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
Two Classes of Parallel Programming Models

- **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
  - Traditionally needs a separate “master” task for scheduling
  - 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**
  - tasks are assigned to a worker process by master process when 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

- Easily implemented in MPI
- Solves some problems
  - implements dynamic load balancing
  - termination
  - dynamic task creation
  - can implement workflow structure of tasks
- 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

- No explicit master for load balancing; workers make 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 shared work queue
  - MPI 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_Ireserve( ... )
  - ADLB_Get_Reserved( handle, buffer )
  - 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)
API Notes

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

- Return codes (defined constants)
  - `ADLB_SUCCESS`
  - `ADLB_NO_MORE_WORK`
  - `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 (production implementation)
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
- MPI profiling library is used to understand application/ADLB behavior.
rc = MPI_Init( &argc, &argv );
aprintf_flag = 0;           /* no output from adlb itself */
num_servers = 1;            /* one server might be enough */
use_debug_server = 0;       /* default: no debug server */

rc = ADLB_Init( num_servers, use_debug_server, aprntf_flag, num_t, type_vec, &am_server, &am_debug_server, &app_comm );

if ( am_server ) {
    ADLB_Server( 3000000, 0.0 ); /* mem limit, no logging */
}
else {                        /* application process */
    code using ADLB_Put and ADLB_Reserve, ADLB_Get_Reserved, etc.
}
ADLB_Finalize();
MPI_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 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
        end while

Work unit =
    partially completed “board”
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
  - 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)
A Physics Application - Parameter Sweep in Material Science Application

- 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 - A Complex Application

- Green’s Function Monte Carlo -- the “gold standard” for *ab initio* calculations in nuclear physics at Argonne (Steve Pieper, PHY)
- 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 $^{12}\text{C}$, 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 master/slave 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 on 100,000 processes
Early Experiments with GFMC/ADLB on BG/P

- Using GFMC to compute the binding energy of 14 neutrons in an artificial well ("neutron drop" = teeny-weeny neutron star)
- A weak scaling experiment

<table>
<thead>
<tr>
<th>BG/P cores</th>
<th>ADLB Servers</th>
<th>Configs</th>
<th>Time (min.)</th>
<th>Efficiency (incl. serv.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>130</td>
<td>20</td>
<td>38.1</td>
<td>93.8%</td>
</tr>
<tr>
<td>8K</td>
<td>230</td>
<td>40</td>
<td>38.2</td>
<td>93.7%</td>
</tr>
<tr>
<td>16K</td>
<td>455</td>
<td>80</td>
<td>39.6</td>
<td>89.8%</td>
</tr>
<tr>
<td>32K</td>
<td>905</td>
<td>160</td>
<td>44.2</td>
<td>80.4%</td>
</tr>
</tbody>
</table>

- Recent work: "micro-parallelization" needed for $^{12}$C, OpenMP in GFMC.
  - a successful example of hybrid programming, with ADLB + MPI + OpenMP
Progress with GFMC

Efficiency = compute_time/wall_time – 25 Feb 2010

Number of nodes (4 OpenMP cores per node)

Efficiency in %

12C ADLB+GFMC

Feb 2009
Jun 2009
Oct 2009
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

- Relies 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
  - documentation
  - download button

- What you get:
  - source code
  - configure script and Makefile
  - README, with API documentation
  - Examples
    - Sudoku
    - Batcher
      - Batcher README
    - 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
Future Directions

- **API design**
  - Some higher-level function calls might be useful
  - User community will generate these

- **Implementations**
  - The one-sided version
    - implemented
    - single server to coordinate matching of requests to work units
    - stores work units on client processes
    - Uses MPI_Put/Get (passive target) to move work
    - Hit scalability wall for GFMC at about 8000 processes
  - The thread version
    - uses separate thread on each client; no servers
    - the original plan
    - maybe for BG/Q, where there are more threads per node
    - not re-implemented (yet)
Conclusions

- There are benefits to limiting generality of approach
- Scalability need not come at the expense of complexity
- ADLB might be handy
The End