DOLIB Shared Memory Library Simplifies Programming for PVM and Paragon

Ed D’Azevedo and Chuck Romine
Mathematical Sciences Section

9 November 1994

http://www.epmornl.gov/~romine/
or
http://www.epmornl.gov/~efdazedo/
Shared- vs. Distributed-memory

“That’s the effect of living backwards,” the Queen said kindly: “it always makes one a little giddy at first —”

“Living backwards!” Alice repeated in great astonishment. “I never heard of such a thing!”

“— but there’s one great advantage in it, that one’s memory works both ways.”

– Lewis Carroll
Through the Looking Glass
and what Alice found there
Premise

- Message-passing (distributed-memory) programming model
  - Requires careful matching between sends and receives
  - Places burden of problem decomposition on programmer
  - Makes dynamic load balancing impractical
  - Treats “naturally shared” data as distributed

- Shared-memory programming model
  - Natural expression of parallelism
  - Program decomposition determined at runtime
  - Dynamic load balance natural
  - Direct access to “naturally shared” data.
DOLIB

- Distributed Object LIBrary
  - Uses IPX from Brookhaven National Laboratory
  - Fortran and C callable library of subprograms to support shared-memory programming model for PVM and Paragon (currently limited PVM support from IPX)
  - Core routines written in (Fortran-callable) C, to take advantage of (portable) dynamic memory allocation
  - Globally shared arrays (byte, int, float, double) allocated and freed dynamically
  - Access to array elements is through gather and scatter primitives
  - Automatic caching of read-only data to enhance performance
  - No explicit locking mechanism needed
  - No compiler extensions or operating system support needed

- User is free to mix message-passing and DOLIB for best programmability/performance

- Serial version of the library available for easy debugging of DOLIB parallel code
Goal of DOLIB

- Ease of use
- More rapid parallelization of serial code
- Simpler debugging environment with serial version of DOLIB
- Competitive performance
- Use aggregate memory as a huge resource for “Grand Challenge” problems
  - Molecular Dynamics calculations on huge systems of atoms
  - Atmospheric Modeling at high resolution
  - Groundwater Modeling at high resolution
DONIO

- Distributed Object Network I/O Library
  - Designed to solve I/O bottleneck of the Intel Paragon
  - Uses DOLIB to create a disk cache copy of file in the aggregate memory of the processors
  - Fortran and C callable library of subprograms to mimic standard UNIX I/O calls (lseek, read, write, etc.)
  - Able to handle large files (DONIO on xps35 can store 2GByte file using 4MBytes per processor).
  - DONIO automatically translates read and write calls into DOLIB gather and scatter calls, respectively
  - Actual disk I/O done
    * in large contiguous blocks to take advantage of RAID 0 striping
    * only during do_open for read and read-write files
    * only during do_close or do_flush for read-write and write-only files
  - DONIO can use full bandwidth of the Paragon network of processors
  - File checkpointing provided to avoid catastrophic loss
DOLIB Routines

- `do_init` – initialize DOLIB subsystem
- `doDeclare` – declare and allocate space for a global array
- `doDestroy` – destroy global array and free space
- `doEnable` – enable caching for global array
- `doDisable` – disable caching for global array
- `doGather` – collect specified global array entries
- `doScatter` – update specified array entries
- `doGSync` – enhanced barrier (to prevent starvation)
- `doCheck` – check for gather/scatter/update requests
- `doAxpbY` – update array \( y \leftarrow \alpha x + \beta y \). Accumulate operation useful for finite element matrix assembly.
- `doAxpbYz` – update array, returning previous value of \( y \). Useful for load balancing, among other things.
DONIO Routines

- `do_nio` - initialize DONIO subsystem
- `do_open` - allocate global cache for file, reading if it exists
- `do_close` - write file if updated, then destroy cached copy
- `do_lsize` - set file size
- `do_lseek` - set local file pointer
- `do_read` - “read” from globally cached file
- `do_write` - “write” to globally cached file
- `do_flush` - write out current copy of cached file (checkpoint)

Note: `do_flush` or `do_close` is required for altered files.
Dynamic Load Balancing

- Difficult with message-passing paradigm
- Important for applications where the message traffic depends upon data, for example
  - Groundwater flow and transport modeling
  - Atmospheric modeling
- Load balancing made simple with DOLIB do\_axpbyz call
Structure of DOLIB Global Arrays

- Global arrays are decomposed into fixed size blocks (blocksize) of fixed size pages (pagesize).
- blocksize and pagesize are user-supplied at array declaration time
- Blocks are wrap-mapped to the processors
- Data movement is in pages, not individual entries
  - Provides automatic “prefetching” of data
  - Simplifies implementation of caching
- DOLIB relies on caching to reduce message-passing overhead
Caching in DOLIB

- A single cache for all global arrays (for simplicity)
- Unit of cache storage is a page
- User determines which arrays are cached, and when (with do_enable and do_disable)
- Current cache implementation is doubly linked list with linear searches
- Empirical studies of cache effects show
  - Performance of user program is sensitive to size of cache
  - Cache overhead is small, so simple implementation sufficient (for now)
Comparison of DONIO with NX

- Example problem:
  - simulated finite-element disk I/O
  - multiple direct access seeks, reads and writes
  - three grid sizes: $41 \times 41 \times 31$, $81 \times 81 \times 61$, and $121 \times 121 \times 91$

- Results are summarized below:

<table>
<thead>
<tr>
<th>Processors</th>
<th>Problem Size</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small (1.5 MBytes)</td>
<td>Medium (12.3 MBytes)</td>
<td>Large (41.5 MBytes)</td>
<td></td>
</tr>
<tr>
<td>NX DONIO</td>
<td>NX DONIO DONIO</td>
<td>DONIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>98.6 23.0</td>
<td>427.7 115.6</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>104.7 15.6</td>
<td>408.3 64.7</td>
<td>201.9</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>114.6 10.7</td>
<td>431.2 46.6</td>
<td>136.4</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>134.4 8.5</td>
<td>476.4 31.5</td>
<td>105.2</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>211.9 7.3</td>
<td>524.8 27.7</td>
<td>99.6</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>– –</td>
<td>– –</td>
<td>81.6</td>
<td></td>
</tr>
</tbody>
</table>

Even more impressive gains in GCT.
Semi-Lagrangian Transport (SLT) with DOLIB

- CHAMMP computational kernel
  - simple advection of scalar fields (e.g., moisture)
  - backward in time Lagrangian one-step particle tracking
  - transformation to avoid singularities at the earth’s poles may induce load imbalance

- Initial parallelization used domain-decomposition and explicit message-passing
  - extended each subdomain with “ghost region” and exchanged neighboring flow field information
  - high cost in memory use and communication volume, or
  - severe time-step constraint

- Using DOLIB:
  - Identified critical do-loops
  - performed gathers before entering loop
  - performed scatters upon exiting loop
• On resolution T42 (64 latitudes, 128 longitudes, 18 levels) averaged time per step (time on slowest processor and excluding I/O) is 16.8 sec (16 processors) and 11.2 sec (32 processors).

• Runtimes were insensitive to size of time step. Runtimes changed by 5% with time step twice as large.

• Host/node version with explicit message passing takes 19.2 sec on 16 processors.

• In a high resolution simulation (T63), 96 mesh layers are estimated to be required for a simulated time of 30 minutes per step.
Molecular Dynamics with DOLIB

- Large-scale MD code based on SOTON_PAR
- DOLIB version employs dynamic load balancing
- More memory efficient than previous parallel version
  - Total memory requirement is 40 bytes per atom (52 bytes if forces are accumulated in double precision)
  - We believe it is possible to model 1000 million atoms on Paragon undergoing testing in Beaverton (1000 node machine)
- Current tests show runtimes competitive with other parallel MD codes
- LJ6-12 potential, $50 \times 50 \times 50$ lattice (500,000 atoms), $T = 0.72$, $\rho = 0.8442$, $R_{\text{cut}} = 2.5$, $dt = .00462$.

<table>
<thead>
<tr>
<th>Processors</th>
<th>Time per step</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>110.0</td>
</tr>
<tr>
<td>8</td>
<td>57.5</td>
</tr>
<tr>
<td>16</td>
<td>30.8</td>
</tr>
<tr>
<td>32</td>
<td>15.0</td>
</tr>
<tr>
<td>64</td>
<td>9.3</td>
</tr>
</tbody>
</table>
Future Work

- Improve caching strategy
- Continue to explore load-balancing with DOLIB
- Enhance performance
- Enhance DONIO to work on larger files (> 2GBytes)
- Full PVM implementation
- Incorporate DOLIB into PICS GCT Groundwater model
Limitations

- **DOLIB**
  - Supports only 1-dimensional arrays. User must treat multidimensional arrays as 1-D
  - No support for more general objects
  - Caching support only for read-only data (no attempt to check for cache coherency)
  - Currently very limited PVM cluster support (i.e., nearly-homogeneous networks)

- **DONIO**
  - Current file size limitation of 2GBytes
  - UNIX compatible I/O only. No support for Fortran unformatted binary files
  - User must estimate eventual size of write-only or read-write files (with dolsize)
DOLIB Code Fragment

allocate global storage for matrices

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

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

reqid(nreq) = dobdgather(IA, nsizeA, istrt, Abuf(1, icol))

call dowait(reqid(nreq))
DONIO Code Fragment

```
#define IOINIT(myid,nproc)
    call donio(myid,nproc)
#define LSEEK dolseek
#define ROPEN( fd, filename)
    fd = doopen( filename, rflags,mode)
#define WOPEN( fd, filename)
    fd = doopen( filename, wflags,mode)
#define LSIZE( fd, newsize )
    call dolsize( fd, newsize )
#define CREAD(fd, ibuffer,nbytes)
    call doread(fd, ibuffer, nbytes )
#define CWRITE(fd, ibuffer, nbytes)
    call dowrite( fd, ibuffer, nbytes )
#define CCLOSE( fd )
    call doclose(fd)
```