Guide to SAM

Dan Scales

November 2, 1994

1 Introduction

In this guide, we describe in detail a system called SAM for distributed shared memory (DSM) in software on distributed-memory multiprocessors. SAM provides a set of primitive operations that present a simple shared memory abstraction and can be used in building higher-level distributed programming systems. All shared data is communicated and accessed in terms of user-defined data types, rather than in fixed-sized units such as pages. SAM provides basic primitives for expressing the fundamental data relationships in parallel programs. First, one computation may produce a value that is required by another computation. Such a relationship is expressed by using a data value with a single-assignment semantics. Second, several computations may each need to modify a piece of data; the final value of the data is unaffected by the order of the modifications, but the modifications may not occur at the same time. This data relationship is enforced by mutual exclusion, and is expressed via a concept called an “accumulator” from functional languages. Accumulators model data which must be updated a number of times to compute its final value, but whose final value does not depend on the order in which the updates are performed.

SAM provides support for these two data relationships by providing two kinds of data items, values and accumulators. (All SAM data items can refer to any user-defined type, not just a machine data type.) As in single-assignment languages, every value has a name and is immutable once created. For a process to read a value means that it must wait for the creation of the value and that the value must be communicated. An accumulator may be modified multiple times in any order by different processes. However, SAM ensures mutual exclusion between modifications to the accumulator. All synchronization and communication in SAM programs is provided by these two types of data items.

SAM attempts to combine the advantages of shared-memory and message-passing programming in a DSM environment. It provides the convenience of a shared-memory model, in which shared data can be accessed by name without any knowledge of where the data resides or which processor must send the data. SAM automatically replicates data values on multiple processors, to allow for multiple readers of the data and to provide faster access. Because all values in the system have distinct names, there is no need for a consistency protocol when values are replicated across processors. Accumulators are automatically migrated between processors as processors attempt to modify the accumulators. Because data items are communicated in user-defined units, there is no problem of “false sharing”.

Although DSM systems ease the task of parallel programming by providing a shared-memory view of data, applications written in a shared-memory style lose many of the efficiencies of pure message-passing programs. Message-passing programs minimize communication by sending data via point-to-point messages between nodes. The latency of message sends is usually partially hidden by attempting to send messages before they are needed remotely. All necessary synchronization between tasks occurs during the communication of data via messages. In contrast, data communication in DSM systems usually involve at least one round-trip delay to send a request message and receive a reply. Extra synchronization operations must be used to guard access to shared data and ensure accesses are properly ordered.

SAM provides the primitives to support message-passing efficiency in a DSM environment. Because synchronization is built in to the access to values and accumulators, no extra synchronization operations are required to create producer-consumer or mutual exclusion relationships. A task that creates a data item can send (“push”) that item to a remote processor to reduce latency and eliminate request overhead if that remote processor is likely to access the item. Data can be fetched asynchronously, thus allowing for the possibility of prefetching, pipelining reads of data, and/or
hiding latency with concurrency. Higher-level systems based on SAM can use these primitives to improve performance by applying communication optimizations based on higher-level knowledge available to the system.

SAM has been implemented on the Intel iPSC/860, the Intel Paragon, the Thinking Machines CM-5, IBM SP1, and networks of workstations using PVM.

2 Files in the Distribution

The distribution includes the source for the SAM run-time library, sources for additional run-time routines needed for an implementation of the Jade parallel programming language in SAM, sources for a SAM/Jade preprocessor that automatically generates the functions necessary for SAM to transmit user-defined data types and also parses Jade constructs, and sources for several complete applications written using SAM or Jade or both. Jade is a complete parallel programming language that allows the programmer to express parallelism in serial programs by specifying how different parts of the program access shared data.

The top-level directory ‘sam’ contains subdirectories ‘bin’, ‘doc’, ‘fe’, and ‘src’. ‘bin’ contains the scripts and executables necessary to run the SAM/Jade preprocessor. This directory should be included in your shell path, if you are using the preprocessor. ‘doc’ contains this document and a document describing the Jade language. ‘fe’ contains the sources for the SAM/Jade preprocessor. ‘src’ contains the basic run-time source for SAM, which builds the library ‘libsam.a’, and additional run-time source for a Jade implementation in SAM, which builds ‘libjade.a’. There are four subdirectories of ‘src’ which contain the sources for several parallel applications written using SAM and/or Jade:

- BH - Barnes-Hut n-body simulation
- GROBNER - symbolic algebra code for computing the Grobner basis of a set of polynomials
- BCF - code to do a Cholesky factorization of a sparse matrix using a block decomposition, and code to do a multiple minimum degree ordering of a sparse matrix to determine the optimal pivoting to minimize fill
- WATER - water simulation code (from the SPLASH benchmark suite)
- SEARCH - simulation of the interaction of several electron beams using Monte Carlo methods

These applications illustrate a variety of ways of using SAM and/or Jade. Water and Search are strictly Jade applications. The Grobner basis code uses SAM primitives directly, but use the Jade threads package to create tasks dynamically. The Barnes-Hut code, Block Cholesky code, and multiple minimum degree ordering code are written entirely using SAM in an SPMD (one process per processor) style.

Also under ‘src’ is an ‘include’ directory that contains the public include files. The file <sam.h> contains definition of all the data types and prototypes of all the functions described in the following sections. It should be included in any file that calls SAM primitives. The file <jade.h> should be included in any file that uses Jade constructs. (It automatically includes <sam.h>). The files <put.h>, <get.h>, and <size.h> should be included in any file that defines type functions, as described in Section 4.

Each application directory under ‘src’, as well as ‘src’ itself, contains several sample makefiles, ‘iPSC.Makefile’, ‘PMAX.Makefile’, ‘CMMD.Makefile’, and ‘SP1.Makefile’, to be used in compiling for the iPSC/860, the DecStation (running PVM), the CM-5, and the IBM SP1 (running PVM) respectively. Appropriate makefiles for SGI, Sparc, or IBM RS6000 workstations (running PVM) can be obtained by modifying the ‘PMAX.Makefile’ to define ‘-DSGI’, ‘-DSUN4’, or ‘-DRS6K’ instead of ‘-DPMAX’. Additionally, the makefile in ‘src’ must be changed to include ‘sgi_context.o sgi_invoke.o’, ‘sparc_invoke.o’, or ‘rios_invoke.s’ in libjade.a, rather than ‘pmax_context.o pmax_invoke.o’. For SGI workstations, the extra text ‘-lsun’ must be added to the link line for the applications. On Suns (and hence the CM-5), ‘gcc’ or ‘acc’ must be used for compilation, since some of the SAM and application source use function prototypes. The workstation implementation using PVM works with either PVM 2.4.1 or PVM 3.3, which should be obtained from elsewhere. (Send mail to netlib@ornl.gov with a subject line of “send index” or access it via the World Wide Web at http://www.netlib.org/pvm3.) To use PVM3, all makefiles should be modified to include an additional define ‘-DPVM3’ on the compile line. For PVM and PVM3, the makefiles must also be
modified to reference the correct path name of the installed PVM libraries. The iPSC/860 makefiles are for doing a cross-compilation in which the name of the cross-compiler is ‘icc’. An appropriate makefile for the Paragon can be obtained by adding the `-DNX` define to each ‘iPSC.Makefile’, removing all references to the cross-compiler ‘icc’ (since everything can be compiled on the Paragon itself using ‘cc’), changing ‘ar860’ to ‘ar’, and adding ‘-nx’ to the link lines.

3 Basic Types

Under SAM, each data item produced by a computation is named explicitly. All items are named by an ordered pair of numbers, which are typically written as (object id, version id). In the common case of modeling imperative data, the “object id” can be used to specify a particular object, and the “version id” specifies a particular version of the object. The types of object ids and version ids are `object_id` and `version_id`, respectively, which are really just unsigned 32-bit integers:

```c
typedef unsigned int object_id;
typedef unsigned int version_id;
```

The user can either explicitly choose the object ids and version ids for each value created, or use utility functions (described below) that provide unique ids each time they are called.

The functions that create and access values or accumulators return a local pointer to the data. The data type of this pointer is ‘object’:

```c
typedef void *object;
```

A few of the functions use or return a processor number. The data type for a processor number is ‘proc’:

```c
typedef int proc;
```

Some functions take as an argument an arbitrary user-defined pointer that is intended to indicate the “task” making the call. The data type for these task identifiers is:

```c
typedef void *taskid;
```

4 Type Functions

SAM can be set up to run in either homogeneous or heterogeneous environments (heterogeneous by default). There are several differences in the interface to SAM depending on whether it is used for homogeneous or heterogeneous systems, which we will describe at various points below. One big difference occurs in describing the data types of the data items that are created. On homogeneous systems, SAM only needs to know the size of a particular data in order to manipulate it properly. Because the data representation of all machines involved is the same, data of any particular type can just be transmitted in a message as a fixed number of bytes. However, on heterogeneous systems, the system must know more about the type of data that is stored in each value, so that it can pack up and unpack the data for transmission in a message and properly account for different data representations on different machines. All of this can be encoded in a single function associated with each data type called the ‘type function’. This function encodes the type, by providing the ability to pack, unpack, free, or determine the size of any data of that type. In addition, because it can describe in a general way how to handle complex data, this type function can be used to manipulate complex data types that contain pointers and/or consist of data that is not necessarily all allocated contiguously.

It is important to note that it is not typically necessary to deal with type functions in any way, because the SAM/Jade preprocessor can automatically generate the appropriate type functions for most C data types. It is only necessary to write type functions for data types that should be packed in some special way.

Each type function should have a prototype as follows:
int typefunc(object p, int operation);

If the value of operation is TYPE_BASESIZE, then typefunc should return the basic size of the data of the corresponding type on the local machine. The basic size includes just the size of the basic storage, excluding any other storage necessary because of pointers in the data type. For example, for an array of floats, the basic size is the entire size of the array. However, for a type that is a structure with pointers to a bunch of other data, the basic size is just the size of the structure, excluding the storage required for all the other data. If the value of operation is TYPE_PUT, then typefunc should pack up the corresponding type pointed to by p for transmission in a message. This packing operation should be built using the functions described below for packing up all the basic data types. Similarly, if the value of operation is TYPE_GET, then typefunc should unpack the corresponding type from a message into a pre-allocated area pointed to by p, again using functions described below. The pre-allocated memory is just for the basic size of the data type; any other storage required (because of pointers in the data type) should be dynamically allocated using malloc. If operation is TYPE_FREE, then typefunc should free up (using destroy_part_data) all the storage associated with the data pointed to by p except the basic storage. If operation is TYPE_MSGSIZE, then typefunc should compute and return the number of bytes (size) required to send the data associated with p in a message. The functions used in computing these sizes are described next.

A variety of functions are provided for doing the packing, unpacking, and size computations for basic types. The type functions for more complex types are built up using these basic types. For efficiency, there are also operations to manipulate arrays of each basic type as well. Note below that the put and size operations on scalar basic types take as an argument the actual value of the data. The get operations take a pointer to the data, since the get operations must return the data that has been unpacked. All array operations take as arguments a pointer to an array of the basic type and a count of how many elements there are in the array.

void byte_put(unsigned char b);
void byte_array_put(unsigned char *bp, int n);
void char_put(char c);
void char_array_put(char *cp, int n);
void short_put(short s);
void short_array_put(short *sp, int n);
void int_put(int i);
void int_array_put(int *ip, int n);
void float_put(float f);
void float_array_put(float *fp, int n);
void double_put(double d);
void double_array_put(double *dp, int n);

void byte_get(unsigned char *bp);
void byte_array_get(unsigned char *bp, int n);
void char_get(char *cp);
void char_array_get(char *cp, int n);
void short_get(short *sp);
void short_array_get(short *sp, int n);
void int_get(int *ip);
void int_array_get(int *ip, int n);
void float_get(float *fp);
void float_array_get(float *fp, int n);
void double_get(double *dp);
void double_array_get(double *dp, int n);

int byte_size(unsigned char b);
int byte_array_size(unsigned char *bp, int n);
5 Initialization Functions

The SAM system is initialized by calling the `start_sam` and `set_num_machine_proc()` functions and terminated via the `stop_sam` function:

```c
void start_sam(proc this_proc, proc num_proc);
void set_num_machine_proc(proc n);
void stop_sam(int waitflag);
```

The `start_sam` function should be called on each processor with an argument indicating the node number of the local processor (`this_proc`) and the total number of processors (`num_proc`) involved in the computation. The `set_num_machine_proc()` function should be called with the number of processors in the “partition” that the computation is running on. For example, on the CM-5 or iPSC/860, the user may be running a 12-processor computation on a 16-processor partition. SAM will use the first `num_proc` of those processors in the partition for the computation. SAM must know the total number of processors in the partition to initialize the unused processors properly and handle hardware broadcasts. `stop_sam` should be called on each processor when the computation is complete. If `waitflag` is TRUE, `stop_sam` includes an implicit barrier which blocks processors at the `stop_sam` call until all processors have called it. This barrier should be used, unless the application already contains an implicit barrier that ensures that no processors exit until the computation has finished. (Processors must not exit prematurely, since they may need to respond to incoming request messages.)

The values of `this_proc` and `num_proc` can be accessed at any later time during the computation using the `get_this_proc` and `get_num_proc` functions, respectively:

```c
proc get_this_proc();
proc get_num_proc();
```

The value of `num_proc` can also be reset after the `start_sam()` call:

```c
void set_num_proc(proc np);
```

One specialized function `sam_read_conf` is provided that does much of the above initialization automatically for straightforward applications:

```c
void sam_read_conf(char *procname, *pp, *np);
```

This function attempts to determine the appropriate values of `this_proc` and `num_machine_proc`. For PVM applications, this function reads a file ‘host.list’ in the same directory as the executable that lists the hosts on which PVM processes should be started. It starts up these processes using PVM and determines the value of `this_proc` for each process. It returns the values of `this_proc` and `num_machine_proc` in `*pp` and `*np` respectively. For other machines, this routine does similarly, except that it does not have to start up processes on the other nodes. The value of `num_machine_proc` is the size of the machine partition that the application is running in. The application
often determines the value of \texttt{num\_proc} (the actual number of nodes participating in the parallel computation) via an argument supplied by the user.

When a processor attempts to access a value that was created on another processor, SAM automatically replicates the value on the local processor for faster access. These replicated values are essentially managed as a cache on the local processor so that they are immediately available if the processor attempts to access the value again in the future. The user can set the size of this ‘cache’ via the \texttt{set\_limit\_mem\_value} function:

\[
\textbf{void set\_limit\_mem\_value(int limit);}\]

If the total storage size of the values on an individual processor reaches this limit, then SAM will discard the least-recently-used data that is cached but unneeded, so as to keep the memory use below this limit.

6 Basic Functions of SAM

6.1 Shared Functions

Another difference between homogeneous and heterogeneous systems is the need to have a representation of a function pointer that is independent of architecture. The address of the function on an individual processor is not sufficient, since the address of the function may vary from machine to machine in a heterogeneous environment. The SAM/Jade preprocessor allows a programmer to specify and reference such \textit{shared functions} easily, as described in section 10.2.

The preprocessor uses the following primitives to handle shared function pointers properly. These functions may be used directly if the preprocessor is not used:

\[
\textbf{void set\_shared\_function(int function\_number, int (*fn)());} \\
\textbf{int allocate\_shared\_function(int n);} \\
\textbf{int (*)() get\_shared\_function(int function\_number);}\]

The \texttt{set\_shared\_function} function associates a global number \texttt{function\_number} with a particular function \texttt{fn}. It should be called during initialization on all processors to associate the local address of the function with the global identifier (function number) for the function. The value of \texttt{function\_number} should be obtained using \texttt{allocate\_shared\_function}, which allocates a block of \texttt{n} shared function numbers and returns the first function number. If calls to \texttt{allocate\_shared\_function} are made in the same order on each processor, then the function numbers returned will be identical on each processor, thereby ensuring that the same shared function numbers are associated with the shared functions. A typical use of these functions is as follows:

\[
n = \texttt{allocate\_shared\_function}(2); \\
\texttt{set\_shared\_function}(n, \texttt{shared1}); \\
\texttt{set\_shared\_function}(n+1, \texttt{shared2});\]

\texttt{get\_shared\_function} is used to get the local address of the indicated function.

6.2 Creating Names

The next group of functions are for creating names for values:

\[
oi = \texttt{new\_object\_id}(); \\
v1 = \texttt{new\_version\_id}(); \\
v12 = \texttt{next\_version\_id(version\_id v1)};\]

\texttt{new\_object\_id}() and \texttt{new\_version\_id}() create new object and version identifiers, respectively, that are unique system-wide. \texttt{next\_version\_id}() is used to create the next in a sequence of unique versions, where it makes sense to consider a set of versions in an ordered sequence. All version ids in that sequence will be unique with respect to the
ids returned by other interleaved calls to new_version_id. The user may employ these functions, or assign unique object and version ids to values using his own system.

In the SAM implementation, the “home” processors for a value (or accumulator) is the processor that allocated the object id of the processor. That “home” processor will service remote requests for that data when a requesting processor does not know where the data is located. Sometimes, it may make sense to ensure that the home processor of a value or sequences of values matches the processor that created the value. This may be done by allocating an object id on the processor that will create values associated with the object id, or via the following function:

oi = new_proc_object_id(proc p);

On any processor, new_proc_object_id returns a unique object id whose “home” processor is p.

6.3 Creating Values

The following functions are used to create a value:

object begin_create_value(object_id oi, version_id vi, int typespec,
int nitems, int flags);
void end_create_value(object_id oi, version_id vi);

begin_create_value creates a value with the specified object and version ids. On homogeneous systems, the typespec argument is just the size of the type being created. On heterogeneous systems, typespec is the global id (as created using set_shared_function) of the type function corresponding to the type of data being created. It is usually easiest to use the SAM/Jade preprocessor to generate the type functions automatically. In this case, the user would specify the appropriate type using the typename construct described in Section 10.2. nitems specifies how many items of the specified type the value will hold. The appropriate amount of storage is allocated for the value, and a pointer to the storage is returned. The storage provided is not guaranteed to be initialized to zeros. The user may then initialize the contents of the value in any desired way. The value should at least be initialized sufficiently so that there are no stray pointers in the value. These uninitialized pointers will cause the packing function for the value to pack incorrect data if the value is transmitted to another processor. The flag is normally zero; we describe some other possible values for flag below that affect the way in which the value is managed. If the user wishes to allocate more storage (because the data type contains pointers), he should use malloc (or create_part_data, if using the SAM/Jade preprocessor. See Section 10.2).

When the value’s contents have been fully initialized, the value is “completed” by calling end_create_value. The value is then available to be used by any other computation in the system. After end_create_value is called, the local pointer returned by begin_create_value should no longer be used.

6.4 Accessing Values

The following primitives are used for accessing a value:

object begin_use_value(taskid tsk, void (*fn)(), object_id oi, version_id vi);
void end_use_value(object_id oi, version_id vi);
object access_value(object_id oi, version_id vi)
int is_created_value(taskid tsk, void (*fn)(), object_id oi, version_id vi);

The function begin_use_value attempts to access the indicated value and returns a pointer to the value if it is locally available. begin_use_value is asynchronous (non-blocking), in that it returns immediately whether the value is available or not. If the value is not immediately available, then begin_use_value returns a NULL pointer. In addition, the caller is notified later when the value has been fetched to the local processor by a call to the user-supplied function fn. The prototype of fn should be as follows:

void fn(taskid tsk, object_id oi, version_id vi, object p);
When the value is available locally, the function \texttt{fn} is called with the user-supplied \texttt{tsk} argument, the object id, the version id, and a local pointer to the value that has been fetched. The intent is that the \texttt{tsk} argument can be used to specify the task that attempted to use the value. A blocking access can easily be built using the non-blocking access operation. For example, we give in Section 12 the definition of a version of \texttt{begin\_use\_value} that spins until the value becomes available. If a task package is in use, it is easy to write a version that suspends the current task if the value is not available and resumes the task when the value is available locally.

Because the function \texttt{fn} is called by the message-handling code within the SAM system, it should not do extensive computation or stall indefinitely waiting for an event. However, it may call other SAM primitives. Often, the callback function may just set a application “flag” variable that signals the main application code that the value has been received.

The user can access the local copy of the value using the pointer returned by \texttt{begin\_use\_value} or provided as an argument to \texttt{fn}. Although there is no checking by the system, the copy of the value is read-only and the user should not attempt to modify any of the data. To create a new value which is a modification of an existing value, the user should use the \texttt{rename\_value} function described below. \texttt{end\_use\_value} is used to indicate when the caller has finished accessing a value that it has fetched. After the call to \texttt{end\_use\_value}, the local pointer returned by \texttt{begin\_use\_value} is no longer valid. After the user has called \texttt{begin\_use\_value} and before calling \texttt{end\_use\_value}, the local pointer to the value may also be obtained by a call to \texttt{access\_value}. Such functionality is useful when many computations in a task are accessing the value, and it is desirable to avoid passing the local pointer around everywhere.

\texttt{is\_created\_value} checks if a value has been created yet, without actually bringing it to the local processor. It can be used to hold off on starting a computation until all the values it requires are available. \texttt{is\_created\_value} is also non-blocking; it returns immediately with a return value indicating whether the value has been created yet. If the value has not yet been created, it notifies the caller later when the value is created via a call to the function \texttt{fn}. The prototype of \texttt{fn} should be as follows:

\begin{verbatim}
void fn(taskid *tsk, object_id oi, version_id vi, proc p);
\end{verbatim}

The function \texttt{fn} is called with the the user-supplied \texttt{tsk} argument, the object id, the version id, and a process argument that indicates a processor on which the value is currently available.

### 6.5 Transmitting Values

The \texttt{push\_value} and \texttt{move\_value} primitives send a value to a processor:

\begin{verbatim}
void push\_value(object_id oi, version_id vi, proc p);
void move\_value(object_id oi, version_id vi, proc p);
\end{verbatim}

\texttt{push\_value} “pushes” a copy of the indicated value to a remote processor. The argument \texttt{p} specifies the processor to which the value is sent. As a special case, a value of \texttt{-1} for \texttt{p} in a call to \texttt{push\_value} broadcasts the value to all other processors. This operation may be more efficient than pushing individually to all processors on machines that have hardware support for broadcasts. \texttt{push\_value} has no effect if the value is not available on the local processor.

\texttt{move\_value} moves a value and frees up the local copy of the value if there are no other local users. \texttt{move\_value} is almost identical to \texttt{push\_value}, except that if it is called on the processor that originally created the value, it moves the value and changes the destination processor to be the owner of the value. In this way, the value can be freed on the source processor, since it is no longer the owner of the value. \texttt{move\_value} is typically used immediately after creating a value. \texttt{move\_value} is identical to \texttt{push\_value} on any processor that does not own the value. Because of the semantics of \texttt{move\_value}, it does not make sense to specify a broadcast by specifying \texttt{p} as \texttt{-1}.

These primitives support the efficiency of message passing by allowing a value to be sent to a processor so that the value is available locally when that process requests access. In this way, the two-way request-response protocol typically of shared memory access is replaced by the more efficient one-way protocol typical of message passing. \texttt{move\_value} optimizes communication further by making the destination process the owner of the value that is
transmitted. No further communication with the source processor is needed to ensure that the value is freed when longer needed. If the destination processor is the only user of the value, then all further communication is avoided. Note, however, that both primitives are purely optimizations and do not ever affect the semantic behavior of a SAM program.

The efficiency of dealing with values in this message-passing style can be increased further by creating these values with the NO_REMOTE flag. Use of this flag eliminates, where possible, system messages associated with providing global access to the value. It is assumed that the value will always be pushed or moved to the processors that will require the value, so no extra system messages are sent in order to allow shared memory access to the value.

6.6 Memory Management

There are two primitives that are crucial for doing proper storage management of values:

```c
void free_value(oi, vi);
void update_value_count(oi, vi, hold, amount);
```

`free_value` indicates that all potential accesses to a particular value by all the processes in the system have occurred. A call to this primitive indicates that the last copy of a value can be removed whenever any remaining local accesses to the value have finished. The `update_value_count` call provides a more dynamic way of determining when the final copy of a value can be reclaimed. It provides similar information, but in the form of the total number of potential users of a value. This global count contains two components, as indicated by the last two arguments of the call. The “amount” component is increased to indicate a corresponding increase in the number of users. The “hold” component is increased to indicate an indefinite number of potential users by a particular part of the program, thereby putting a hold on the value (preventing it from being removed). When the number of users in this part of the program is determined, the user count can be increased by the appropriate amount and the hold count decremented via another call to `update_value_count`. When the number of users has been fully specified (no holds left), then the last copy of the value can be removed when the value has been accessed by the indicated number of users. In order to use this method of tracking the number of users of a value dynamically, the value must be created with a flag value of TRACK_USE. All values start out with a hold value of 1. Thus, a value will never be completely removed from the system until the initial hold is removed via a call to either `free_value` or `update_value_count`.

6.7 Giving New Names to Values

The following primitive is for creating new values from existing values:

```c
object rename_value(taskid *tsk, void (*fn)(), object_id oi, version_id vi1,
                     version_id vi2);
```

This primitive is intended to support the creation of values that represent different versions of a single piece of data. Hence, it only allows the creation of new values from old values with the same object ids. `rename_value` should only be called after the source value has been accessed (via `begin_use_value`) or created (via `begin_create_value`) on the local processor. It attempts to rename the source value named by `(oi, vi1)` to have a new version id `(oi, vi2)`. If it is immediately successful in renaming the value locally, it returns a pointer to the new value. The value `(oi, vi2)` is still in an ‘incomplete’ state, and can now be modified as desired by the user. When the user has done the necessary modification, the value is completed by calling `end_create_value` (just as if the value had been created via `begin_create_value`).

`rename_value` may be delayed if there are local computations that are still accessing the local copy of the value to be renamed, or if there are remote computations that wish to access the value, but do not yet have a copy. If so, `rename_value` returns a NULL pointer. When the rename has succeeded, the function `fn` is called; `fn` should have the same prototype as the function passed to `begin_use_value`.

A value is given two different names via the following primitive:

```c
void equate_value(oi, vi1, vi2);
```
This call makes the two names \((oi, vi1)\) and \((oi, vi2)\) refer to the same value. This call is legal even if a value does not yet exist under one or both of the names. However, it is an error if both names already specify existing values. This primitive is useful for modularity reasons: different parts of a computation can refer to the same value by different names that have meaning within their particular module.

7 Accumulators

Accumulators capture the notion of data that is updated by a number of processes in any order. A simple example of such data is a running total, which is used to add up the results returned by a number of independent processes. The steps in the summing process are commutative and associative, so the individual values may be accumulated in any order. Any normal value can be converted into an accumulator, and this accumulator can then be updated by a number of tasks before producing a “final” result.

One way to think of accumulators is as a sequence of values whose names are implicitly managed by SAM. Thus, when a process calls the function to update an accumulator, it specifies that it wants to fetch the “current” value in the accumulator sequence and create the “next” value in the sequence. SAM takes care of the naming details and ensures that only one process is ever updating the current value.

Another common idiom for which it is appropriate to leave the management of names to the system is chaotic access to data. Chaotic algorithms are parallel computations in which different processes do not always use the most up-to-date data. Chaotic algorithms are useful for iterative computations that will converge and produce an acceptable result even when “old” data is sometimes used during an iteration. SAM supports the idiom of chaotic computation via functions that provide access to a “recent” value of the accumulator. This recent value is immutable and can only be read; i.e. it cannot be updated. It is not guaranteed to be the most current value in the accumulator sequence. However, a “recent” value may be all that is necessary for some kinds of computation. The chaotic read operation can often proceed without communication because a recent copy of the accumulator is available on the local processor.

7.1 Creating and Accessing Accumulators

The prototypes of the functions for creating and accessing accumulators are as follows:

```c
object begin_create_accum(object_id oi, version_id vers,
            version_id dvers, int s, int n,
            int flags);
void create_accum_value(object_id oi, version_id vi1, version_id vi2,
            version_id vi3);
object begin_update_accum(taskid *tsk, void (*fn)(), object_id oi, version_id vi);
void end_update_accum(object_id oi, version_id vi);
object begin_read_accum(taskid *tsk, void (*fn)(), object_id oi, version_id vi);
void end_read_accum(object_id oi, version_id vi);
```

The `begin_create_accum` and `create_accum_value` calls are used to create an accumulator which can be updated a number of times before it finally becomes immutable. `begin_create_accum` creates an accumulator with name \((oi, vers)\) and returns a pointer to the accumulator. The accumulator can then be initialized; the initialization is terminated by a call to `end_update_accum`. With the `create_accum_value` primitive, the accumulator is created by renaming the value named by \((oi, vi1)\) to \((oi, vi2)\), which is the name by which the accumulator is referenced. The accumulator is fetched and prepared for an update by `begin_update_accum`, which returns a local pointer to the accumulator. As with `begin_use_value`, `begin_update_accum` is asynchronous (non-blocking). If the accumulator is not available locally or is currently being updated by another task, then `begin_update_accum` returns a NULL pointer. In addition, the caller is notified later when the accumulator is present locally and not being updated by any other task by a call to the user-supplied function `fn`. The prototype of `fn` is the same as for `begin_use_value`:
7.2 Modifying an Accumulator via RPC

The function `begin_update_accum` operates by getting mutual exclusion on the accumulator and bringing the latest accumulator data to the local process for update. An accumulator can also be updated remotely via RPC with the `modify_accum` function:

```c
int modify_accum(taskid tsk, void (*fn)(), object_id oi, version_id vi, sharedfn modifyfn, int arg, int *rp);
```

This primitive determines the processor that contains the current value of the accumulator and makes an RPC call to that processor to modify the accumulator. On that processor, the shared function indicated by `modifyfn` is invoked as follows when mutual exclusion on the accumulator has been obtained:

```c
modifyfn(object_id oi, version_id vi, object p, int arg)
```

The argument `p` is a pointer to the local copy of the accumulator and `arg` is an integer-sized argument passed to the `modify_accum` call. The `modifyfn` function may also have an integer-sized return value. `modify_accum` returns `TRUE` if the accumulator is available locally and the modification can take place immediately. If so, then the return value of the `modifyfn` call is returned in `*rp`. (`rp` can be `NULL`, in which case any return value of `modifyfn` is ignored.) Otherwise, `modify_accum` returns `FALSE`. If `fn` is non-NULL, then the caller is notified when the RPC is complete via a callback to `fn`, as follows:

```c
fn(taskid *tsk, object_id oi, version_id vi, int rv)
```

Here, `rv` is the return value of the call to `modifyfn`.

7.3 Moving an Accumulator

Analogous to the `move_value` operation, an accumulator can be moved to another processor via `move_accum`:

```c
void move_accum(object_id oi, version_id vi, proc p);
```

The accumulator should be present on the local processor, but not currently being updated. If the accumulator is not on the local processor, then `move_accum` does nothing.

8 Useful Higher-level Functions

Also provided with SAM are several higher-level functions that can be built from the basic primitives:

```c
void create_barrier(object_id oi, int numprocs);
void spin_wait_barrier(object_id oi);

object spin_begin_use_value(object_id oi, version_id vi);
object spin_rename_value(object_id oi, version_id sv, version_id dv);
object spin_begin_update_accum(object_id oi, version_id vi)
object spin_begin_read_accum(object_id oi, version_id vi)
```

`create_barrier` is used to create a spinning barrier with the specified object id and for the specified number of tasks or processors. When a processor/task calls `spin_wait_barrier`, it will block (via spinning) until `numprocs` other processors/tasks have also called `spin_wait_barrier`. While spinning, the local processor will still serve requests by other processors.

`spin_begin_use_value` provides a synchronous version of `begin_use_value`. It will attempt to access the specified value and return a pointer to a local copy of the value. If the value is not immediately available, `spin_begin_use_value` will spin until the value is available locally and then return a pointer to the local copy. `spin_rename_value`, `spin_begin_update_accum` and `spin_begin_read_accum` operate similarly. These accesses are ended as normal by `end_use_value`, `end_update_accum`, and `end_read_accum`.
9 Utility Functions

The following functions are for gathering time information about SAM runs:

```c
void get_time(long *tp)
double ticks_to_seconds_time(long t)
```

get_time returns in *tp a machine-dependent long integer that indicates the current time on the local processor. ticks_to_seconds_time converts a long integer that is the difference between two values returned by get_time into a double-precision floating point number that represents the seconds elapsed between the two calls to get_time.

Because the malloc implementations on many machines are quite slow, SAM includes its own version of malloc, realloc, calloc, and free that are very fast. However, SAM’s memory allocator trades off memory use for speed by only allocating memory in sizes that are powers of two. Therefore, an application may use up an unexpectedly large amount of memory if it allocates a lot of memory in chunks that are slightly bigger than powers of two.

10 Jade Implementation in SAM

Included with the SAM library is an implementation of the Jade language. The support for Jade includes a number of extra components:

- a preprocessor ‘jcc’ that converts Jade constructs to run-time calls and automatically generates type functions for Jade objects
- a linking script ‘jld’ that, in addition to doing the necessary linking, creates the initialization functions that make the automatically generated type functions accessible as shared functions
- a threads package that provides automatic load balancing and placing of tasks on specific processors
- an implementation of the Jade run-time calls in terms of the SAM primitives and the functions in the thread package

The Jade language is described in the file ‘jadedoc.ps’ in the ‘doc’ directory. One limitation of the SAM implementation is that it does not implement hierarchical objects.

Because Jade is implemented using SAM, it is easy to use Jade constructs and SAM primitives in the same application. For example, important distributed data structures may be implemented directly in terms of SAM primitives, whereas the rest of the data in the application is written using Jade. In the Barnes-Hut n-body simulation algorithm, the oct-tree and list of bodies are implemented directly using SAM primitives, while the rest of the shared data is handled using Jade.

10.1 Using Jade

The makefiles for the applications included in the distribution illustrate the use of the ‘jcc’ and ‘jld’ programs for compiling Jade programs. Full Jade applications must be linked with both the ‘libjade.a’ and ‘libsam.a’ libraries. In addition, the main function must be renamed to start, so that the main function provided in ‘libjade.a’ can do the proper initialization of the Jade run-time system.

Various parameters of the Jade execution are then controlled by a file in the directory containing the executable called ‘jade.hosts’. The ‘jade.hosts’ file lists the hosts involved in the computation and the values for several parameters on each of the hosts. When SAM and Jade are being used on workstations with PVM, the host names are the actual names of the workstations involved in the computation. A workstation (or SGI multiprocessor) may be named several times in the file to have more than one process run on that machine. On the CM-5, iPSC/860, and Paragon multiprocessors, the host names given are irrelevant. The lines of the ‘jade.hosts’ just specify consecutive nodes on the machine; the number of lines in the ‘jade.hosts’ file therefore determines the number of nodes used in the computation. A sample ‘jade.hosts’ file is as follows:
The first entry on each line is the host name. The second entry is amount of main memory (in bytes) dedicated to caching Jade objects (values and accumulators in SAM programs). The third and fourth entries on each line are parameters for preventing unlimited spawning of tasks from overwhelming the resources of the system. For our example ‘jade.hosts’ file, if a node ever has 800 outstanding tasks in its queue, any task that attempts to spawn a new task on that node will be suspended. The spawning task will be suspended until less than 700 tasks exist on the node. The fifth parameter determines the number of tasks for which the Jade implementation will attempt simultaneously to prefetch their initially requested data. A larger value will attempt the fetching of data for more tasks at once. A value of ‘1’ for the sixth parameter turns off all dynamic load balancing. Tasks can still be placed on remote processors using withonly with the ‘@’ construct, but tasks are not otherwise migrated. The final parameter specifies which directory the application should run in on each host with respect to the directory where the executable is.

The Jade run-time system will abort with an error message if a shared object is accessed by a task in a way which the task did not declare in its access specification section. Currently, the easiest way to find out where the bad access is occurring is to run the program under dbx and put a breakpoint in exit. When the error occurs, the program will stop in exit, and you can look at the stack to see where the access is occurring.

At the end of the run, the Jade run-time system outputs a bunch of statistics and a list of some of the compile-time and run-time execution parameters.

### 10.2 The SAM/Jade Preprocessor

The option ‘-FEdDJ’ should always be supplied to the Jade/SAM preprocessor ‘jcc’. The option ‘-comp’ can be used to specify the use of a C compiler other than ‘cc’. Except for the ‘-FE’ and ‘-comp’ options, ‘jcc’ takes exactly the same arguments as the C compiler. The output of ‘jcc’ for a file x.c can be examined by looking at file .x.c. The additional option -FEl to ‘jcc’ tells the front end not to put in line directives, so dbx will reference the .x.c file, rather than x.c.

The SAM/Jade preprocessor can also be used to automatically generate the type functions used in applications written strictly in terms of SAM primitives. A type function can automatically be generated by using the typename construct in a SAM program. For example, typename(struct node) causes the preprocessor to generate a type function for the struct node type and to replace the typename expression with an expression that will generate a shared function identifier that specifies that type function. Hence, a typical use of typename in a SAM program is as follows:

```c
p = begin_create_value(oi, vers, typename(struct node), 1, 0);
```

If a SAM program uses typename anywhere, but does not link with the Jade library, it must follow the call to start_sam on each processor with the following initialization call:

```c
init_type_table(allocate_shared_function(num_type_table()));
```

The init_type_table and num_type_table functions are generated automatically by jld.

One change must be made to the way SAM data is allocated if the automatically generated type functions are used. The user must use special functions to allocate, reallocate, and free any extra data that is part of a value or an accumulator (instead of just using malloc, realloc, and free):

```c
object create_part_data(int s, int n);
object realloc_part_data(object o, int s, int n);
void destroy_part_data(object po);
```
These functions are necessary so that the automatically generated type functions can keep track of the length of variable-sized data allocated inside data items. As with `malloc`, `create_part_data` does not guarantee that the contents of the allocated memory is initialized to zero.

The preprocessor can also be used to manage pointers to shared functions properly. Any function can be made into a ‘shared’ function via the `shared` keyword:

```c
void shared
    sharedfn(int a, int b)
{
    ...
}
```

If that function name is then used as a function pointer (as in a `modify_accum` call), the preprocessor automatically does all the necessary bookkeeping to ensure that the function pointer is handled correctly even in heterogeneous environments.

## 10.3 The Threads Package

The threads package in the Jade implementation can also be used in SAM programs. Tasks can be created via the `withonly` construct, without including any Jade access specifications. In this case, the Jade/SAM preprocessor must be used, and the Jade library must be linked in. The optional `@` construct can be used to run tasks on specific processors. If tasks are not placed on specific processors, then they are automatically moved as appropriate for dynamic load balancing. In addition, the priority of a task can be set by calling the function `set_priority` in the access declaration section of the `withonly`:

```c
void set_priority(int prio);
```

Any task which has been given a priority has a higher priority than any task that has not been given a priority. Whenever a processor is looking for work, it will always run the highest priority task that is locally available that is ready to run. It will not, however, necessarily run the highest priority task in the whole system.

To use the Jade threads package, an SAM application must link with ‘libjade.a’ as well as ‘libsam.a’. As described above with Jade applications, its main function must be renamed to `start`, and its execution parameters are then governed by the ‘jade.hosts’ file.

## 11 Polling and Interrupts

By default, SAM does not use interrupts to serve incoming messages. That is, it only deals with incoming messages when the user calls a SAM function or when the user explicitly calls the function `poll_msg()`. For some coarse-grain applications, performance may be greatly affected by the frequency of polling, and it may be very important for the user to call `poll_msg` periodically during long periods of computation that have no calls to SAM functions.

On the iPSC/860, Paragon, and CM-5, the SAM implementation can optionally be set up to serve incoming messages using an interrupt-driven message handler (via the `ASYNC_MSG` compiling option in `<sam.h>`). While the user does not then have to worry about calling `poll_msg`, the user does have to deal with the fact that the message handler may run at any time during a computation. In particular, the callback functions associated with `begin_use_value`, `rename_value`, etc. may be called at any time. The user can ensure that the message handler does not run during a critical section by surrounding that section with the following two functions:

```c
void acquire_system()
void release_system()
```

At the very least, if interrupts are used, the user must surround all calls to the following functions with `acquire_system` and `release_system`:
These functions do not automatically disable interrupts on every call, since the operation to disable interrupts may be expensive. Interrupts are already disabled during execution of the callback functions associated with begin_use_value, begin_rename_value, begin_update_accum, and begin_read_accum.

12 Sample Program

Below is a sample SPMD program using SAM on the iPSC. In this example, SAM is configured for a heterogeneous environment and for message handling by polling. Note that the header file <sam.h> should be included; it defines all the necessary types and includes prototypes for all the SAM functions.

#include <sam.h>

main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{
  int p, n;
  int *ip;
  object_id oi;

  /* Programs loaded onto the iPSC/860 automatically start up running on
      * each node. mynode() and numnodes() are iPSC/860 built-in functions. */
  p = mynode();
  n = numnodes();
  start_sam(p, n);
init_type_table(allocate_shared_function(num_type_table()));
set_limit_mem_value(4000000)
set_num_machine_proc(n);

/* Create an integer value on processor 0 and access the value on */
/* processor 1. */
if (p == 0) {
   ip = (int *) begin_create_value(2, 1, typename(int), 1, 0);
   *ip = 4;
   end_create_value(2, 1);
}
else if (p == 1) {
   ip = (int *) spin_begin_use_value(2, 1);
   printf("int = %d\n", *ip);
}
end_sam(TRUE);

/* A definition of spin_begin_use_value() in terms of begin_use_value(). */
char * volatile fetch_p = NIL;

static void
fetch_callback(tsk, oi, vers, p)
taskid *tsk;
object_id oi;
version_id vers;
void *p;
{
   fetch_p = (char * volatile) p;
}

object
spin_begin_use_value(oi, vers)
object_id oi;
version_id vers;
{
   void *p;

   fetch_p = NIL;
   p = begin_use_value(NIL, fetch_callback, oi, vers);
   if (p)
      return p;
   while (fetch_p == NIL)
      poll_msg();
   return (void *) fetch_p;
}