Chapter 1
DOLIB: Distributed Object Library[*]

E.F. D'Azevedo[†] C.H. Romine[†]

Abstract

DOLIB (Distributed Object Library) emulates global shared memory in distributed memory environments intended for scientific applications. Access to global arrays is through explicit calls to gather and scatter. Use of DOLIB does not rely on language extension, compiler or operating system supports. Shared memory provided by DOLIB was also used by DONIO (Distributed Network I/O Library) as large disk caches that gave improvements of 15 to 30 fold on the Intel Paragon. DOLIB shared memory simplifies the parallelization of the CHAMMP Semi-Lagrangian Transport (SLT) code that has particle tracking as the kernel computation.

1 Introduction

DOLIB (Distributed Object Library) [1] is a library of routines that provide support for accessing emulated global shared memory on distributed memory 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 supports automatic caching of *read-only* data for high performance. DOLIB also supports atomic accumulation and update operations that avoid explicit locking/unlocking for use in parallel finite element matrix assembly.

Section 2 contains details in implementing DOLIB. Currently DOLIB is implemented on the IPX message system developed at Brookhaven National Laboratory. The original ipsc860 implementation of IPX relies heavily on the hrecv() preemptive interrupt handling capability of the Intel machines. We have modified a more portable version of IPX to use polling. Performance of DOLIB on the synthetic and application codes will be described.

We have used DOLIB to create DONIO (Distributed Object Network I/O Library) [2] for faster disk I/O performance on Intel machines, described in §3. DONIO uses DOLIB to store a "cached" copy of the entire disk file in the aggregate memory of the multiprocessor. All disk I/O routines are then translated into memory updates (again using the DOLIB gather and scatter operations) to exploit the faster high bandwidth network for moving data. Actual disk I/O operations are performed in large blocks using a few I/O processors to take advantage of RAID 0 striping across multiple disks for optimal disk performance. Tests on both synthetic codes and actual application codes in §4 show that DONIO improves disk I/O

[*]This research was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

[†]Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831–6367.

performance by a factor of 15 to 30 compared to native Intel NX I/O calls to their parallel file system `cfs` or `pfs`.

Virtual shared memory provided by `DOLIB` also simplified the parallelization of particle tracking kernels of the CHAMMP Semi-Lagrangian Transport (SLT) code as discussed in §5. By storing the flow field in shared memory, we can eliminate the need for an artificial constraint on time step.

2 Implementation of DOLIB

There has been much research on software emulation of virtual shared memory in distributed memory environments.

The `CHAOS` library [5] 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 `x(ia(i)) = x(ia(i)) + y(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` presents a simplified view of shared memory for use in scientific applications. `DOLIB` is implemented in C using standard message passing protocols (PVM or PICL) and does not rely on language extension, compiler or operating system support. We are also considering an implementation of `DOLIB` in lower level primitives such as Active Messages [4] and Split-C [6] for higher performance.

The performance of virtual shared memory on a distributed memory system requires an effective caching strategy. `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, updates to global arrays occur at well-defined phases in the computation so cache coherency is commonly 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.

`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 map array references to pages and processors. Pages are dynamically `malloc`'ed or `free`'ed.

`DOLIB` for the Intel ipsc860 and Paragon machines is implemented using the `IPX` (Inter Process eXecution) [3] system developed at Brookhaven National Laboratory.[1] The ipsc860 version of `DOLIB` (`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

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

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

We have also developed a portable version of `DOLIB` (based on a polling version of `IPX`) that does not require a preemptive interrupt mechanism and uses standard message systems such as PICL or PVM. Each call to `DOLIB` primitive actively polls a message queue to service requests for shared memory.

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

$$y(ix(:)) \leftarrow \alpha x(:) + \beta y(ix(:)),$$

where α and β are constants, y is a globally shared array in `DOLIB`, and $ix(:)$ is an index vector 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 without explicit locks. We are experimenting with a generalization of this atomic operation `do_axpby` to

$$z(:) \leftarrow y(ix(:)); \qquad y(ix(:)) \leftarrow \alpha x(:) + \beta y(ix(:)),$$

return a copy of the value of y just before being modified. This may be used as a "test-and-set" facility for use in implementing queues and in dynamic load balancing.

3 Distributed Object Network I/O Library (DONIO)

`DONIO` is designed to speed up the I/O for distributed-memory parallel applications where all processors open a large multi-megabyte shared file for simultaneous access. To access a shared file, each processor relocates its own private copy of the file pointer with `lseek`'s to specific places in the file and then performs input/output operations. (Simultaneous output to overlapping regions in a shared file is nondeterministic; therefore, we assume that output operations do not overlap among processors). Such file access patterns are common in finite element codes that are based on subdomain decomposition. For example, the data for material properties or boundary conditions are commonly stored in shared files. This arrangement provides flexibility in solving the same problem with varying numbers or configurations of processors without rearranging the data files.

A disadvantage of large shared files is that the overhead induced by many processors attempting to access the disk file concurrently can be quite large. Machines like the Intel i860 and Paragon attempt to support simultaneous access through a special file system (CFS for the i860, PFS for the Paragon). Even with this support, the cost for concurrent access to the same file can significantly degrade the performance of a parallel program. The performance of the current generation of Intel's CFS and PFS file systems is hampered by strict adherence to the OSF/1 standard. This in effect serializes the I/O to prevent any anomalous behavior of the file system.

`DONIO` offers a UNIX-like interface consisting of the 'C' callable primitives `do_open`, `do_read`, `do_write`, `do_lseek`, `do_lsize`, `do_flush` and `do_close`, which are similar to UNIX and NX routines.

`DONIO` uses the simple idea of caching the *entire* disk file into the memory on the multiprocessor. Each processor has a limited amount of memory, so the cached data must be distributed among all processors. `do_read` and `do_write` access the cached copy in the

Table 1
DONIO routines on $81 \times 81 \times 61$ *grid, file size is* 12,288,000 *bytes.*

processor	wopen	write	wclose	ropen	read	rclose
4	0.72	13.53	13.05	20.76	67.56	0.01
8	0.57	7.33	13.53	11.72	31.51	0.01
16	0.59	3.85	10.85	14.98	16.30	0.01
32	1.09	2.15	8.80	10.86	8.55	0.01
64	0.66	1.29	8.84	12.07	4.87	0.01

aggregate memory instead of the disk file. Actual disk operations in `DONIO` are performed only during `do_open` for read-only and read-write files, and `do_close` for read-write and write-only files.

Most parallel supercomputers support a high performance parallel disk partition where disk records are striped across multiple disk for fast access. On the Intel machines, disk requests are serviced by dedicated I/O processors. Any actual disk I/O that is performed by `DONIO` operates on large blocks of contiguous data using the available I/O processors to take full advantage of RAID 0 striping across multiple disks. Note that a 2 gigabyte file can be comfortably stored in `DONIO` with 4 megabytes each on 512 processors.

4 Performance of DONIO on Intel Paragon

In this section we present a rough comparison of disk performance by `DONIO` versus native `NX` routines. We devised an example that simulates the disk I/O common in finite element codes by performing multiple direct access `lseek`'s, `read`'s and `write`'s to a single file to generate the element-to-vertex list for a three dimensional $nnx \times nny \times nnz$ grid. Elements along the vertical direction are grouped before writing, to obtain better disk performance. Note that the element-to-vertex list file is independent of the number of processors. The same file is later read again.

We present results on a medium $81 \times 81 \times 61$ (384,000 elements), and a large $121 \times 121 \times 91$ (1,296,000 elements) problem. Timings for `NX` native routines on the largest problem were over 1,000 seconds. These times were highly variable since the machine was not dedicated to our application, and hence they are not reported.

Tables 1–3 list the runtimes obtained from `dclock()`. `wopen` (`wclose`) denotes the time for opening (closing) a file for write-only access; similarly, `ropen` and `rclose` apply to read-only access. Note that most time-consuming actual disk operations are performed in `DONIO` during `wclose` and `ropen`. Only 4 processors were used to perform real disk I/O, hence actual disk I/O time is largely insensitive to the total number of processors.

Note that `read` and `write` times in `DONIO` decrease with the addition of more processors; since as more processors are used, fewer messages *per* processor are generated. On the other hand, `NX` disk operations are handled by 6 dedicated I/O processors. For a given problem the total number of disk requests is fixed, and hence I/O times do not decrease with more processors. We see that on all test cases, *total* time for `DONIO` is over 15 times faster than using native `NX` routines.

5 Semi-Lagrangian Transport (SLT)

We have used `DOLIB` to parallelize a serial version of the CHAMMP Semi-Lagrangian Transport (SLT) code to perform simple advection of scalar fields, such as moisture.

Table 2
NX routines on $81 \times 81 \times 61$ *grid, file size is* 12,288,000 *bytes.*

processor	wopen	write	wclose	ropen	read	rclose
4	1.71	241.02	0.25	1.75	182.80	0.18
8	4.24	237.12	0.46	3.31	162.70	0.46
16	9.52	231.62	0.78	9.54	179.01	0.74
32	16.95	247.22	1.32	23.92	185.91	1.05
64	51.18	239.68	3.12	45.47	182.56	2.79

Table 3
DONIO routines on $121 \times 121 \times 91$ *grid, file size is* 41,472,000 *bytes.*

processor	wopen	write	wclose	ropen	read	rclose
8	0.95	20.69	46.13	56.91	77.19	0.01
16	0.81	10.79	41.44	47.00	36.32	0.01
32	0.61	5.65	36.97	40.80	21.16	0.01
64	0.64	3.03	41.64	42.67	11.57	0.01
128	0.65	1.80	33.05	39.22	6.90	0.02

One goal of developing SLT is to ultimately couple semi-Lagrangian advection [7] with a global spectral transform dynamical model such as the Parallel Community Climate Model (PCCM) code. SLT uses a backward in time Lagrangian one-step particle tracking to determine, given an arrival point (nodal point on a grid), the departure point in the previous time step. Let (λ_A, ϕ_A), (λ_D, ϕ_D) denote the coordinates of the arrival and departure points, and $u(\lambda, \phi, t)$, $v(\lambda, \phi, t)$ the velocities. The departure points are given by

$$\lambda_D = \lambda_A - \int_{(\lambda_D,\phi_D,t)}^{(\lambda_A,\phi_A,t+\delta t)} \frac{u(\lambda,\phi,t)}{\cos\phi} dt \ , \tag{1}$$

$$\phi_D = \phi_A - \int_{(\lambda_D,\phi_D,t)}^{(\lambda_A,\phi_A,t+\delta t)} v(\lambda,\phi,t) dt \ . \tag{2}$$

SLT uses a centered-in-time point along the trajectory for evaluating the integral quantities in (1) (mid-point quadrature rule),

$$\lambda_D = \lambda_A - \delta t \left[\frac{u(\lambda_M, \phi_M, t+\delta t/2)}{\cos\phi_M}\right] \ , \tag{3}$$

$$\phi_D = \phi_A - \delta t \left[v(\lambda_M, \phi_M, t+\delta t/2)\right] \ . \tag{4}$$

The locations of the centers of the trajectories (λ_M, ϕ_M) are found by iteration

$$\lambda_M^{k+1} = \lambda_A - \tfrac{1}{2}\delta t \left[\frac{u(\lambda_M^k, \phi_M^k, t+\delta t/2)}{\cos\phi_M^k}\right] \ , \tag{5}$$

$$\phi_M^{k+1} = \phi_A - \tfrac{1}{2}\delta t \left[v(\lambda_M^k, \phi_M^k, t+\delta t/2)\right] \ , \tag{6}$$

The velocity components at (λ_M^k, ϕ_M^k) are found by shape preserving interpolation. SLT has special coordinate transformation to avoid singularities in the vicinity of the poles, however, this extra transformation may lead to load imbalance among processors.

A host/node version of SLT was parallelized by John Drake by assigning grid subdomains to processors. The velocity/flow field was replicated in an extended region surrounding each subdomain (processor). The overall time step and buffer region were determined to guarantee no particle can escape this extended region. At the start of each time step, each processor exchanged velocity values with neighboring processors to fill this extended region. No further communication of velocity values were required until the next time step.

Although fairly straight-forward to implement, the extended region approach suffers from high memory use and high communication volume. In a high resolution simulation (T63), 96 mesh layers are estimated to be required for a simulation with time step of 30 minutes. Exchange of the velocity field would also produce a high communication volume.

We parallelized SLT by storing arrays associated with the velocities into global shared memory emulated in `DOLIB`. We identified critical do-loops and performed gathers before entering do-loop calculations and immediately scattering results back into global memory. Most of the debugging was done on a serial processor using a serial version of `DOLIB` (gather/scatter's are simply memory copies). Synchronization primitives were added to prevent race conditions in the parallel code. Further gains may be possible by rewriting the original code to issue long vector gather/scatter operations for better message passing performance.

5.1 Performance of SLT on Intel Paragon

Here we present runtimes for one step of SLT on the Intel Paragon. All times exclude time for I/O and averaged for one time step of problem T42 (64 latitudes, 128 longitudes, 18 levels). Due to load imbalance, we measured runtime for the slowest processor. On 16 processors, each time step required 16.8 seconds and 11.2 seconds on 32 processors. A `DOLIB` cache size of 64 pages (approximately 580Kbytes) was used. The runtimes were insensitive to size of time step (δt), changing by about 5% with time step twice as large ($2\delta t$). For comparison, a host/node version of SLT, which was parallelized by John Drake using an extended velocity field around each subdomain, required 19.2 seconds on 16 processors for a timestep. The `DOLIB` shared memory approach to Lagrangian particle tracking hold promise for taking longer time steps in a higher spatial resolution.

References

[1] E. F. D'Azevedo and C. H. Romine, *DOLIB: Distributed object library*, Tech. Rep. ORNL/TM-12744, Oak Ridge National Laboratory, 1994.

[2] ——, *DONIO: Distributed object network I/O library*, Tech. Rep. ORNL/TM-12743, Oak Ridge National Laboratory, 1994.

[3] B. Marr, R. Peierls, and J. Pasciak, *IPX – Preemptive remote procedure execution for concurrent applications*, tech. report, Brookhaven National Laboratory, 1994.

[4] R. Riesen, A. B. Maccabe, and S. R. Wheat, *Split-C and active messages under SUNMOS on the Intel Paragon*, submitted to Super Computing 94, (1994).

[5] 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.

[6] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, *Active messages:A mechanism for integrated communication and computation*, in Proceedings of the 19th International Symposium on Computer Architecture, Gold Coast, Australia, ACM Press, May 1992.

[7] D. L. Williamson and P. J. Rasch, *Two-dimensional semi-lagrangian transport with shape-preserving interpolation*, Monthly Weather Review, (1989), pp. 102–129.