

A Programming Framework for Incremental Data Distribution in Iterative Applications

Philip Chan

David Abramson

MESSAGE Laboratory

Caulfield School of Information Technology, Monash University
900 Dandenong Road, Caulfield East, Victoria, Australia

{pchan,davida}@csse.monash.edu.au

Abstract—Successful HPC over desktop grids and non-dedicated NOWs is challenging, since good performance is difficult to achieve due to dynamic workloads. On iterative data-parallel applications, this is addressed by dynamic data distribution. However, current approaches migrate an application from one distribution to another in one single phase, which can impact performance. In this paper, we present D³-ARC, a programming framework to support adaptive and incremental data distribution, so that data migration takes place over several successive iterations. D³-ARC consists of a runtime system and an API for specifying the distribution of arrays as well as how data redistribution takes place. We demonstrate how D³-ARC can be used to develop an incremental strategy for data distribution in a Poisson solver, utilising a runtime feedback mechanism to determine how much data to migrate during each iteration.

I. INTRODUCTION

Desktop grids [1], networks of non-dedicated workstations [2], and peer-to-peer systems are cost-effective alternatives to high-end clusters and specialised supercomputers. However, due the fluctuating workloads experienced by nodes on such systems, achieving consistent and good application performance is a challenge. One approach to improving application performance is dynamic reconfiguration [3], [4].

Reconfigurable (or adaptively parallel [5], [6] or malleable [7]) applications involve either (a) *dynamic data distribution* – altering the data assignment among processes to improve load balance [8], [2]; and/or (b) *support for dynamic process joins and departures* – adjusting the degree of parallelism in response to changes in availability of resources [3]. Writing these applications in MPI [9], [10] requires incorporating additional program logic amongst the (often) already complex code for data distribution and message-passing. Thus, high-level primitives (e.g., DParLib [8], DyRecT [6], Zoltan [11], KeLP [12], PETSc [13], SRS [7], and Dyn-MPI [2]) have been proposed to improve programming productivity.

Fig. 1 outlines data distribution in an iterative application. A data distribution specifies how arrays are partitioned and mapped among processes [14]. With the initial distribution denoted by $\mathcal{D}_0$, multiple redistributions (lines 07 & 08) in an

```

01  local initialisation and data distribution
02  for (t=0; t<n_iters; t++) {
03      computations on owned data
04      message passing to update ghost points
05      other code (convergence tests, etc.)
06      if (conditions indicate redistribution) {
07          determine new distribution
08          perform data movements
09      }
10  }
```

Fig. 1. Outline for Dynamic Distribution

application may be represented as a sequence:

$$\underbrace{\mathcal{D}_0 \xrightarrow{r_0} \mathcal{D}_1}_{\text{redistribution from } \mathcal{D}_0 \text{ to } \mathcal{D}_1} \xrightarrow{r_1} \mathcal{D}_2 \xrightarrow{r_2} \mathcal{D}_3 \cdots \mathcal{D}_{n-2} \xrightarrow{r_{n-2}} \mathcal{D}_{n-1}$$

where $\mathcal{D}_i$ is the i -th distribution and r_i represents a redistribution from $\mathcal{D}_i$ to $\mathcal{D}_{i+1}$.

With current approaches, a data redistribution is performed in a single operation, e.g., in DyRecT [6], it is triggered by the `DYR_Reconfigure()` primitive. So, when an application undergoes a redistribution r_i , the application is stalled pending the migration of all relevant data to effect the new distribution $\mathcal{D}_{i+1}$. The cost for r_i is directly proportional to the amount of data required to migrate the application's current distribution $\mathcal{D}_i$ to its new distribution $\mathcal{D}_{i+1}$.

However, on a non-dedicated cluster/NOW environment, the workloads on each individual processor may change erratically as other processes arrive and depart. It is not possible, in general, to determine a priori if application performance will improve in a new distribution. Under these conditions, optimal data distribution may not be possible and repeated data migrations will be needed. Consequently, it may be helpful to amortise the redistribution over multiple iterations.

In particular, we propose that redistribution should occur as a sequence of *incremental* changes, performed over multiple iterations rather than in one step. We present this incremental approach within a programming framework called D³-ARC. Our goal is a framework that is flexible enough to allow the implementation of various strategies that use the basic idea of incremental redistribution. We demonstrate this by presenting a simple but effective heuristic and employ this

heuristic for incremental data redistribution. We demonstrate its effectiveness with a Poisson solver using successive over-relaxation.

II. THE D³-ARC FRAMEWORK

We present a programming framework called D³-ARC, *Data-Decomposition-Distribution Awareness for Reconfigurable Computing*, specifically for iterative data parallel applications. Iterative applications use data parallelism with distributed arrays and nearest-neighbor communication at each iteration; and are used in partial differential equation (PDE) solvers, finite-element models, and heat/wave equation solvers.

D³-ARC extends the message-passing model through abstractions that express a global view of the data, aimed at reducing the programming effort required to write reconfigurable MPI programs. Our abstractions were strongly influenced by extensive body of research on global view languages (e.g., HPF [15], ZPL [16], etc.).

D³-ARC provides medium-level abstractions that automate the tedious aspects of data distribution while flexible control on operations that impact performance. With high-level abstractions, the underlying performance model is hidden from the programmer, inhibiting the use of coding techniques that enhance performance. MPI, on the other hand, exposes the (almost) raw functionality of the underlying machine, requiring all aspects of data distribution to be manually coded. This is tedious but it makes hand-coded optimisations possible.

In D³-ARC, the programmer retains explicit control over *when* operations are performed. For example, the D³-ARC runtime will not automatically perform data redistributions unless the appropriate primitive is invoked. This ensures that specific details such as the logical boundaries of arrays are not modified without the programmer's knowledge, allowing MPI primitives to be used.

A. Distributed Arrays

Distributed arrays and their distributions are declared using two primitives: `d3_darray()` – declares array structure, including agglomeration units; and `d3_dist()` – specifies its distribution. For a consistency, all processes invoke these primitives with identical arguments. The `d3_darray()` primitive declares the dimensions, element data type, and logical blocks of a distributed array, returning its descriptor, but does not allocate the array. The fifth argument specifies¹ how the array elements are grouped into logical blocks, for block-cyclic distributions and row-wise distributions of arrays. The logical blocks are the smallest units of data migration for the array.

The `d3_dist()` primitive specifies how the logical blocks identified from `d3_darray()` are to be distributed among processes. In a distribution, each logical block is assigned to exactly one process owner, but a process may own several blocks (e.g., cyclic distribution). The basic block distribution is specified as `D_BLOCK` and its cyclic equivalent `D_CYCLIC`. In a block/cyclic distribution along more than one axis, it

```
#include "darc.h"
int main(int argc, char **argv)
{
    int w[2], d[2], v[4], *bp;
    vp_t vt[4];
    declare other variables
    MPI_Init(&argc, &argv);
    d3_init(&argc, &argv);
    w[0] = w[1] = 1000;
    ❶ dA = d3_darray("A", 2, w, D3_FLOAT, w);
    d[0] = D_BLOCK; d[1] = D_GENERAL;
    compute the weights vector w
    ❷ d3_dist(dA, G_DEFAULT, d, NULL, w);
    vt[0]=vt[1]=vt[2]=vt[3]=VP_REL;
    v[0]=v[1]=v[2]=v[3]=1;
    ❸ d3_viewport(dA, vt, v);
    if (my_id == 0) {
        ❹ A = d3_map_DA(dA, NULL);
        initialise array A
    }
    ❺ d3_effect_dist(dA);
    or_id = 0;
    ❻ A = d3_map_OR(dA, or_id, NULL);
    ❼ bp = d3_get_OR_bounds(dA, or_id);
    s_i = bp[0]; e_i = bp[1];
    s_j = bp[2]; e_j = bp[3];
    for (t=0; t<n_iters; t++) {
        for (i = s_i; i <= e_i; i++)
            for (j = s_j; j <= e_j; j++)
                A[i][j] = f(A, i, j);
    }
    ❽ d3_update_OR(dA, or_id);
    redistribute
}
d3_finalize();
MPI_Finalize();
}
```

Fig. 2. Example D³-ARC Program with Nearest-Neighbor Communication

is natural to arrange the processes to form a logical grid. This is provided in D³-ARC in two ways: (a) the default grid (`G_DEFAULT`); and (b) the `d3_grid()` primitive. When applying p processes on a 2-dimensional distributed array, `G_DEFAULT` represents $\sqrt{p} \times \sqrt{p}$. The `d3_grid()` primitive can be used when a different geometry is required.

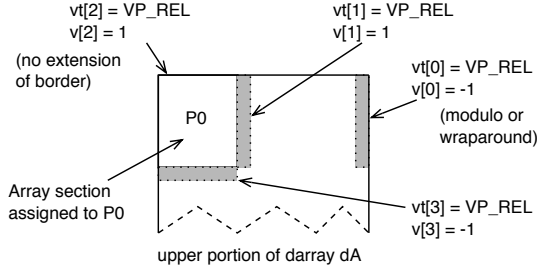
Generalised distributions are also supported by the `d3_dist()` primitive, using `D_GENERAL`. This results in a distribution that is weighted according to the measured load/performance information. The fifth argument is the weight vector that will be used for this purpose, which is explained later in Section III.

In Fig. 2, we present how D³-ARC primitives are used in an iterative program with nearest-neighbor data exchanges. All D³-ARC primitives are invoked within the pair of primitives: `d3_init()` and `d3_finalize()`. ❶ declares a logical distributed array `dA` of 1000×1000 floats. ❷ specifies how this array is partitioned and distributed among a square-shaped process grid using `BLOCK` $\times$ `GENERAL` distribution. The weights vector `w` is discussed in Section III. For the remainder of this paper, we will refer to this sample program as other D³-ARC primitives are introduced.

B. The D³-ARC Viewport Abstraction

For nearest-neighbor data exchanges, e.g., updating ghost cells/points often employed in data parallel programs, D³-ARC

¹Please see the Appendix for the details of the primitives.



The effect of `d3_viewport(dA, vt, v)`;

Fig. 3. Viewport Settings

provides the *viewport* abstraction. The `d3_dist()` primitive defines non-overlapping orthogonal regions, each owned by exactly one process. A viewport extends the logical boundary on the process' owned region, enabling a process to gain read-only access overlapped neighboring orthogonal regions.

In Fig. 2, a viewport (using `d3_viewport`) is declared at ❸ with all 4 borders set to `VP_REL` and width (array `v`) set to 1. When `vt[0] = VP_REL` and corresponding `v[0] = 1`, `D3-ARC` will define the viewport to extend on the lower end of the x-axis by 1 column. Fig. 3 shows the other options for viewports. When the viewport width is a negative integer, it means extending the border by its absolute value, but also performs wraparound extension if at the edge.

C. Mapping to Local Address Space

The logical structure and the local memory address space may be mapped either at: (a) entire array level using `d3_map_DA()`¹ – useful for initialising arrays that can fit in the memory of one process; (b) array block level as determined by the `d3_dist()` primitive. At Fig. 2 ❹, the use of the `d3_map_DA()` maps the entire array on process 0.

However, mapping the entire distributed array may be expensive if its size exceeds the available primary memory. In this case, mapping of the orthogonal regions that are owned by the process can be performed via the `d3_map_OR()` primitive. Each orthogonal region is identified by an integer (starting at 0). In the example, the `d3_map_OR()` is used at ❻ to map the first (and only) orthogonal region (`or_id = 0`) owned by the process onto its local address space.

D. Putting Distributions into Effect

The `d3_dist()` primitive only computes the array distribution. The `d3_effect_dist()` primitive has to be invoked by all participating processes to put this distribution into effect (See ❺ in Fig. 2). It has two consequences: (a) `d3_map_DA()` may only be used *before* the `d3_effect_dist()` is called; afterwards, only `d3_map_OR()` can be used to map owned regions of the array; (b) if `d3_map_DA()` was not used before `d3_effect_dist()`, no data movement takes place as the entire array is considered as uninitialised.

¹DA stands for distributed array.

```

initialise variables
for (t=0; t<n_iters; t++) {
    mark start time
    perform iteration
    timing = mark end time
    d3_update_OR(dA, or_id);
    ❸ d3_collectall(timing, t_vect, &len);
    determine if we want to redistribute
    ❹ w = d3_norm(t_vect, len, 1, 1.5);
    ❶ dist_desc = d3_new_dist(dA, w, delta, &nblks);
    est. cost = nblks * unit_cost
    if cost too high, reduce delta
    invoke d3_new_dist() again
    when cost is acceptable, apply the migration
    ❷ actual_cost = d3_migrate(dA, dist_desc);
    update unit_cost
}

```

Fig. 4. Dynamic Data Migration in `D3-ARC`

E. After Distribution

After a distribution is effected, mapping of owned orthogonal regions into the local address space is performed using `d3_map_OR()`. The `d3_get_OR_bounds()` primitive is used to obtain the upper/lower bounds of the mapped region (see ❷). `or_id` is set to 0 to indicate the first (and in this example, only) orthogonal region assigned to each process. The next two statements show which values of the returned array of `ints` indicate upper and lower bounds of the assigned orthogonal region. The `d3_update_OR()` primitive is used at line ❸. Invoking this at the end of the iteration causes the runtime to perform any updates necessary to effect the viewports, hiding all data-exchanges needed to update ghost cells.

III. DYNAMIC DATA DISTRIBUTIONS

In this paper, we focus on adjusting the distribution of data at runtime. This is but part of the features that `D3-ARC` offers for the purpose of improving application performance by adaptation.

A. Determining Need for Redistributions

A data redistribution is performed when there is a perceived loss in performance. An integral component of any redistribution system is the mechanism to detect this performance loss. `D3-ARC` supports two methods: (a) use monitored system load information; and (b) use measured timing information from previous/recent iterations. We use a similar technique employed by Dyn-MPI [2] in obtaining this information from the `ps` command. We found that the second method is as equally effective and measures performance impacts of changing system loads on the application directly.

In Fig. 4, we show how `D3-ARC` supports redistribution. The timing information is measured using a set of time/event mark functions (not shown due to lack of space). The user has to indicate which sections of the program is to be timed. In our tests, we measured the elapsed times of the computation only. The elapsed times are then distributed to all processes via the `d3_collectall()` (see Fig. 4 ❸) primitive which employs an `MPI_Allgather()` operation. A simple condition may be

used to check the values of the timings if there is a need to attempt a data distribution.

The timings are normalised at Fig. 4 (10) so that their values fall between 1.0 to 1.5, ensuring that no process is assigned 50% more data than another. The formula (1) for normalisation (for n processes) is given as follows, where $x = \langle x_0, x_1, \dots, x_{n-1} \rangle$ represents the load vector and $w = \langle w_0, w_1, \dots, w_{n-1} \rangle$ is the resulting normalised weights vector. The parameters lb and ub determine the allowable range for the weights.

$$w_i = lb + \frac{(ub - lb)(\max \{x\} - x_i)}{\max \{x\} - \min \{x\}} \quad (1)$$

B. Incremental Data Redistributions

In D³-ARC, a data redistribution is performed by the `d3_new_dist()` primitive (Fig. 4 (1)), which moves the logical boundaries determine the orthogonal regions. The weights vector and the Δ are passed as input arguments to this function to obtain a new distribution. This vector is used to proportionally compute the new logical bounds for the distributed array. `d3_new_dist()` returns a handle for the new distribution and also sets the fourth argument to the number of logical blocks that will be moved by all processes to take the array from the current distribution to the new one. Invoking `d3_new_dist()` does not perform the migration itself, so to effect this movement, the `d3_migrate()` has to be invoked.

Incremental redistribution in D³-ARC is obtained by allowing the user to specify a parameter Δ ($0 \leq \Delta \leq 1$) that controls how much of the redistribution during each iteration. This parameter controls how much of the array is to be migrated between processes, assuming that a decision to proceed with the migration has been made. When $\Delta = 0$ means no data is redistributed and $\Delta = 1$ means perform the redistribution according to the proportional function. A low value means that redistributions are made in small increments, e.g., at 0.1, we observed some iterations where only 1 or 2 rows were moved between neighbouring processes.

We demonstrate the usefulness of D³-ARC by presenting an adaptive incremental strategy for computing Δ using runtime timing data from previous migrations. This is used as a heuristic to estimate the cost of an upcoming migration. At the first migration, the initial Δ is used. Since the `d3_migrate()` function returns the total time for this migration, its value is used to compute the unit cost (in seconds) for each logical block that is migrated. In our experiments, we stored the measured migration unit costs into a sorted list. And to estimate the cost for an upcoming migration, an average of the values except the largest and smallest is used as the estimated unit cost. An outline of this strategy is given in Fig. 4. The timings enable custom and more elaborate functions to be designed (e.g., exponential weighted averaging).

IV. D³-ARC IMPLEMENTATION

D³-ARC consists of a library for the API and runtime system (Fig. 5). The library consists of several components, implemented as C modules: (a) the decomposition/distribution

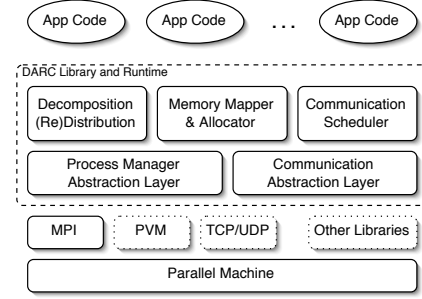


Fig. 5. D³-ARC System Components

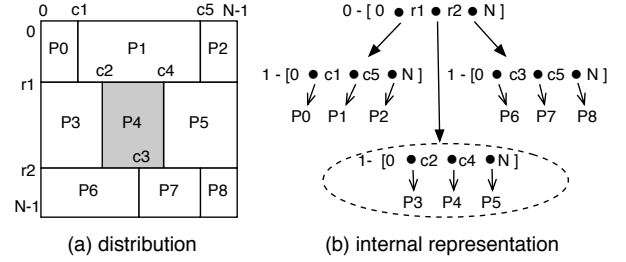


Fig. 6. Internal Representation

component – implements the `d3_darray()`, and `d3_dist()` primitives; (b) memory mapper – memory allocation and deallocation support; (c) communication scheduler – determines how communication takes place; (d) process manager abstraction layer – abstracts spawning of processes so that other systems like PVM may be supported; and (e) communication abstraction layer – wrapper functions to hide upper layers from the details of a particular message-passing library.

A. Internal Representation of Distributions

Internally, a data distribution is represented as an m -ary tree of vectors. Each vector represents the logical boundaries along one axis of a data distribution. In Fig. 6(a), we present an example generalised distribution of an $N \times N$ array onto a grid of 3×3 processes. The horizontal (row) boundaries of the array are indicated by the values $r1$ and $r2$, while the vertical (column) boundaries are indicated by $c1, c2, c3, c4, c5$. Process P4 is assigned an orthogonal region (shaded) of the array from row $r1$ until $r2-1$ and from column $c2$ until $c4-1$, inclusive. Fig. 6 (b) shows the internal data structure. During a redistribution, the orthogonal regions affected will be resized by moving the logical partitions. For example, to repartition the array among processes P3–P5, it only requires updating the interval values on the associated list (encircled in Fig. 6(b)).

The D³-ARC runtime manages one tree of intervals for each distributed array and this tree is replicated on all processes. During the `d3_new_dist()`, each process must apply the same weights vector to obtain a new distribution, resulting in a new tentative tree. When the `d3_migrate()` is invoked, the new

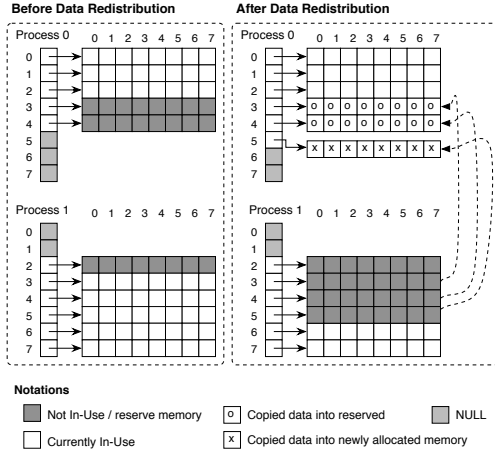


Fig. 7. Memory Allocation in D³-ARC

tree replaces the current one. All processes perform the same updates to keep the replicated data structures consistent.

B. Dynamic Array Allocations

In Fig. 7, we show how array allocation is performed by the runtime, when `d3_map_DA()` is invoked with the second argument set to `NULL`. In D³-ARC, a distributed array is allocated locally as an array of rows. Additional rows are included but left unused. These rows are pre-allocated into the structure in anticipation of future migration. In this example, process 0 has 2 additional rows allocated (see left of Fig. 7). After a redistribution, three rows from process 1 are migrated to process 0. This leads to using the two reserved rows and the allocation of a new one.

V. EXPERIMENTAL RESULTS

We used an iterative algorithm for a Poisson solver using successive over-relaxation for our experiments. Array sizes of 16000×16000 doubles are distributed in row-wise fashion, and ghost cells are updated at the end of each iteration. This application was also generalised to use an arbitrary number of arrays as real applications operate of multiple arrays. Our tests were conducted on a IBM Power5 cluster connected via Myrinet using MPICH version 1.2.7.

A. Incremental Redistribution Parameter (Δ)

To evaluate the impact of various settings for this Δ parameter, we conducted runs for each of two scenarios: (a) when only one node is loaded; and (b) when two nodes are loaded. Artificial load is created by 5 processes. In this case, two processes perform matrix multiplication; and three processes perform array sort. We designed these so that they can execute only with a given amount time limit (in seconds).

For Fig. 8 and Fig. 9, the x -axis specifies the 9 different values for Δ . For each value, we ran the application on three settings, namely: 8 (left), 16 (middle) and 32 processes (right). The results are relative to the times we obtained when running the application without data redistributions but with overloaded nodes, i.e., smaller bars mean better performance.

In Fig. 8, the best performance is achievable when $0.3 \leq \Delta \leq 0.6$. We also observe that high Δ values are dependent on how well the computed new distribution addresses the observed load imbalance. On the other hand, low Δ values cause the application to linger in an unbalanced state, affecting performance. However, the selection of an appropriate value for Δ will require understanding of the dynamic characteristics of the application. Thus, we argue that the ability to compute Δ at runtime can lead to an improvement of application performance over that when a static Δ is used.

With D³-ARC primitives, a custom strategy that computes Δ on-the-fly can be devised as in Fig. 4. In Fig. 9, we show the results when this adaptive incremental strategy is employed under one-loaded node scenario. The Δ on the x -axis represent the *initial* value used by the algorithm. For any initial value, the algorithm is able to adjust Δ at runtime and ensure that migration costs are controlled. Fig. 10 compares the static approach with the adaptive strategy when only one node is loaded on 32 processes while Fig. 11 compares this when two nodes are loaded, instead of one. The latter shows that the adaptive approach consistently outperforms the static one.

B. Timing Results Per Iteration

We present timing results under two scenarios: (a) introducing load into one node, followed by that on a second node; and (b) recurring loads on one node. In Fig. 12 shows the breakdown of the per-iteration timings for scenario (a). The corresponding timings in Fig. 14 show the relationships between the Δ value as it changes over time with the migration overhead (in seconds and in number of rows).

The values for Δ changes dynamically according to the ratio between differences in the computation times (which represents workload imbalance) and the estimated cost of the migration. If the estimated cost is higher, then Δ is reduced so that less data will be migrated. On the other hand, Δ is increased if the cost is below the differential. Observe that the data migration was performed over multiple iterations (from #52 through #70), amortising the costs over that period, yielding a reduction in execution time.

From these results, we observed that both computation and nearest-neighbor communication are affected by load changes. Note that during each iteration, a `d3_collectall()` is performed to regularly update all processes of the current weights which is used to determine the next distribution. The costs for this operation are indicated as Coord Time (in blue) on the graph.

In Fig. 14 and Fig. 15, the relationship between the changes to the Δ parameter, the amount of data migrated, and the corresponding timing costs of the migration are presented. First, the time it takes to migrate is clearly directly proportional to the amount of data migrated. Thus, the application performance can be affected by these migration costs. With an adaptive strategy, an application can use runtime timing information to control the amount of data to migrate. In D³-ARC, this is achieved by increasing or reducing Δ during each migration. In Fig. 14, it can be seen how the feedback mechanism causes Δ to change as migration costs increases.

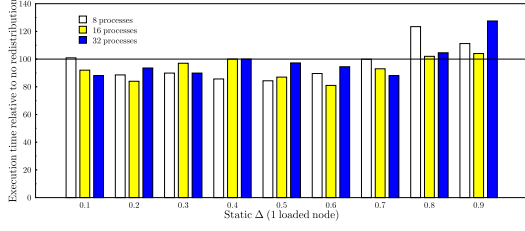


Fig. 8. Static Δ , one loaded node

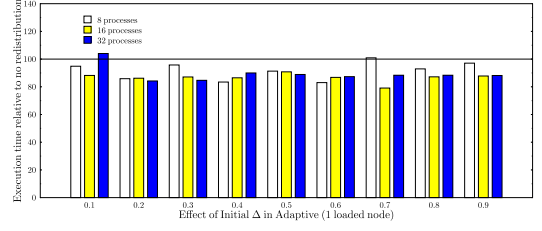


Fig. 9. Adaptive Δ , one loaded node

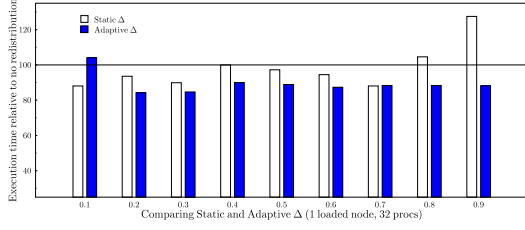


Fig. 10. Static vs. Adaptive Δ , one loaded node, 32 processes

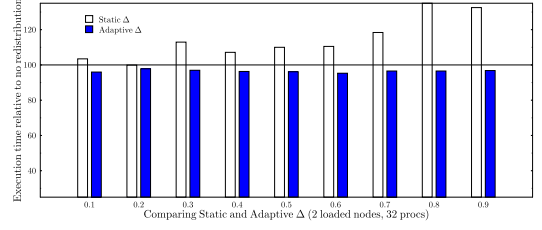


Fig. 11. Static vs. Adaptive Δ , two loaded nodes, 32 processes

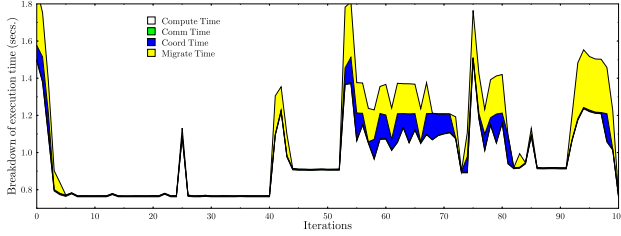


Fig. 12. Timings with two loaded nodes

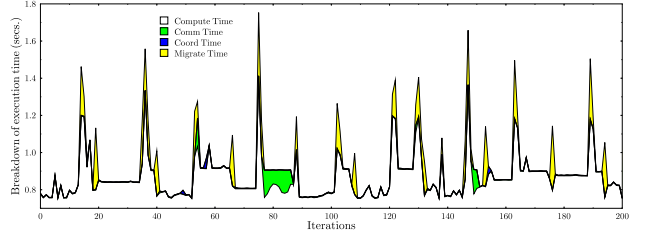


Fig. 13. Timings with recurring loads

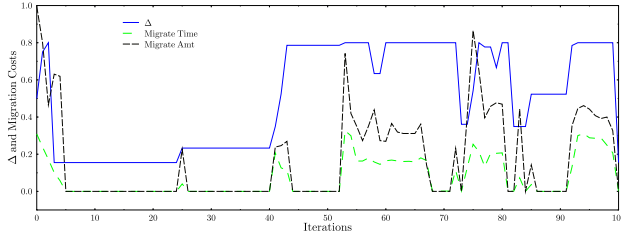


Fig. 14. Δ and migration costs, two loaded nodes

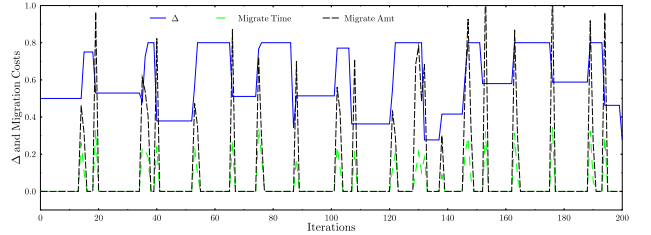


Fig. 15. Δ and migration costs, recurring loads

We summarise other tests as follows: (a) The migration cost is directly proportional to number of distributable arrays, and our adaptive algorithm compensates by reducing the Δ . (b) If the parallelism of the application is increased, the migration cost increases as well, since a redistribution can potentially involve all processes. We observe that small changes to the distribution incurs a minimal amount of overhead on the application. However, when changes are made too slowly, the additional load can increase the computation times.

C. Evaluation

We have shown how incremental data distributions can be effective for improving application performance and how

D³-ARC supports this. The adaptive algorithm for computing Δ is only a demonstration of its utility for flexible data migration. Thus, a thorough evaluation of this adaptive algorithm has yet to be conducted. The key idea here was to show that such adaptive algorithms can be custom-made for each application, made to suit its particular characteristics.

With the use of D³-ARC primitives, such adaptive algorithms can be designed. An algorithm may use the timing data returned by `d3_migrate()` at runtime along with other heuristics to compute Δ on-the-fly. This enables different ways to compute the weights vector for the `d3_new_dist()` to determine the new distribution. This vector along with Δ provides a simple but flexible control over how redistributions are applied.

VI. RELATED WORK

Dyn-MPI [2] provides reconfiguration primitives that complements MPI. Dyn-MPI provides functions to register arrays with the underlying runtime so that it can make redistributions at runtime. These redistributions are done automatically without programmer control. This approach is different from D³-ARC, where programmer control is given more preference over automation. We believe that flexible control facilitates a more predictable application performance. DParLib offers the specification of various array distributions and D³-ARC shares several common aspects including specification of global arrays and distributions. D³-ARC differs by providing weighted and incremental distributions.

Other approaches like the PCM library [4] and the Program Control Language (PCL) [17], provide primitives for modifying configuration parameters. PCL, on the other hand, provides a more comprehensive set of adaptation functions. It includes functions to create/remove tasks, handle events, and change parameters. Similar to D³-ARC, performance instrumentation is explicit in PCL. PCL is targeted for general distributed applications, so distribution of data and other reconfiguration operations has to be performed manually.

Zoltan [11], is a mature software toolkit with a comprehensive set of data management operations: data partitioning, migration, distributed directories, and unstructured communication. Zoltan's library is more comprehensive than that of D³-ARC, enabling partitioning of generalised distributed objects, while D³-ARC only supports contiguous arrays. Due to D³-ARC being limited to arrays, reconfiguration is expressed in terms of moving the logical boundaries that determine array partitioning. This also enables D³-ARC to perform array packing and unpacking for the programmer. Furthermore, D³-ARC focuses on supporting incremental distribution.

PETSc [13], [18] is another mature software library for parallel linear and nonlinear equation solvers that is intended for use in C, C++, Fortran and Python. It provides parallel matrix and vector management routines with the capability to overlap communication and computation. Unlike Zoltan and PETSc, D³-ARC is an ongoing project (currently with bindings on for C), but the novelty with D³-ARC is in its support for incremental data distribution by moving the logical partitions that determines the array distribution.

The Distributed Resource Management System (DRMS) system proposed by Moreira and Naik [3] provides extensive support for dynamic reconfiguration for Fortran applications. DRMS presents the programmer with directives (like HPF) for reconfiguration. Our approach is to incorporate a high-level view over that of MPI, while exposing the underlying performance model. Prior to DRMS, the Adaptive Multiblock PARTI (AMP) library [5] was proposed for dynamically reconfigurable PVM applications. Similarly, Dome [19] provides reconfiguration in PVM and requires data to be declared as Dome objects, limiting its applicability. The DyRecT [6] framework also provides reconfiguration and checkpointing using LAM MPI [20]. However, D³-ARC provides a more

flexible mechanism for incremental data distribution using Δ .

KeLP [21], [12] provides a flexible set of primitives for geometry-based representations and data movements of distributed data structures. In KeLP, several primitives are invoked to exchange ghost cells, e.g., `grow()`, `CopyOnIntersection()` and `execute()`, while on D³-ARC, the intersections are computed from the description of the viewports and using the `d3_update_OR()` primitive to distribute the updated values. In this case, D³-ARC primitives are simpler for regular arrays but less flexible.

VII. CONCLUSIONS

We have presented a programming framework D³-ARC, for data redistribution in iterative parallel programs. This framework consists of primitives for specifying array distribution and redistribution, with particular support for incremental data redistributions and flexible control over data migration. To facilitate efficient redistributions, we employ a tree-structured interval-based representation of the logical array partitioning. This representation allows logical boundaries of the partitioned array data to be moved by a few localised changes.

We have implemented the D³-ARC API and runtime system and presented results from a Poisson solver. For this application, we devised a simple adaptive algorithm demonstrating the use of incremental distribution within D³-ARC. Using timing information from previous migrations, our algorithm makes an estimate to determine the value of Δ , the parameter that controls how much data is migrated. The results show the benefits of incremental data migration on an application with up to 32 processes. In the future, other heuristics and adaptive strategies can be considered for use with our framework.

ACKNOWLEDGMENT

We thank the Victorian Partnership for Advanced Computing (VPAC) for the use of their facilities and for their continuing funding to support this project. Philip Chan thanks the Monash eResearch Centre for its support.

REFERENCES

- [1] D. Kondo, M. Taufer, C. L. Brooks, III, H. Casanova, and A. A. Chien, "Characterizing and Evaluating Desktop Grids: An Empirical Study," in *Proc. of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Press, 2004.
- [2] D. B. Weatherly, D. K. Lowenthal, M. Nakazawa, and F. Lowenthal, "Dyn-MPI: Supporting MPI on Medium-scale, Non-dedicated Clusters," *Journal of Parallel and Distributed Computing*, vol. 66, no. 6, pp. 822–838, 2006.
- [3] J. E. Moreira and V. K. Naik, "Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications," *IBM Journal of Research and Development*, vol. 3, no. 41, pp. 303–330, May 1997.
- [4] K. E. Maghraoui, T. Desell, B. K. Szymanski, J. D. Teresco, and C. A. Varela, "Towards a Middleware Framework for Dynamically Reconfigurable Scientific Computing," in *Grid Computing: The New Frontier of High Performance Computing*, L. Grandinetti, Ed. Amsterdam, The Netherlands, The Netherlands: Elsevier North-Holland, Inc., 2005, pp. 275–301.
- [5] G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz, "Data Parallel Programming in an Adaptive Environment," in *Proc. of the 9th Int'l Parallel Processing Symposium*. Santa Barbara, CA, United States: IEEE Press, April 1995.

- [6] E. Godard, S. Setia, and E. White, "DyRecT: Software Support for Adaptive Parallelism on NOWs," in *Proc. of the IPDPS Workshop on Run-Time Systems for Parallel Programming*, ser. Lecture Notes in Computer Science, vol. 1800. Springer-Verlag, May 2000, pp. 1168–1175.
- [7] S. S. Vadhiyar and J. J. Dongarra, "SRS – A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems," *Parallel Processing Letters*, vol. 13, no. 2, pp. 291–312, 2003.
- [8] W. George, "Dynamic Load-Balancing for Data-Parallel MPI Programs," in *Message Passing Interface Developer's and User's Conference (MPIDC'99)*, March 1999.
- [9] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," May 1994.
- [10] W. Gropp, "Learning from the Success of MPI," in *HiPC '01: Proc. of the 8th International Conference on High Performance Computing*. London, UK: Springer-Verlag, 2001, pp. 81–94.
- [11] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan Data Management Services for Parallel Dynamic Applications," *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–96, March–April 2002.
- [12] S. J. Fink, S. B. Baden, and S. R. Kohn, "Efficient Run-Time Support for Irregular Block-Structured Applications," *Journal of Parallel and Distributed Computing*, vol. 50, no. 1, pp. 61–82, 1998.
- [13] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.
- [14] J. E. Moreira, V. K. Naik, and S. P. Midkiff, "Dynamic Data Distribution and Processor Repartitioning for Irregularly Structured Computations," *Journal of Parallel and Distributed Computing*, vol. 50, no. 1, pp. 28–60, 1998.
- [15] HPF Forum, "High-Performance Fortran Language Specification (Version 2.0)," Center for Research on Parallel Computation, Rice University, Houston, TX, Report, January 1997, available as ftp://softlib.rice.edu/pub/HPF/hpf_v20.ps.gz.
- [16] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Abstractions for Dynamic Data Distribution," in *Proc. of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*. Virginia: IEEE Press, April 2004, pp. 42–51.
- [17] B. Ensink, J. Stanley, and V. Adve, "Program Control Language: A Programming Language for Adaptive Distributed Applications," *Journal of Parallel and Distributed Computing*, vol. 63, no. 6, pp. 1082–1104, 2003.
- [18] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Users Manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 2.1.5, 2004.
- [19] J. N. C. Arabe, A. B. B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, "Dome: Parallel Programming in Heterogeneous Multi-User Environment," in *Supercomputing '95*. ACM Press, 1995.
- [20] LAM-MPI Homepage, "The Local Area Multicomputer (MPI Implementation) Project," [online] URL: <http://www.lam-mpi.org> (May, 2001), 2002.
- [21] S. J. Fink, S. B. Baden, and S. R. Kohn, "Flexible Communication Mechanisms for Dynamic Structured Applications," in *Proc. of the 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems*, ser. Lecture Notes in Computer Science, A. Ferreira, J. D. P. Rolim, Y. Saad, and T. Yang, Eds., vol. 1117. Springer-Verlag, August 1996, pp. 203–215.

APPENDIX

The D³-ARC primitives in detail. `real` is `double`.

```
int d3_init(int argc, char **argv)
    For initialisation, follows MPI_Init().

int d3_darray(str a, int n, int *w, dtype_t e, int *b)
    Describes the dimensions of a distributed array with name a
    and n dimensions; w is an array of integers representing the
    dimensions of the array; and e is the type of each element. b
    specifies the sizes of the logical blocks, which must have n
    elements. It returns a unique integer descriptor for this array.
```

```
pg_t *d3_grid(int n, int *w)
    Specifies a process grid with n dimensions, and w specifies the
    extents of each dimension. On success, returns the address of a
    pg_t structure, NULL otherwise.

int d3_dist(int d, pg_t *g, dist_t *t, int *r, real *w)
    Specifies how D3-ARC performs the logical distribution of array
    d, using the process grid g and with the details of the distribution
    in t and priority in r. dist_t is an enumerated type: {D_NONE,
    D_BLOCK, D_CYCLIC, D_GENERAL}. w provides the weights to
    use for a general distribution, ignored when D_GENERAL is not
    used. The array of integers r specifies the order in which the
    array dimensions are distributed. For example, if r = [0, 1]
    means that we apply distribution by row first, then by column,
    while r = [1, 0] does it the other way around. If r = NULL,
    then it means r = [0, 1]. Returns 0 if success, -1 otherwise.

int d3_viewport(int d, vp_t *vt, int *v)
    For specifying a viewport for ghost cells updating. Enumeration
    vp_t consists of VP_REL – for relative coordinates; and VP_ABS
    – for absolute coordinates. Returns 0 if success, -1 otherwise.

void *d3_map_DA(int d, void *buff)
    Allocates (if buff == NULL) and maps the entire distributed
    array d onto the local address space of the calling process.
    For multi-dimensional arrays, the application is responsible to
    typecast the return value to the appropriate pointer.

void *d3_map_OR(int d, int id, void *buff)
    Allocates (if buff == NULL) and maps the orthogonal region to
    the local address space. id identifies which orthogonal region to
    map (starts at 0).

int *d3_get_OR_bounds(int d, int id)
    Used to retrieve the logical boundaries of the orthogonal region
    identified by id in distributed array d.

real d3_effect_dist(int d)
    Invoked to effect the distribution for array d, which includes
    determining if there is a need to copy modified orthogonal
    regions to other processes. If d3_map_DA() was invoked before
    this, it is assumed that there is a need to scatter the array
    elements. Otherwise, no data movement is performed. Returns
    the elapsed time of this distribution.

int d3_update_OR(int d, int id)
    This is used to inform the runtime that the orthogonal region
    id of array d was updated. And it also causes ghost cells (as
    defined in the d3_viewport() function) to be exchanged between
    processes.

int d3_collectall(real t, real *tv, int len)
    Performs an all gather operation to construct the timing
    information vector tv.

real *d3_norm(real *tv, real min, real max)
    Applies equation 1 on tv and returns the normalised values in
    an array of doubles. This function overwrites the values in tv
    with results, and returns tv. It requires that max > min > 0.

int d3_new_dist(int d, real *w, real delta, int *nb)
    Computes a new distribution using the weight vector w and  $\Delta$ 
    in delta. Upon return, nb is set to the number of blocks that
    will be moved to achieve the new distribution from the current
    one. This function returns a unique handle to represent this
    distribution, for subsequently use in d3_migrate().

real d3_migrate(int d, int dist_id)
    Applies the distribution as represented by the handle dist_id
    to the distributed array d. This involves actual data movements
    and updating the internal data structure for the new distribution.
    Returns the elapsed time of this migration.

int d3_finalize()
    Cleans up the D3-ARC runtime. Must be invoked before
    MPI_Finalize().
```