For example, to address a three-dimensional array of double’s, use ArrayAccess(array, LinearFrom3(x, y, z), double). This access is defined using #define macros for speed. - B

```
#define LinearFrom1(a, x)           ((x))
#define LinearFrom2(a, x, y)        (((x)+(a)->dimension[0]*(y)))
#define LinearFrom3(a, x, y, z)     LinearFrom2(a, x, (y)+(a)-
      >dimension[1]*(z))
#define LinearFrom4(a, x, y, z, w)  LinearFrom3(a, x, y, (z)+(a)-
      >dimension[2]*(w))
#define LinearFrom5(a, x, y, z, w, v)     LinearFrom4(a, x, y, z, (w)+(a)-
      >dimension[3]*(v))
#define LinearFrom6(a, x, y, z, w, v, u)  LinearFrom5(a, x, y, z, w, (v)+(a)-
      >dimension[4]*(u))
#define LinearFrom7(a, x, y, z, w, v, u, t)     LinearFrom6(a, x, y, z, w, v,
      (u)+(a)->dimension[5]*(t))
#define LinearFrom8(a, x, y, z, w, v, u, t, s)  LinearFrom7(a, x, y, z, w, v,
      u, (t)+(a)->dimension[6]*(s))
#define ElementAddress(a, x)        ((a)->p+(a)->elementSize*(x))
#define ArrayAccess(a, x, type)           (((type*)ElementAddress(a, x))[0])
```

SETReduce operation constants

These constants are passed in the operation parameter of SETReduce. For SETReduce to operate correctly, the data type of the array should be set using the above Array Flag Constants when created. - F

```
enum {
      SETReduce_Sum     =     0L,
      SETReduce_Max     =     1L,
      SETReduce_Min     =     2L
};
```

SET array matrix element data type

The matrix element data type of the element is based on this constant. - F and B

```
#define MatrixElementType                 double
```

SETInitializeStruct selector constants

When calling SETInitialize, the selector field of the SETInitializeStruct is filled with one of these constants to specify the format of the following union. - F

```
#define SETInitializeNodesTasks           'NdTs'
```

SET parameter constants

The SETSetParameter and SETGetParameter routines access parameters of the SET runtime environment. These constants are passed into the selector parameter. - F

```
#define SETTimeOutParameter               'TmOt'      /* double */
```

SET common constants

```
#define SETVerboseFlag                    0x00000100L
#define SETInitializeVerboseFlag          SETVerboseFlag
```

References

Reference materials

Texts used in the creation of this project include descriptions of the MPI standard and guides to parallel computing.

1. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI-The Complete Reference*, second edition, MIT Press, Cambridge MA, 1998.

2. G. F. Pfister, *In Search of Clusters*, Prentice Hall, 1997.

3. V. K. Decyk, "How to Write (Nearly) Portable Fortran Programs for Parallel Computers", *Computers In Physics*, **7**, p. 418 (1993).

4. V. K. Decyk, "Skeleton PIC Codes for Parallel Computers", *Computers Physics Communications*, **87**, p. 87 (1995).

For additional information

Useful web sites for learning about parallel computing and techniques to use them are on this site:

http://daugerresearch.com/vault/

It includes links to eight articles on writing parallel code:

- Parallelization - introduces the basic issues when writing parallel code
- Parallel Zoology - compare and contrast the different parallel computing types
- Parallel Knock - exhibition of basic message-passing code
- Parallel Adder - tutorial on parallelizing a single-processor code of independent work
- Parallel Pascal's Triangle - tutorial on parallelizing propagation-style code requiring local communication
- Parallel Circle Pi - tutorial on creating a load-balancing code divisible into independent work
- Parallel Life - tutorial on parallelizing propagation-style code requiring two-dimensional local communication
- Visualizing Message-Passing - a tutorial on using a graphical monitor window to debug and optimize parallel code

as well as other publications on cluster computing.

Contact information

For further questions or suggestions email set@advclustersys.com