*11th International Parallel Processing Symposium, Geneva, Switzerland, April 1997.*

# DPF: A Data Parallel Fortran Benchmark Suite

Yu Hu
Div. Eng. and Applied Sc.
Harvard University
hu@deas.harvard.edu

S. Lennart Johnsson
Computer Science Dept.
University of Houston
johnsson@cs.uh.edu

Dimitris Kehagias
Fixed Income Sales & Anal.
Merrill Lynch
dkehagia@fir.ml.com

Nadia Shalaby
Div. Eng. and Applied Sc.
Harvard University
nadia@deas.harvard.edu

## Abstract

*We present the Data Parallel Fortran (DPF) benchmark suite, a set of data parallel Fortran codes for evaluating data parallel compilers appropriate for any target parallel architecture, with shared or distributed memory. The codes are provided in basic, optimized and several library versions. The functionality of the benchmarks cover collective communication functions, scientific software library functions, and application kernels that reflect the computational structure and communication patterns in fluid dynamic simulations, fundamental physics and molecular studies in chemistry or biology. The DPF benchmark suite assumes the language model of High Performance Fortran, and provides performance evaluation metrics of busy and elapsed times and FLOP rates, FLOP count, memory usage, communication patterns, local memory access, and arithmetic efficiency as well as operation and communication counts per iteration. An instance of the benchmark suite was fully implemented in CM–Fortran and tested on the CM–5.*

## 1 Introduction

### 1.1 Motivation, Functionality and Scope

High performance is the main motivation for scalable architectures, while portability of user codes is critical for making scalable architectures economically feasible for all but a few applications. These requirements represent a significant challenge for all software developers, whether they are developing compilers, run–time systems, operating systems or software libraries. The goal in developing the Data Parallel Fortran (DPF) benchmark suite was to produce a means for evaluating such high performance software suites. In particular, we target data parallel Fortran compilers; such as any of the High Performance Fortran (HPF) [5] compilers, Fortran–90 [12] compilers, the Fortran–Y or CRAFT [4] compiler, as well as the Connection Machine Fortran (CMF) [15] compilers. At the time the benchmarks were developed, CMF was the only data parallel Fortran language with a production quality compiler available. Hence, the benchmarks were all written in CMF. Conversion to any Fortran standard, in particular HPF, should be straight–forward given the limited differences between CMF and HPF.

The functionality of the benchmarks cover collective communication functions, scientific software library functions, and application kernels. Communication functions are intended to measure data motion in memory hierarchies. In fact, efficient exploitation of spatial and temporal locality of reference is the main objective of compilers for high performance. Some functions, such as gather and scatter, require efficient run–time system support. For conventional vector architectures, gather and scatter have been implemented as special instructions, and array transposition has been included in some languages, like Fortran–90, as an intrinsic function. Reduction and broadcast operations are other examples of operations on collection of variables that are incorporated in modern languages. On scalable architectures these functions are usually implemented as part of a collective communications library, which may be part of the run–time system or a separate library. Several of these functions are incorporated into the emerging Message–Passing Interface (MPI) standard [11].

Scientific software library functions, particularly in the early years of new architectures, may offer significantly higher performance by being implemented, at least in part, in lower level languages to avoid deficiencies in compiler technology, or in the implementation of compilers and run–time systems. However, low level library implementation tends to be very costly, often meaning that good performance may not be available until late in the hardware production cycle. Thus, the amount of low level code in software libraries should be minimized not only for direct cost reasons.

The DPF benchmark suite also contains a set of small application codes containing typical "inner loop" constructs that are critical for performance, but that are typically not found in libraries. An example is stencil evaluations in explicit finite difference codes. The benchmarks were chosen to complement each other, such that a good coverage would be obtained of language constructs and idioms frequently used in scientific applications, and for which high performance is critical for good performance of the entire application. The application benchmarks were selected so as to represent the dominating applications on large data parallel machines. Much of the resources at supercomputer centers are consumed by codes used in fluid dynamic simulations, in fundamental physics and in molecular studies in chemistry or biology, and the DPF application codes reflect this fact.

Some of the objectives for the DPF benchmark suite are similar to that of several other collections of programs. The NAS parallel benchmarks [1] are "paper and pencil" benchmarks intended for vendors and implementors using algorithms and programming models appropriate to their particular platforms. The NAS parallel benchmarks 2.0 [2] are an MPI–based source implementation. However, to our knowledge, this suite is the first focused entirely on data parallel software environments.

The benchmark suite is divided into two groups, the *library functions*, and the *applications oriented codes*. *Library functions* are of two types: *communication*, which include four functions, and *linear algebra*, which consist of eight function suites. The *application codes* are comprised of twenty different application benchmarks.

Here, we provide an overview of the DPF benchmark suite. A detailed description and instructions for the use of the suite are covered in [7] and in the online documentation at the URL `http://www.das.harvard.edu/cs/research/dpf/root.html`. Sources, examples of DPF benchmark use and produced output are also available there. In all, there are 32 benchmarks in the suite, comprising about 17,000 lines of source code. The full DPF benchmark suite, including the sample data files, occupies 2.64 MBytes.

After presenting the code versions, architectural model, language aspects and performance evaluation in sections 1.2, 1.3, 1.4 and 1.5, respectively, we discuss the library functions for communication in Section 2, the library functions for linear algebra in Section 3 and the applications oriented codes in Section 4. Our intent is to provide an overview of the benchmark codes for prospective users to understand which language or compiler feature a particular benchmark attempts to evaluate. Therefore, for each of the codes, we document several aspects of the employed data structures, the floating–point computation count, the data structures' distribution among memory modules, the dominating communication patterns, the primary local memory access patterns, as well as how all these aspects are implemented in the benchmark code. In sections 3 and 4, we summarize these aspects in comprehensive tables to facilitate assessment and comparison. These tables should be used as a primary guide in selecting the appropriate code (or group of codes) from the entire benchmark suite, according to a given set of goals and criteria.

## 1.2 DPF Code Versions

A number of the benchmarks exist in several forms, as tabulated in Table 1. A **basic** CMF version is provided in most cases, intended to represent a "typical" user code produced by a knowledgeable user without a lengthy optimization process. In some cases an **optimized** CMF version is also provided, representing the kind of code a highly performance oriented, advanced programmer with good knowledge of the compiler and the architecture would produce.

| Benchmark Name | basic | optimized | library | CMSSL | C/DPEAC |
|---|---|---|---|---|---|
| boson | × | | | | |
| conj–grad | × | | | | |
| diff–1D | | | | × | |
| diff–2D | | | | × | |
| diff–3D | × | | | | |
| ellip–2D | × | | | | |
| fem–3D | | | | × | |
| fermion | × | | | | × |
| fft | × | | | × | |
| gather | × | | | | |
| gauss–jordan | × | | | | |
| gmo | | × | | | |
| jacobi | × | | | | |
| ks–spectral | × | | | × | |
| lu | | | | × | |
| matrix–vector | × | | | × | |
| md | × | | | | |
| mdcell | × | | | | |
| n–body | × | | | | |
| pcr | × | | | × | |
| pic–gather–scatter | | | × | | |
| pic–simple | | | | × | |
| qcd–kernel | | × | | | |
| qmc | | | × | | |
| qptransport | | | × | | |
| qr | × | | | × | |
| reduction | × | | | | |
| rp | × | | | | |
| scatter | × | | | | |
| step4 | × | | | | |
| transpose | × | | | × | |
| wave–1D | × | | | × | |

**Table 1. Benchmark suite code versions**

Optimization can also be achieved via calls to highly optimized routines, which may be implemented in languages other than CMF. Such routines are typically found in a run–time system library, a scientific software library or, in the case of the Connection Machine systems, in the CMF library augmenting the intrinsic functions of the language. For optimizations resorting to source language library functions, the code version is termed **library**, whereas for those codes calling the specialized, in our instance, Connection Machine Scientific Software Library (CMSSL) [16] functions, we refer to the code version as **CMSSL**. In other cases, rather than resorting to library calls, some segment of the code, critical to the benchmark performance, is identified and implemented in the lower level language C/DPEAC [17]. This code version is termed **C/DPEAC** and is assumed to give the programmer finer control over the underlying architecture.

## 1.3 Architectural Model

Constructing a benchmark suite suitable for compiler evaluation in addition to a language definition requires both a hardware model and an execution model. Most benchmarks in the DPF suite are appropriate for any parallel architecture, whether the memory is distributed among the processors or shared. Some of the benchmarks focus on evaluating how well the local memory hierarchy in a distributed memory multiprocessor is used. However, such benchmarks may also be very useful in architectures with distributed caches, such as the Stanford Dash and Flash architectures [10]. Other benchmarks also contain constructs related to an execution model in which one processor is primarily responsible for the execution control of single thread programs, such as a typical HPF (no extrinsic procedures) and CMF (no local–global features) program. Such processors act very much in the same fashion as the scalar unit in a typical vector processor.

For distributed memory architectures, the efficiency of a parallel code is highly dependent on minimizing communication between the processors, maximizing data locality, and exploiting the memory hierarchy. For shared memory architectures, the parallel code efficiency is affected by how the data referencing pattern interplays with maintaining cache coherence, specifically, whether cache coherence is maintained at the level of the hardware [9, 10], run–time system [14], or compiler [13]. Other architectural features affecting benchmark performance are interconnection network topology, nodal architecture, availability of built–in hardware for certain specialized communication patterns, such as broadcast and reduction, available hardware support for program control execution, as well as parallel I/O.

### 1.4 Language Aspects

The DPF benchmark suite intended language is HPF, and we developed an instance of it in CMF with the CM–5 as the target platform with its environment of run–time systems, available libraries and most importantly, its data–parallel Fortran compiler. In our terminology, we refer to the standard notions in general terms adhering to the HPF standard. For instance, we refer to the axes of parallel arrays as *local* and *parallel*.

The performance evaluation and analysis is based on the execution semantics of HPF. An example is the execution of the statement `vtv = sum(v*v, mask)`, where the self inner product of the vector `v` is executed for all elements, rather than only the unmasked ones. Thus, when analyzing performance for unmasked operations, we take into consideration the entire vector and not only the unmasked elements. In short, every time an ambiguity emerges, we resolve it by adhering to HPF conventions, since we assume all HPF compilers to adhere to the standard.

### 1.5 Performance Evaluation

The DPF codes produce the following performance metrics:
(1) *Busy time (sec.):* non–idle exectution time,
(2) *Elapsed time (sec.):* total benchmark execution time,
(3) *Busy floprate (MFLOPs):* number of million floating–point operations per second (FLOP count by busy time),
(4) *Elapsed floprate (MFLOPs):* million floating–point operations per second (FLOP count by elapsed time).

In some of the application codes, namely boson, fem-3D, md, mdcell, pic-gaussian, qcd-kernel, qptransport and step4, the above measures are provided for code segments, rather than the entire benchmark. Similarly, performance metrics for different modules of a benchmark may also be reported separately. For instance, the factorization and solution times for qr and lu, as well as the the constituents of the kernel in diff-1D and diff-2D, are timed separately.

We quantify performance by the following attributes:
(1) *FLOP count:* In counting the FLOPs, we adopt the operation counts suggested in [6], assuming one FLOP for real addition, subtraction and multiplication, four FLOPs for division and square root, and eight FLOPs for logarithmic and trigonometric functions. The reduction operations and parallel prefix operations, such as the intrinsic *SUM* and segmented scans, are counted for their sequential FLOPs, which is $N-1$ in this case. For computations involving masks, we seek the most accurate FLOP count: reporting an exact FLOP count when the outcome of a mask is deterministic, and resorting to the upper bound, when the mask outcome cannot be determined at compile time. Redundant operations are sometimes included in the FLOP count, as a consequence of the semantics dictated by HPF.
(2) *Arithmetic efficiency (%):* Only computed for linear algebra functions, by dividing the busy FLOP rate by the peak FLOP rate of all the participating processors[1].
(3) *Memory usage (in bytes):* We assume the standard data type sizes, with an associated symbolic notation: 4(t), 4(l), 4(s), 8(d), 8(c), 16(z) for integer, logical, single–precision real, double–precision real, single–precision complex and double–precision complex, respectively. We count the memory of all the user declared data structures including all the auxiliary arrays required by the algorithm's implementation. However, temporaries generated by the compiler are not accounted for. In the case where a lower dimensional array $L$ is aligned with a higher dimensional array $H$, and $L$ effectively takes up the storage of *size*$\{H\}$, we report the collective memory of $L$ and $H$ to be *2 size*$\{H\}$.
(4) *Communication pattern:* We specify the types of communication that the algorithm exhibits, and the language constructs with which they are expressed. These communication patterns include stencils, gather, scatter, reduction, broadcast, all–to–all broadcast communication (AABC), all–to–all personalized communication (AAPC) [8], butterfly, scan, circular shift (cshift), send, get, and sort. It should be noted that more complex patterns (such as stencils and AABC) can be implemented by more than one simpler com-

[1] In the case of the CM–5, the peak FLOP rate is 32 MFLOPs per second per vector unit (VU) and for the CM–5E it is 40 MFLOPs per second.

munication function (e.g. cshifts, spreads, etc.).

(5) *Operation count per iteration (in FLOPs):* We give the number of floating–point operations per data point, by dividing the total FLOP count of the benchmark by the problem size. This metric serves as a first approximation to the computational grain size of the benchmark, giving an insight into how the program scales with increasing problem sizes.

(6) *Communication count per loop iteration:* We group the communication patterns invoked by this benchmark and specify exactly how many such patterns are used within the main computational loop. This metric, together with the operation count per iteration, gives the relative ratio between computation and communication in the benchmark.

(7) *Local memory access:* This attribute reports the local memory access pattern for the primary data structures in the main loop of the benchmark. This local access scheme is labeled as *N/A* where no local axes are present, *direct* where the local axis is only indexed directly by the loop variable, *indirect* where the local axis is indexed by another array and *strided* where the local axis is indexed by a triplet subscript.

## 2 Library Functions for Communication

The library communication functions measure particular communication patterns, not bundled with computation. These codes allow for evaluating the implementation of communication operations in library functions or intrinsic functions in data parallel languages. The DPF communication benchmarks are gather, scatter, reduction and transpose. The gather and reduction codes measure various forms of many–to–one communication, the scatter code one–to–many, and the transpose is implemented as an AAPC. The gather and scatter operations appear frequently in basic linear algebra operations for arbitrary sparse matrices, for histogramming and many other applications, such as finite element codes for unstructured grids. The global reduction is an essential component of the language's intrinsic functions and library routines, and the transpose, apart form being an indispensable operation in linear algebra and other numerous applications, may be used to confirm advertised bisection bandwidths. The communication library functions, except the reduction function, do not perform any floating–point operations, which is why no FLOP count is produced by theses codes.

## 3 Library Functions for Linear Algebra

The linear algebra library subset of the DPF benchmark suite is provided to enable testing the performance of compiler generated code against that of any highly optimized library, such as the CMSSL. CMSSL was created for data parallel languages and distributed memory architectures and attempts to make efficient use of the underlying system architecture with its careful choice of data layout, an efficient implementations of interprocessor data motion and optimal management of local memory hierarchy and data paths in each processor. These are all primary issues of investigation in modern compiler design for parallel languages and on parallel machine architectures.

The DPF linear algebra subset is comprised of matrix–vector multiplication (matrix–vector), two different dense matrix solvers, based on LU factorization and solution (lu) and QR factorization and solution (qr), two different tridiagonal system solvers, based on parallel cyclic reduction (pcr) and the conjugate gradient method (conj–grad), a dense eigenanalysis routine (jacobi) and an FFT routine (fft). Where possible, the interface conventions are kept identical with those of CMSSL. In many cases, different layouts are accepted and analyzed before calling the common interface.

| Code | Arrays (":serial" for local axes, ":" for parallel axes) | | | |
|---|---|---|---|---|
| | 1–D | 2–D | 3–D | 4–D |
| matrix–vector: (1) | X(:) | X(:,:) | | |
| (2) | | X(:,:) | X(:,:,:) | |
| (3) | | X(:serial,:) | X(:serial,:serial,:) | |
| (4) | | X(:,:) | X(:serial,:,:) | |
| lu | | | X(:,:,:) | |
| qr | | X(:,:) | | |
| gauss–jordan | X(:) | X(:,:) | | |
| pcr: (1) | X(:) | X(:serial,:) | | |
| (2) | | X(:,:) | X(:serial,:,:) | |
| (3) | | | X(:,:,:) | X(:serial,:,:,:) |
| conj–grad | X(:) | | | |
| jacobi | X(:) | X(:,:) | | |
| fft: 1–D | X(:) | | | |
| 2–D | | X(:) | | |
| 3–D | | | X(:) | |

**Table 2.** **Data representation and layout for dominating computations in linear algebra kernels**

| Communication Pattern | Arrays | | |
|---|---|---|---|
| | 1–D | 2–D | 3–D |
| Reduction (array dimensions for source) | | matrix–vector (1)<br>gauss–jordan<br>qr<br>lu | matrix–vector (2,3,4) |
| Broadcast (array dimensions for destination) | | matrix–vector (1)<br>gauss–jordan<br>qr<br>jacobi | matrix–vector (2,3,4)<br>lu |
| AAPC | fft 1–D | fft 2–D | fft 3–D |
| cshift | conj–grad<br>jacobi<br>fft 1–D<br>pcr (1) | jacobi<br>fft 2–D<br>pcr (2) | fft 3–D<br>pcr (3) |
| Send/Get | | gauss–jordan<br>jacobi | |

**Table 3.** **Communication of linear algebra kernels**

For ease of reference and clarity, we summarize and contrast the properties of of the linear algebra benchmarks in three tables. Table 2 gives an overview of the data representation and layout for the dominating computations of the linear algebra kernels. Table 3 shows the benchmarks classified by the communication operations that they use, along with their associated array ranks. Finally, Table 4 demonstrates the computation (FLOP count) to communication ratio in the main loop of each linear algebra benchmark, memory usage for the implemented data types, as well as

| Code | FLOP Count (per iteration) | Memory Usage (in bytes) | Communication (per iteration) | Local Memory Access |
|---|---|---|---|---|
| matrix–vector | s,d: $2nmi$<br>c,z: $8nmi$ | s: $4(n+nm+m)i$<br>d: $8(n+nm+m)i$<br>c: $8(n+nm+m)i$<br>z: $16(n+nm+m)i$ | 1 Broadcast, 1 Reduction | direct |
| lu: factor<br>lu: solve | $2/3\, n^2 i$<br>$2rni$ | d: $8n(n+2r)i$<br>d: $8n(n+2r)i$ | 1 Reduction, 1 Broadcast<br>1 Reduction | N/A<br>N/A |
| qr: factor | s,d: $(5.5m - 0.5n)n$<br>c,z: $4(5.5m - 0.5n)n^2$ | s: $24mn$<br>d: $36mn$<br>c: $40mn$<br>z: $68mn$ | 2 Reductions, 2 Broadcasts | N/A |
| qr: solve | s,d: $(8m - 1.5n)n$<br>c,z: $4(8m - 1.5n)n^2$ | s: $24mn + 4m(r+1)$<br>d: $44mn + 8m(r+1)$<br>c: $48mn + 8m(r+1)$<br>z: $92mn + 16m(r+1)$ | 2 Reductions, 4 Broadcasts | N/A |
| gauss–jordan | $n + 2 + 2n^2$ | s: $28n^2 + 16n$ | 1 Reduction, 3 Sends,<br>2 Gets, 2 Broadcasts | N/A |
| pcr | s,d: $(5r+12)ni$<br>c,z: $4(5r+12)ni$ | s: $4(r+4)ni$<br>d: $8(r+4)ni$<br>c: $8(r+4)ni$<br>z: $16(r+4)ni$ | $(2r+4)$ CSHIFTs | direct |
| conj–grad | $15n$ | d: $40n$ | 4 CSHIFTs, 3 Reductions | N/A |
| jacobi | $6n^2 + 26n$ | s: $44n^2 + 28n$<br>d: $88n^2 + 4n$ | 2 CSHIFTs on 1–D arrays,<br>2 CSHIFTs on 2–D arrays,<br>2 Sends, 4 1–D to 2–D Broadcasts | N/A |
| fft: 1–D | $5n$ | c: $60n$<br>z: $100n$ | 2 CSHIFTs, 1 AAPC | N/A |
| fft: 2–D | $10n^2$ | c: $76n^2$<br>z: $115n^2$ | 4 CSHIFTs, 2 AAPC | N/A |
| fft: 3–D | $15n^3$ | c: $92n^3$<br>z: $136n^3$ | 6 CSHIFTs, 3 AAPC | N/A |

**Table 4. Computation to communication ratio in the main loop of linear algebra library codes**

the local memory access pattern for the local axes of the arrays in the main loop of the benchmark. The tables are not representative of inherent algorithmic properties; rather, reflect the chosen implementation.

## 4 Applications Oriented Codes

These benchmarks are intended to cover a wide variety of scientific applications typically implemented on parallel machines. The DPF application benchmarks consist of quantum many–body simulation for bosons on a 2D lattice (boson), solution of the diffusion equation in 1D via a tridiagonal solver (diff-1D), in 2D via the direction implicit algorithm (diff-2D), and in 3D via an explicit finite difference method (diff-3D), solution of Poisson's equation by the Conjugate Gradient method (ellip-2D), iterative solution of finite element equations in three dimensions (fem-3D), quantum many–body computation for fermions on a 2D lattice (fermion), a highly generalized moveout seismic kernel for all forms of Kirchhoff migration and Kirchhoff DMO (gmo), integration of the Kuramoto–Sivashiniski equation by a spectral method (ks-spectral), molecular dynamics codes for Leonard–Jones force law for local forces only (mdcell) and for long range forces (md), a generic direct 2D N–body solver for long range forces (n-body), a particle–in–cell code in 2D using a straightforward implementation (pic-simple) and a sophisticated implementation (pic-gather-scatter), a staggered fermion Conjugate Gradient code for Quantum Chromo–Dynamics (qcd-kernel), a Green's function quantum Monte–Carlo (qmc) code, a quadratic programming problem on a bipartite graph (qptransport), solution of nonsymmetric linear equations using the Conjugate Gradient method (rp), an explicit finite difference method in 2D (step4), and the simulation of the inhomogeneous 1D wave equation (wave-1D).

| Code | Arrays (":serial" for local axes, ":" for parallel axes) | | | |
|---|---|---|---|---|
| | 1-D | 2–D | 3–D | 4–D, 6–D, 7–D |
| boson | | | X(:serial,:,:) | |
| diff–1D | x(:) | | | |
| diff–2D | | x(:serial,:) | | |
| diff–3D | | | x(:,:,:) | |
| ellip–2D | | x(:,:) | | |
| fermion | | | x(:,:serial,:serial) | |
| gmo | x(:) | x(:serial,:) | | |
| ks–spectral | | x(:,:) | | |
| mdcell | | | | x(:serial,:,:,:) |
| md | x(:) | x(:,:) | | |
| n–body | | x(:serial,:) | | |
| pic–simple | | x(:serial,:) | x(:serial,:,:) | |
| pic–gather–scatter | | x(:serial,:) | x(:serial,:,:) | |
| qcd–kernel | | | | x(:serial,:,:,:,:,:)<br>x(:serial,:serial,:,:,:,:,:) |
| qmc | | x(:,:) | | x(:serial,:serial,:,:) |
| qptransport | x(:) | | | |
| rp | | | x(:,:,:) | |
| step4 | | | x(:serial,:,:) | |
| wave–1D | x(:) | | | |
| | Unstructured grid | | | |
| fem–3D | | | x(:serial,:,:)<br>x(:serial,:serial,:) | |

**Table 5. Data representation and layout for dominating computations in the Application codes.**

Table 5 lists the data representation and layout for the dominating computations in the application codes. Table 7 summarizes the communication patterns in the codes. Table 8 summarizes the implementation techniques for the

| Code | FLOP Count (per iteration) | Memory Usage (in bytes) | Communication (per iteration) | Local Memory Access |
|---|---|---|---|---|
| boson | $4(258+36/n_t)n_t n_x n_y$ | s: $20n_x n_y + 64n_t + 6000$ $+2000m_b + 768n_t n_x n_y$ | 38 CSHIFTs | strided |
| diff–1D | $13n_x + 4P\log P - 8$ | d: $32n_x$ | 1 3–point Stencil, substructuring w/ pcr | N/A |
| diff–2D | $10n_x^2 - 16n_x + 16$ | d: $32n_x^2$ | 1 3–point Stencil, 1 AAPC | strided |
| diff–3D | $9(n_x-2)(n_y-2)(n_z-2)$ | d: $8n_x n_y n_z$ | 1 7–point Stencil | N/A |
| ellip–2D | $38n_x n_y$ | d: $96n_x n_y$ | 4 CSHIFTs, 3 Reductions | N/A |
| fem–3D | $18n_{ve}n_e$ | s: $56n_{ve}n_e + 140n_v$ $+1200n_e$ | 1 Gather, 1 Scatter w/combine | direct |
| fermion | local matmul | d: $144n^2 + 6ln + 48p$ | N/A | indirect |
| gmo | $6p$ | s: $p \cdot (4 \cdot \mathrm{ns}_{in} \cdot \mathrm{ntr}_{in} +$ $4 \cdot \mathrm{ns}_{out} \cdot (\mathrm{ntr}_{out} + 2) +$ $8 + 12 \cdot n_{vec})$ | N/A | indirect |
| ks–spectral | $(76 + 40\log_2 n_x)n_x n_e$ | d: $144n_x n_e$ | 8 1–D FFTs on 2–D arrays | N/A |
| mdcell | $(101+392n_p)n_p n_c{}^3$ | d: $(184+160n_p)n_x n_y n_z$ | 195 CSHIFTs, 7 Scatter on local axis | indirect |
| md | $(23+51n_p)n_p$ | d: $160n_p + 80n_p^2$ | 6 1–D to 2–D SPREADs, 3 1–D to 2–D sends, 3 2–D to 1–D Reductions | N/A |
| n–body | | | | |
| broadcast | $17n^2$ | s: $36n$ | 3 Broadcasts | direct |
| broadcast w/fill | $17n^2$ | s: $20n + 36m$ | 3 Broadcasts | direct |
| spread | $17n^2$ | s: $36n$ | 3 SPREADs | direct |
| spread w/fill | $17n^2$ | s: 20n+36m | 3 SPREADs | direct |
| cshift | $17n(n-1)$ | s: $36n$ | 3 CSHIFTs | direct |
| cshift w/fill | $17n(n-1)$ | s: $20n + 36m$ | 3 CSHIFTs | direct |
| cshift w/sym. | $13.5n(n-1) + 17n \bmod (n,2)$ | s: $48n$ | 3 CSHIFTs | direct |
| cshift w/sym.fill | $13.5n(n-1) + 17n \bmod (n,2)$ | s: $20n + 44m$ | 2.5 CSHIFTs | direct |
| pic–simple | $n_p + 15n_x n_y(\log n_x + \log n_y)$ | d: $60n_p + 72n_x n_y$ | 1 Gather w/ add 1–D to 2–D, 3 FFT, 1 Gather 3–D to 2–D | direct |
| pic–gather–scatter | 270 | s: $12n_x^3 + 88n_p$ | 81 Scans, 27 Scatters w/ add, 27 1–D to 3–D Scatters, 27 3–D to 1–D Gather | indirect |
| qcd–kernel | $606n_x n_y n_z n_t$ | s: $360n_x n_y n_z n_t i$ | 4 CSHIFTs | direct |
| qmc | $[(42 + 2n_o n_{maxw})n_p n_d n_w n_e +$ $(142n_o + 251)n_w n_e]n_b$ | d: $16n_p n_d + 96n_w n_e$ | $n_{maxw}$ SPREADs 3–D to 1–D, 5 Reductions 2–D to 1–D, $(n_p n_d + 4)$ Scans on 2–D, $(n_p n_d + 1)$ Sends, 3 Reductions 2–D to scalar | direct |
| qptransport | $34n$ | d: $160n$ | 10 Scatters 1–D to 1–D, 1 Sort, 5 Scans, 1 CSHIFT, 1 EOSHIFT, 3 Reductions | N/A |
| rp | $44n_x n_y n_z$ | s: $60n_x n_y n_z$ | 2 Reductions, 12 CSHIFTs (2 7–point Stencils) | N/A |
| step4 | 2500 | s: $500n_x n_y$ | 128 CSHIFTs (8 16–point Stencils) | direct |
| wave–1D | $29n_x + 10n_x \log n_x$ | d: $64n_x$ | 12 CSHIFTs, 2 1–D FFTs | N/A |

**Table 6. Computation to communication ratio in the main loop of the Application codes.**

stencil, gather/scatter, and AABC communication patterns. Table 6 lists the computation to communication ratio for the main loop in the application codes, memory usage for the implemented data types, as well as the local memory access pattern for the local axes of the arrays.

From the standpoint of computational structures and communication patterns, the applications may be divided into a number of classes. These classes are meant to be neither mutually exclusive nor exhaustive, but rather, demonstrate an attempt to assess the performance of different applications according to some inherent properties that inevitably dictate their computational structure and communication pattern [3]. For the class of grid–based codes, we categorize the applications according to the dichotomies from (1) to (6), and for non–grid–based codes from (7) to (11).

(1) Grid structure: The grids may be *structured* (boson, diff-1D, diff-2D, diff-3D, ellip-2D, rp, ks-spectral, pic-simple, pic-gaussian, wave-1D) which can be mapped into a Cartesian space, and tend to use communication on Cartesian grids such as *stencils* and *cshift*s. Otherwise, the grids may be *unstructured* (fem-3D) with irregular connectivity, and tend to use communication primitives tailored for general communication, such as *send–with–combiner*. There are also algorithms that do not use an Eulerian mesh but rather employ a *Lagrangian* description of the spatial layout.

(2) Linearity: These are divided into *linear* (diff-1D, diff-2D, diff-3D, ellip-2D, rp, step-4, wave-1D) and *nonlinear* (ks-spectral) differential equations. For linear differential equations with structured grids, a *stencil* primitive can be provided to retrieve the data from several neighbors simultaneously and to pipeline the combining of the data. For linear equations with unstructured grids a *send–to–queue* would get data from neighbors into a local array, which may then be combined with benefits from local optimization. In the nonlinear case, a *pshift* [16] primitive could be provided for structured grids, whereas a *send–to–queue* with local optimization would deal with unstructured grids.

(3) Solution method: These divide into *direct solvers* (diff-

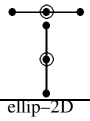
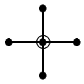
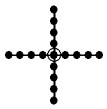
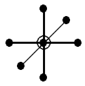
| Communication Pattern | Arrays 1–D | 2–D | 3–D | 4–, 6–, 7–D |
|---|---|---|---|---|
| Stencil | diff–1D<br><br>wave–1D<br> | diff–2D<br><br>ellip–2D<br><br>step4<br> | rp, diff–3D<br> | |
| Gather | | pic–simple | pic–gather–scatter | |
| Gather w/ combine | | pic–simple | | |
| Scatter | qptransport | | pic–gather–scatter | mdcell |
| Scatter w/ combine | | qmc | pic–gather–scatter | |
| Reduction | qptransport | ellip–2D<br>ks–spectral<br>md<br>qmc | | |
| Broadcast | | gaussj<br>md<br>n–body | | qmc |
| AABC | | md<br>n–body | | |
| AAPC | | diff–2D | | |
| Butterfly (FFT) | wave–1D | pic–simple<br>ks–spectral | | |
| Scan | qptransport | qmc | pic–gather–scatter | |
| cshift | wave–1D | ellip–2D<br>step4 | boson<br>rp | mdcell<br>qcd–kernel |
| Send/Get | | md | | |
| Sort | qptransport | pic–gather–scatter | | |
| | Unstructured grid | | | |
| Gather | | | fem–3D | |
| Scatter w/ Combine | | | fem–3D | |

**Table 7. Communication patterns in application codes**

1D, diff-2D, diff-3D), which tend to make heavy use of *spreads* and *scans*, and *iterative solvers* (ellip-2D, rp, fem-3D, step-4), which also retrieve and combine data from neighbors at each iteration step, thus making use of such primitives as *scans* and *spreads* as well.

(4) Homogeneity: *Homogeneous* (diff-1D, diff-2D, diff-3D, ks-spectral) grids have no factors that depend explicitly on spatial position. Thus, the corresponding codes may employ stencils with constant coefficients. Otherwise, the equation is *inhomogeneous* (ellip-2D, wave-1D) and stencils with variable coefficients would be required.

(5) Boundary conditions: *Periodic* (boson, ks-spectral, wave-1D) boundary conditions on a Cartesian grid point to the use of *cshift*s, whereas *Dirichlet* (ellip-2D, rp) or *Neumann* boundary conditions on the surfaces of a Cartesian grid would necessitate an *eoshift* or *cshift* with conditionalization to freeze values at the boundaries. *Constant* (diff-1D, diff-2D, diff-3D) boundary conditions on the surfaces of a

| Communication Pattern | Code | Implementation Techniques |
|---|---|---|
| Stencil | boson<br>wave–1D<br>ellip–2D<br>rp<br>mdcell | CSHIFT |
| | step4 | chained CSHIFT |
| | diff–1D<br>diff–2D<br>diff–3D | Array sections |
| Gather | fem–3D | CMSSL partitioned gather utility |
| | pic–gather–scatter<br>pic–simple | FORALL w/ indirect addressing |
| Gather w/ combine | pic–simple | FORALL w/ SUM |
| Scatter | mdcell | CMF_aset_1D or FORALL w/ indirect addressing |
| | pic–gather–scatter | FORALL w/ indirect addressing indirect addressing |
| | qptransport | indirect addressing |
| Scatter w/ combine | fem–3D | CMSSL partitioned scatter utility |
| | pic–gather–scatter | CMF_send_add or FORALL w/ indirect addressing |
| | qmc | CMF_send_overwrite |
| AABC | md | SPREAD |
| | n–body | CSHIFT, SPREAD, broadcast |

**Table 8. Implementation techniques for stencil, gather/scatter and AABC communication**

Cartesian grid often employs *array sections* to select the interior elements.

(6) Locality: Within the realm of partial differential equations, the communication is *local to the grid* (diff-1D, diff-2D, diff-3D, step-4). However, the simulation of an integral or integrodifferential equation requires more distant communication which might benefit from primitives such as *global–local transpose* operations.

7) Spectral methods: They (ks-spectral) are closely related to non–local methods for grid problems and frequently benefit from a *global–local–transpose* primitive. For Cartesian grids with periodic boundary conditions the FFT is appropriate. For other grid geometries, transforms such as spherical harmonics, Fourier–Bessel transforms, wavelet transforms, etc. may be used. All these methods benefit from a fast *global–local transpose*.

(8) Particle–in–cell codes: These applications (pic-simple, pic-gather-scatter) maintain not only a spatial grid data structure, but also a data structure for a set of particles. These particles generally possess some quantity whose density determines the force acting collectively on all of them. The particles use *send–with–combiner* to get the density on the spatial grid, some sort of elliptic solver (often done with transform methods) to get the force from the density, and *get–with–collisions* to get the force back to the particles. Since both primitives are highly sensitive to data–router collisions (this occurs at local regions of high density), the particles may first be sorted according to their destination on the lattice, and then a *sum–scan* performed prior to the router operation, which would require the *scan–with–combiner* primitive.

(9) Monte Carlo simulation: These applications all need a fast random number generator to simulate a stochastic process. The process may consist of *random walks* (qmc) or may be *lattice–based* (boson, fermion). In the former kind,

each processor locally determines how many new processes it must spawn. This is accomplished by algorithms that involve *sum–scans*, general *sends* and *segmented copy scans*. The latter kind is effectively Monte Carlo simulations on a grid which involves fast stencil–like communication.

(10) General N–body problems: In this class of applications (md, n-body), every element needs to communicate with every other element. The most efficient implementation would be mapping the communication structure to the machine hardware, but it would be useless to employ in a general purpose benchmark. Thus, the algorithms can make use of *cshifts*, *get–from–processor* and global *broadcast*. For smaller data structures, it is often possible to parallelize over particle–particle interactions, rather than particles. This would require general *send* and *sum–scan*.

(11) Molecular dynamics problems: In this class of applications (mdcell), a data structure of interacting particles is constructed, where the interaction range is short, and particles need only interact with other nearby particles, making the general N–body approach wasteful. To utilize this fact, an interaction list is determined for each particle at each instant. Good approaches would involve use of *send–to–queue* to get the particle onto the spatial grid, *cshift* or *pshift* to determine who is a neighbor and compute the force, and general *collisionless sends* to retrieve the force.

Another important aspect of the application codes is local memory moves. For some applications with local axes, this aspect can be made efficient by means of *local optimization*, i.e. the vectorization of operations on local axes, as well as indirection for local axes, so that vector–valued subscripts on local axes become efficient. Among the application codes, gmo and fermion are the only two embarrassingly parallel.

## 5 Summary

We presented the DPF benchmark suite, a set of data parallel Fortran codes for evaluating data parallel compilers appropriate for any target parallel architecture, with shared or distributed memory. The codes are provided in basic, optimized and several library versions. The functionality of the benchmarks cover collective communication functions, scientific software library functions, and application kernels that reflect the computational structure and communication patterns in typical scientific applications, particularly fluid dynamic simulations, fundamental physics and molecular studies in chemistry or biology. Assuming the language model of HPF, we provided performance evaluation metrics in the form of busy and elapsed times, busy and elapsed FLOP rates, and quantify performance according to the FLOP count, memory usage, communication pattern, local memory access, arithmetic efficiency as well as operation and communication counts per iteration. An instance of the benchmark suite was fully implemented in CM–Fortran and tested on the CM–5. We expect the DPF benchmark suite to serve an important role in the development and benchmarking of data parallel compilers.

## Acknowledgment

The DPF benchmark suite was developed by Thinking Machines Corp. with partial support from ARPA under subcontract DABT63-91-C-0031 with the Computer Science Department of Yale University and the Northeast Parallel Architectures Center (NPAC) at Syracuse University. Verification, debugging, documentation and analysis were conducted by the Parallel Computation Research Group at Harvard University.

## References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, California, Mar. 1994.

[2] D. Bailey, T. Harris, W. Saphir, R. Wijngaartand, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, California, Dec. 1995.

[3] B. Boghosian. CM–5 performance metrics suite. TMC internal memo, 1993.

[4] M. C. Chen, J. Cowie, and J. Wu. CRAFT: A framework for F90 compiler optimization. In *5th Workshop on Compilers for Parallel Computers, Malaga, Spain*, June 1995.

[5] HPF Forum. High performance fortran; language specification, version 1.0. *Scientific Prog.*, 2(1 - 2):1–170, 1993.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[7] Y. Hu, S. L. Johnsson, D. Kehagias, and N. Shalaby. DPF: A data parallel Fortran benchmark suite. Technical Report TR–36–95, Harvard University, Sep. 1995.

[8] S. L. Johnsson and C.-T. Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comp.*, 38(9):1249–1268, Sep. 1989.

[9] Kendall Square Research, Waltham, MA. *Kendall Square Research Technical Summary (of KSR-1)*, 1992.

[10] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Trans. Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[11] Message–Passing Interface Forum. MPI: A message-passing interface standard. 1994.

[12] M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford Scientific Publications, 1991.

[13] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: a high–level, machine independent language for parallel programming. *Computer*, June 1993.

[14] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *first Symp. OSDI*, Nov. 1994.

[15] TMC *CM Fortran Reference Manual, Version 2.1*, 1993.

[16] TMC *CMSSL for CM Fortran, Version 3.1*, 1993.

[17] TMC *DPEAC Reference Manual, CMOST Version 7.1*, 1993.