# Compositional Oil Reservoir Simulation in Fortran D: A Feasibility Study on Intel iPSC/860

*Ulrich Kremer*  *Marcelo Ramé*

**CRPC-TR93335**
**September, 1993**

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

To be pulished in the International Journal of Supercomputer Applications (IJSA).

# Compositional Oil Reservoir Simulation in Fortran D: A Feasibility Study on Intel iPSC/860*

Ulrich Kremer[†] kremer@rice.edu

Marcelo Ramé[‡] marcelo@rice.edu

[†]Department of Computer Science
[‡]Department of Computational and Applied Mathematics
Rice University
P.O. Box 1892
Houston, Texas 77251

*This research was supported by the U.S. Department of Energy under contract DE-FG-05-92ER25142 and by the State of Texas under contract 1059. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

## Abstract

This paper describes a study of the use of data-parallel languages such as Fortran D or High Performance Fortran (HPF) and their compilation systems for existing large scientific applications. The central question addressed in this study is: Do such languages allow the expression of the parallelism available in the applications so that the underlying compilation systems are able to generate efficient code for the specified parallelism?

Our results are based on Fortran D and the current prototype implementation of the Fortran D compilation sytem. This compilation system represents the state-of-the-art in compiler technology with respect to compile time optimizations, which is the main point of interest in this study.

We test the feasibility of Fortran D on a small section of a reservoir simulation code of wide use in the oil industry. The code is written in a style to take advantage of the machine characteristics of a vector supercomputer such as a Cray. However, this machine dependent programming style inhibits many state-of-the-art compile time optimizations. This paper provides an insight into a data-parallel programming style that allows state-of-the-art compilers to generate efficient code across a variety of machines.

Our experiments on Intel's iPSC/860 distributed-memory multiprocessor indicate that the Fortran D language and its compilation system can exploit the parallelism that exists in computations related to reservoir modeling, provided that the code is written in data-parallel programming style. Additional experiments show that other machines and their compilation systems can also exploit the compile time information available for programs written in a data-parallel programming style. This line of work will help to design enhancements to current parallel languages and compilation techniques, allowing the parallelization of large existing codes not written in data-parallel programming style.

## 1 Introduction

This study presents preliminary numerical experiments using the data-parallel Fortran D language and its compilation system to convert parts of a reservoir simulator written for high performance vector computers into a program that will run on distributed-memory parallel processors.

Enhanced and improved recovery of oil is becoming an increasingly necessary activity due to the exhaustion of oil fields all across the United States. Present economic considerations discourage tertiary recovery of oil due to the cost associated with those techniques. In this regard, a detailed and accurate computer simulation of recovery processes would provide a test bed at reasonably low cost.

Reservoir simulation has been used for three decades in the oil industry as a predicting tool in the design of drilling strategies and reservoir management. Distributed memory parallel computing appears as a way of obtaining the necessary computing power at an acceptable cost. However, this poses the problem of having to convert (i.e., partly rewrite) simulation codes written for high performance vector computers to run on distributed memory parallel machines. Since the number of man-hours invested in the majority of simulators currently used in industry is quite considerable, it is critical to develop a language and compiler capable of mapping general code written for sequential machines onto the processors of a distributed machine. Fortran D has been designed to be such a machine independent parallel programming system. Its success in the context of oil reservoir simulations will depend on whether it is able to express the parallelism available in such applications, and whether its compiler can generate efficient code for the specified parallelism. The feasibility study presented in this paper is based on the current prototype implementation of the Fortran D compiler [Tse93]. The prototype performs a variety of advanced compile time optimizations.

The simulator used in this study is UTCOMP, developed at the Petroleum Engineering Department at the University of Texas, Austin [Cha90, CLPS91, CPS90]. The code is a equation-of-state compositional miscible gas flood simulator in three space dimensions. The range of physical processes modeled include tracer floods, hydrocarbon miscible floods and carbon dioxide floods (developed miscibility). The numerical model can handle both horizontal and vertical wells and four distinct phases. The code uses higher-order, block-centered, finite differences for the discretization and declares the variables as linear arrays, which are mapped onto the three-dimensional space by a prescribed numbering scheme of the grid blocks. On the other hand, UTCOMP does not have all of the generality (e.g., fully implicit option, coupled wellbore models or a model for surface facilities) displayed by other (usually proprietary) code of wider use in industry.

The paper is structured as follows. After an introduction to the Fortran D compilation system in Section 2, we discuss the single routine chosen as the basis of our feasibility study in Section 3. Section 3.1 and Section 3.2 describe our experiments and their results, respectively. The paper concludes with a short summary of our findings.

## 2 Fortran D Compilation System

To make distributed-memory machines easier to use, several researchers have developed languages such as Fortran D that provide a global name space and annotations that let the user specify the mapping of the data onto the distributed-memory machine. Based on these annotations, a sophisticated compilation system transforms the program into a message-passing program that can be executed on a distributed-memory machine.

In the following, we will give a short overview of the Fortran D language and its compilation system.

## 2.1 Fortran D Language

Fortran D is a version of Fortran that allows the user to specify the program's data layout in two steps using `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` statements. A decomposition is an abstract problem or index domain; it does not require any storage. Each element of a decomposition represents a unit of computation. The `DECOMPOSITION` statement declares the name, dimensionality, and size of a decomposition.

The `ALIGN` statement maps arrays onto decompositions. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or across dimensions. The alignment of arrays to decompositions is specified by placeholders `I, J, K, ...` in the subscript expressions of both the array and decomposition. In the example below,

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(J-2,I+3)
```

$A$ is declared to be a two dimensional decomposition of size $N \times N$. Array $X$ is then aligned with respect to $A$ with the dimensions permuted and offsets within each dimension.

After arrays have been aligned with a decomposition, the `DISTRIBUTE` statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes are `BLOCK`, `CYCLIC`, and `BLOCK_CYCLIC`. The symbol ":" marks dimensions that are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine. In the following example,

```
DECOMPOSITION A(N,N), B(N,N)
DISTRIBUTE A(:, BLOCK)
DISTRIBUTE B(CYCLIC,:)
```

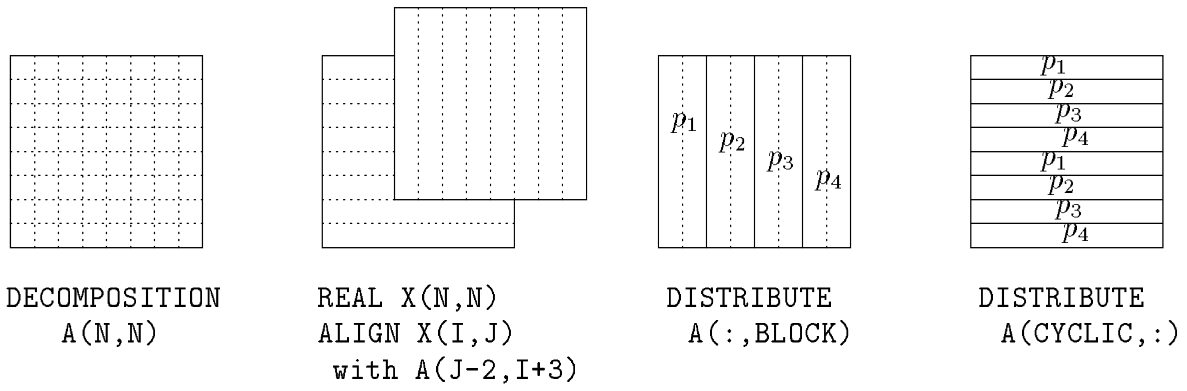

Figure 1: Fortran D Data Layout Specifications

distributing decomposition $A$ by `(:,BLOCK)` results in a column partition of arrays aligned with $A$. Distributing $B$ by `(CYCLIC,:)` partitions the rows of $B$ in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 1.

We should note that the goal in designing Fortran D is not to support the most general data decompositions possible. Instead, the intent is to provide decompositions that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs. Fortran D is a language with semantics very similar to sequential Fortran. As a result, it should be easy to use by computational scientists. In addition, we believe that the two-phase strategy for specifying data decomposition is natural and conducive to writing modular and portable code. Fortran D bears similarities to HPF [Hig93], CM Fortran [TMC89], Kali [KM91], and Vienna Fortran [CMZ92]. The complete language is described in detail elsewhere [FHK+90].

## 2.2 The Compilation System

A Fortran D compilation system translates a Fortran D program into a Fortran 77 SPMD (single program multiple data) node program that contains calls to library primitives for interprocessor communication. A vendor-supplied Fortran 77 node compiler is used to generate an executable that will run on each node of the distributed-memory target machine. A Fortran D compiler may support optimizations that reduce or hide communication overhead, exploit parallelism, or reduce memory requirements. Procedure cloning or inlining may be applied under certain conditions to improve context for optimization. A Fortran D compiler may relax the *owner computes* rule for reductions and parallel prefix operations, and for scalars or arrays that are recognized to be temporaries [HKT91, HKT92, HHKT91, Tse93]. Node compilers may perform optimizations to exploit the memory hierarchy and instruction-level parallelism available on the target node processor [Car92, Wol92, Bri92]. At present, the principal target of the prototype Fortran D compilation system [Tse93] is the Intel iPSC/860.

## 3 The Test Subroutine

For the purpose of this feasibility study, one routine of the reservoir simulator UTCOMP was chosen (procedure DISPER), which displays some of the most typical features of the computations in the whole code. This particular subroutine models the dispersion phenomena in the model for transport of chemical components, due both to molecular diffusion and to hydrodynamic dispersion, which occurs as a result of the mixing process at different length scales of the permeable rock matrix.

From the numerical model standpoint, this procedure uses a variety of finite-difference stencils, some of which are asymmetric, i.e., the intervening directions in the computational molecule depend on the runtime values of other process variables. In the original code, this directionality is implemented through indirect addressing, i.e., by defining an integer array at each grid block, that contains the runtime location the upstream grid block in each coordinate direction. This design was intended to achieve high performance on vector computers.

## 3.1 Description of Experiment

The original version (CRAY version) of the DISPER routine contains 500 noncomment lines of code. A program fragment of this code is shown in Figure 3. The shown code solves a problem of size (8x256x8).

As mentioned above, the vector version contains indirect addressing. The first form of indirection is due to the linearization of arrays. For each grid block, the physical neighboring relation is represented by the indirection array `nblk`. The array has six entries for each grid block, two entries for each coordinate direction. The second form of indirection is used to select one out of two possible computational stencils, in each coordinate direction, at runtime. The indirection array `nblkup` contains for each grid block, phase, and coordinate direction the index of the upstream grid block.

Most of the compiler optimizations described in Section 2.2 require that data access patterns are known at compile time. Therefore, most compile time optimizations cannot be applied in the presence of indirections, resulting in the generation of inefficient code by most compilers. The following steps were taken to eliminate the two types of indirections:

- We delinearized "long vectors" and expressed boundary conditions through bounds of newly introduced loops.
- We made the selection of the stencils explicit by replicating the stencil computation for the two cases under a selection guard.

The modified code for the problem size (8x256x8) is shown in Figure 4. We refer to this version as DISPER written in a *data-parallel programming style.* The transformation of DISPER into data-parallel programming style tripled the code size, i.e. the modified version has 1500 noncomment lines. This increase in code size is mostly due to the replication of the stencil computations. Note that rewritting other programs into data-parallel programming style may result in code size increases of more than a factor of three or no code size increases at all.

For our experiments we chose four different problem sizes. For all of the problem sizes, the data layout that distributes the second physical dimension ($y$-dimension) is the best choice. Such a data layout can be easily expressed in Fortran D. The data layout specifications for the data-parallel programming style version of DISPER are shown in Figure 4. The specifications are inserted after the array declarations.

We compiled the resulting Fortran D program using the current Fortran D prototype compiler [Tse93]. The prototype requires the specification of the number of processors available on the distributed-memory target machine. The message-passing code generated for eight processors and problem size (8x256x8) is shown in Figure 5 and Figure 6. The compiler was able to perform several communication optimizations. All communications have been hoisted out of loops. Redundant messages have been eliminated and messages of independent data have been aggregated into single messages if the communication occurs between the same pair of processors. The quality of the compiler generated node program compares to that of a hand-generated node program. This is due to the fact that the compiler was able to perform the for the overall performance crucial hoisting of communication out of innermost loops.

Note that due to deficiencies in the current implemenation of the prototype Fortran D compiler, we had to make some minor changes to the compiler generated node program in order to ensure its correctness. However, all communication statements were generated by the prototype compiler.

## 3.2 Results of Experiment

Four sets of numerical experiments were run on an Intel Hypercube iPSC/860, all of them with a constant x-z cross section, having 8 grid blocks in each of these directions, and 64, 128, 256 and 512 grid blocks in the y-direction, respectively. The model size was scaled in one direction only since the available implementation of the Fortran D compiler can only distribute a single dimension. For the chosen problem sizes and number of processors used, the computational load is perfectly balanced in every experiment.

The lower part of Table 1 shows the execution times of the automatically generated node programs on the Intel Hypercube iPSC/860. For comparison purposes, the upper part of the table contains the execution times of the data-parallel programming style versions on a sequential workstation (Sparc2), two superscalar workstations (Sparc10 and RS6000/550), and a vector supercomputer (Cray Y-MP). The comparision shows that the four problem sizes are non-trivial in the sense that vector and parallel supercomputers can solve the problems much faster than any of the listed workstations.

Figure 2 shows the speed-up results of the iPSC/860 experiments. The performance of the parallel code improves, as expected, as the model size in the y-direction increases from 64 to 512 grid blocks. This is naturally due to the surface-to-volume ratio of the subdomains becoming smaller as the model size increases. The speed-up results are outstanding even upto subdomain sizes of $8^3$, which gives a surface-to-volume ratio of 0.25, according to the one-dimensional decomposition chosen for the experiments. In spite of the fact that all communication calls inserted by Fortran D are of synchronous type, the observed communication to computation ratios are significantly smaller than those suggested by the corresponding surface to volume ratios of the subproblems.

The actual communication cost was measured for each of the cases run and these figures

| machine | main memory per processor | compiler | #procs | execution times (in seconds) | | | |
|---|---|---|---|---|---|---|---|
| | | | | 8x64x8 | 8x128x8 | 8x256x8 | 8x512x8 |
| **Sparc2** | 64Mbytes | f77 -O4 | 1 | 6.44 | 13.93 | 29.93 | 60.47 |
| **Sparc10** | 96Mbytes | f77 -O4 | 1 | 2.97 | 5.73 | 12.03 | 25.56 |
| **RS6000/550** | 192Mbytes | xlf -O | 1 | 2.70 | 6.78 | 17.51 | 37.44 |
| **Cray Y-MP** | 128 banks, 8Mbytes each | cf77 -Zv -Wf | 1 | 0.19 | 0.36 | 0.71 | 1.39 |
| **iPSC/860** | 16Mybtes | if77 -O4 | 1 | 8.68 | 18.09 | $*$ | $*$ |
| | | | 2 | 4.09 | 8.20 | 16.65 | $*$ |
| | | | 4 | 2.04 | 4.12 | 8.21 | 16.66 |
| | | | 8 | 1.10 | 2.05 | 4.12 | 8.21 |
| | | | 16 | 0.58 | 1.10 | 2.05 | 4.13 |
| | | | 32 | 0.35 | 0.58 | 1.09 | 2.03 |
| | | | 64 | 0.22 | 0.35 | 0.58 | 1.09 |

Table 1: Performance of the data-parallel programming style version of DISPER. A '$*$' indicates that there was not enough main memory available to run the problem size.

Figure 2: Normalized speed-up for different problem sizes

were compared to the estimated communication overhead. The estimation of the communication overhead was based on assuming that the total elasped time can be split up into a computing time, $t_c$, and a message-passing time, $t_m$. The message-passing time was assumed constant for all runs with the same global discretization size. This happens to be true exactly because of the one-dimensional data decomposition, but it would not be so in the case of multi-dimensional decompositions (i.e., message-passing in more than one coordinate direction). As for the computing time, it was assumed that no serial computations remained in the code after parallelization by Fortran D. Expressing the speed-up, S, as the ratio of the serial elapsed time to the elapsed time in parallel, and the communication overhead as the ratio of the communication time to the total time [HJ89], respectively, i.e.,

$$S = \frac{t_s}{t_t} \ \ and\ OH = \frac{t_m}{t_t} \quad ,$$

one can extract the communication overhead from knowing only the speed-up and the number of CPU's, i.e.,

$$OH = 1 - \frac{S}{N_{CPU}}.$$

In the procedure analyzed here, the incidence of serial computations is negligible, and, therefore, the estimated and measured communication overhead figures agree remarkably well.

(Re)writting programs in data-parallel programming style should not only be profitable in the context of a distributed-memory compilation system, but should also allow compilers for sequential, superscalar, and vector machines to perform more optimizations and therefore generate more efficient code. Table 2 lists the execution times of the original version of DISPER (Cray version) and the corresponding data-parallel programming style version on a Sparc2, Sparc10, RS6000/550, and a single processor of a Cray Y-MP. The experiments used the same machine configurations and compilers as shown in Table 1. The resulting numbers show that the SUN compilers for the Sparc2 and Sparc10 did not exploit the additional opportunities for compile time optimizations available in the data-parallel programming style. The IBM compiler for the RS6000/550 performed roughly the same on the three smaller problem sizes, but improved the execution time of the biggest problem size by 14%. For the Cray system, the improvement in execution time was between 32% and 47%. The latter result is of particular interest since the original version was written for a Cray machine.

| machine | programming style | execution times (in seconds) 8x64x8 | 8x128x8 | 8x256x8 | 8x512x8 |
|---|---|---|---|---|---|
| **Sparc2** | original | 5.67 | 12.55 | 27.86 | 56.58 |
| | data-parallel | 6.44 | 13.93 | 29.93 | 60.47 |
| **Sparc10** | original | 2.59 | 5.52 | 11.38 | 25.82 |
| | data-parallel | 2.97 | 5.73 | 12.03 | 25.56 |
| **RS6000/550** | original | 2.81 | 6.53 | 17.03 | 43.62 |
| | data-parallel | 2.70 | 6.78 | 17.51 | 37.44 |
| **Cray Y-MP** | original | 0.36 | 0.55 | 1.05 | 2.05 |
| | data-parallel | 0.19 | 0.36 | 0.71 | 1.39 |

Table 2: Comparison between data-parallel programming style and original style.

Programs written in data-parallel programming style have the advantage of being more portable across different machines, but have the disadvantage of a potential increase in code size as compared to a vector style version. In addition, rewritting existing programs into a data-parallel programming style is a time consuming process. To address some aspects of the latter problem, Liebrock and Kennedy have investigated compiler directives that will allow the compiler to perform delinearization of linearized arrays in 'long vector style' codes without actually modifying the source code. Preliminary results of their research can be found in [LK93]. Another option that might be worthwile to explore is to use compilation techniques for irregular problems in the context of 'long vector style' programs with array indirection. Von Hanxleden discusses such state-of-the-art compilation techniques for irregular problems for the Fortran D language [vH93].

## 4 Summary

Our feasibility study shows that data-parallel languages such as Fortran D can express the parallelism available in computations considered typical for oil reservoir simulations. However, in order to take full advantage of advanced compile time optimizations in state-of-the-art compilers, the codes have to be (re)written in a data-parallel programming style. We converted a single routine in the oil reservoir simulator UTCOMP into this programming style. Experiments show that the Fortran D compiler is able to generate code that achieves nearly linear speed-up for most problem sizes and numbers of processors. Supporting the conversion of programs into data-parallel programming style will be a challenge for language designers as well as compiler researchers.

## Acknowledgement

We would like to thank Victor Parr from Cray Research Inc. for providing us with the presented execution times on the Cray Y-MP.

## References

[Bri92] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[Car92] S. Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, September 1992.

[Cha90] Y. Chang. *Development and Application of an Equation of State Compositional Simulator.* PhD thesis, UT Austin, 1990.

[CLPS91] Y. Chang, M.T. Lim, G.A. Pope, and K. Sepehrnoori. Carbon dioxide flow patterns under multiphase flow, heterogeneous, field scale conditions. In *SPE 66th Annual Technical Conference and Exhibition*, Dallas, Texas, October 1991.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[CPS90] Y. Chang, G.A. Pope, and K. Sepehrnoori. A higher-order finite-difference compositional simulator. *J. Pet. Sci. Eng.*, 5(35), 1990.

[FHK$^+$90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[HHKT91] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. Technical Report TR91-169, Dept. of Computer Science, Rice University, November 1991.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993. To appear in *Scientific Programming*, vol. 2, no. 1.

[HJ89] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2*, chapter 1, pages 111–116. Adam Hilger, Bristol-Philadelphia, 1989.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT92] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[KM91] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[LK93] L. Liebrock and K. Kennedy. Parallelization of linearized applications in Fortran D. Technical Report CRPC-TR93-342-S, Center for Research on Parallel Computation, Rice University, November 1993.

[TMC89] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, Houston, TX, January 1993. Rice COMP TR93-199.

[vH93] R. von Hanxleden. Handling irregular problems with Fortran D — a preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as technical report CRPC-TR93-339-S, Rice University.

[Wol92] M.E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992.

```
c
      subroutine disper
cccccc...........................................................ccc
c      purpose: computes the dispersion term.
c      calls:    none
cccccc...........................................................ccc
      implicit real*8 ( a-h, o-z )
c
      parameter ( npm = 4 )
      parameter ( ncm = 5 )
      parameter ( nbm = 8 * 256 * 8 )
      parameter ( ncmp1 = ncm + 1 )
c
      common /com100/ nb, nx, ny, nz, nw, np, nc, ncp1, nbwp
      common /com210/ nblk(nbm,6), nblkup(nbm,npm,3)
      common /com440/ ddx(nbm), ddy(nbm), ddz(nbm)
      common /com470/ porstd(nbm), por(nbm), vb(nbm),
      common /com480/ diffun(npm,ncmp1), tau, alphal(npm), alphat(npm)
      common /com510/ sat(nbm,npm), denml(nbm,npm), denms(nbm,npm),
      logical lsat, lomfr, lpmfr
      common /com515/ lsat(nbm,npm), lomfr(nbm,ncm), lpmfr(nbm,npm,ncm)
      common /com540/ omfr(nbm,ncm), pmfr(nbm,npm,ncm),
      common /com850/ disp(nbm,ncmp1), dispx(nbm), dispy(nbm),
     &                         dispz(nbm)
      common /com870/ coexx(nbm,npm), coexy(nbm,npm), coexz(nbm,npm),
     &                coeyx(nbm,npm), coeyy(nbm,npm), coeyz(nbm,npm),
     &                coezx(nbm,npm), coezy(nbm,npm), coezz(nbm,npm)
c
c
          .
          .
          .

      do 900 k = 1, nc

        do 850 j = 2, np
          do 840 i = 1, nb

            ilyp1 = nblk(i,2)

            iupy = nblkup(i,j,1)

c
c          ———————
c          y-direction
c          ———————
c
            if ( ( ilyp1.gt.0 ).and.( lsat(i,j) )
     &                   .and.( lsat(ilyp1,j) ) ) then
              grady = 2. * ( pmfr(ilyp1,j,k) - pmfr(i,j,k) )
     &               / ( ddy(i) + ddy(ilyp1) )
              dispy(i) = dispy(i)
     &              + denml(iupy,j) * ( por(iupy) * sat(iupy,j)
     &                            * diffun(j,k) + coeyy(i,j) )
     &                         * grady
            endif
            .
            .
            .

 840      continue
 840     continue
            .
            .
            .

 900   continue
```

Figure 3: original code fragment (CRAY version)

```
      double precision  ddx(256,8,8),ddy(256,8,8),ddz(256,8,8)
      double precision  por(256,8,8),pmfr(256,8,8,4,5),diffun(4,6)
      double precision  alphal(4),alphat(4),sat(256,8,8,4),denml(256,8,8,4)
       logical           lsat(256,8,8,4)
      double precision  potx(256,8,8,4),poty(256,8,8,4),potz(256,8,8,4)
      double precision  disp(256,8,8,6),dispx(256,8,8),dispy(256,8,8),
     &                  dispz(256,8,8)
      double precision  coexx(256,8,8,4),coexy(256,8,8,4),coexz(256,8,8,4),
     &                  coeyx(256,8,8,4),coeyy(256,8,8,4),coeyz(256,8,8,4),
     &                  coezx(256,8,8,4),coezy(256,8,8,4),coezz(256,8,8,4)

c ———- FORTRAN D ———-
      integer    n$proc
      parameter (n$proc = 8)
c     decomposition dd(256)
c     align ltemp(i),grady(i)                         with dd(i,j,k,l,m)
c     align ddy(i,j,k),r(i,j,k),dispy(i,j,k)          with dd(i,j,k,l,m)
c     align poty(i,j,k,l),coeyy(i,j,k,l)              with dd(i,j,k,l,m)
c     align sat(i,j,k,l),lsat(i,j,k,l),denml(i,j,k,l) with dd(i,j,k,l,m)
c     align pmfr(i,j,k,l,m)                           with dd(i,j,k,l,m)
c     distribute dd(block,:,:,:,:)
c ————————————-
          . . .

      do 900 k = 1, 5

        do 850 j = 2, 4
          do 840 i3 = 1, 8
            do 840 i2 = 1, 8
              do 840 i1 = 1, 256

                if(poty(i1,i2,i3,j).lt.0) then
c                 ———–
c                 y-direction
c                 ———–
                  if ( i1.ne. 256 ) then
                    ltemp(i1) = lsat(i1,i2,i3,j).and.lsat(i1+1,i2,i3,j)
                    if ( ltemp(i1) ) then
                      grady(i1) = 2.*(pmfr(i1+1,i2,i3,j,k)
     &                          -pmfr(i1,i2,i3,j,k))
     &                          / ( ddy(i1,i2,i3) + ddy(i1+1,i2,i3) )
                      dispy(i1,i2,i3) = dispy(i1,i2,i3) + denml(i1,i2,i3,j)
     &                      * ( por(i1,i2,i3) * sat(i1,i2,i3,j)
     &                      * diffun(j,k) + coeyy(i1,i2,i3,j) )
     &                      * grady(i1)
                    endif
                  endif
                  . . .

                else
                  . . .

                  if ( i1.ne. 256 ) then
                    ltemp(i1) = lsat(i1,i2,i3,j).and.lsat(i1+1,i2,i3,j)
                    if ( ltemp(i1) ) then
                      grady(i1) =2.*(pmfr(i1+1,i2,i3,j,k)
     &                          - pmfr(i1,i2,i3,j,k))
     &                          / ( ddy(i1,i2,i3) + ddy(i1+1,i2,i3) )
                      dispy(i1,i2,i3) = dispy(i1,i2,i3)
     &                      + denml(i1+1,i2,i3,j) * ( por(i1+1,i2,i3)
     &                      * sat(i1+1,i2,i3,j)
     &                      * diffun(j,k) + coeyy(i1,i2,i3,j) )
     &                      * grady(i1)
                    endif
                  endif
                  . . .

                endif
                . . .

 840          continue
 850        continue

 900      continue
```

Figure 4: Code fragment in data parallel style with Fortran D data layout specifications

```
      double precision ddx(33, 8, 8), ddy(0:33, 8, 8), ddz(33, 8, 8)
      double precision por(33, 8, 8), pmfr(0:33, 8, 8, 4, 5)
      double precision diffun(4, 6), alphal(4), alphat(4)
      integer mdisp
      double precision sat(33, 8, 8, 4), denml(33, 8, 8, 4)
      logical lsat(0:33, 8, 8, 4)
      double precision potx(32, 8, 8, 4), poty(32, 8, 8, 4), potz(32,
     #8, 8, 4)
      double precision disp(32, 8, 8, 6), dispx(32, 8, 8), dispy(0:32,
     # 8, 8), dispz(32, 8, 8)
      double precision fluxx(32, 8, 8, 4), fluxy(32, 8, 8, 4), fluxz(3
     #2, 8, 8, 4)
      double precision coexx(32, 8, 8, 4), coexy(32, 8, 8, 4), coexz(3
     #2, 8, 8, 4), coeyx(32, 8, 8, 4), coeyy(32, 8, 8, 4), coeyz(32, 8,
     #8, 4), coezx(32, 8, 8, 4), coezy(32, 8, 8, 4), coezz(32, 8, 8, 4)
      double precision wksp1(32, 8, 8), wksp2(32, 8, 8), wksp3(32, 8,
     #8)
      double precision aleff(32, 8, 8, 4), ateff(32, 8, 8, 4)

      integer n$proc

C      --- Fortran D variable declarations ---
      common /FortD/ n$p, my$p, my$pid
      integer n$p, my$p, my$pid, numnodes, mynode, mypid
      logical l$buf(300)
      double precision dp$buf(1500)

C      --- Fortran D initializations ---
      n$p = numnodes()
      if (n$p .ne. 8) stop
      my$p = mynode()
      my$pid = mypid()
          .
          .
          .
C      --- Send por(1, 1:8, 1:8) ---
      if ( my$p .gt. 0) then
        call buf3D$dp(por, 1, m1, 1, 8, 1, 8, 1, 1, 1, 1, 8, 1, 1, 8,
     #1, dp$buf(1))
        call csend(115, dp$buf, 8 * 8, my$p - 1, my$pid)
      endif
C      --- Recv por(m1, 1:8, 1:8) ---
      if ( my$p .lt. maxproc) then
      call crecv(115, dp$buf, 8 * 8)
      call unbuf3D$dp(por, 1, m1, 1, 8, 1, 8, m1, m1, 1, 1, 8, 1, 1, 8
     #, 1, dp$buf(1))
      endif
C      --- Send denml(1, 1:8, 1:8, 2:4), sat(1, 1:8, 1:8, 2:4) ---
      if ( my$p .gt. 0) then
        call buf4D$dp(denml, 1, m1, 1, 8, 1, 8, 1, 4, 1, 1, 1, 1, 8, 1
     #, 1, 8, 1, 2, 4, 1, dp$buf(1))
        call buf4D$dp(sat, 1, m1, 1, 8, 1, 8, 1, 4, 1, 1, 1, 1, 8, 1,
     #1, 8, 1, 2, 4, 1, dp$buf(193))
        call csend(116, dp$buf, 384 * 8, my$p - 1, my$pid)
      endif
C      --- Recv denml(m1, 1:8, 1:8, 2:4), sat(m1, 1:8, 1:8, 2:4) ---
      if ( my$p .lt. maxproc) then
      call crecv(116, dp$buf, 384 * 8)
      call unbuf4D$dp(denml, 1, m1, 1, 8, 1, 8, 1, 4, m1, m1, 1, 1, 8,
     # 1, 1, 8, 1, 2, 4, 1, dp$buf(1))
      call unbuf4D$dp(sat, 1, m1, 1, 8, 1, 8, 1, 4, m1, m1, 1, 1, 8, 1
     #, 1, 8, 1, 2, 4, 1, dp$buf(193))
      endif
C      --- Send ddy(1, 1:8, 1:8) ---
      if ( my$p .gt. 0) then
        call buf3D$dp(ddy, 0, m1, 1, 8, 1, 8, 1, 1, 1, 1, 8, 1, 1, 8,
     #1, dp$buf(1))
        call csend(117, dp$buf, 64 * 8, my$p - 1, my$pid)
      endif
C      --- Recv ddy(m1, 1:8, 1:8) ---
      if ( my$p .lt. maxproc) then
      call crecv(117, dp$buf, 64 * 8)
      call unbuf3D$dp(ddy, 0, m1, 1, 8, 1, 8, m1, m1, 1, 1, 8, 1, 1, 8
     #, 1, dp$buf(1))
      endif
c
```

Figure 5: PART 1: Compiler generated, message passing Fortran 77 SPMD node program

```
C        —- Send pmfr(1, 1:8, 1:8, 2:4, 1:5) —-
        if ( my$p .gt. 0) then
          call buf5D$dp(pmfr, 0, m1, 1, 8, 1, 8, 1, 4, 1, 5, 1, 1, 1, 1,
     # 8, 1, 1, 8, 1, 2, 4, 1, 1, 5, 1, dp$buf(1))
          call csend(119, dp$buf, 8*8*3*5 * 8, my$p - 1, my$pid)
        endif
C        —- Recv pmfr(m1, 1:8, 1:8, 2:4, 1:5) —-
        if ( my$p .lt. maxproc) then
        call crecv(119, dp$buf, 8*8*3*5 * 8)
        call unbuf5D$dp(pmfr, 0, m1, 1, 8, 1, 8, 1, 4, 1, 5, m1, m1, 1,
     #1, 8, 1, 1, 8, 1, 2, 4, 1, 1, 5, 1, dp$buf(1))
        endif
C        —- Send lsat(1, 1:8, 1:8, 2:4) —-
        if ( my$p .gt. 0) then
          call buf4D$l(lsat, 0, m1, 1, 8, 1, 8, 1, 4, 1, 1, 1, 1, 8, 1,
     #1, 8, 1, 2, 4, 1, l$buf(1))
          call csend(122, l$buf, 8*8*3 * 4, my$p - 1, my$pid)
        endif
C        —-Recv lsat(m1, 1:8, 1:8, 2:4) —-
        if ( my$p .lt. maxproc) then
        call crecv(122, l$buf, 8*8*3 * 4)
        call unbuf4D$l(lsat, 0, m1, 1, 8, 1, 8, 1, 4, m1, m1, 1, 1, 8, 1
     #, 1, 8, 1, 2, 4, 1, l$buf(1))
        endif
c
        do 900 k = 1, 5
C
          do 850 j = 2, 4
            do 840 i3 = 1, 8
              do 840 i2 = 1, 8
                do 840 i1 = 1, 32
C
                  if (poty(i1, i2, i3, j) .lt. 0) then
C
                    if (my$p .lt. 7 .or. i1 .ne. 32) then
                      ltemp(i1) = lsat(i1, i2, i3, j) .and. lsat(i1 + 1,
     # i2, i3, j)
                      if (ltemp(i1)) then
                        grady(i1) = 2. * (pmfr(i1 + 1, i2, i3, j, k) - p
     #mfr(i1, i2, i3, j, k)) / (ddy(i1, i2, i3) + ddy(i1 + 1, i2, i3))
                        dispy(i1, i2, i3) = dispy(i1, i2, i3) + denml(i1
     #, i2, i3, j) * (por(i1, i2, i3) * sat(i1, i2, i3, j) * diffun(j, k
     #) + coeyy(i1, i2, i3, j)) * grady(i1)
                      endif
                    endif
C
                    .
                    .
                    .

                  else
C
                    .
                    .
                    .

                    if (my$p .lt. 7 .or. i1 .ne. 32) then
                      ltemp(i1) = lsat(i1, i2, i3, j) .and. lsat(i1 + 1,
     # i2, i3, j)
                      if (ltemp(i1)) then
                        grady(i1) = 2. * (pmfr(i1 + 1, i2, i3, j, k) - p
     #mfr(i1, i2, i3, j, k)) / (ddy(i1, i2, i3) + ddy(i1 + 1, i2, i3))
                        dispy(i1, i2, i3) = dispy(i1, i2, i3) + denml(i1
     # + 1, i2, i3, j) * (por(i1 + 1, i2, i3) * sat(i1 + 1, i2, i3, j) *
     # diffun(j, k) + coeyy(i1, i2, i3, j)) * grady(i1)
                      endif
                    endif
                    .
                    .
                    .

                  endif
                  .
                  .
                  .

 840        continue
 850      continue
            .
            .
            .

 900    continue
```

Figure 6: PART 2: Compiler generated, message passing Fortran 77 SPMD node program