# DyRecT: Software Support for Adaptive Parallelism on NOWs

Etienne Godard Sanjeev Setia Elizabeth White

Department of Computer Science, George Mason University

**Abstract.** In this paper, we describe DyRecT (Dynamic Reconfiguration Toolkit) a software library that allows programmers to develop adaptively parallel message-passing MPI programs for clusters of workstations. DyRecT provides a high-level API that can be used for writing adaptive parallel HPF-like programs while hiding most of the details of the dynamic reconfiguration from the programmer. In addition, DyRecT provides support for making a wider variety of applications adaptive by exposing to the programmer a low-level library that implements many of the typical tasks performed during reconfiguration. We present experimental results for the overhead of dynamic reconfiguration of several benchmark applications using DyRecT.

## 1 Introduction

Parallel applications executing on clusters of workstations have to be able to "withdraw" from a workstation if its owner returns. This is because workstation owners are typically unwilling to share their workstation with parallel applications while they are using it for doing interactive tasks. Thus, it is necessary to ensure that parallel applications execute only on idle workstations.

To address this issue, several run-time libraries and environments provide mechanisms for process migration [1]. When owner activity is detected on a workstation being used by a parallel application, the process executing on that workstation is migrated to an idle workstation. If no idle workstation is available, the parallel application is either suspended until more resources are available or multiple processes that compose the parallel application are scheduled on the same processor.

Several studies [5, 7] have shown that a more desirable approach from the performance viewpoint would be to dynamically reconfigure the parallel application so that its parallelism matched the number of processors available for execution. Such dynamically reconfigurable applications have been referred to as adaptive parallel or malleable parallel applications. Unlike conventional parallel applications, adaptive parallel applications can adapt to changes in the availability of underlying resources by dynamically shrinking or expanding their degree of parallelism.

While the performance benefits of supporting adaptively parallel applications seem clear, most parallel programming environments do not provide mechanisms for dynamically changing the degree of parallelism of executing applications. In

this paper, we describe DyRecT (Dynamic Reconfiguration Toolkit), a software library that allows programmers to develop adaptively parallel message-passing MPI programs for clusters of workstations.

Ideally, writing adaptive parallel applications should be no more difficult than developing conventional parallel applications. To this end, several run-time systems [1, 6] have been designed that support adaptive parallel applications in a user-transparent fashion. Some of these systems, however, require all applications to be written using a master-slave programming paradigm. This can lead to poor performance for several classes of applications [3]. Other systems support adaptive parallelism for specific classes of applications, e.g., Adaptive Multiblock Parti [2] supports adaptive parallel structured and block-structured parallel applications.

Recently two systems have been developed that have a wider applicability than the systems discussed above. DRMS [3] supports adaptive parallelism for grid-based message-passing programs on the IBM SP2, while in [4], Scherer et al describe a system for adaptively parallel shared memory programs that use the OpenMP programming model. The wider applicability of these systems arises from the fact that they support the OpenMP and HPF programming models that are used for several classes of applications.

DyRecT resembles DRMS in that one of its goals is to support grid-based message-passing programs. To this end we provide a high level API that can be used by the programmer for writing adaptive parallel HPF-like programs. It differs from DRMS in two important ways. First, we provide support for making a wider variety of applications adaptive by exposing to the programmer a low-level library that implements many of the typical tasks performed during reconfiguration. Second, we support adaptive parallelism on NOWs consisting of potentially heterogeneous workstations by providing support for saving and restoring the stack of an executing process in an architecture-independent fashion.

Our approach is motivated by the observation that while the details of the actions that need to be taken during reconfiguration depend upon the application, there are common tasks that typically need to performed, e.g., spawning processes, synchronizing the application, capturing and restoring the stack, exchanging data, etc. For example, to move from the first configuration in Figure 1 to the second, the four starting processes must synchronize at some point in the computation where a consistent grid exists across the processes. At that point, data must be moved so that it is distributed across three of the processes. The process leaving the computation must be terminated. Finally, any required changes to the communication bindings must be made. At this point, the grid computation can continue.

In the case of regular grid-based iterative applications, most of these reconfiguration related tasks are performed by our high-level library and are hidden from the programmer. However, the high-level API provided with DyRecT is only suitable for certain classes of grid-based applications. Using the low-level library, discussed in Section 3, a programmer can develop reconfiguration code

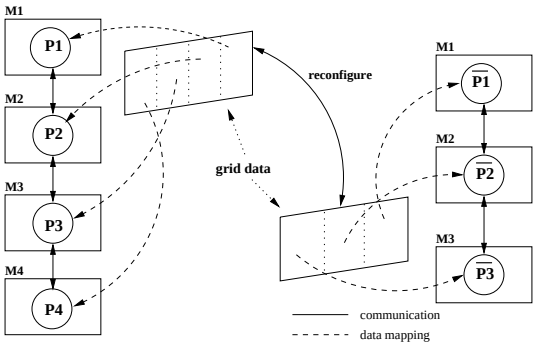


**Fig. 1.** Changing the level of parallelism by moving between configurations in a grid-based parallel application.

for other classes of applications with considerably less effort than if they had to develop the code from scratch.

## 2 High-Level Primitives

There are several different types of high-level primitives provided by the toolkit: initialization and finalization, synchronization, data distribution registration, runtime data support and reconfiguration data distribution. These primitives hide many of the details that user would typically have to deal with when making iterative grid-based applications adaptive; describing how the partitioning is related to the number of processes, moving data between processes at reconfiguration time, performing some data initialization, spawning and terminating processes and synchronizing to ensure that a consistent grid is repartitioned.

As an example, consider a typical iterative grid-based application as shown in Figure 1. For each process, every iteration consists of doing a local computation, exchanging information with neighboring processes, and synchronizing to decide convergence. When the global grid is uniformly distributed across the participating processes, this application can be made adaptive by instrumenting the source code with our high-level primitives. These calls provide to the runtime system basic information about how the grids are partitioned across any number of processes. The code for this, described below, is shown in Figure 2.

The data partitioning high-level primitives allow users to define uniform partition schemes over multi-dimensional data. In the example, the user specifies a *block* partition along the first (row) dimension (`DYR_Block()`) combined with a *collapsed* (non-partitioned) partition (`DYR_Collapsed()`) for the second (column) dimension. Two grids, one for the current iteration and one for the previous, that are partitioned using this scheme, are registered with the library using the `DYR_Register_data()` calls.

After providing information about the data to be repartitioned, the user decides where in the component source code it is legal for repartitioning to

```
int main (int argc, char *argv[]) {
  int local_dims[2],dims[2];
  double **mydata, **mydata_next;
  DYR_Disttype dist_types[2];
  DYR_Disthandle strips;
  MPI_Comm Compute_context;
  MPI_Init(&argc, &argv);
  DYR_Init(&Compute_context);                    /* initialize DYRECT */
  DYR_Save((void *) &iter, 1, MPI_INT);
    /* save variable(s) needed across all nodes */
  DYR_Block(&dist_types[0]);                     /* globally distributed data */
  DYR_Collapsed(&dist_types[1]);
  DYR_Borders_uniform (1, 1, &dist_borders[0]);
  dist_borders[1] = dist_borders[0];             /* define borders */
  DYR_Create_distribution (2, dist_types, dist_borders, &strips);
  DYR_Register_data (&mydata,2, dims,MPI_DOUBLE,0,0,strips);
  DYR_Register_data (&mydata_next,2,dims,MPI_DOUBLE,0,0,strips);
  if (DYR_Init_node()) {
    DYR_Local_shape (&mydata, local_dimens);     /* new local size */
        /* Put standard initialization calls, etc. from original program */
    iter = 0; init_data (local_dimens, mydata_next);     /* initialize data area */
  }

  do { /* iterate using Jacobi relaxation until block has converged */
        /* check for reconfiguration */
    if (DYR_Check_reconf(0)) {
      DYR_Reconfigure(1, &Compute_context); /* reconfigure */
      DYR_Local_shape (&mydata, local_dimens);
    } /* if */
    DYR_Update_borders ( &mydata_next, 0, 0);
    copy_data (local_dimens, mydata_next, mydata);
    calc_area (local_dimens, mydata, mydata_next);
    iter++;
  } while (cont_iter (local_dimens, mydata, mydata_next);
  MPI_Finalize(); DYR_Final();
}

cont_iter(...) {
  /* compute local norm */
  DYR_Sync_MS(...,result,comp_norm,set_flag);
  return result;
}
```

**Fig. 2.** Abbreviated source code for the Jacobi Application. Code added for dynamic reconfiguration is shown in boldface.

occur. The start of each iteration is used for the Jacobi application. At that point, the user adds an invocation to `DYR_Reconfigure()` guarded by a call to `DYR_Check_reconfig()`. The `DYR_Reconfigure()` function uses the data registration information to take care of all of the repartitioning calculations, data exchange, and process creation and termination required for the new set of processes.

The toolkit provides two different synchronization mechanisms, both of which assume that the application is iteration based and that reconfiguration must occur when all processes are at the same iteration. Both synchronization functions are responsible for setting a flag that is used by the `DYR_Check_reconfig()` function. The synchronization mechanism used in the example extends the existing global synchronization at the end of an iteration. In the master process, this function takes over the details of receiving the data and computing convergence using a user-provided function. It determines if a reconfiguration is needed and and informs the other processes about both convergence and reconfiguration in the return message.

If a parallel application has a variable that needs to hold the same value across all participating processes, (such as `iter` in Figure 2), it is registered with the toolkit using `DYR_Save()`. If a process joins the application at reconfiguration time, the toolkit ensures that it is initialized appropriately. It is sometimes necessary to transform the control flow of the components depending upon whether or not the process was one of the initial processes. Function `DYR_Init_node()` only returns true for processes that were part of the application at start time. In Figure 2, this function is used so that the initial processes can initialize their local data and variables. When new processes enter later in the application execution, they skip this code and immediately enter the loop, perform their reconfiguration and get information using `DYR_Local_shape()` about their data set. Then they execute normally. This primitive can also be used to guard code that only new processes should execute.

## 3 Low-Level Primitives

In addition to primitives tailored toward one class of parallel applications, our toolkit also provides to the user a set of low-level primitives. The primary reason for providing these primitives is to allow programmers to more easily handle situations where the standard high-level functionality is not sufficient. There are several different types of primitives we provide for specialized partitioning, physical resource control, tailoring of work done at reconfiguration points, and dealing with data on the runtime stack. We have found these types of low-level primitives useful for several different types of applications.

As an example, consider the case where there is variation in the relative processor speeds in the workstation cluster. In this situation, it makes sense to give processes on faster processors larger local grids than processes on slower processors. While high-level functions may provide solutions for some aspects of the problem (synchronization for example), support for non-uniform partitioning

schemes, e.g., recursive bisection, are not supported by the high-level primitives. However, using the low-level primitives provided by DyRecT, the user can tailor the actions taken during reconfiguration such that non-uniform partitioning schemes can be handled.

The default assumption is that the given reconfiguration points are placed in the main program. While this is not atypical of this class of applications, for some members of this class, more efficient reconfiguration can be achieved by placing reconfiguration points in other locations in the source code where they are encountered more frequently. For example, a multigrid V-cycle can be implemented recursively and one logical place for reconfiguration is inside the recursive function. However, this placement of reconfiguration points raises the question of how to create the correct runtime stack for new processes and how to update data (typically variables tied to grid size and pointers to intermediate grids) that may be on the stack in existing processes. Our low-level primitives include functions to deal with these problems and some rudimentary source-to-source transformation tools that deal with some of the difficult issues of the placement of these functions.

## 4 Performance Results

In this section, we describe the results of experiments in which we measured the cost of dynamically reconfiguring several parallel applications. The main goals of these experiments were to demonstrate the feasabilty of using DyRecT for supporting adaptive parallelism on NOWs and to identify the various components that contribute to the overhead of dynamic reconfiguration.

Our experimental environment consists of 16 PCs connected by a switched 100 Mbps Ethernet. Each machine has one or two 200 MHz Intel Pentium Pro processors and between 128 and 256 MB RAM. The computers run Linux 2.2.10. Our reconfiguration software was built on top of the LAM (version 6.2b) implementation of MPI.

We measured the cost of reconfiguration for five benchmark applications. The first two applications (referred to as Jacobi and RB) use the Jacobi relaxation method to solve Poisson's equation on a square grid. In Jacobi, a strip partitioning scheme is used to distribute the grid among the processors, while RB uses recursive bisection to partition the grid. The third benchmark (BC) employs a block cyclic data decomposition technique to allocate grid data to processors. The next two applications, Multigrid and Integer Sort are taken from the NAS parallel benchmarks.

We reconfigured each application several times and measured the adaptation time under different scenarios. These scenarios are representative of fluctuations in resource availability that can occur in non-dedicated clusters of workstations such as new nodes joining the computation, nodes leaving the computation, and migration of a process from one node to another.

In our experiments, an executing parallel application reconfigures itself when it receives a signal sent via the LAM "doom" command. The delay before the

application resumes execution after reconfiguration consists of two components. The main component is the actual cost of reconfiguration itself (as discussed below). In addition, before the reconfiguration can be initiated each process in the computation needs to reach the next "safe" point in its execution. This synchronization delay is application-specific since it depends on the location and frequency of occurrence of reconfiguration points. For example, in the case of the Jacobi, RB, and BC benchmarks, the reconfiguration point occurs at the end of each iteration whereas in the case of multigrid, reconfiguration points occur at each level of the multigrid V-cycle. For our benchmark applications, the synchronization delay varied from 0.07 to 3.77 seconds depending on the number of processors and the data set size of the application.

The reconfiguration cost can be broken down into several components corresponding to the different steps involved in the dynamic reconfiguration of parallel applications. These steps are: (i) spawning any new processes (ii) re-establishing the logical configuration of the application (iii) figuring out the new logical data partitioning, e.g. by invoking the recursive bisection algorithm (iv) allocating memory for any newly assigned data (v) figuring out the overlap of the current data assignment with the future data assignments, and (vi) exchanging data between nodes to account for the new configuration. Figure 3 shows the costs for each benchmark for two reconfiguration scenarios: changing the parallelism from 8 to 16 nodes, and vice versa. The time for steps (i) through (vi) is labeled spawn, init, part, alloc, overlap, and redist respectively.

Our experiments showed that the main component of the total reconfiguration time was the data redistribution time, which is proportional to the amount of data that needs to be redistributed between the processors. The reconfiguration time for our benchmarks ranged from hundreds of milliseconds to around 15 seconds depending mainly on the data set size of the application. For a more thorough discussion of our performance results, the reader is referred to [8].

## 5 Conclusion

Efficient and non–intrusive use of NOWs for parallel applications requires easy to use mechanisms for providing adaptive behavior. This paper describes research into providing both high– and low–level functionality for achieving this. The high–level primitives, tailored to iterative grid–based applications, provide simple to use mechanisms for many of the common tasks in this domain. When this functionality does not capture some required feature of the application, the user can use the provided low–level functions to provide additional flexibility.

This work is ongoing in that we are still refining both the API and the functionality provided by the API. One natural next step is to look at how high–level APIs for other classes of applications can be constructed on top of our low–level primitives. Research into efficient algorithms for data exchange within this framework is also of interest.

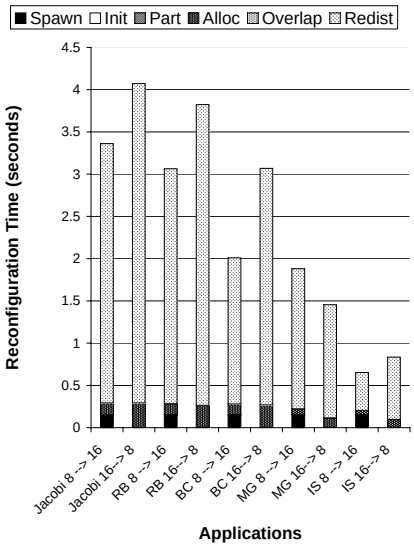


**Fig. 3.** The components of the reconfiguration overhead for five benchmark applications. The data set sizes for the benchmarks are as follows: Jacobi, RB, and BC – 144 MB, MG – 55 MB, IS – 24 MB.

## References


1. J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of Workstation Cluster s. In *Journal of Future Generation Computer Systems*, Vol. 12, 1996.
2. G. Edjlali et al. Data Parallel Programming in an Adaptive Environment. Technical Report CS-TR-3350, University of Maryland, 1994.
3. J. Moreira, V. Naik and M. Konuru. Designing Reconfigurable Data-Parallel Applications for Scalable Parallel Computing Enviromnents. Technical Report RC 20455, IBM Research Division, May 1996.
4. A. Scherer, H. Lui, T. Gross, W. Zwaenepoel. Transparent Adaptive Parallelism on NOWs using OpenMP. In Proc. of PPoPP'99, May 1999.
5. A. Acharya, G. Edjlali, J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proc. of ACM Sigmetrics '97*, 1997.
6. N. Carriero, E. Freeman, D. Gelernter. Adaptive Parallelism and Piranha. *IEEE Computer*, pp. 40-49, Jan 1995.
7. A. Chowdhury, L. Nicklas, S. Setia, E. White Supporting Dynamic Space-sharing on Non-dedicated clusters of Workstati ons. In *Proc. of ICDCS '97*, 1997.
8. E. Godard, S. Setia, E. White. DyRecT: Software Support for Adaptive Parallelism on NOWs. Technical Report GMU-TR00-01, Department of Computer Science, George Mason University, January 2000.


This article was processed using the LaTeX macro package with LLNCS style