Data Flow Pattern Analysis of Scientific Applications

February 28, 2005
Michael Frumkin
Intel
Michael.A.Frumkin@intel.com

Abstract

Many parallel and distributed applications have well defined structure which can be described by few simple relations. Some structures are so common that they are abstracted to parallel and grid benchmarks, other structures, are data and application dependent, become evident only at run-time. Many of these patterns reflect flow of data in the applications. Data Flow Graphs (DFG) provide an abstraction that allows to express data flow in the applications and to decompose the application into tasks. Applications represented by task graphs can be translated into MPI or Java RMI parallel programs having a consistent set of Send/Receive calls or set of remote method invocations. In this paper we describe some data flow patterns observed in the Computational Fluid Dynamic (CFD) applications and used in the NAS Parallel Benchmarks and in the NAS Grid Benchmarks. Then we describe new approach to the data flow analysis of the traces of parallel programs via the Data Cube Operator. The approach uses On-Line Analytic Processing (OLAP) to create various views of application trace files. The detected patterns can help in application performance tuning and improving compiler support for the commonly arising patterns. Finally, we show examples of using DFGs in parallel and distributed scientific programming.

1 Introduction

Scientific applications solving fluid dynamics, molecular dynamics or other computational physics problems abstract space, time, local and long-range interactions in programming constructs. In spite that the implementations may significantly vary, there are fundamental commonalities in these abstractions: space is represented by structured or unstructured grid or set of grids, time is modeled by outmost iteration loop, interactions are modeled by system of linearized equations, which are solved by iterative explicit or implicit methods. Not all applications fit this scheme but dozens important NASA applications use it [3, 10, 13, 16].

As a result, a number of similar constructs, design patterns, and data flows are used in various scientific applications. Some of these constructs are explicit, other are implicit, third are data dependent. Analysis of scientific applications which reveals these constructs usually results in a good understanding applications, providing high-level programming constructs for other applications and help to map the applications on computer architecture. The most commonly used programming constructs and typical applications are used as benchmarks (Livermore Loops, NAS Parallel Benchmarks, HPC Challenge Benchmarks). Some data dependent patterns which can be observed only during run-time are used for dynamic load balancing and in JIT compilers.

In this paper we present a data flow and design pattern analysis of some NASA applications. Formalization and refinement of these patterns resulted in the NAS Parallel and the NAS Grid Benchmarks. Then we present an OLAP based method for detection patterns in trace files and illustrate it on a trace of High Performance Linpack (HPL). Finally, we present Data Flow Graph computing paradigm for expressing some of these constructs and demonstrate a method of converting a data flow program can be into a consistent MPI or a JAVA-RMI parallel program.

2 Some NASA Applications and Their Task Decompositions

A number of NASA applications solve fluid dynamic problems using structured or unstructured cubical grids. The structured grids are used in simple domains, while unstructured grids are used to approximate very complex domains. To approximate complex domains with a set of overlapping structured grids the Chimera overset grid method have been developed [4]. It allows iterations of the solution algorithm to be performed on each grid independently, followed by an update of the

solution at the boundary points and at the overlapping parts of the grids. Load balancing, cache optimization, and overlapping of computations and communications for structured grids are well studied and efficiently implemented in mentioned NASA codes. They use bin packing algorithms for load balancing of overset grids, space-filling curves [7] for improving cache reuse, and Jacobi method to update boundary points instead of Gauss-Seidel method for parallelization iterations over the grids. On the other hand, some industry standard methods such as Metis partitioner [15] have not been used in these codes.

To compare efficiency of different computers for running these codes, the NASA Advanced Supercomputing (NAS) Division has developed the NAS parallel benchmarks (NPB) and the NAS grid benchmarks GridNPB. These benchmarks represent computationally or data intensive segments of these codes and serve as a tool for performance estimation of real applications on particular computer architectures. Currently NPB include simulated applications and kernel benchmarks. There are six simulated applications: CFD applications BT, SP, and LU, data intensive applications DC and DT, and a computational chemistry application UA. There are five kernel benchmarks FT, MG, CG, ED and IS. The original benchmarks are written in Fortran and C and are parallelized using MPI and OpenMP. Some benchmarks are available in HPF and Java. The HPF and Java versions use for parallelization data distribution and multithreading respectively. Each benchmark comes with a verification method, and a performance model.

Because parallelism of these benchmarks limited by few hundred processors these benchmarks become harder and harder to scale for the computational grids and for massively parallel computers having thousands of processors. On large systems, insignificant serial sections, data reductions and data transpositions become dominated contributors to the execution time. For these reasons another level of parallelism was added to the GridNPB [8] and Mutizone benchmarks (NPB-MZ) [11]. The GridNPB use the task graphs with the basic task is an NPB problem. The task graphs are acyclic and work in dataflow manner: a task is executed as soon as data along all incoming arcs have arrived. The granularity of the inter tasks communicated data is chosen to be about two orders of magnitude smaller than intratask communications. As a result, GridNPB scale well if communication time between grid nodes are not more than 100 times slower than communications within each node. The NPB-MZ benchmarks add another level of structured grids to the NPB. The top level structured grid is partitioned onto smaller subgrids. Within each subgrid it solves one of the application benchmarks BT, SP, or LU. The solutions on the boundaries of these subgrids are exchanged after each iteration of the solver. Since the tasks working on the subgrids are loosely coupled, the NPB-MZ benchmarks can be scaled to the thousands of processors on clusters of SMPs.

3 Patterns used in the NAS Parallel and Grid Benchmarks

3.1 The NAS Parallel Benchmarks

The early versions of the NPB include simulated CFD applications BT, SP, and LU. These applications solve a discretization of the Navier-Stokes equation

$$K(u^{q+1} - u^q) = Lu^q,$$

where u is a five-dimensional vector of density, moments and energy, K is a discretization of the Navier-Stokes operator, and L is the right-hand side operator. For three-dimensional grids K and L usually are represented as seven-diagonal block matrix of 5x5 blocks. The benchmarks represent three methods to approximate (or split) K as a product of linear operators. The BT benchmark splits K into a product of three **B**lock **T**ridiagonal operators one along each dimension of the computational mesh, SP uses the Beam and Warming split that involves three **S**calar **P**entadiagonal operators interleaved with diagonal matrices of 5x5 blocks, and LU employs a **L**ower and **U**pper diagonal splitting.

The first two methods represent so called Alternative Directions Implicit (ADI) pattern. In this pattern algorithm is structured so that it performs solutions of linear systems in x, y, and z directions iteratively, Figure 1. The L and U operators

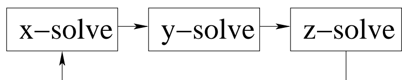


Figure 1. The ADI pattern.

of the LU benchmark have the following dependence matrices:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Hence, LU can be structured as a two-dimensional pipeline or as a hyperplane based computation. In the two-dimensional pipeline, on k^{th}-iteration the computations are performed in each horizontal plane along the diagonal $x + y = w$, where

$w = k - z$ or $w = 3N - k - z$ depending on whether L or U operator is applied, Figure 2 (N is the number of mesh points in each direction). In the hyperplane version of LU (LU-hp), the update of the flow vectors performed simultaneously at all points of a hyperplane $x + y + z = k$, Figure 2. The ADI, two-dimensional pipeline, and hyperplane patterns represent most

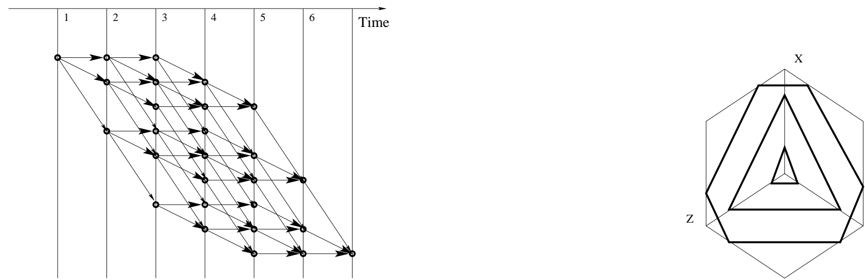


Figure 2. Seven stages of two-dimensional pipeline for processing points of 3x3x3 lattice (left pane). Various stages of processing in the hyperplane algorithm are shown on the right pane.

common patterns for sweeping through the points of three-dimensional lattice used in CFD applications.

The EP benchmark lunches independent tasks which combine their results, as shown on Figure 5. The MG (**M**ulti **G**rid) benchmark uses well known multigrid V-cycle pattern. It starts from projecting of the residual from the finest grid to the coarser grids. Then, it finds a solution on the coarsest grid. Finally, it lifts the solution on the finest grid by an interlaced sequence of interpolation and smoothing operations, Figure 3, left pane. The FT benchmark performs a few iterations of the three-dimensional Fast Fourier Transformation followed by an evolution of the transformed array. As in ADI, the three-dimensional Fourier Transformation is performed as a sequence of transformations along x, y, and z directions. The transformation along each direction is a two-dimensional array of one-dimensional Fourier transformations. Each one-dimensional Fourier is performed by a vectorized variant of self-sorting Stocham algorithm attributed to Swarztrauber. Hence, on the low level, the FT benchmark is a (vectorized) shuffle pattern, (Figure 3, right pane) recursively built of butterfly patterns. Interestingly, the FFTE algorithm, part of the HPC Challenge benchmark suite [12], on the higher level demonstrates the matrix transpose pattern, rather than ADI pattern, but on the lower level in addition to the binary shuffle it demonstrates radix-3 and radix-5 shuffles. The CG (Conjugate Gradient), IS (Integer Sort), and UA (Unstructured Adaptive) benchmarks are working

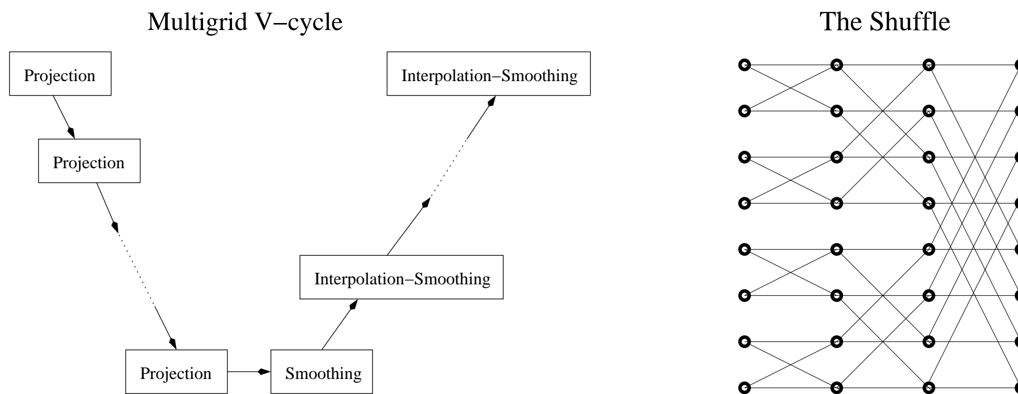


Figure 3. Multigrid V-cycle pattern and FFT shuffle.

with randomly generated matrix, randomly generated data set, and an adaptive grid respectively. These benchmarks do not expose any definite data flow patterns. On the other hand, on each iteration CG uses basic linear algebra operations: sparse matrix by vector multiplication and scalar product. The IS benchmark uses bucket sorting algorithm and parallel prefix computation of partial sums (key ranking). Newly added to the NPB suit data intensive benchmarks DC (Data Cube) and DT (Data Traffic) also work with randomly generated data but they use trees and a shuffle as data flow patterns. The DC benchmark builds RB-tree to sort tuples from a dataset. DC's ability to benchmark various levels of memory hierarchy depends on the size of the tree. The tree fits in L1-cache (class S) and grows beyond main memory (class B). The DT benchmark uses quad-trees (black hole and white hole) and the binary shuffle (Figure 3, right pane) as the task graphs.

3.2 The NAS Grid Benchmarks

With automation in data processing, the results generated by some programs are consumed by other programs which, in turn, feed inputs to other programs. One example of such programming approach is the `Cart3d` package developed by M. Aftosmis at the NASA Ames Research Center [3] and recently used for ”Return to Flight” simulations on the NAS Columbia machine. This package automates the computing process from description of the aircraft geometry to the actual computation of the drag, lift, and side force coefficients. The `Cart3d` package involves a number of applications working in data flow mode: triangulation of the aircraft surface, mesh generation, partition of the mesh with use of the space filling curves, mutigriding for acceleration of the convergence, solution of the flow equations, postprocessing and visualization of the final results, Figure 4. The actual inspection of data by the users becomes an infrequent event.

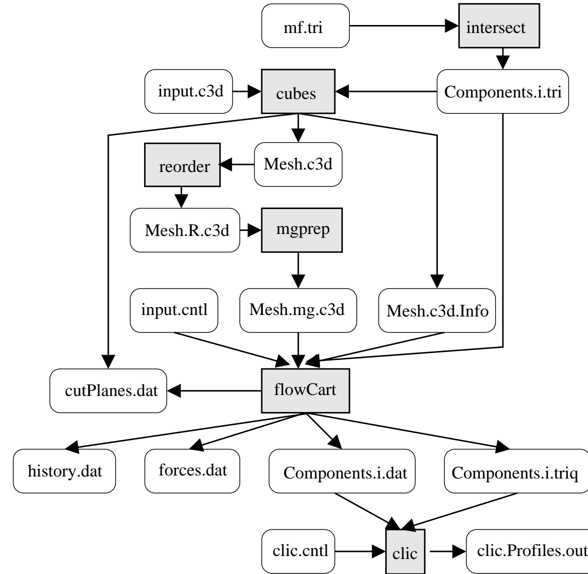


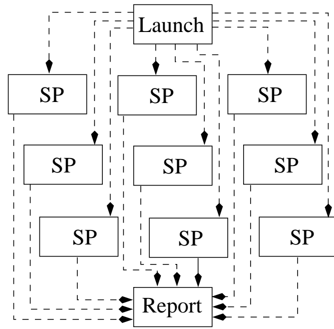
Figure 4. A flowchart of the Cart3D package. Grey boxes indicate executables, clear boxes indicate I/O files.

Observation of this trend resulted in task graph programming paradigm which encapsulates tasks in nodes and data communicated between tasks in arcs. The general perception in the community is that as the task graphs will be growing, the amount of inter task parallelism will be growing as well. These applications will be able to scale to thousands of processors and to the computational grids. This trend, coupled with the observation that two basic grid services, “create task” and “communicate” are sufficient for data flow computations, prompted NAS to create the NAS Grid Benchmarks [8]. A GridNPB of a particular class (problem size) is specified by a data flow graph which encapsulates NGB tasks (NPB codes) and communications between these tasks. The Report node of the graph is endowed with a verification test to determine correctness of the computational result. The GridNPB use NPB codes BT, SP, LU, MG, and FT, since these codes are well-studied, well-understood, portable, and widely accepted as scientific benchmarking codes.

GridNPB employs graphs named Embarrassingly Distributed (ED), Helical Chain (HC), Visualization Pipeline (VP), and Mixed Bag (MB), as shown in Figures 5 and 6. ED represents the so called parameter studies, which constitute multiple independent runs of the same program, but with different input parameters. At NASA Ames, flow solvers—symbolized by SP—are often used for such studies. HC represents long chains of repeating processes, such as a set of flow computations that are run one after the other, as is customary when breaking up very long running simulations into series of tasks. VP represents chains of compound processes, like those encountered when visualizing flow solutions as the simulation progresses. MB is similar to VP, but it the emphasis is on introducing asymmetry. Different amounts of data are transferred between different tasks, and some tasks take longer to run than others, presenting a tough job to a grid scheduler.

4 Pattern Analysis of Program Traces

The data flow patterns listed in the previous section are explicit, frequently used in applications as design patterns [14]. Dynamic event patterns occur during a specific workload or run of a program. These patterns may depend on data, and may be not obvious to the designer of a program. Identification and analysis of these patterns may be useful for improving application



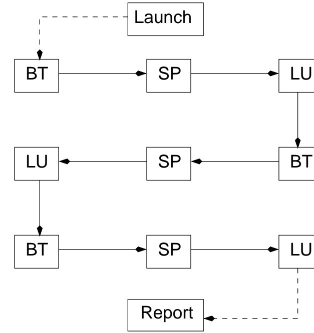
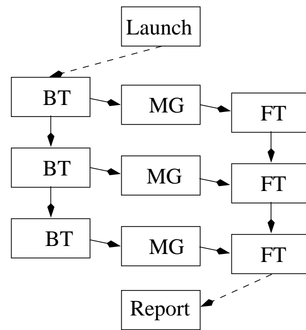


Figure 5. Embarrassingly Distributed and Helical Chain patterns.



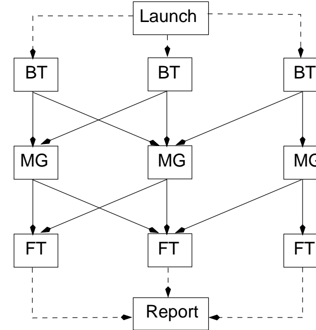


Figure 6. Visualization Pipeline and Mixed Bag patterns.

performance, adding compiler optimizations and hardware support to handle these patterns. One example of using patterns observed in scientific computations is implementation of vector registers and pipelined functional units in vector computers. Another example, are JIT compilers which optimize code by analyzing patterns in its execution.

The pattern analysis of programs is similar to the Fourier analysis of signals: in both cases an observed subject is expanded over a set of basis functions. The difference is that the pattern analysis is highly nonlinear. Any recursive function can be build of a few elementary functions, but building operators: composition, recursion, and minimization are highly non-linear. As it was demonstrated by S. Wolfram [17], a simple cellular automaton can generate a computational trace lacking any recognizable pattern.

One way to identify dynamic patterns is to use system-wide hardware counters to collect distributions and correlations of various events. The correlations, indicate that some hardware support for the correlated events would speed up the workload. The system-wide approach in blind to the applications: it is not usually possible to trace these patterns back to the application code and even to a specific application. Detection of dynamic event patterns during application execution can be done through pattern analysis of trace files.

In our approach to identify frequent patterns in program traces we apply the Data Mining method used in On-Line Analytic Processing (OLAP). Particularly, we use the data mining algorithm implemented in DC parallel benchmark [6]. This approach applies the Data Cube Operator (DCO) to identify the most common patterns in the trace. The DCO computes views (or group-bys) of a dataset. For a chosen subset of k attributes, a view is a sorted set of k-tuples containing only the chosen attributes with accumulated measures of the duplicates. DCO computes views of interesting subsets of the attributes.

To demonstrate this approach we use the trace file of the HPL program running on four processors. For analysis we used only 1920 messages presented in this file. No knowledge about the MPI implementation of HPL or about its mapping onto the system were used. The task communication graph is shown in Figure 7 and one view of the trace file generated by a DCO is shown in Table 1. Many other views and patterns may be extracted by this method with a minimal effort. For example some views on quantified communication time indicate that there are only four messages have duration over ten seconds: two $1 \rightarrow 2$ and two $1 \rightarrow 3$. The reason for this large delay may shed a light onto improving HPL performance in current hardware configuration.

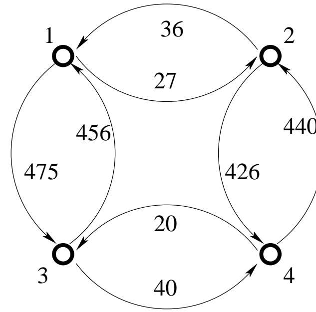


Figure 7. Number of messages sent among processors in HPL algorithm.

Table 1. A view of the trace file generated by an OLAP system.

Sender	Receiver	Number of Messages	Communication Time	Bytes Sent	Bandwidth
1	2	27	84784	724380	10
1	3	475	113849	1123108	9
2	1	36	10872	733840	67
2	4	426	32943	1017360	30
3	1	456	48683	1041896	21
3	4	40	9127	758700	83
4	2	440	39457	1081912	27
4	3	20	7429	667816	89

5 Parallel Programming with Data Flow Graphs

To take advantage of patterns observed in the programs a programmer should be able to express these patterns easily. Unfortunately, procedural programming languages do not have constructs to express patterns other than arrays. This deficiency of programming languages is even more striking if we recall that internal representation of any program is a flow graph, sometimes called ”parse tree”. The nodes of the flow graph are basic blocks of the program, the arcs indicate possible iterblock transitions. Many problems related to compilation, program optimization, dependence analysis, and parallelization would be alleviated if the users will have simple language constructs to express the patterns in the terms close to the program flow graph. The flowcharts, popular in early days of programming, are still used by hardware designers in form of schematics and data flow graphs [1]. The DFG concept was well studied as a programming concept for Data Flow Machines [5]. In spite of this, the existing programming languages still do not provide a simple way to express common patterns. Here we demonstrate one approach to express the patterns.

In a rudimentary form, the DFGs are implemented in Java version of the NAS Grid Benchmarks and in C/MPI version of DT benchmark. OO languages such as Java and C++ are very natural for declaring DGArc, DGNode, and DGraphs as classes, [8]. The same goal, however, can be achieved with C data structures (C/MPI version of DT), see Appendix A.

Representation of a parallel program via DFGs has essential advantage that translation of the program to an MPI program can be done almost automatically with a guarantee that each send is matched with a receive. The translation can be performed in two steps. First, we assign each node to a separate MPI process. Then, for each arc we issue a pair of send-receive in the processes corresponding to the tail and head of the arc and use the arc ID as a tag. For acyclic DFGs, theoretically, this translation should generate a deadlock-free MPI program. In practice, however, extra caution should be used, since only a limited number of send (receive) call can be pending due to the system limitations. For example, a deadlock may happened for the graph shown in Figure 8, if the sends will be posted in the ascending order of the arc ID, but the receives in the descending order of arc ID.

The DFGs in the GridNPB benchmarks (Java version) also allow simplify use of Java RMI interface in distributed environment. This version reads benchmark DFG as an input which the user can annotate with the grid machines where she wants to assign the tasks. Each task waits for inputs on all incoming arcs, then processes these inputs, and, finally, sends the results along all outgoing arcs to the destination tasks. If the DFG is acyclic, this execution will automatically be deadlock free.

The main advantage of programming with DFGs is that they increase level of programming abstractions and relieve the programmer from explicit details of the patterns she works with. The programming with the DFGs has also well known limitations: the graph must be acyclic and there is no generally accepted way to include control into the graph. Also, if during the execution one of the processors (grid machines) fails, a reassignment of a task to a new processor requires another control

layer on the top of the DFG. In spite of these limitations the DFG is a useful programming concept employed implicitly or explicitly in many applications and benchmarks.

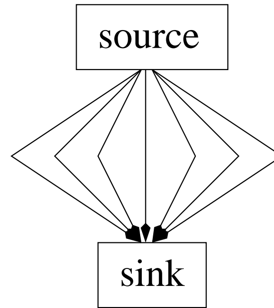


Figure 8. Let `source` and `sink` are assigned to different MPI processes and send (receive) calls are posted in increase (decrease) of arc ID. If the system limits only three simultaneous send (receive) calls, an MPI program will deadlock.

6 Conclusions

A number of static patterns exist in the parallel programs. These patterns are well understood and many of them are implemented in parallel benchmarks. The dynamic patterns, appearing during program runs, are data, application, or hardware dependent. These patterns can be detected by OLAP methods used in data mining. Many static patterns can be expressed by Data Flow Graphs which facilitate development of parallel and distributed programs.

7 Acknowledgements

The authors want to thank Scott McMillan of Intel for providing the trace file, Timothy Mattson of Intel for discussion of pattern analysis, Michael Aftosmis of NASA Ames for providing the Cart3D package, and Leonid Shabanov of CorssZ Solutions USA for generation Table 1 using an OLAP system.

8 Appendix A: DGraph Declaration in C

Below we list implementations of the DGArc, DGNode, and DGraph which are used in the DT benchmark. Similar implementations in Java are used in the Java version of the GridNPB. Some fields such as `length` nad `width`, are not necessary for the implementation of the data Flow Graph, but used to store additional information about arcs and nodes in the benchmarks.

```
typedef struct{
  int id;
  void *tail,*head;
  int length,width,attribute,maxWidth;
} DGArc;

typedef struct{
  int maxInDegree,maxOutDegree;
  int inDegree,outDegree;
  int id;
  char *name;
  DGArc **inArc,**outArc;
  int depth,height,width;
  int color,attribute,address,verified;
  void *feat;
} DGNode;
```

```
typedef struct{
  int maxNodes,maxArcs;
  int id;
  char *name;
  int numNodes,numArcs;
  DGNode **node;
  DGArc **arc;
} DGraph;

DGArc *newArc(DGNode *tl,DGNode *hd);
void arcShow(DGArc *ar);
DGNode *newNode(char *nm);
void nodeShow(DGNode* nd);

DGraph* newDGraph(char *nm);
int AttachNode(DGraph *dg,DGNode *nd);
int AttachArc(DGraph *dg,DGArc* nar);
void graphShow(DGraph *dg,int DetailsLevel);
```

References

[1] C. Bobda. “Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement.” *Dissertatin, Padderborn University.*, May 2003, 90 pp.

[2] M.J. Aftosmis, M. J. Berger, and S. M. Murman. “Applications of Space-Filling Curves to Cartesian Methods for CFD.” *NAS Technical Report, NAS-04-002, http://www.nas.nasa.gov*, March 2004.

[3] Cart3D. http://people.nas.nasa.gov/∼aftosmis/cart3d/cart3Dhome.html.

[4] W. M. Chan, S. E. Rogers, S. M. Nash, P. G. Buning. “User’s Manual for Chimera Grid Tools”. http://rotorcraft.arc.nasa.gov/cfd/CFD4/CGT/man.html

[5] J.B. Dennis. “Data Flow Supercomputers.” *Computer 13*, 11, pp. 48-56, November 1980.

[6] M.A. Frumkin, L.V. Shabanov. “Benchmarking Memory Performance with the Data Cube Operator.” *Proceedings of ISCA 17th International Conference Parallel and Distributed Computing Systems, September 2004, pp 165-171, see also NAS Technical Report, NAS-04-013, http://www.nas.nasa.gov*,

[7] M. Frumkin, R. Hood, and J. Yan. “On the Information Content of Program Traces.” *NAS Technical Report, NAS-98-008, http://www.nas.nasa.gov*, March 1998.

[8] M.A. Frumkin, Rob F. Van der Wijngaart. “NAS Grid Benchmarks: A Tool for Grid Space Exploration.” *Cluster Computing* 5, pp. 247-255, 2002.

[9] NAS parallel and NAS grid benchmarks web site. http://www.nas.nasa.gov/Software/NPB.

[10] D. C. Jespersen “Parallelism and OVERFLOW” *NAS Technical Report, NAS-98-013, http://www.nas.nasa.gov*, October 1998.

[11] H. Jin, Rob F. Van der Wijngaart. “Performance Characteristics of the Muti-Zone NAS Parallel Benchmarks.” *Proceedings of 18-th International Parallel & Distributed Processing Symposium, IPDPS 2004.*

[12] HPC Challenge Benchmark. http://icl.cs.utc.edu/hpcc.

[13] H. Jin, M. Hribar, and Jerry Yan. “Parallelization of ARC3D with Computer-Aided Tools”. *NAS Technical Report, NAS-98-005, http://www.nas.nasa.gov*, 1998.

[14] T.G. Mattson, B.A. Sanders, B.L. Massingill. “Patterns for Parallel Programming.”, Addison-Wesley, 2004.

[15] METIS Family of Multilevel Partitioning Algorithms. http://www-users.cs.umn.edu/∼karypis/metis.

[16] S. E. Rogers, D. Kwak, C. Kiris. "Numerical Solution of the Incompressible Navier-Stokes Equations for Steady-State and Time-Dependent Problems", *AIAA Journal.* 29(4), April 1991, pp. 603–610. Also see *people.nas.nasa.gov/∼rogers/ins3d/ins3d.html*.

[17] S. Wolfram. "Cellular Automaton Supercomputing". *High-Speed Computing: Scientific Applications and Algorithm Design, ed. Robert B. Wilhelmson, University of Illinois Press, 1988.*