Asynchronous Dynamic Load Balancing (ADLB)

A high-level, non-general-purpose*, but easy-to-use programming model and portable library for task parallelism

Rusty Lusk
Mathematics and Computer Science Division
Argonne National Laboratory

*But more than you might think…
Two General Approaches to Parallel Algorithms

- **Data Parallelism**
  - Parallelism arises from the fact that physics is largely local
  - Same operations carried out on different data representing different patches of space
  - Communication usually necessary between patches (local)
    - global (collective) communication sometimes also needed
  - Load balancing sometimes needed

- **Task Parallelism**
  - Work to be done consists of largely independent tasks, perhaps not all of the same type
  - Little or no communication between tasks
  - Dependencies among tasks must be managed (statically or dynamically)
  - Load balancing fundamental
Load Balancing

- **Definition:** the assignment (scheduling) of tasks (code + data) to processes so as to minimize the total idle times of processes

- **Static load balancing**
  - all tasks are known in advance and pre-assigned to processes
  - works well if all tasks take the same amount of time
  - requires no coordination process

- **Dynamic load balancing (old-fashioned version)**
  - A task is assigned to a worker process by a master process when the worker process becomes available by completing previous task
  - Requires communication between manager and worker processes
  - Tasks may create additional tasks
  - Tasks may be quite different from one another
Generic Manager/Worker Algorithm for Dynamic Load Balancing

- Easily implemented in MPI
- Solves some problems
  - implements dynamic load balancing
  - termination
  - dynamic task creation
  - can implement workflow structure of task dependencies
- Provides some scalability problems
  - Manager can become a communication bottleneck (granularity dependent)
  - Memory can become a bottleneck (depends on task description size)
The ADLB Idea

- A task is described by a string of bytes (defined by application)
  - Tasks have types
- No explicit master for load balancing; workers make put/get calls to ADLB library; those subroutines access local and remote data structures (remote ones via MPI).
- Simple Put/Get interface from application code to distributed work queue hides MPI calls
- Potential proactive load balancing in background
The ADLB Model (no master)

- Doesn’t really change algorithms in workers
- Not a new idea (e.g. Linda)
- But need scalable, portable, distributed implementation of the shared work queue
  - Considerable complexity hidden here
API for a Simple Programming Model

- **Basic calls**
  - `ADLB_Init(num_servers, am_server, app_comm)`
  - `ADLB_Server()`
  - `ADLB_Put(type, priority, len, buf, target_rank, answer_dest)`
  - `ADLB_Reserve(req_types, handle, len, type, prio, answer_dest)`
  - `ADLB_GetReserved(handle, buffer)`
  - `ADLB_Ireserve(...)`
  - `ADLB_Set_Done()`
  - `ADLB_Finalize()`

- **A few others, for optimizing and debugging**
  - `ADLB_{Begin,End}_Batch_Put()`
  - Getting performance statistics with `ADLB_Get_info(key)`

- **Both C and Fortran bindings**
API Notes

- Types, answer_rank, target_rank can be used to implement some common patterns
  - Sending a message
  - Decomposing a task into subtasks

- Return codes (defined constants)
  - ADLB_SUCCESS
  - ADLB_DONE
  - ADLB_DONE_BY_EXHAUSTION
  - ADLB_NO_CURRENT_WORK (for ADLB_Ireserve)

- Batch puts are for inserting work units that share a large proportion of their data
More API Notes

- If some parameters are allowed to default, this becomes a simple, high-level, work-stealing API
  - examples follow

- Use of the “fancy” parameters on Puts and Reserve-Gets allows variations that allow more elaborate patterns to be constructed

- This allows ADLB to be used as a low-level execution engine for higher-level models
  - ADLB is being used as execution engine for the Swift workflow management language
How It Works (current production implementation)

- Application Processes
- ADLB Servers

put/get
The ADLB Server Logic

- **Main loop:**
  - `MPI_Iprobe` for message in busy loop
  - `MPI_Recv` message
  - Process according to type
    - Update status vector of work stored on remote servers
    - Manage work queue and request queue
    - (may involve posting `MPI_Isends` to `isend` queue)
  - `MPI_Test` all requests in `isend` queue
  - Return to top of loop

- **The status vector replaces single master or shared memory**
  - Circulates among servers at high priority
ADLB Uses Multiple MPI Features

- ADLB_Init returns separate application communicator, so application processes can communicate with one another using MPI as well as by using ADLB features.
- Servers are in MPI_Iprobe loop for responsiveness.
- MPI_Datatypes for some complex, structured messages (status)
- Servers use nonblocking sends and receives, maintain queue of active MPI_Request objects.
- Queue is traversed and each request kicked with MPI_Test each time through loop; could use MPI_Testany. No MPI_Wait.
- Client side uses MPI_Ssend to implement ADLB_Put in order to conserve memory on servers, MPI_Send for other actions.
- Servers respond to requests with MPI_Rsend since MPI_Irecvs are known to be posted by clients before requests.
- MPI provides portability: laptop, Linux cluster, BG/Q, Cray
- MPI profiling library is used to understand application/ADLB behavior.
Typical Code Pattern

\[ \text{rc} = \text{MPI}_\text{Init}( \&\text{argc}, \&\text{argv} ); \]
\[ \text{aprintf\_flag} = 0; \quad /* \text{no output from adlb itself} */ \]
\[ \text{num\_servers} = 1; \quad /* \text{one server might be enough} */ \]
\[ \text{use\_debug\_server} = 0; \quad /* \text{default: no debug server} */ \]

\[ \text{rc} = \text{ADLB}\_\text{Init}( \text{num\_servers}, \text{use\_debug\_server}, \text{aprintf\_flag}, \text{num\_t}, \]
\[ \quad \text{type\_vec}, \&\text{am\_server}, \&\text{am\_debug\_server}, \&\text{app\_comm} ); \]

\[ \text{if ( am\_server ) } \{ \]
\[ \quad \text{ADLB}\_\text{Server}(3000000, 0.0); \quad /* \text{mem limit, no logging} */ \]
\[ \}\]
\[ \text{else } \{ \quad /* \text{application process} */ \]
\[ \quad \text{code using ADLB\_Put and ADLB\_Reserve, ADLB\_Get\_Reserved, etc.} \]
\[ \}\]
\[ \text{ADLB}\_\text{Finalize}(); \]
\[ \text{MPI}\_\text{Finalize}(); \]
Some Example Applications

- Fun – Sudoku solver
- Simple but useful Physics application – parameter sweep
- World’s simplest batch scheduler for clusters
- Serious – GFMC: complex Monte Carlo physics application
A Tutorial Example: Sudoku

(The following algorithm is not a good way to solve this, but it uses ADLB and fits on one slide.)
Parallel Sudoku Solver with ADLB

Program:

```
if (rank = 0)
    ADLB_Put initial board
    ADLB_Get board (Reserve+Get)
while success (else done)
    ooh
    find first blank square
    if failure (problem solved!)
        print solution
        ADLB_Set_Done
    else
        for each valid value
            set blank square to value
            ADLB_Put new board
            ADLB_Get board
        endif
    end if
end while
```

Work unit =
    partially completed “board”
How it Works

- After initial Put, all processes execute same loop (no master)
Optimizing Within the ADLB Framework

- Can embed smarter strategies in this algorithm
  - ooh = “optional optimization here”, to fill in more squares inside the main loop
  - Even so, potentially a lot of work units for ADLB to manage

- Can use priorities to address this problem
  - On ADLB_Put, set priority to the number of filled squares
  - This will guide depth-first search while ensuring that there is enough work to go around
    - How one would do it sequentially

- Exhaustion automatically detected by ADLB (e.g., proof that there is only one solution, or the case of an invalid input board)
Finding materials to use in luminescent solar concentrators
- Stationary, no moving parts
- Operate efficiently under diffuse light conditions (northern climates)
- Inexpensive collector, concentrate light on high-performance solar cell

In this case, the authors never learned any parallel programming approach other than ADLB
The “Batcher”: World’s Simplest Job Scheduler for Linux Clusters

- Simple (100 lines of code) but potentially useful
- Input is a file (or stream) of Unix command lines, which become the ADLB work units put into the work pool by one manager process
- ADLB worker processes execute each one with the Unix “system” call
- Easy to add priority considerations
Green’s Function Monte Carlo -- the “gold standard” for *ab initio* calculations in nuclear physics at Argonne (Steve Pieper, Physics Division)

A non-trivial manager/worker algorithm, with assorted work types and priorities; multiple processes create work dynamically; large work units

Had scaled to 2000 processors on BG/L, then hit scalability wall.

Needed to get to 10’s of thousands of processors at least, in order to carry out calculations on $^{12}$C, an explicit goal of the UNEDF SciDAC project.

The algorithm threatened to become even more complex, with more types and dependencies among work units, together with smaller work units

Wanted to maintain original manager/worker structure of physics code

This situation brought forth ADLB

Achieving scalability has been a multi-step process

– balancing processing
– balancing memory
– balancing communication

Now runs with 100’s of thousands of processes
Scalability of GFMC/ADLB

12C – GFMC+ADLB – BG/Q
Weak scaling, 2 configs/rank
An Alternate Implementation of the Same API

- Motivation for 1-sided, single-server version:
  - Eliminate multiple views of “shared” queue data structure and the effort required to keep them (almost) coherent
  - Free up more processors for application calculations by eliminating most servers.
  - Use larger client memory to store work packages
- Relied on “passive target” MPI-2 remote memory operations
- Single master proved to be a scalability bottleneck at 32,000 processors (8K nodes on BG/P) not because of processing capability but because of network congestion.
Getting ADLB

- Web site is [http://www.cs.mtsu.edu/~rbutler/adlb](http://www.cs.mtsu.edu/~rbutler/adlb)
  - documentation
  - download button

- What you get:
  - source code
  - configure script and Makefile
  - README, with API documentation
  - Examples
    - Sudoku
    - Batcher
    - Traveling Salesman Problem

- To run your application
  - Configure, make to build ADLB library
  - Compile your application with mpicc, use Makefile as example
  - Run with mpiexec

- Problems/questions/suggestions to {lusk,rbutler}@mcs.anl.gov
A Problem Arising With Large Work Units

- As work units get larger (as they do when we apply Argonne’s GFMC to more nucleons) memory gets tight.
- To store large numbers of large work units, more servers are needed.
- But then they aren’t available for application computations.
- ADLB is complicated enough without trying to integrate a solution for this problem into it.
- So we chose an orthogonal approach...
DMEM: a Simple Library for Distributed Memory Management of Large Items

- API summary: put, get, copy, free, get-part, update

- User (application or another library) refers to a memory object via a (small) `handle`, which encodes its location and size.

- DMEM manages memory on all clients. Runs as separate thread, sharing memory with application processes, so local operations are fast.

- Optimization: put and copy operations are local if possible.

- ADLB is then free to manage only DMEM handles, which are tiny, thus reducing requirement for lots of servers just for memory reasons.

- Looking ahead, object size is of type MPI_Aint, which is typically a long int in C and an integer*8 in Fortran.
The DMEM API

- The C API:
  
  int DMEM_Init(MPI_Comm user_comm, MPI_Aint init_memsize)
  int DMEM_Finalize()
  int DMEM_Put(void *pkg_addr, MPI_Aint pkg_len, DMEM_handle dh)
  int DMEM_Get(DMEM_handle dh, void *buf_addr)
  int DMEM_Copy(DMEM_handle orig, DMEM_handle *copy)
  int DMEM_Get_part(DMEM_handle dh, MPI_Aint offset, MPI_Aint len, void *buf_addr)
  int DMEM_Update(DMEM_handle dh, MPI_Aint offset, MPI_Aint len, void *buf_addr)
  int DMEM_Free(DMEM_handle dh)

- The Fortran API is similar, with an extra argument for return codes, as in MPI

- Status: implemented, GFMC converted to use it

- Available from same web site as ADLB: [http://www.cs.mtsu.edu/~rbutler/adlb](http://www.cs.mtsu.edu/~rbutler/adlb)
A Lurking Future Problem (LFP)

- (Near) future machines are going to have lots of memory per node (for huge work units) and lots of threads (hardware and software) per node (to work on them).
- What if an ADLB (or even just a DMEM) application wants to utilize work units whose size is larger than 2 GB (approximately the size of a 32-bit integer)?
- ADLB and DMEM are agnostic about the internal structure of work units, so their internal messages use MPI_BYTE as their message type, so the count argument in MPI communications is the size (in bytes) of the message.
- MPI_{Send/Recv} specifies the count argument as an integer (still 32 bytes on most systems).
- The MPI-3 forum decided not to change this, because “long” messages could be sent/received on an MPI-compliant implementation by using MPI datatypes to lower the count argument into the 32-bit range.
- But:
  - Some people (even me, a computer scientist) consider MPI datatypes inconvenient.
  - Some important MPI implementations are not MPI-compliant! (e.g. Mira’s)
- Solution: a long-message library for anyone who needs it
MPIL - MPI Long Messages

- API
  - MPIL_Init(comm)
  - MPIL_Send(*buf, MPI_Count count, datatype, rank, tag, comm)
  - MPIL_Recv(*buf, MPI_Count count, datatype, rank, tag, comm, MPIL_Status &status)
  - MPIL_Finalize(comm)
  - MPIL_Probe(...) (the tricky one)
  - MPIL_Bcast(...) (etc.)

- Implementation (in progress)
  - For MPI-standard-conforming implementations:
    - Construct datatype consisting of large number of user’s datatypes
    - MPISend/Recv using this datatype and 32-bit value of count. Use (hidden) struct datatype if division has remainder
  - For implementations where the underlying communication layer can only handle 32-bit-size messages:
    - Divide user message into multiple smaller messages (chunks).
    - Send header with first chunk, so MPIL_Recv knows how many MPI_Recvs to post.
    - Use hidden communicator to help with MPIL_Probe.
Summary

- ADLB demonstrates that by giving up generality, a programming model can provide scalability without complexity.
- GFMC motivated ADLB, which motivated DMEM, which motivated MPIL.
  - But all 3 are small, portable, generally useful libraries
- DMEM was a big help to ADLB, but is potentially useful in a more general context. (e.g. to exploit multiple types of memory in a hierarchical memory system).
- MPIL will be a simple, portable way to provide long message support to any MPI program.
The End