PPPE Deliverable 4.3.a (Part 2 of 2)
Analysing various HPF Parallelisation Strategies

Organisation: GMD, St. Augustin, Germany
Contributor: Thomas Brandes
Deliverable: 4.3.a (Part 2 of 2)
Date: Aug 31, 1993

Automatic Parallelization/Partitioning: A Case Study on the AVL Fire Benchmark Codes

Thomas Brandes
German National Research Center for Computer Science,
P.O. Box 1316, 53731 Sankt Augustin, FRG
e-mail: brandes@gmd.de

Abstract

The goal of this study was to investigate how tools can actually help for the automatic parallelization and automatic partitioning of application codes that work on unstructured grids by using indirect addressing schemes. For this purpose the AVL Fire benchmark codes have been chosen where the iterative solvers work exactly on such structures. Parallelization of I/O has not been considered.

The results will show that the automatic parallelization of codes with indirect addressing is very simple if the code is written in the right way. But the efficiency of the parallel code is very dependent on the run-time system and on the opmization strategies of the underlying compiler. Furthermore, it will be important for the user that he can choose a domain decomposition though the compiler itself will only support data decompositions.

This work has been funded by the ESPRIT-6643 project PPPE (Portable Parallel Programming Environment).

1 Tools

For the study the automatic parallelization tool CMAX and the automatic partitioning tool ADAPTOR have been chosen. It should be mentioned that especially the combination of two such tools might be interesting for many application codes.

The reason for the above choice was especially the authors familiarity with these tools. But the essential results of this study might be independent of the considered tools.

In the following the functionality of these tools is described and presented for a simple example program.

1.1 Automatic Parallelization with CMAX

The CMAX Converter [Cor92] translates Fortran 77 programs into Connection Machine Fortran programs. So this tool converts sequential programs into data parallel programs. This is done by looking at the dependences of the serial loops and transforming them to array operations and forall loops if possible. Furthermore, the layout or distribution of all array variables is determined by using interprocedural analysis.

The CMAX tool has been derived from the FORGE tool of APR (Applied Parallel Research). It can greatly simplifiy the task of porting serial Fortran 77 programs to the massively parallel Connection Machines system. The converter was available in a Beta release.

Figure 1: Automatic Parallelization and Partitioning

The following code shows a serial Fortran 77 program.

```
      program simple
      real A(100), S
      integer I
      do I=1,100
        A(I) = I
      end do
      S = 0.0
      do I=1,100
        S = S + A(I)
      end do
      print *, 'Sum of A = ', S, ' (should be 1000.0) '
      end
```

The following program is generated by CMAX:

```
CMAX: [Version 0.8 Beta 2 (0.8.05 2/5/93)]
CMAX: ----------------------------------------------------
CMAX: Transformation of SIMPLE from simple.f
CMAX: ----------------------------------------------------
      program simple
      real A(100), S
      integer I
CMF$  LAYOUT a(:NEWS)
      FORALL (i = 1:100) a(i) = i
      s = 0.0
      s = s + sum(a)
      PRINT *, 'Sum of A = ',s,' (should be 1000.0) '
      END
```

1.2 Automatic Partitioning with Adaptor

Adaptor (Automatic Data Parallelism Translator) is a tool that transforms data parallel programs written in Fortran with array extensions, parallel loops, and layout directives to parallel programs with explicit message passing [Bra93]. The current version supports especially the translation of Connection Machine Fortran programs to message passing programs, but it can also deal with some kind of MIMD parallelism.

Adaptor is not a compiler but a source to source transformation that generates Fortran 77 host and node programs with subroutine calls to a special library. This library realizes message passing and global operations on distributed arrays. The new generated source codes can run on most parallel architectures, but also on a single node or on a single workstation. By this way it is possible to write data parallel programs that are portable for a wide range of parallel machines.

The following message passing program is generated by Adaptor from the data parallel CM Fortran program:

```
SUBROUTINE NODEMODULE ()
INTEGER*4 A_LOW, A_HIGH,
INTEGER*4 A_START, A_STOP, A_INC
INTEGER*4 A_DIM1
INTEGER*4 A_OFS
REAL*4 A (1:100)
REAL*4 S
INTEGER*4 I
REAL*4 S_TMP1
INTEGER*4 I_1
INTEGER*4 dalib_pid
EXTERNAL dalib_pid
call dalib_array_pardim (100,A_LOW,A_HIGH)
A_DIM1 = A_HIGH-A_LOW+1
A_OFS = 1-A_LOW
DO I=A_LOW,A_HIGH
   A(A_OFS+I) = I
END DO
S = 0.0
S_TMP1 = 0.0
DO I_1=A_LOW,A_HIGH
   S_TMP1 = S_TMP1+A(A_OFS+I_1)
END DO
call dalib_reduction (S_TMP1,8)
S = S+S_TMP1
IF (dalib_pid() .eq. 1) THEN
   PRINT *,'Sum of A = ',S,' (should be 1000.0) '
END IF
END
```

2 The FIRE Code

FIRE is a registered trade-mark of AVL LIST Gmbh.

2.1 Applications for FIRE

FIRE(TM) is a fully interactive fluid dynamics package, including powerful, menu-driven pre- and post-processing, developed specially for computing compressible and incompressible turbulent fluid flow as encountered in engeneering environments. 3D and 2D unsteady or steady simulations of flow and heat transfer within arbitrary complex geometries with moving or fixed boundaries can be performed. Additionally FIRE offers the capabilities to predict ignition, combustion, particle transport, spray propagation and evaporation, cavitating two-phase flows and flows in porous elements. FIRE incorporates restart and backup capabilities at any time step. The equations, namely the momentum, mass and energy conservation equations, the k-epsilon model equations and equations for a passive scalar and fuel vapour concentrations are solved according to the finite volume method for structured and unstructured meshes with an ICCG solver and at massive parallel hardware platforms with a Gauss-Seidel solver.

The range of applications is aerodynamics, air conditioning, biomechanical flows, cooling purposes, engine development, simulation of avalanches. Markets in which FIRE is used: aerospace, automotive, chemical, electrical/electronics, environmental safety, food, fuel, health care.

2.2 The Benchmark Codes

Three solvers are delivered for the benchmarks:

- gaus is a solver that uses Gauss-Seidel relaxation
- gccg uses orthomin, diagonal scaling
- xccg uses orthomin, modified lu preconditioning

For the first two solvers hand-coded message passing programs based on domain decomposition were also available. Both versions were realized by using the portable message passing interface PARMACS [Hem91]. The Gauss solver uses red/black relaxation.

3 Data Structures for Unstructured Grids

All solvers are iterative ones that work on unstructured grids 2. While structured grids are represented by two- or three-dimensional arrays, unstructured grids are represented by one-dimensional arrays to reduce the memory requirements. But then neighbourhood information must be represented in integer arrays that will be used for indirect addressing within all programs.

The dimension of the problem is described by the number of the cells `NNCELL`. For the solver itself the subset of the interior cells `NNINTC` is used. Values of the cells are stored in one-dimensional arrays like `DIREC1`.

```
INTEGER NNCELL     ! number of cells
INTEGER NNINTC     ! number of interior/active cells
```

Figure 2: Structured and Unstructured Grids (2D Case)

```
      PARAMETER (NNCELL = 65000, NNINTC = 50000)

      DOUBLE PRECISION DIREC1(NNCELL)
      DOUBLE PRECISION DIREC2(NNINTC)
```

The common way of addressing the 6 main directions (in 3D) is by naming them North, East, South, West, High and Low. It is illustrated by the following example:

The neighbourhood information of the cells is contained in the integer array `LCC`.

```
      INTEGER LCC (1:6,1:NNINTC)   ! neighbour cells of a cell

   the south neighbour of cell 12 is LCC(1,12) = 20
   the east  neighbour of cell 12 is LCC(2,12) = 13
   the north neighbour of cell 12 is LCC(3,12) =  4
   the west  neighbour of cell 12 is LCC(4,12) = 11
   the upper neighbour of cell 12 is LCC(5,12) = 194
   the lower neighbour of cell 12 is LCC(6,12) = 13884
```

The reason why it is called indirect addressing is, that in order to get the coefficient of the east neighbour, you first have to retrieve the array index from the `LCC` array and then access this element in other arrays.

The following statement computes a new value for cell `NC` depending on older values of its six neighboured cells.

```
      DIREC2 (NC) = f (DIREC1(LCC(1,NC)), DIREC1(LCC(2,NC)),
     +                 DIREC1(LCC(3,NC)), DIREC1(LCC(4,NC)),
     +                 DIREC1(LCC(5,NC)), DIREC1(LCC(6,NC)))
```

The communication pattern for irregular problems with unstructured grids depends on the input data.

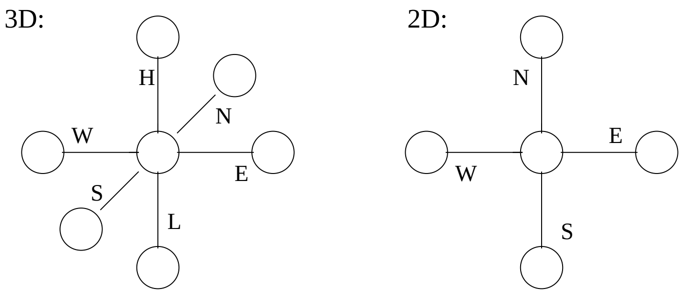

Figure 3: Neighbourhood in 2D and 3D case

4 Partitioning of Data Structures

For the distributed memory parallelization of the solvers the grid will be decomposed and distributed among the available nodes.

Data decomposition means that you take the arrays and split them according to the number of processors onto the available nodes, e.g. if you have 2 processors, one would give processor 1 indices 1-16 and processor 2 indices 17-32.

Domain decomposition on the other hand should take the underlying geometry into account when splitting up the data between the processors. By this way, one gets smaller boundaries between the processors (see figure 4).

Therefore domain decomposition is a numbering of the cells in such a clever way, that by pure (and easy) data decomposition you can get (nearly) optimal small boundaries between processors. In other words, domain decomposition is a way of reducing the necessary communication to a minimum.

When using the layout directives of High Performance Fortran [Hig93] the distribution can be specified in the following way:

```
      INTEGER NNCELL, NNINTC
      PARAMETER (NNCELL = 65000, NNINTC = 50000)

      DOUBLE PRECISION DIREC2(NNINTC)
      INTEGER LCC (1:6,1:NNINTC)

!HPF$ TEMPLATE CELLS (1:NNINTC)
!HPF$ DISTRIBUTE CELLS (BLOCK)
!HPF$ ALIGN DIREC2 (I) WITH CELLS(I)
```

Figure 4: Data Decomposition vs. Domain Decomposition

```
!HPF$ ALIGN LCC (*,I) WITH CELLS(I)
```

The alignment directives guarantee that the neighbourhood information of one cell is always available on the same processor where the value of the cell is contained. The value of a neighboured cell will only be on the same processor if `LCC(j,NC)` is a cell that is on the same processor as the cell `NC` itself.

5 Indirect Addressing

Due to the importance of the indirect addresing it is discussed in this section a little bit more in detail.

5.1 Data Parallel Loop with Indirect Addressing

This is a typical loop in the codes (here gccg) where indirect addressing is used. All iterations are independent of each other as old values from `DIREC1` are used to compute new ones in `DIREC2`.

```
      DOUBLE PRECISION DIREC1(NNCELL), DIREC2(NNINTC)

      DO  NC = 1, N
        DIREC2(NC)=BP(NC)*DIREC1(NC)
     X           -BS(NC)*DIREC1(LCC(1,NC))
     X           -BW(NC)*DIREC1(LCC(4,NC))
     X           -BL(NC)*DIREC1(LCC(5,NC))
```

```
     X           -BN(NC)*DIREC1(LCC(3,NC))
     X           -BE(NC)*DIREC1(LCC(2,NC))
     X           -BH(NC)*DIREC1(LCC(6,NC))
      END DO
```

The arrays DIREC1, LCC and DIREC2 are all distributed. DIREC2(I) and LCC(*,I) are aligned. Due to the indirect addressing of DIREC1 the processors will use data from other processors. Therefore communication becomes necessary.

5.2 Replication

The idea of replication is that all processors will exchange their values of the whole array DIREC1. Of course it might happen that a node gets many values that it does not really need. This is especially the case when using many processors.

```
      DIMENSION DIREC1REP (NNCELL)
!hpf$ distribute DIREC1REP (*)
      ...
      DIREC1REP = DIREC1       ! every processor has all values
      DO NC= 1, NNINTC
        DIREC2(NC)=BP(NC)*DIREC1REP(NC)
     X           -BS(NC)*DIREC1REP(LCC(1,NC))
     X           -BW(NC)*DIREC1REP(LCC(4,NC))
     X           -BL(NC)*DIREC1REP(LCC(5,NC))
     X           -BN(NC)*DIREC1REP(LCC(3,NC))
     X           -BE(NC)*DIREC1REP(LCC(2,NC))
     X           -BH(NC)*DIREC1REP(LCC(6,NC))
      END DO
```

The advantage of this approach is that the replication of the distributed array has regular communication patterns that can be realized efficiently. Furthermore, this approach will also work well if no domain decomposition has been taken into account.

But as more communication is needed, the replication of the array is an operation that does not scale with the number of processors. Also the need of memory for the full array DIREC1REP on every processor will become a problem for bigger input data sets.

5.3 Scheduling

If the array DIREC1 is distributed, the access to DIREC1(LCC(k,NC)) will require communication if LCC(k,NC) does not refer to a cell that is on the same processor.

For the realization of such loops in High Performance Fortran compilers a typical strategy will be to split up the data parallel loop in a global get operation and a local loop. By this way all non-local data will become local before the iterations of the loop are executed.

```
      REAL DIREC1V (6,NNINTC)
!HPF$ ALIGN DIREC1V(*,NC) WITH CELLS (NC)
      ...
      DO NC = 1, NNINTC
```

```
      DO I = 1, 6
        DIREC1V (I,NC) = DIREC1(LCC(I,NC))
      END DO
    END DO
    DO NC= 1, NNINTC
      DIREC2(NC)=BP(NC)*DIREC1V(NC)
   X            -BS(NC)*DIREC1V(1,NC) - BW(NC)*DIREC1V(4,NC)
   X            -BL(NC)*DIREC1V(5,NC) - BN(NC)*DIREC1V(3,NC)
   X            -BE(NC)*DIREC1V(2,NC) - BH(NC)*DIREC1V(6,NC)
    END DO
```

Only the first loop nesting will require communication, the second loop is a local one where the iterations can be executed without any communication.

The first loop is often called a global get operation. Such an operation can be optimized for multiprocessor machines with distributed memory.

```
    CALL GLOBAL_GET (DIREC1V, DIREC1, LCC)
```

The global get operation has two phase:

- In the first phase every processor determines the values that it needs from other processors. Then it asks these processors for these values by sending them a corresponding message with index values.
- In the second phase every processor sends other processors the values which was asked for.

By this way the information to be transmitted is collected into relatively large messages. This avoids the hgih startup cots for communication.

The first phase has not to be repeated if the global get is done again with the same communication pattern, this means, the `LCC` array has not been changed. This is the case for the solvers of the AVL benchmark codes.

Most work on implementing such operations for irregular problems and irregular grids has been done in the PARTI project (Parallel Autmated Runtime Toolkit at ICASE) [SCMB90, BSS91]. They have realized primitives for the retrieval of globally indexed but irregularly distributed data-sets.

The runtime system *DALIB* of Adaptor provides also subroutines that realize the get operation for distributed arrays. In contrary to the PARTI primitives no hash tables are used, so it might be possible that not only a single copy of each off-processor datum is fetched.

5.4 The Solution with Overlapping Areas

The solution discussed before will require a temporary array (`DIREC1V`). This array has six entries for every cell and is distributed in the same way as the cells.

When this array is computed most values will be local ones if a good domain decomposition is given. Therefore many local data movement is executed that would not be necessary.

Therefore a better solution is using an overlapping area for the array `DIREC1`. This technique is used in the hand-coded message passing programs.

Figure 5: Domain Decomposition with Overlap Area

```
      DOUBLE PRECISION DIREC1(MY_NNCELL), DIREC2(MY_NNINTC)
      INTEGER LCC (6, MY_NNINTC)
```

The entries of `DIREC1(1:MY_NNINTC)` contain the local values. But the entries `DIREC1(MY_NNINTC+1:MY_NCELL)` will hold the values of neighboured processors. These entries build the overlap area.

If `LCC(k,NC)` is a non-local index, it will be replaced with an index that points in the overlap area. So every off-processor reference is an on-processor reference in the overlap area. By this way it is possible that on every node nearly the same code as in the original loop is executed but restricted to the local cells.

```
      DOUBLE PRECISION DIREC1(MY_NNCELL), DIREC2(MY_NNINTC)
      INTEGER LCC (6, MY_NNINTC)
      ....
      DO  NC = 1, MY_NNINTC
        DIREC2(NC)=BP(NC)*DIREC1(NC)
     X           -BS(NC)*DIREC1(LCC(1,NC))
     X           -BW(NC)*DIREC1(LCC(4,NC))
     X           -BL(NC)*DIREC1(LCC(5,NC))
     X           -BN(NC)*DIREC1(LCC(3,NC))
     X           -BE(NC)*DIREC1(LCC(2,NC))
     X           -BH(NC)*DIREC1(LCC(6,NC))
      END DO
```

After every iteration an update of the overlap area is required. Therefore translation tables are necessary that contain the global index information. The computation of such tables is rather complicated.

Figure 6: Red-Black Ordering

6 Red-Black Relaxation

Red-Black ordering is a way of separating the grid into two halfs, a red and a black one with the additional constraint that a red cell is only surrounded by black cells in its non diagonal directions and vice versa.

One solves the linear equation system first on the red cell subset by using the (old) values of the black cells and then computes the solution on the black cells by using the (new) values of the red cells. These steps are repeated until the normalized residual sum is reduced to a certain level. The normalisation quantity is the residual sum at iteration zero. Within the red and black cell subsets the order of the cells being computed is arbitrary, because those cells are totally decoupled.

In order to access the cells in the correct order, another array is introduced, NBOARD. On it first the indices of the red cells and afterwards the indices of all black cells are stored. For a typical test-case it looks like

```
      INTEGER NBOARD (NNCELLS)

NBOARD: 1,3,5,7,10,12,14,16,17,19,21,23,26,28,30,32,    (red cells)
        2,4,6,8, 9,11,13,15,18,20,22,24,25,27,29,31     (black cells)
```

One might call this double indirect addressing, because you have to retrieve the index into `LCC` from `NBOARD` and then to use this index for other arrays.

Red-Black relaxation is required for the parallelization of the Gauss Solver (see section 11).

7 Input Data Sets and Results

Two input files (tjunc.dat, drall.dat) were available.

```
                      tjunc.dat       drall.dat
-------------------------------------------------
NNCELL (no of cells)     19061          60394
NNINTC (interior cells)  13845          47312
```

All solvers will need a definite number of iterations that is required until the ratio of the new residual and the reference residual is smaller than `1.0d-10`. This number depends on the solver and on the input data set.

After all iterations the results of all cells are sorted and the ten minimal and ten maximal values are printed. These values must always be the same when using different machines or different parallelization strategies. This output can be used to prove correctness of the implemented algorithms.

For the timing only the iterations of the solvers are measured, not the sorting and the I/O.

These are the results for the first data set tjunc:

```
Iterations for gaus  : 678
Iterations for gccg  : 338
Iterations for xccg  :  51

Addresses Solution (Minima)      Addresses Solution (Maxima)
===========================      ===========================

13667  -3782.692365980521         5889   1824.507667963191
13668  -3781.468254774083         5928   1823.323989883303
13666  -3780.493189325922         6396   1823.051893314409
13669  -3777.207897749402         5382   1822.909292548036
13681  -3774.960307045700         5967   1822.338629704606
13665  -3774.506815005687         6435   1822.135006768962
13682  -3773.497114804752         5421   1822.018688257849
13680  -3773.152138812593         6006   1821.269939369986
13670  -3770.369967974590         6474   1821.265901696900
13683  -3769.183153198038         5460   1821.176816753117
```

These are the results for the second data set 'drall':

```
Iterations for gaus  : 453
Iterations for gccg  : 335
Iterations for xccg  :  56

Addresses Solution (Minima)      Addresses Solution (Maxima)
===========================      ===========================

19500  -893.7184656811423        5863   295.8683079933826
19501  -869.9497745017825        6391   295.8486488543076
19502  -805.4362126107151        6919   295.5001635119620
21224  -721.7879305103476        5864   295.4904414812605
19410  -718.4597659554876        5335   295.4619966124145
19536  -696.5315580631791        6392   295.4422100208253
19503  -690.8569483790839        5336   295.1291511964123
21225  -688.5407097807794        6920   295.0117977033000
19535  -685.9272791610492        7447   294.8441422510693
19381  -681.0842776776066        4807   294.6234953064007
```

The following results were already available for the sequential versions before this study started:

Benchmark 1 : TJUNC (13845 cells)

CPU [sec]	GAUS	GCCG	XCCG	XVCG	XOCG	I/O
IBM RS6000/580	7.9	5.7	1.2	3.3	0.9	0.9
IBM RS6000/560	10.5	6.9	1.5	4.0	1.2	1.1
IBM RS6000/350	15.6	10.6	2.3	5.4	1.7	1.2
DEC ALF 3000-500	20.0	12.5	2.7	3.4	2.1	0.6
HP 735	22.4	14.7	3.0	na	na	0.5
SGI CRIMSON	31.2	22.1	4.4	7.0	3.6	0.6
CRAY YMP-EL	15.2	4.0	3.8	1.7	4.5	0.07

Benchmark 2 : DRALL (47312 cells)

CPU [sec]	GAUS	GCCG	XCCG	XVCG	XOCG	I/O
IBM RS6000/580	19.3	19.2	4.5	17.3	3.4	3.0
IBM RS6000/560	24.3	24.2	5.6	20.8	4.5	4.0
IBM RS6000/350	36.9	36.6	8.6	15.8	6.7	3.9
DEC ALF 3000-500	49.2	46.7	11.3	19.5	9.0	2.0
HP 735	57.3	55.2	12.7	na	na	1.7
SGI CRIMSON	82.9	83.7	18.5	40.4	15.9	2.1
CRAY YMP-EL	34.6	13.8	14.2	6.3	16.9	0.07

Symbols:

```
na  : not available
GAUS: point iterative gauss-seidel solver with overrelaxation and
      red/black ordering
GCCG: Orthomin with diagonal scaling
XCCG: Orthomin with modified ILU preconditioning
XVCG: fully vectorized variant of XCCG - hyperplane ordering
XOCG: cache optimized variant of XCCG - offset ordering
```

Comments: A Cray optimized version of GAUS needs 7.5 seconds for TJUNC and 19.6 seconds for DRALL on the YMP-EL. Similar optimizations for IBM gave 5.8 seconds for TJUNC and 15.4 seconds for DRALL on the RS 6000/580.

Here are the specifications of the machines:

Computer	Memory [MByte]	Clock [MHz]	Cache [kB] Data/Instr.	Compiler Options
IBM 6000/580	256	62	64/32	xlf 2.3 -O
IBM 6000/560	512	50	64/32	xlf 2.3 -O
IBM 6000/350	192	50	32/32	xlf 2.3 -O
DEC ALF 3000-500	256	150	8/8	f77 3.3 -O
HP 735	64	99	256/128	
IRIS CRIMSON	64	-	-/-	f77 3.4.1 -mips2
CRAY YMP-EL	256	33	-	cf77 5.0 -dp

On the Alliant FX/2800 one gets the following times for one node:

tjunc	iterations	time	MFlops
gaus	678	88.5 s	1.8
gccg	338	59.4 s	2.4
xccg	51	10.2 s	2.1

drall	iterations	time	MFlops
gaus	453	257.7 s	1.4
gccg	335	230.6 s	2.1
xccg	56	53.6 s	1.5

8 Shared Memory Parallelization

Every code that should run on a distributed memory machine must also run on a shared memory parallel machine. A code that cannot be parallelized for a shared memory architecture will not run on a distributed memory architecture.

Due to the shared memory the indirect addressing makes no problems on this architecture. The following results have been verified on an Alliant FX/2816, a 16 processor machine with 128 MByte shared memory. The autoparallelizing Fortran compiler was used (fxfortran 1.4.0 on concentrix 2.2.00).

8.1 GCCG

There is no problem for the automatic parallelization of this code. A special subroutine for the dotproduct was inserted.

These are the results for the Alliant FX/2800:

tjunc (13845 cells)	time	MFlops
Sequential Version	59.4 s	2.4
Parallel Version, 1 node	64.7 s	2.2
Parallel Version, 2 nodes	37.9 s	3.8
Parallel Version, 4 nodes	22.4 s	6.5
Parallel Version, 8 nodes	13.1 s	11.1

drall (47300 cells)	time	MFlops
Sequential Version	230.6 s	2.1
Parallel Version, 1 node	261.0 s	1.9
Parallel Version, 2 nodes	146.9 s	3.3
Parallel Version, 4 nodes	85.7 s	5.7
Parallel Version, 8 nodes	51.4 s	9.6

8.2 GAUS

In the Gaus algorithm the following loop was the loop that must be parallelized:

```
      DO 1 KI=NINTCI,NINTCF
        NC=NBOARD(KI)
        RES=BS(NC)*VAR(LCC(1,NC))+BL(NC)*VAR(LCC(5,NC))
     x     +BE(NC)*VAR(LCC(2,NC))+BH(NC)*VAR(LCC(6,NC))
     x     +BN(NC)*VAR(LCC(3,NC))+BW(NC)*VAR(LCC(4,NC))
     x     -VAR(NC)+SU(NC)
        VAR(NC)=VAR(NC)+1.7*RES
        RESNEW=RESNEW+RES*RES
    1 CONTINUE
```

Due to the variable VAR there might be dependences between the iterations of the loop, the Gaus-Seide algorithm cannot be parallelized.

Therefore a red-black colouring is required and the one loop will be splitted up into two loops (red-black relaxation). Therefore it is useful to define a mask. NBOARD is a renumbering of the cells and therefore just a permutation vector. The first positions are red cells the latter ones are black cells.

```
      NBR = (NINTCF + 1)/2
      DO I = 1, NBR
         RED_POINTS (NBOARD(I)) = .true.
      END DO
      DO I = NBR+1, NINTCF
         RED_POINTS (NBOARD(I)) = .false.
      END DO
```

Afterwards one can split the original loop in the following way:

```
cvd$  select (concurrent)
cvd$  ivdep
      DO KI=1,NINTCF
        if (RED_POINTS(KI)) then
        RESVEC(KI) = BS(KI)*VAR(LCC(1,KI))+BL(KI)*VAR(LCC(5,KI))
     x     +BE(KI)*VAR(LCC(2,KI))+BH(KI)*VAR(LCC(6,KI))
     x     +BN(KI)*VAR(LCC(3,KI))+BW(KI)*VAR(LCC(4,KI))
     x     -VAR(KI)+SU(KI)
        VAR(KI)=VAR(KI)+1.7*RESVEC(KI)
        end if
      END DO

cvd$  select (concurrent)
cvd$  ivdep
      DO KI=1,NINTCF
        if (.not. RED_POINTS(KI)) then
        ...
        END
      END DO
```

The iterations within every loop are now independent of each other, they can be executed in parallel.

The compiler must be told that now there are no dependences. This is done by the compiler directive *ivdep*. This directive causes the compiler to ignore vector dependences for the next loop.

The parallelization gave the following results.

tjunc (13845 cells)	time	MFlops
Sequential Version	88.5 s	1.8
Parallel Version, 1 node	115.4 s	1.4
Parallel Version, 2 nodes	63.0 s	2.5
Parallel Version, 4 nodes	38.9 s	4.1
Parallel Version, 8 nodes	20.1 s	7.9

drall (47300 cells)	time	MFlops
Sequential Version	257.7 s	1.4
Parallel Version, 1 node	337.1 s	1.1
Parallel Version, 2 nodes	186.2 s	2.0
Parallel Version, 4 nodes	102.0 s	3.6
Parallel Version, 8 nodes	60.2 s	6.1

8.3 XCCG

The following two loops must be executed sequentially as the values computed in one iteration can be used in one of the next iterations.

```
      DO 10 NC=NINTCI+1,NINTCF
        DIREC1(NC)=RESVEC(NC)
     X            +PDUM00(NC)*DIREC1(LCC(1,NC))
     X            +PDUM01(NC)*DIREC1(LCC(4,NC))
     X            +PDUM02(NC)*DIREC1(LCC(5,NC))
   10 CONTINUE

      DO 11 NC=NINTCF-1,NINTCI,-1
        DIREC2(NC)=BN(NC)*DIREC1(LCC(3,NC))
     X            +BE(NC)*DIREC1(LCC(2,NC))
     X            +BH(NC)*DIREC1(LCC(6,NC))
        DIREC1(NC)=(DIREC1(NC)+DIREC2(NC))*CGUP(NC)
   11 CONTINUE
```

A possible strategy for the parallelization might be to find hyperplanes. These are sets of points which have no dependences within one plane. Due to the high overhead and the fine-grained parallelism this optimization has not been considered. Furthermore, such a transformation cannot be done automatically.

Therefore no further parallelization of this solver has been considered.

9 SIMD Parallelization of GCCG

In this section only the SIMD parallelization of the GCCG solver is discussed. The parallelization of the GAUSS solver is very similiar and is discussed in section 11. The parallelization of the XCCG solver is not possible.

For the SIMD parallelization the loops have to be transformed to array operations. This has been done with the help of the tool CMAX (see section 1.1). For better results it is necessary to make some hand optimizations.

9.1 Using CMAX for the GCCG solver

```
(in)  cmax avl -addfiles= *.f
(out) cmax <avl> [Version 1.0 (6/22/93)]
(out) Creating new package
(out) Adding fortran source file(s) to package <avl>
(out)     gccg.f
(out)     xread.f
(out)     xtime.f
(out)     xwrite.f
(out) CMAX execution complete.

(in)  cmax avl -addfiles= *.f
(out) cmax <avl> [Version 1.0 (6/22/93)]
(out) Translating package <avl>
(out) CMAX execution complete.
```

CMAX did a quite good job on this code and translated every serial loop to array or forall statements, except one. This one was a reduction where the reduction variable was a scalar of a parallel array. After introducting a temporary scalar this loop became also a dotproduct reduction.

```
c    this reduction loop is not recognised by CMAX
     do i = nintci, nintcf
        cnorm(nor) = cnorm(nor) + a(i) * b(i)
     end do
c    this will work
     do i = nintci, nintcf
        cnorm1 = cnorm1 + a(i) * b(i)
     end do
```

In some cases CMAX made a parallel array of a small array that should be better a serial one. This can be enforced by using layout directives before applying CMAX.

It should be mentioned that there were some run-time erros on the CM5 that disappeared after replacing the variable `nintci` with the value 1.

Here are the results of the code on a CM-5 with 32 or 64 nodes after using CMAX. The CM Fortran Compiler (CM5 VecUnit 2.1, Beta 0.1) was used, optimization was switched on compiler flags `'-O -vu'`.

tjunc (13845 cells)	time	MFlops
CM-5/32VU	27.3 s	5.32
CM-5/64VU	17.7 s	8.19

This very poor result can be improved by further hand optimizations. With the compiler flag $-list$ one gets information about statements that require communication and should be optimized. Indeed the first version produced by CMAX did require about 16 send and 6 get operations. These communication routines are very expensive and should be avoided.

9.2 Optimizations of the SIMD Code

The most send operations were due to the fact that some arrays are dimensioned for the number of all cells and some only for the number of interior cells. Operations between these arrays will not be aligned.

```
      DIMENSION resvec (nncell)
      DIMENSION su (nnintc)
      ...
c     statement will require send communication
      resvec(:nintcf) = su(:nintcf)
```

After introducing a new dimension for the array `resvec` the arrays are aligned and the assignment becomes a local operation.

```
      DIMENSION resvec (nnintc)
      DIMENSION su (nnintc)
      ...
c     statement will require no communication
      resvec(:nintcf) = su(:nintcf)
```

In the same way some other arrays can be dimensioned that most send and get operations will disappear.

Another optimization is to reduce the number of get operations in the parallel loop with indirect addressing. The original version will require six get operations.

```
      FORALL (nc = 1:nintcf) direc2(nc) = bp(nc) * direc1(nc) - bs(nc)
     &    * direc1(lcc(1,nc)) - bw(nc) * direc1(lcc(4,nc)) - bl(nc)
     &    * direc1(lcc(5,nc)) - bn(nc) * direc1(lcc(3,nc)) - be(nc)
     &    * direc1(lcc(2,nc)) - bh(nc) * direc1(lcc(6,nc))
```

The get operations can be reduced to one get operation when the loop is splitted up in the following way (see section 5.3):

```
      DIMENSION direc1v (6, NNINTC)
      INTEGER lcc (6, NNINTC)
CMF$  LAYOUT direc1v (:serial,:news)
CMF$  LAYOUT lcc(:SERIAL,:NEWS)
      ...
C-----
      direc1v = 0.0

      FORALL (j=1:6, nc = 1:nintcf, lcc(j,nc) .le. nintcf)
     &    direc1v (j, nc) = direc1 (lcc(j,nc))
C-----
      FORALL (nc = 1:nintcf) direc2(nc) = bp(nc) * direc1(nc) - bs(nc)
     &    * direc1v(1,nc) - bw(nc) * direc1v(4,nc) - bl(nc)
     &    * direc1v(5,nc) - bn(nc) * direc1v(3,nc) - be(nc)
     &    * direc1v(2,nc) - bh(nc) * direc1v(6,nc)
C-----
```

After all these optimizations there was only one global get operation for the forall loop. All other loops became local ones. Now one gets the following results:

tjunc (13845 cells)	time	MFlops
CM-5/32VU, opt	8.43 s	17.20
CM-5/64VU, opt	7.05 s	20.57

It might be possible that the results can be improved when fixing the communication patterns by using some special routines of the CMSSL (Connection Machine Scientific Software Library). This has not been tested.

Nevertheless it is very disappointing to get only about 20 MFlops from a machine that has a peak performance of 8000 MFlops.

10 MIMD Partitioning with Adaptor

In this section only the automatic partitioning of the GCCG solver is discussed.

For translation with Adaptor the hand-optimized CM Fortran code of the GCCG algorithm was taken. Three versions have been considered as it was discussed in section 5.

- **replicate** is a version that uses a replicated array

```
      DIMENSION DIREC1REP (NNCELL)
      DIMENSION DIREC1    (NNINTC)

!HPF$ DISTRIBUTE DIREC1    (BLOCK)      ! distributed
!HPF$ DISTRIBUTE DIREC1REP (*)          ! replicated
      ...
      DIREC1REP = DIREC1

      DIREC2(1:NNINTC)=BP(1:NNINTC)*DIREC1REP(1:NNINTC)
     X                -BS(1:NNINTC)*DIREC1REP(LCC(1,1:NNINTC))
     X                -BW(1:NNINTC)*DIREC1REP(LCC(4,1:NNINTC))
     X                -BL(1:NNINTC)*DIREC1REP(LCC(5,1:NNINTC))
     X                -BN(1:NNINTC)*DIREC1REP(LCC(3,1:NNINTC))
     X                -BE(1:NNINTC)*DIREC1REP(LCC(2,1:NNINTC))
     X                -BH(1:NNINTC)*DIREC1REP(LCC(6,NC))
```

- **global get** is a version that realizes the required communication for indirect addressing with distributed arrays.

```
      DIMENSION DIREC1V (6, NNINTC)
      LOGICAL   LCCMASK (6,NNINTC)
      INTEGER   LCC     (6, NNINTC)
CMF$  LAYOUT LCCMASK (:serial,:news)
CMF$  LAYOUT DIREC1V (:serial,:news)
CMF$  LAYOUT LCC     (:SERIAL,:NEWS)
      ...
C-----
      direc1v = 0.0
```

```
        mask = (lcc .le. nintcf) .and. (lcc. ge. 1))

        call global_get (DIREC1V, DIREC1, LCC, LCCMASK)

        DIREC2(1:NINTCF)=BP(1:NINTCF)*DIREC1(1:NINTCF)
     X          -BS(1:NINTCF)*DIREC1V(1,1:NINTCF)
     X          -BW(1:NINTCF)*DIREC1V(4,1:NINTCF)
     X          -BL(1:NINTCF)*DIREC1V(5,1:NINTCF)
     X          -BN(1:NINTCF)*DIREC1V(3,1:NINTCF)
     X          -BE(1:NINTCF)*DIREC1V(2,1:NINTCF)
     X          -BH(1:NINTCF)*DIREC1V(6,1:NINTCF)
C-----
```

- **fixed schedule** is the same version as global get, but the communication pattern used for the indirect addressing is computed only once. This is done by moving the subroutine call that computes the communication pattern outside of the iteration loop.

10.1 Single Processor Version

Adaptor can also generate a Fortran 77 version with message passing and a Fortran 77 version running on a single node.

When comparing the original sequential Fortran 77 program with the Fortran 77 program generated by Adaptor from the CM Fortran version, one can identify the following main differences:

- some arrays are now dimensioned for the number of interior cells and not for the number of all cells
- there is the new temporary array `DIRECV1(6,NNINTC)` that will contain all the values of the neighboured cells
- the update loop is splitted in two loops: in the first loop the values of all

The version with the replicated arrays has some overhead due to the copying of the data. The execution time increases from 59 to 70 seconds.

In the other version the global get operations requires much more overhead that increases the execution time from 59 s to 94 seconds.

The single processor version should not be mistaken for the parallel version running on a single node. This version has communication statements and only at run-time it can be seen that there is only one node.

10.2 Results on the Alliant

Though the Alliant has a shared memory and not a distributed memory, the generated programs of Adaptor can also run on this machine by emulating the message passing between processes by a shared memory segment.

tjunc (13845 cells)	1	2	4	8
replicate	65 s	55 s	55 s	na
global get	334 s	181 s	114 s	62 s
fixed schedule	167 s	93 s	59 s	35 s

The sequential code needs 57.7 s on the Alliant FX/2800, this figure increases a little bit for replicate due to the copy of the array. The gather operation itself has a very high overhead that can be reduced when reusing the communication pattern. But the gather operation itself scales while the replication does not.

10.3 Results on the CM-5

All of the following results are on a CM-5 without using the vector units as these units cannot be used with Adaptor in its current version.

These are the results on a CM-5 using the CMMD message passing library. The first results have been achieved by using the replicate function of the DALIB, the second results are due to the CMMD subroutine CMMD_concat_with_nodes. Though the latter one is far more efficient the replication does not scale.

Code	1	4	8	16	32	64
replicate	85 s	57 s	62 s	s	109 s	147 s
concat		–	–	–	45 s	44 s

The following results are given when using the global gather operation.

Code	4	8	16	32	48	64
global get	149 s	77 s	42 s	25 s	29 s	31 s
fix schedule	71 s	37 s	25 s	16 s	14 s	11 s

About 32 processors there is no more speed-up. This situation might be better when using bigger input data sets, e.g. drall.

It should be mentioned that some more optimizations are possible.

10.4 Comparison with the hand-coded message passing program

As already mentioned there was a hand-coded message passing version of the gccg algorithm available.

When comparing this program with the generated version by Adaptor one can idenitify that the hand-codes version uses overlap areas and communication patterns that will be reused for all the itertions of the solver.

Due to the overlapping many memory overhead and local copy of data can be avoided. This results in runtimes that are only about 30 to 40 the Adaptor version.

11 Parallelization and Partitioning of the GAUSS solver

11.1 Parallelization with CMAX

There is no chance for CMAX when using the sequential algorithm, therefore the red-black relaxation algorithm was taken.

But CMAX must be told in a similiar way as it is done for the Alliant that there are no dependences.

11.2 Partitioning with Adaptor

```
      LOGICAL red_cell (NNINTC)
      DIMENSION VARIND (6,NNINTC)
      LOGICAL LCCMASK (6,NNINTC)
      ...
      LCCMASK = (LCC .le. NINTCF)
      red_cell = .false.
      IRB = (NINTCF+1) / 2
      RED_CELL (NBOARD(1:IRB)) = .true.

      call global_get (VARIND, VAR, LCC, LCCMASK .AND. .NOT. RED_CELL)
      WHERE (RED_CELL)
        DIREC1(1:NINTCF) =  BS(1:NINTCF)*VARIND(1,1:NINTCF)
     x                    + BL(1:NINTCF)*VARIND(5,1:NINTCF)
     x                    + BE(1:NINTCF)*VARIND(2,1:NINTCF)
     x                    + BH(1:NINTCF)*VARIND(6,1:NINTCF)
     x                    + BN(1:NINTCF)*VARIND(3,1:NINTCF)
     x                    + BW(1:NINTCF)*VARIND(4,1:NINTCF)
     x                    - VAR(1:NINTCF) + SU(1:NINTCF)
        VAR(1:NINTCF)=VAR(1:NINTCF)+1.7*DIREC1(1:NINTCF)
      END WHERE

      call global_get (VARIND, VAR, LCC, LCCMASK .AND. RED_CELL)
      WHERE (.NOT. RED_CELL)
        DIREC1(1:NINTCF) =  ...
        VAR(1:NINTCF)=VAR(1:NINTCF)+1.7*DIREC1(1:NINTCF)
      END WHERE
```

The following results were obtained on the Alliant FX/2800 (always 678 iterations) for the tjunc input data set:

tjunc (13845 cells)	1	2	4	8
replicate	142 s	136 s	160 s	na
global get	-	665 s	389 s	62 s
fixed schedule	-	-	-	-

- the original sequential version needs 89.77 s (1.78 MFlops)
- the sequential red-black gaus needs 97.0 s (1.65 MFlops), overhead is caused by dividing all iterations in two iteration sets depending on a mask.

- after splitting a own phase for local get realized via Fortran 77 and additional masking gives about 225 s.
- compiling for one node with Adaptor results in a program that needs 428.5 s (overhead by dalib_local_get operation).

12 Summary

In many scientific application codes the underlying geometries are represented by unstructured grids and the given algorithms use indirect addressing schemes.

If data parallel operations on these grids are coded in the right way they are no problem for the automatic parallelization. This means that the compiler has to see that there are no dependences in the data parallel loops.

For the partitioning of the data structures that represent the data of the unstructured grid, the user has to know how the data will be distributed or how the data decomposition is done. This is necessary to make sure that the data decomposition goes conform with a domain decomposition and only communication on the domain boundaries will be required. This is absolutely necessary to have efficient communication patterns.

When an indirect addressing scheme is used, this scheme will usually require communication by a global get operation. For such an operation a communication pattern or communication schedule has to be computed before the actual data is sent and received. This computation is very complicated if implemented by hand, but will be part of most runtime systems for data parallel compilers. When reusing a scheme it is important that the previously computed communication schedule has been saved and can be used again.

When an unstructured grid has been partitioned by a domain decomposition it is useful to have an overlap area. This overlap area will not reduce the amount of communication between the processors but will reduce the copy of local data. This technique should become part of future compilers for data parallel languages.

When the structure of the grid itself changes during the runtime the communication schedule must be recomputed in any case. This was not the case for the AVL benchmark codes, but might be the case in other application codes. There might be many different optimizations possible if only some slight changes will come up with every iteration. This could be subject of another study.

References

[Bra93] T. Brandes. Automatic Translation of Data Parallel Programs to Message Passing Programs. In *International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction*, Saarbrücken, March 1993.

[BSS91] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[Cor92] Thinking Machines Corporation. Using the CMAX Converter. Manual, TMC, September 1992.

[Hem91] R. Hempel. The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model. User's guide and reference manual, Gesellschaft für Mathematik und Datenverarbeitung mbH, November 1991.

[Hig93] High Perforamnce Fortran Forum. High Performance Fortran Language Specification. Final Version 1.0, Department of Computer Science, Rice University, May 1993.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.