

ORNL/TM-12744

Engineering Physics and Mathematics Division

Mathematical Sciences Section

DOLIB: DISTRIBUTED OBJECT LIBRARY

E.F. D'Azevedo
C.H. Romine

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Date Published: October 1994

Research supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Contents

1	Introduction	1
2	Previous Work	2
3	User Interface	3
	do_axpby	4
	do_declare	7
	do_destroy	9
	do_disable	10
	do_wait	11
	do_enable	12
	do_gather	13
	do_init	16
	do_isavail	17
	do_scatter	18
	do_gsync	19
	do_check	20
	do_setchsize	21
4	Implementation Details	22
	4.1 Caching	23
	4.2 Accessing a Global Shared Array	25
	4.3 Reclaiming Memory	26
5	Performance of DOLIB on Intel iPSC/860	26
6	Implementation of DOLIB using Polling	27
7	Summary	29
8	Obtaining the Software	29
9	Appendix	30
10	References	39

DOLIB: DISTRIBUTED OBJECT LIBRARY

E.F. D'Azevedo

C.H. Romine

Abstract

This report describes the use and implementation of **DOLIB** (Distributed Object Library), a library of routines that emulates global or virtual shared memory on Intel multiprocessor systems. Access to a distributed global array is through explicit calls to **gather** and **scatter**. Advantages of using **DOLIB** include: dynamic allocation and freeing of huge (gigabyte) distributed arrays, both **C** and **FORTRAN** callable interfaces, and the ability to mix shared-memory and message-passing programming models for ease of use and optimal performance. **DOLIB** is independent of language and compiler extensions and requires no special operating system support. **DOLIB** also supports automatic caching of read-only data for high performance. The virtual shared memory support provided in **DOLIB** is well suited for implementing Lagrangian particle tracking techniques. We have also used **DOLIB** to create **DOMIO** (Distributed Object Network I/O Library), which obtains over a 10-fold improvement in disk I/O performance on the Intel Paragon.

1. Introduction

Distributed memory multiprocessors have proven to be scalable and offer high performance. However, the limited memory on each processor node generally requires programmers to perform their own data decomposition, carefully moving needed data among nodes by explicit message passing. Writing parallel application code using message passing is both intricate and error prone. Virtual shared memory enables a programmer to make full use of the total aggregate (gigabyte) memory resources while avoiding the difficulties of message passing.

DOLIB (Distributed Object Library) supports virtual shared memory on the Intel Paragon and iPSC/860 distributed memory supercomputers. DOLIB enables all processors to operate directly on any part of a distributed global array through explicit calls to `gather` and `scatter`. Advantages of using DOLIB include: dynamic allocation and deallocation of huge (gigabyte) distributed arrays, both C and FORTRAN callable interfaces, and the ability to mix shared-memory and message-passing programming models for ease of use and optimal performance. DOLIB is independent of any language or compiler extensions, and requires no special operating system support. DOLIB supports automatic caching of *read-only* data for high performance.

Section 2 discusses previous work related to DOLIB. In section 3, we describe the user interface in the form of manual pages for the DOLIB procedures. In section 4, we provide implementation details of DOLIB, to aid the programmer in optimizing the use of DOLIB's global arrays. Section 5 illustrates the effectiveness and ease of use of DOLIB with a parallel matrix-matrix multiply. We have also used DOLIB to create DONIO, the Distributed Object Network I/O Library; details can be found in D'Azevedo and Romine [1]. DONIO obtains more than a 10-fold improvement in performance of concurrent disk I/O on the Intel Paragon. Finally, section 7 gives a short summary and a preview of upcoming developments in future versions of DOLIB.

2. Previous Work

The performance of virtual shared memory on a distributed memory system requires an effective caching strategy. Thus, much of the research into virtual shared memory, such as Li [2], Li and Hudak [3], and Stumm and Zhou [7], concerns intricate network protocols that maintain cache coherency in the presence of multiple concurrent updates.

Shiva [4] is a shared virtual memory system for the Intel iPSC/2 hypercube multiprocessor. **Shiva** uses the Memory Management Unit (MMU) page fault mechanism on each Intel i386 node to generate memory requests for remote pages. The implementation requires low level hardware and operating system support.

The **CHAOS** library [6] is an attempt to provide support for the parallel solution of irregular problems; that is, problems whose communication patterns are not easily predictable. **CHAOS** is a runtime library that can analyze the pattern of indirect addressing of arrays (such as $\mathbf{x}(\mathbf{ia}(i)) = \mathbf{x}(\mathbf{ia}(i)) + \mathbf{y}(\mathbf{ib}(i))$), and automatically devise an optimized schedule of communication. **CHAOS** supports irregular assignment of data arrays to processors by using a globally accessible *translation table* to describe the location of elements of the array. The loop iterations are automatically partitioned (or repartitioned) and assigned to processors (based on trying to optimize the resulting load balance and communication volume). A preprocessing phase constructs the required communication schedules for the given distribution of workload and data.

DOLIB avoids the complexity of cache coherency by supporting a restricted virtual shared memory. Specifically, **DOLIB** assumes that any global array with caching enabled contains *read-only* data. If the array is updated, it is the programmer's responsibility to flush the cache to prevent erroneous results. In many important applications such as distributed finite element matrix assembly, parallel sparse matrix factorization and Lagrangian particle tracking, cache coherency is not an issue. For example, global data such as a flow field typically remains constant throughout a time-step for Lagrangian particle tracking. At the beginning of the next time step, the cache can be flushed to prepare for recomputing

the flow field.

In section 4.1, we describe how caching is implemented in DOLIB, and discuss the DOLIB routines available to the user to control the cache.

3. User Interface

This section provides details on the syntax and behavior of each of the DOLIB primitives. They form the manual pages for DOLIB.

do_axpby

*do_axpby performs a vector update $y(list(:)) \leftarrow \alpha * x(:) + \beta * y(list(:))$ operation into the global shared array y .*

Synopsis

```
void do_iaxpby( int Iaf, int nsize, int *list,
               int *xarray, int ialpha, int ibeta )

void do_daxpby( int Iaf, int nsize, int *list,
               double *xarray, double dalpha, double dbeta )

void do_biaxpby( int Iaf, int nsize, int start,
                int *xarray, int ialpha, int ibeta )

void do_bdaxpby( int Iaf, int nsize, int start,
                double *xarray, double dalpha, double dbeta )

subroutine doiaxpby( Iaf, nsize, list, xarray,
                   ialpha, ibeta )
integer Iaf, nsize, list(nsize), xarray(nsize)
integer ialpha, ibeta

subroutine dodaxpby( Iaf, nsize, list, xarray,
                   dalpha, dbeta )
integer Iaf, nsize, list(nsize)
real*8 xarray(nsize), dalpha, dbeta

subroutine dobiaxpby( Iaf, nsize, start, xarray,
                    ialpha, ibeta )
integer Iaf, nsize, start, xarray(nsize)
integer ialpha, ibeta
```

```
subroutine dobdaxpby( Iaf, nsize, start, xarray,  
                    dalpha, dbeta )  
integer Iaf, nsize, start  
real*8 xarray(nsize), dalpha, dbeta
```

Input parameters

- | | | |
|-----------------------------|---|--|
| <code>Iaf</code> | – | <code>Iaf</code> is the array descriptor obtained from <code>do_declare</code> . |
| <code>nsize</code> | – | <code>nsize</code> is number of items involved in the vector update operation. |
| <code>list</code> | – | <code>list</code> holds the index list for the vector update operation, for those versions of <code>axpby</code> that use it. |
| <code>start</code> | – | <code>start</code> is the starting index into the global array referenced by <code>Iaf</code> for the vector update operation, for those versions of <code>axpby</code> that use it. |
| <code>xarray</code> | – | <code>xarray</code> is the (local) buffer area that holds the values for the vector update operation. |
| <code>ialpha, dalpha</code> | – | <code>ialpha, dalpha</code> is the scaling in <code>xarray</code> . |
| <code>ibeta, dbeta</code> | – | <code>ibeta, dbeta</code> is the scaling in the global array. |

Description

`do_axpby` (and all of its variants) performs a distributed vector update operation into the global shared array out of the specified buffer. The index set is described in `list` (`do_daxpby`), or is derived as a contiguous block of `nsize` elements starting at index `start` (`do_bdaxpby`). It is an error to have `nsize` \leq 0.

The `axpby` operation can be used in assembly of finite element matrices into a global stiffness matrix. The accumulation of individual entries is performed as a critical section by the owner of the data page and hence eliminates the need for exclusive access or locking of the global array.

Note that the scatter operation can be implemented as a special case of an `axpby` update with $\alpha = 1$, $\beta = 0$.

This is not a synchronous call. The calling process does not wait (block) until the `axpby` request is completed. Note that it is an error to access past the declared array size, `gsize`, set in `do_declare`.

do_declare

`do_declare` allocates a distributed global shared array and returns a descriptor to the distributed object. An implicit global synchronization is performed.

Synopsis

```
void do_declare( int *Iaf, char *name, int gsize,
                char *ctype, int pagesize, int blocksize )

subroutine dodeclare( Iaf, name, gsize, ctype,
                    pagesize, blocksize)

integer Iaf, gsize, pagesize, blocksize
character*(*) name, ctype
```

Input parameters

- `name` – `name` is a *null terminated* character string that distinguishes the global array declared. It is used for displaying informative error messages. A null terminated string can be generated as `name = "arrayname" // char(0)` in FORTRAN.
- `gsize` – `gsize` is the number of items (not bytes) in the global array.
- `ctype` – `ctype` is a *null terminated* character string that describes the type of the global array. Valid string values are “int” or “integer”, “double”, “real*8” or “double precision” and “char” or “character” (all either lower or upper case).
- `pagesize` – `pagesize` is the number of items in a page.
- `blocksize` – `blocksize` is the number of pages in a block. Note there are `blocksize*pagesize` entries in a block.

Output parameter

`Iaf` – `Iaf` is a descriptor to the global shared array.
`do_declare` returns a positive value in `Iaf` on success.

Discussion

`do_declare` allocates a distributed global shared array and returns a descriptor (positive integer) to the distributed object. The array is composed of fixed-size pages grouped as blocks. If enabled, caching is performed on a page as a unit. The blocks are distributed among all processors in wrap-mapped fashion. Actual storage is obtained by `malloc()` in the C language.

Important note: indexing conforms to the conventions of the language being used. That is, a global array is indexed with `1:gsiz` in FORTRAN, `0:(gsiz-1)` in C. The region of memory is not initialized in any way — assume it is garbage.

All processors must participate in `do_declare`. An implicit global synchronization is performed.

do_destroy

do_destroy deallocates global shared resources associated with the array descriptor. An implicit global synchronization is performed.

Synopsis

```
void do_destroy( int Iaf )  
  
subroutine dodestroy( Iaf )  
integer Iaf
```

Input parameters

Iaf – Iaf is the array descriptor obtained from `do_declare`.

Description

`do_destroy` deallocates the global shared resources associated with the array descriptor `Iaf`. Memory is returned to the heap by `free()` in the C language.

All processors must participate in `do_destroy`. An implicit global synchronization is performed.

do_disable

`do_disable` disables caching of data pages associated with the global array.

Synopsis

```
void do_disable( int Iaf )  
  
subroutine dodisable( Iaf )  
integer Iaf
```

Input parameters

`Iaf` – `Iaf` is the array descriptor obtained from `do_declare`.

Discussion

DOLIB supports automatic caching of data pages to reduce communication and enhance performance. `do_disable` disables caching of data pages in the global array associated with descriptor `Iaf`. All data pages associated with global array descriptor `Iaf` in the cache are purged.

Important note: it is the programmer's responsibility to disable caching while the global array can be asynchronously updated. There is no implicit enforcement of cache coherency since only unmodified read-only pages should be cached.

Purging of data pages in the cache can be done by calling `do_disable` or by setting the cache size to be zero with `do_setchsize`. Caching can be turned on by calling `do_enable`.

`do_wait`

`do_wait` blocks and waits till the asynchronous gather request is satisfied.

Synopsis

```
void do_wait( int reqid )  
  
subroutine dowait( reqid )  
integer reqid
```

Input parameters

`reqid` – request descriptor returned from an asynchronous gather operation such as `do_bgather` or `do_agather`.

Description

`do_wait` blocks and waits until the data requested by an asynchronous gather has been copied into the user buffer. The request id `reqid` is released and invalidated after return from `do_wait`. Calling `do_wait` with an invalidated request id is an error.

do_enable

`do_enable` enables caching of data pages associated with the specified global array.

Synopsis

```
void do_enable( int Iaf )  
  
subroutine doenable( Iaf )  
integer Iaf
```

Input parameters

`Iaf` – `Iaf` is the array descriptor obtained from `do_declare`.

Discussion

DOLIB supports caching of commonly used data pages to reduce communication and enhance performance. `do_enable` enables caching of read-only data pages in the global array associated with descriptor `Iaf`. Caching is performed on a page (determined by `pagesize` in `do_declare`) as a unit.

Important note: It is the programmer's responsibility to ensure only read-only unmodified data is cached. There is no implicit enforcement of cache coherency since only unmodified read-only pages should be cached. Although not required, a global synchronization (`do_gsync`) is strongly recommended before calling `do_enable` to ensure correctness.

Purging of data pages in the cache can be done by calling `do_disable` or by setting the cache size to be zero with `do_setchsize`.

do_gather

do_gather performs a gather operation out of the global shared array into the specified buffer.

Synopsis

```
void do_gather( int Iaf, int nsize, int *list, void *buf )
```

```
subroutine doigather( Iaf, nsize, list, buf )
```

```
integer Iaf, nsize, list(nsize)
```

```
integer buf(nsize)
```

```
subroutine dodgather( Iaf, nsize, list, buf )
```

```
integer Iaf, nsize, list(nsize)
```

```
real*8 buf(nsize)
```

Input parameters

- `Iaf` – the array descriptor obtained from `do_declare`.
- `nsize` – number of items to be collected in the gather operation.
- `list` – the index list for the gather operation.
- `buf` – the buffer area to hold the result of the gather operation.

Description

`do_gather` performs a collect operation out of the global shared array into the specified buffer. The index set is described in `list`.

This is a synchronous call. The calling process waits (blocks) until the gather request is completed. Note that it is an error to attempt access beyond the declared array size, `gsize`, set in `do_declare`.

do_bgather

do_bgather performs an asynchronous block gather operation out of the global shared array into the specified buffer.

Synopsis

```
int do_bgather( int Iaf, int nsize, int istart, void *buf )
```

```
integer function dobiggather( Iaf, nsize, istart, buf )
```

```
integer Iaf, nsize, istart
```

```
integer buf(nsize)
```

```
integer function dobdgather( Iaf, nsize, istart, buf )
```

```
integer Iaf, nsize, istart
```

```
real*8 buf(nsize)
```

Input parameters

- `Iaf` – the array descriptor obtained from `do_declare`.
- `nsize` – number of items to be collected in the gather operation.
- `istart` – the starting index for the gather operation. Indices range from `istart` to `istart + nsize - 1` are collected.
- `buf` – the buffer area to hold the result of the gather operation.

Description

`do_bgather` performs an asynchronous block gather operation out of the global shared array into the specified buffer. The index set ranges from `istart` to `istart + nsize - 1`. This is an asynchronous call with the function returning an integer request descriptor to be used with `do_isavail` and `do_wait`. The buffer `buf` should be treated as invalid until either

`do_isavail` returns `.true.` or 1, or until `do_wait` is called. The calling process returns immediately without blocking. It is an error to attempt access beyond the declared array size, `gsize`, set in `do_declare`.

do_init

do_init enables emulation of global shared memory. An implicit global synchronization is performed.

Synopsis

```
void do_init( int myid, int nproc )
```

```
subroutine doinit( myid, nproc )
```

```
integer myid, nproc
```

Input parameters

- `myid` – the unique identifier for each processor, $0 \leq \text{myid} \leq (\text{numproc} - 1)$. On Intel systems, `myid` can be obtained from the NX routine `mynode()`.
- `nproc` – `nproc` is the total number of processors. On Intel systems, `numproc` can be obtained from the NX routine `numnodes()`.

Discussion

`do_init` initializes the DOLIB library and enables the emulation of global shared memory. `do_init` must be called before any use of DOLIB routines.

An implicit global synchronization is performed.

do_isavail

do_isavail checks whether the data requested by an asynchronous gather are available.

Synopsis

```
int do_isavail( int reqid )
```

```
logical function doisavail( reqid )
```

```
integer reqid
```

Input parameters

`reqid` – request descriptor returned from an asynchronous gather operation such as `do_bgather`.

Description

`do_isavail` returns `.true.` or `1` if the data requested by an asynchronous gather are available. The request descriptor `reqid` is released and is invalid as soon as `do_isavail` returns `.true.` or `1`. Calling `do_isavail` with an invalid request descriptor is an error.

do_scatter

`do_scatter` performs a scatter (distributed copy) operation into the global shared array out of the specified buffer.

Synopsis

```
void do_scatter( int Iaf, int nsize, int *list, void *buf )
```

```
subroutine doiscatter( Iaf, nsize, list, buf )
```

```
integer Iaf, nsize, list(nsize)
```

```
integer buf(nsize)
```

```
subroutine dodscatter( Iaf, nsize, list, buf )
```

```
integer Iaf, nsize, list(nsize)
```

```
real*8 buf(nsize)
```

Input parameters

`Iaf` – the array descriptor obtained from `do_declare`.

`nsize` – number of items involved in the scatter operation.

`list` – the index list for the scatter operation.

`buf` – the buffer area that holds the values to be scattered.

Description

`do_scatter` performs a distributed copy operation into the global shared array out of the specified buffer. The index set is described in `list`. It is an error to have `nsize` \leq 0.

This is an asynchronous call. The calling process does not wait (block) until the scatter request is completed. Note that it is an error for `do_scatter` to attempt to access beyond the declared array size, `gsize`, set in `do_declare`.

`do_gsync`

`do_gsync` performs an explicit global synchronization of all the processors. When the `do_gsync` operation is completed, all previously called DOLIB operations are guaranteed to have completed.

Synopsis

```
void do_gsync( )  
  
subroutine dogsync
```

Discussion

The `do_gsync` routine synchronizes the processors and ensures that all active or pending DOLIB routines (including all `do_gather`, `do_scatter`, and `do_axpby` operations) have completed. `do_gsync` is useful for avoiding potential race conditions (such as when a `do_scatter` is immediately followed by a `do_gather`).

Application codes can freely mix DOLIB with other parallel programming primitives (such as PICL and PVM calls) when desired; however, before making calls to routines in these libraries, `do_gsync` should be called to purge processor buffers of any messages relating to DOLIB.

do_check

do_check causes the calling processor to check its message queue for any pending DOLIB requests. If any are found, they are processed before do_check returns.

Synopsis

```
void do_check( )
```

```
subroutine docheck
```

Discussion

The `do_check` routine causes the calling processor to satisfy all pending DOLIB requests in its message queue. `do_check` is a no-op in the Intel multiprocessor version of DOLIB, since the interrupt handler causes all incoming requests to be processed. However, for the polling version of DOLIB (see section 6), `do_check` is provided so that the programmer can prevent a starvation condition or increase the frequency with which DOLIB requests are handled to enhance performance.

do_setchsize

do_setchsize sets the size of the cache in terms of number of pages to be cached. do_setchsize returns the previous cache size.

Synopsis

```
int do_setchsize( int npages )  
  
integer function dosetchsize( npages )  
integer npages
```

Input parameters

`npages` – `npages` is the size of cache in terms of number of pages to be cached.

Discussion

DOLIB supports automatic caching of commonly used data pages to reduce communication message traffic and enhance performance. A common shared cache pool is used even though page size for each global array may be different. A larger cache would yield a better “hit ratio” but require more memory and entail a higher overhead for associative searching. The programmer can set the cache size appropriate for a specific application for optimal performance.

Note setting the cache size to be zero is a fast way of purging the entire cache.

4. Implementation Details

DOLIB is designed to provide support for globally shared arrays in a distributed-memory environment. In this section, we describe some of the design decisions made that may affect the performance of application codes that use DOLIB.

DOLIB views a large global array as composed of fixed size pages stored in a block wrapped fashion across all processors. This page structure simplifies caching, which is vital for good performance. The restriction to a block wrapped mapping allows DOLIB to easily find the location (processor number and machine address) of any array element. Pages can be easily `malloc`'ed or `free`'ed.

DOLIB for the Intel iPSC/860 and Paragon machines is implemented using the IPX (Inter Process eXecution) [5] system developed at Brookhaven National Laboratory.¹ This version of DOLIB (and IPX) relies heavily on a reliable interrupt mechanism provided by `hrecv` on Intel multiprocessors. If a processor makes a call to `do_gather`, DOLIB first determines where (on which other processors) the requested data reside. For example, suppose that processor A requires data residing on processors B and C. `do_gather` causes processor A to send message requests that interrupt processors B and C from regular computation. These processors package the requested data and send reply messages back to Processor A. They then exit this "interrupt" mode and resume regular computation. At no time is the thread of regular computation "aware" of the interruption. However, when the interrupt mode is entered and assumes control of the processor, there may be two detrimental effects on the regular computation. First, data that reside in the hardware cache may be discarded during interrupt processing, thus increasing memory access time when regular computation is resumed. Second, if the regular computation involves pipelined floating point operations, the pipeline will be interrupted, thus increasing the overhead in reloading the pipe when regular computation is resumed.

The `do_scatter` operation involves a similar sequence of messages as the

¹IPX is available by anonymous FTP from the site `msg.das.bnl.gov` under the directory `/pub/ipx`.

`do_gather`.

The important facility provided by Intel's `hrecv` primitive is that all such signals are caught. For example, if processor B in the example above receives another interrupt while processing the one from processor A, the new interrupt is queued and then processed before processor B returns to normal execution mode. We will discuss these operations in more detail later in this section.

We have also developed a version of DOLIB (based on a polling version of IPX) that does not require a preemptive interrupt mechanism. In section 6, we discuss the implementation of DOLIB in the absence of such reliable signal handling.

DOLIB must be initialized with a call to `do_init` before any other DOLIB calls can be made. The DOLIB routine `do_declare` defines a new global shared array. The user provides: the global array size (total number of elements); the data type (`double` or `real*8`, `int` or `integer`, `char` or `character`); the number of data items per page (page size) and the number of pages per block (block size); and a name for the array (to allow for useful error messages), though accessing the array is always done through the *array descriptor* that is returned from `do_declare`. Since memory for the new global array is immediately allocated and the values of the resulting local data addresses are shared among the processors, `do_declare` implicitly synchronizes the processors. Thus, *all* processors must participate in the `do_declare` operation.

4.1. Caching

Caching is *disabled* by default when an array is declared. The DOLIB routine `do_enable` enables caching of the indicated array. There is a single cache for all the global arrays that have had caching enabled. The DOLIB routine `do_setchsize` can be used to specify the maximum size (number of cache pages) in the cache, which currently defaults to 128. The `do_setchsize` function returns the previous cache size. Note that for a given amount of memory there is a tradeoff between the number of cache pages allowed and the size of each page in the array. Both are chosen at run time, allowing the user to optimize the use of the cache for the

given application. Moreover, since the `pagesize` of a global array is set when it is declared, the pages in the cache may be of different sizes. For the Intel iPSC/860 and Paragon machines, a page size of 8Kbytes is reasonable.

Important note: Some patterns of accessing a global shared array may be unexpectedly memory-intensive. For example, consider a `do_gather` operation in which a single element of every page in the array is requested. The page containing each element is returned to the requesting processor, causing a copy of the *entire* global array to be temporarily created. For large arrays, such an operation may overflow the available memory, causing an error.

DOLIB makes no attempt to maintain cache coherency across processors. It is the user's responsibility to ensure that any information that is cached is read-only. The user may flush the cache pages associated with any given array by calling `do_disable`. No implicit synchronization is done in `do_disable`, though the utility of selectively disabling caching of an array on some processors is unclear.

To illustrate the utility of `do_enable` / `do_disable`, consider the problem of tracking particles along characteristics of a changing flow field, where the flow field is stored in a global shared array. Throughout a given time step, the flow field is assumed constant (and hence, *read-only* data). After all particles have been tracked in the given time step, the flow field must be updated for the next time step. Since the flow field is no longer read-only data, the cache should be purged of the flow field by calling `do_disable`. After the new flow field has been computed and stored in the global array, caching can be re-enabled for the flow field with `do_enable`.

To purge the *entire* cache without the necessity of disabling caching on each global array individually, the user can simply set the cache size to 0 using `do_setchsize` (the previous value is returned by the function), and then reset it to its previous value.

4.2. Accessing a Global Shared Array

The main DOLIB routines that access the globally shared arrays are `do_gather`, `do_scatter` and `do_axpby`. The `do_axpby` routine implements the operation $y \leftarrow \alpha x + \beta y$, where α and β are constants, y is a globally shared array in DOLIB, and x is a local vector. `do_axpby` is a powerful and flexible primitive, and is commonly used in such contexts as finite element matrix assembly.

When a processor (say, processor A) calls the `do_gather` routine requesting a list of elements from a global shared array, the index list is processed to determine where the requested data reside. (If one of the contiguous block versions is called, the starting index and number of items is treated similarly). Items that are not local to processor A are obtained from other processors using the IPX `get_array` call. The information exchanged among the processors during the `do_declare` allows processor A to compute the machine address *on the remote processors* where the data reside. Using this address, the `get_array` call interrupts the remote processor, forcing it to return a message containing a copy of that data.

DOLIB provides both synchronous and asynchronous versions of the `gather` operation. The synchronous version causes the calling processor to block until the gather has been completed. The asynchronous version returns control immediately to the calling processor, providing a *request descriptor* as return value. The calling processor can then query the status of the asynchronous gather using `do_isavail` (with the request descriptor) to determine whether the gather has been completed. Calling `do_wait` will force the processor to block until the specified gather has been completed. The asynchronous call is provided to allow for overlapping of communication with computation; however, the user should be aware that there is some overhead associated with creating a request descriptor and allocating memory to contain the returned pages. The asynchronous version may be more expensive overall in cases where there is little chance of overlapping communication with computation.

Given a list of values for the local array x and a corresponding list of array indices for the global array y , `do_axpby` accumulates the updated values to the

appropriate location in the global array. Global array references that reside on the calling processor (say, processor A) are done directly by assignment to memory; the others require message passing, as follows: Processor A determines the owner processor id, block number, page number and page offset for each array reference, grouping those belonging to the same processor for efficiency. Then A sends an interrupt message to each such processor (say, B and C), followed by a message containing the necessary values and indices. The interrupt message induces processors B and C to complete the `do_axpby` operation with the values contained in the second message.

The procedure for implementing `do_scatter` is identical to that of the `do_axpby` routine. Indeed, the DOLIB `do_scatter` routine is simply a `do_axpby` call with $\alpha = 1$ and $\beta = 0$.

Warning: If a `do_scatter` operation is immediately followed by a `do_gather` operation, a race condition may ensue, and old values (before the `do_scatter` has completed) may be returned to the `do_gather` call. A `do_gsync` should be called in between the `do_scatter` and `do_gather` to prevent this type of error.

4.3. Reclaiming Memory

One major advantage of DOLIB is the ability to dynamically create and destroy global shared arrays. When an array is no longer needed, `do_destroy` will free all memory allocated to the array, including cache pages associated with the given array. To avoid inconsistent views of the array, the processors are first synchronized (to ensure that any outstanding `gather` requests are satisfied), and then the data structures for the array are destroyed. This implies that all processors must participate in the `do_destroy` operation.

5. Performance of DOLIB on Intel iPSC/860

DOLIB is currently implemented using the IPX message system developed at Brookhaven National Laboratory. IPX relies heavily on the `hrecv()` preemptive

interrupt capability on Intel multiprocessors. Access cost to emulated shared memory is due in part to the overhead in DOLIB, IPX and the underlying NX message delay. In this section, we illustrate the performance and use of DOLIB in the context of a parallel matrix-matrix multiply algorithm. The RATFOR source is included in the Appendix.

The algorithm computes the matrix product $C = A * B$ where matrix A is $N \times M$ and B is $M \times M$ on P processors. The computation proceeds by block row partition of A among processors. Block columns in matrix B are gathered to generate parts of matrix C by calling the BLAS routine `dgemm`. The volume of communication for each processor is $O(NM/P)$ for gathering matrix A , $O(M^2)$ for gathering B and $O(NM/P)$ for scattering results back to matrix C .

Tables 5.1 and 5.2 display the run time (in seconds) on an Intel iPSC/860 with 8Mbytes of memory on each node. The run times are measured by calling the NX `dclock()` routine. The items *gathA*, *gathB*, *scattC*, *dgemm* denote the maximum communication time among all processor to gather A , to gather B to scatter C and the computation time in the level 3 BLAS `dgemm`. The total time is the elapsed time from a `gsync()` at the start of the algorithm to another `gsync()` at the end.

The matrix dimension N affects the data distribution of the global matrices A and C since data pages are assigned to processors in a wrapped fashion. If N is exactly divisible by P ($\text{mod}(N, P) = 0$), then all references to matrices A and C are satisfied locally on each processor without external communication. This is reflected in the faster times for the gather of A and scatter of C for $N = 1440, 19200$ in Table 5.1 and $N = 1920, 37440$ in Table 5.2.

6. Implementation of DOLIB using Polling

The version of DOLIB for the Intel iPSC/860 and Paragon machines relies heavily on the reliable interrupt handling capabilities of Intel's multiprocessors. We have implemented a version of DOLIB that relies on explicit polling of the message queue to service DOLIB requests. However, there are several requirements imposed on

Table 5.1: 16 nodes on ipsc/860

(N,M)	gathA	gathB	scattC	dgemm	total time
(1440,1440)	1.1	15	2.1	12	36
(1441,1441)	4.2	16	4.2	12	42
(19200,160)	0.13	1.0	1.2	1.9	4.2
(19201,160)	2.7	1.4	2.0	2.0	7.9

Table 5.2: 32 nodes on ipsc/860

(N,M)	gathA	gathB	scattC	dgemm	total time
(1920,1920)	1.5	29	2.9	15	61
(1921,1921)	6.2	30	12	15	75
(37440,160)	0.14	1.2	1.2	2.0	4.4
(37441,161)	3.6	2.3	2.1	1.9	9.2

the user of this version of DOLIB.

First, every DOLIB routine starts by checking its receive queue for DOLIB requests that must be satisfied. Hence, as long as all processors are executing DOLIB routines on a regular basis, performance should be largely unaffected. However, if any of the processors fails to call DOLIB routines, it will not check its receive queue for DOLIB requests, thus starving any processors attempting to access its array elements. Note that this is not a problem under the Intel version of DOLIB, since the interrupt handler forces the processor to handle incoming requests. The `do_check` routine is provided to allow the programmer to force a processor to check its input queue for DOLIB requests. The `do_check` routine is a no-op under the Intel version of DOLIB.

Our initial port of DOLIB to PVM (using a translation of IPX) is complete; however, because the IPX `get_array` routine is unaware of data types and treats all arrays as a sequence of bytes, DOLIB assumes a consistent integer and double precision format. Hence, currently only heterogeneous networks containing machines that agree on these formats (such as Sun Sparcstations and IBM RS/6000's) are

supported. We are currently rewriting DOLIB to use the PVM3.3 message passing library directly. When this is completed, DOLIB will provide a shared-memory programming paradigm for a wide variety of platforms, including fully heterogeneous clusters of workstations.

7. Summary

We have described the design and implementation of DOLIB, a library of routines that support virtual shared memory on the Intel family of distributed memory machines. DOLIB provides access to globally shared arrays through the explicit use of `gather` and `scatter` primitives. Globally shared arrays are dynamically created and destroyed by the user application, thus allowing efficient use of the available aggregate memory on the multiprocessor. DOLIB also supports automatic caching of read-only memory, to increase the efficiency of global shared memory by reducing the amount of message passing required.

We provided two illustrations of the use of global shared memory: a matrix-matrix multiplication routine, and faster concurrent I/O using a library called DONIO. These examples show that global shared memory can be an effective means of providing the shared-memory programming paradigm on distributed memory machines.

8. Obtaining the Software

To obtain the source code for DOLIB, the reader should send email to the authors: `efdazedo@msr.epm.ornl.gov` or `rominech@ornl.gov`.

Acknowledgements

The authors would like to express appreciation to Bob Marr, Ron Peierls and Joe Pasciak for the IPX package, which simplified the development of DOLIB.

9. Appendix

In this appendix, we list FORTRAN source code to illustrate the use of the DOLIB primitives. The source code given here is for the matrix-matrix multiplication example discussed in section 5.

```
#include "stdinc.h"

#define logdev (10+myid)

#define DEFAULT_ISIZEB 8

/* ***** */
/* perform matrix-matrix multiply with explicit gather */
/* ***** */
program      pllmat
{
    intrinsic      abs, max, min;

#define eps 1.0d-5
#define isnear(x,y) (abs((x)-(y)) <= eps*max( one,max(abs(x),abs(y)) ) )
    real8          cij, gcij;

#if AIX || IBM || RS6K || RIOS
#define EXTRA_UNDERSCORE 1
#endif

#ifdef PICL_VERSION

#ifdef EXTRA_UNDERSCORE

#define OPEN0 open0_
#define BCAST0 bcast0_
#define GSUM0 gsum0_
#define GMAX0 gmax0_

#else

#define OPEN0 open0
#define BCAST0 bcast0
#define GSUM0 gsum0
#define GMAX0 gmax0

#endif

#endif

#define IROOT 0
#define GMAX0_MSGTYPE 88888
#define GSUM0_MSGTYPE (GMAX0_MSGTYPE+1)
```

```
#define DOUBLE_TYPE 5
#define DOUBLE_BYTES 8

#define INTEGER_TYPE 3
#define INTEGER_BYTES 4

#define STOP( message ) { call CLOSEO(); stop message; };

#define GDHIGH( dvalue ) { \
    call GMAXO( dvalue, 1, \
        DOUBLE_TYPE, GMAXO_MSGTYPE, IROOT ); \
    call BCASTO( dvalue, DOUBLE_BYTES, DOUBLE_TYPE, IROOT ); \
};

#define GISUM( ivalue ) { \
    call GSUMO( ivalue, 1, INTEGER_TYPE, GSUMO_MSGTYPE, IROOT ); \
    call BCASTO( ivalue, INTEGER_BYTES, INTEGER_TYPE, IROOT ); \
};

#define DCLOCK CLOCKO

#define masktrap(mask) (mask)
#else
    /* NX version */
    integer      mynode, numnodes;
    external     mynode, numnodes;

#define STOP( message ) { stop message; };

#define GDHIGH( dvalue ) { call gdhigh( dvalue, 1, dwork ); };
#define GISUM( ivalue ) { call gisum( ivalue, 1, iwork ); };

#define DCLOCK dclock
    integer      masktrap;
    external     masktrap;
#endif

    real8       dwork;

#define BDZERO( nsize, dvec) { call dcopy( nsize,dzero,0,dvec,1); }

    integer      dobdgather;
    external     dobdgather, dodgather;

    integer      oldsize, setchsize;
    external     setchsize;

    integer      oldmask;

    STRING      ctype;
    STRING      name;
```

```
STRING      fmt;
integer     pagesize, blocksize;

logical     ismine;
integer     irem, iwork;
integer     myid, nproc, host;

real8      DCLOCK;
external   DCLOCK;

real8      tstart, tend, ttotal, tmegflops;
real8      tstartA, tgatA, tstartB, tgatB, tstartC, tscatC,
real8      tstartG, tgemm;
real8      tlocal;

integer     indev, outdev;
            parameter(indev = 5, outdev = 6);

real8      one, dzero;
            parameter(one = 1.0, dzero = 0.0);
real8      alpha, beta;
character   *1 transA;
character   *1 transB;

integer     iseed;
real       random;
external   random;

/* allocate 2Meg for local buffers */
integer     maxmem;
            parameter(maxmem = 2 * 1024 * 1024 / 8)
real8      dmem(maxmem);
integer     imem(maxmem * 2);
            equivalence(dmem, imem);

integer     Iidx;
#define idx(i) imem((Iidx-1) + (i) )

integer     maxrequest;
            parameter(maxrequest = 2 * 1024);
integer     preid(maxrequest), reqid(maxrequest);
integer     lastcol, nprefreq;
real8      ddummy;

integer     i, j, k, ii, jj, kk;

integer     isize, isizeA, isizeB;
integer     istrtA, iendA, nsizeA;
integer     istrtB, iendB, nsizeB;

integer     nsize, ip, istrt, jstrtA, jendA, nreq, ncount,
```

```
                                ichunk, jstrtB, jendB, jsizeB, irow, icol;

logical          spaceok;
integer          Ifree, IA, IB, IC;
integer          IAbuf, IBbuf, ICbuf;

integer          nrowA, ncolA, nrowB, ncolB, nrowC, ncolC;

#define ALLOC( IA, isize ) { \
    IA = Ifree; Ifree = Ifree + isize; \
    ASSERT( Ifree <= maxmem, 'insufficient memory',maxmem ); \
}

#define FREE( IA, isize ) { \
    ASSERT( IA + isize == Ifree, \
           ' ** can only free last allocated array ', IA ); \
    Ifree = IA; IA = 0; \
}

#define IALLOC( Ip, isize ) { \
    ALLOC( Ip, ((isize/2) + mod(isize,2)) ); \
    Ip = (Ip-1)*2 + 1; \
}

#define IFREE( Ip, isize ) { \
    ASSERT( mod((Ip-1),2) == 0, \
           ' ** IFREE: invalid Ip ', Ip ); \
    Ip = (Ip-1)/2 + 1; \
    FREE( Ip, ((isize/2) + mod(isize,2)) ); \
}

#ifndef index2
#define index2(i,j,  nrow,ncol) ((i)+((j)-1)*(nrow))
#endif

#define A(i,j) dmem((IA-1) + index2(i,j,  nrowA,ncolA))
#define B(i,j) dmem((IB-1) + index2(i,j,  nrowB,ncolB))
#define C(i,j) dmem((IC-1) + index2(i,j,  nrowC,ncolC))

#define Abuf(i,j) dmem((IAbuf-1) + index2(i,j,  nsizeA,ncolA))
#define Bbuf(i,j) dmem((IBbuf-1) + index2(i,j,  nrowB, isizeB))
#define Cbuf(i,j) dmem((ICbuf-1) + index2(i,j,  nsizeA, isizeB))

integer          IAtmp, IBtmp, ICtmp;

#define Atmp(i) dmem((IAtmp-1)+(i))
#define Btmp(i) dmem((IBtmp-1)+(i))
#define Ctmp(i) dmem((ICtmp-1)+(i))

/* initialize */
#ifdef PICL_VERSION
```

```
call          OPENO(nproc, myid, host);
#else
              myid = mynode();
              nproc = numnodes();
#endif
call          doinit(myid, nproc);

/* read in matrix sizes, may be rectangular matrices */

              nrowA = 0;
              ncolA = 0;
              ncolB = 0;

if            (myid == 0) {
write(outdev, *) ' Enter nrowA, ncolA, ncolB ';
read(indev, *) nrowA, ncolA, ncolB;
write(outdev, *) ' nrowA,ncolA,ncolB',
nrowA, ncolA, ncolB;

};

GISUM(nrowA);
GISUM(ncolA);
GISUM(ncolB);
nrowC = nrowA;
ncolC = ncolB;
nrowB = ncolA;

isizeA = nrowA / nproc;
irem = nrowA - isizeA * nproc;
if (irem != 0) {
    isizeA = isizeA + 1;
};

istrtA = 1 + myid * isizeA;
iendA = min(nrowA, istrtA + isizeA - 1);
nsizeA = max(0, iendA - istrtA + 1);

isizeB = max(1, min(DEFAULT_ISIZEB, ncolB / nproc));
irem = ncolB - nproc * isizeB;
if (irem > 0) {
    isizeB = isizeB + 1;
};

/* allocate global storage for matrices */

/* Note: null terminated strings even in Fortran */

pagesize = 1024;
blocksize = 1;
ctype = 'double precision' // char (0);
```

```
name = 'A(nrowA,ncolA)' // char (0);
pagesize = isizeA;
call          dodeclare(IA, name, nrowA * ncolA,
                      ctype, pagesize, blocksize);

name = 'B(nrowB,ncolB)' // char (0);
pagesize = nrowB;
call          dodeclare(IB, name, nrowB * ncolB,
                      ctype, pagesize, blocksize);

name = 'C(nrowC,ncolC)' // char (0);
pagesize = isizeA;
call          dodeclare(IC, name, nrowC * ncolC,
                      ctype, pagesize, blocksize);

/* initialize matrices */

Ifree = 1;

ALLOC(IAtmp, nrowA);

iseed = 13;
doloop(j, 1, ncolA) {

    ismine = (mod(j, nproc) == myid);
    if (ismine) {
        doloop(i, 1, nrowA) {
            Atmp(i) = one / dble(i + j - 1);
        };

        istrtr = index2(1, j, nrowA, ncolA);
        call          dobdscatter(IA, nrowA, istrtr, Atmp(1));
    };
};

FREE(IAtmp, nrowA);

ALLOC(IBtmp, nrowB);

iseed = 17;
doloop(j, 1, ncolB) {

    ismine = (mod(j, nproc) == myid);
    if (ismine) {

        doloop(i, 1, nrowB) {
            Btmp(i) = i + (j - 1) * nrowB;
        };
        istrtr = index2(1, j, nrowB, ncolB);
    };
};
```

```

                                call          dobdscluster(IB, nrowB, istrtr, Btmp(1));
};                                /* end if (ismine) */
};                                /* end do j */

FREE(IBtmp, nrowB);

ALLOC(ICbuf, nsizeA * isizeB);
BDZERO(nsizeA * isizeB, Cbuf(1, 1));

ALLOC(IAbuf, nsizeA * ncolA);

/* make sure all initializations to global matrix is complete */

tgatA = 0.0;
tgatB = 0.0;
tscatC = 0.0;
tgemm = 0.0;
call          dogsync();
tstart = DCLOCK();

iendA = min(nrowA, istrtrA + isizeA - 1);
nsizeA = max(0, (iendA - istrtrA + 1));

/* perform gather */

tstartA = DCLOCK();

ichunk = 2 * nproc;
doloop4(jstrtrA, 1, ncolA, ichunk) {
    jendA = min(ncolA, jstrtrA + ichunk - 1);

    nreq = 0;
    doloop(icol, jstrtrA, jendA) {
        nreq = nreq + 1;
        istrtr = index2(istrtrA, icol, nrowA, ncolA);
        reqid(nreq) = dobdgather(IA, nsizeA,
                                istrtr, Abuf(1, icol));
    };

    doloop(j, 1, nreq) {
        call          dowait(reqid(j));
    };

};                                /* end do j */

tgatA = tgatA + (DCLOCK() - tstartA);

ALLOC(IBbuf, nrowB * isizeB);
doloop4(istrtrB, 1, ncolB, isizeB) {
    iendB = min(ncolB, istrtrB + isizeB - 1);
};
```

```
nsizB = (iendB - istrB + 1);
ASSERT((1 <= nsizB) & (nsizB <= isizB),
       ' ** invalid nsizB ', nsizB);

/* perform gather of B */

tstartB = DCLOCK();

/* one long contiguous gather */

call          dowait(dobdgather(IB, nrowB * (iendB - istrB + 1),
                              index2(1, istrB, nrowB, ncolB),
                              Bbuf(1, 1)));

tgatB = tgatB + (DCLOCK() - tstartB);

/* perform computation */

tstartG = DCLOCK();
alpha = 1.0;
beta = 0.0;
transA = 'N';
transB = 'N';

oldmask = masktrap(1);

call          dgemm(transA, transB,
                  nsizA, nsizB, nrowB,
                  alpha, Abuf(1, 1), nsizA,
                  Bbuf(1, 1), nrowB,
                  beta, Cbuf(1, 1), nsizA);
tgemm = tgemm + (DCLOCK() - tstartG);

oldmask = masktrap(0);

/* scatter back into C */

tstartC = DCLOCK();
doloop(jj, 1, nsizB) {
    icol = istrB + (jj - 1);
    istr = index2(istrA, icol, nrowC, ncolC);
    call          dobdscluster(IC, nsizA, istr,
                              Cbuf(1, jj));
};
tscatC = tscatC + (DCLOCK() - tstartC);

};          /* end do istrB */
FREE(IBbuf, nrowB * isizB);

/* clean up */
```



```
FREE(IAbuf, nsizeA * ncolA);

FREE(ICbuf, nsizeA * isizeB);

tlocal = (DCLOCK() - tstart);
call      dogsync();
tend = DCLOCK();
ttotal = tend - tstart;

if (myid == 0) {
    write(outdev, *) ' total time is ', ttotal;
    tmegflops = 1.0 d - 6 * nrowC * ncolC * ncolA;
    write(outdev, *) ' aggregate MFLOP rate is ',
        tmegflops / ttotal, ' on ', nproc, ' processors ';
};

GDHIGH(tgatA);
GDHIGH(tgatB);
GDHIGH(tscatC);
GDHIGH(tgemm);
GDHIGH(tlocal);

if (myid == 0) {
    write(outdev, 9000) myid, tgatA, tgatB, tscatC;
    9000      format('myid,tgatA,tgatB,tscatC', i4, 3(1 pg12 .2, 1 x));
    write(outdev, 9010) myid, tgemm, tlocal;
    9010      format('myid,tgemm,tlocal', i4, 2(1 pg12 .2, 1 x));
};

call      dodestroy(IA);
call      dodestroy(IB);
call      dodestroy(IC);

STOP('ALL DONE');

}
end
```

10. References

- [1] E. F. D'AZEVEDO AND C. H. ROMINE, *DONIO: Distributed object network I/O library*, Tech. Report ORNL/TM-12743, Oak Ridge National Laboratory, 1994.
- [2] K. LI, *Shared Virtual Memory on Loosely Coupled Multiprocessor*, PhD thesis, Yale University, 1986.
- [3] K. LI AND P. HUDAK, *Memory coherence in shared virtual memory systems*, ACM Transactions on Computer Systems, 7 (1989), pp. 321–359.
- [4] K. LI AND R. SCHAEFER, *Shared virtual memory for a hypercube multiprocessor*, in The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications, March 1989, Monterey California, Golden Gate Enterprises, Los Altos, California, 1989, pp. 371–378.
- [5] B. MARR, R. PEIERLS, AND J. PASCIAK, *IPX – Preemptive remote procedure execution for concurrent applications*, tech. report, Brookhaven National Laboratory, 1994.
- [6] S. SHARMA, R. PONNUSAMY, B. MOON, Y.-S. HWANG, R. DAS, AND J. SALTZ, *Run-time and compile-time support for adaptive irregular problems*. Submitted for Publication.
- [7] M. STUMM AND S. ZHOU, *Algorithms implementing distributed shared memory*, IEEE Computer, (May 1990), pp. 54–64.