PPPE Deliverable 4.3.a (Part 1 of 2)
Analysing various HPF Parallelisation Strategies

Organisation:	GMD, St. Augustin, Germany
Contributor:	Thomas Brandes
Deliverable:	4.3.a (Part 1 of 2)
Date:	Aug 31, 1993

Automatic Parallelization/Partitioning: A Case Study on the ECMWF's Weather Forecast Program (2D Case)

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 is to investigate how tools can actually help for the automatic parallelization and automatic partitioning of more complex application codes. For this purpose the ECMWF's weather predication code (2D case) has been chosen.

The results will show that tools can be very helpful for the parallelization of this code after some modifications. Especially with an automatic partitioning tool much straightforword work can be avoided. Though hand written message passing programs will be faster in any case, with extensions of current tools it will be possible to achieve nearly the same performance.

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.

The following code shows a serial Fortran 77 program.

Figure 1: Automatic Parallelization and Partitioning

```
      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 ECMWF Code

In a cooperation between GMD and ECMWF (European Center for Medium-Range Weather Forecasts) it is intended to parallelize the ECMWF's production code for medium-range weather forecasts, the IFS (Integrated Forecasting System).

For this study a sequential program of the 2D model of the IFS has been investigated. This version contains already all relevant data structures and algorithmic components of the corresponding 3D model. It should be mentioned that the same code has already been parallelized successfully by hand [GJS93].

The sequential code has had some minor bugs. There were uninitialized variables (`IPOSTS`, `NSIM4D`, `NSIM4DL`) in the subroutine `CNT4` which caused some problems on some machines.

3 First Steps

Both tools were after some initial problems able to translate and analyze the code.

Though Adaptor can easily generate a host and node program of the given program, there is no use of it, as the code has no data partitioning, no parallel loops and no array operations in it. This means that every node executes the same program on all data.

The CMAX tool determines automatically the home of the arrays (front end or distributed) and generates array statements wherever possible. Though some array operations on distributed arrays have been generated, most part and of course the relevant part remains sequential code.

Currently, the CMAX generated sources cannot be compiled with the Adaptor tool. There have been some scanner problems with complex formats within continuation lines, but some other problems did also come up. In the next time, Adaptor will be extended in such a way that it can translate the generated sources (during this study there was no real use in it).

4 Analyzing the Code

When using parallelization tools it is required to have a standard, portable and scalable code. Furthermore the data partitioning has to go conform with the load partioning in such a way that the processors operate mostly on their own local data.

In the following it is discussed what kind of parallelism was in the code and how the data could be partitioned. Especially the experiences with the parallelization by hand [GJS93] gave useful insights in the code.

4.1 The Spectral Transform Method

The solution method employed in the IFS is the spectral transform technique, in which three different function spaces are involved in the computations, namely grid point space, Fourier space and spectral space.

Figure 2 shows what kind of transformations between these three different spaces exist.

4.2 Parallelism

The sequential code is already written in such a way that one can easily identify parallel loops that can be executed independently. It is very simple to run the code on a shared memory

Figure 2: Spectral Transform Method

parallel machine (e.g. KSR1, Alliant FX/2800, or Cray Y-MP) where good speed-ups might be achieved.

Furthermore, the given parallelism is already structured in such a way that the processors can work on their own local data. The parallelism is coarse-grained parallelism, the current version has only low potential for fine-grained parallelism.

The following is the essential subroutine structure of the code. But only the main subroutines are mentioned.

```
MASTER
 . CNT0
 .  . SU0INIT     ! most setup
 .  . SU0YOMB     ! many other setup
 .  . CNT1
 .  .  . CNT2
 .  .  .  .  CNT3
 .  .  .  .  . CNT4
 .  .  .  .  .  . SPNORM      ! norm calculation in spectral space
 .  .  .  .  .  . STEPO       ! iteration step
 .  .  .  .  .  .  . LTINVM     ! inverse legendre transform
 .  .  .  .  .  .  . SCAN2M     ! calculations in Fourier/Grid space
 .  .  .  .  .  .  .  . SC2RDF    ! read fourier buffer
 .  .  .  .  .  .  .  . FTINV     ! reverse fourier transform
 .  .  .  .  .  .  .  . CPG2      ! grid point calculations
 .  .  .  .  .  .  .  . FTDIR     ! direct fourier transform
 .  .  .  .  .  .  .  . SC2WRF    ! write fourier buffer
 .  .  .  .  .  .  . LTDIRM     ! direct legendre transform
 .  .  .  .  .  .  . SPC2M      ! spectral space calculations
```

4.3 Data Partitioning

Due to the parallelization of the code by hand there is already a good strategy for the data distribution [GJS93].

Figure 3: Distribution Strategy for the IFS code

For the 2D model of the IFS, a 1D partitioning of the data in each of the different data spaces was adequate. During the computations in grid point space and in Fourier space, the partitioning is over latitudes, i.e. different processes are responsible for the calculations of different latitudes. The spectral coefficients are partitioned with respect to wave numbers m such that for a given m all corresponding values are mapped to the same process.

Due to the different distributions there is a redistribution of the data in the Fourier space required. One distribution is with respect to latitudes and the second one is with respect to wave numbers m.

Nevertheless this seemed to be a good distribution strategy for starting with. But the identification of the data structures in the Fortran sources shows that the logical two- or three-dimensional data structures are mapped to one-dimensional ones that are nothing else than working spaces.

The following kind of data structures is used for the Fourier space:

```
      DOUBLE PRECISION GT1BUF(JP_NLENGT1B)
      INTEGER NSTAGT1B(JP_NDGSA:JP_NDGEN)
c     NSTAGT1B is a pointer array
      COMMON /LI_YOMGT1B/ GT1BUF,NSTAGT1B

c     GT1 (IJR,JGL) -> GT1BUF (NSTAGT1B(JGL)+IJR)
```

The advantage of these data structures is of course that the memory is used optimally. But with these data structures there is absolutely no chance for an automatic partitioning with Adaptor. The same is also true for a transformation with the CMAX tool.

A similar data structure exists for the legendre polynomials.

```
    DOUBLE PRECISION RPNM(JP_NLEI3,JP_NSPOLEGP)
    INTEGER NPM(0:JP_NSMAX)

C NPM   : address for legendre polynomial for local M (partitioning
```

Figure 4: Sparse representation of an Array

```
C         wrt. M)
C         e.g RPNM(.,NPM(JM)+JN) gives m=JM,n=NSMAX+2-JN in partitioning
C         over M
```

Also the Spectral space has a triangular structure that is stored in this way. In the Spectral space there are also some arrays that are equivalenced with each other.

In the Grid point space the Grid point arrays exist only for one latitude at one time.

Due to these kind of data structures there is no chance for an automatic parallelization or automatic partitioning.

5 Transformation of the Code

There is only a chance for using the tools when the one-dimensional work spaces are changed to two or three-dimensional arrays within the code.

This restructuring cannot be done automatically, especially as it requires a deep understanding of the code. The work spaces will also be used as dummy arguments and that makes the analysis very complicated.

At the moment these transformations have been done by hand. Indeed they make the code more readable, but there is more memory space required.

5.1 Restructuring the Fourier Space

The Fourier space is given by the four arrays GTnBUF (n=0,1,5,9).

```
      DOUBLE PRECISION GTnBUF(JP_NLENGTnB)
      INTEGER NSTAGTnB(JP_NDGSA:JP_NDGEN)
c     NSTAGTnB is a pointer array
      COMMON /LI_YOMGTnB/ GTnBUF,NSTAGTnB
```

As these data structures are logically three-dimensional ones they will be replaced with:

```
      PARAMETER (JP_NMAXnB = ...)
c     note:  max is given by NBLIOIFD or NBLIOIFB for GT0BUF
c     note:  max is given by NBLIODFA or NBLIODFB for GT1BUF
      DOUBLE PRECISION GTnBUF (JP_NMAXnB, 0:JP_NSMAX, JP_NDGSA:JP_NDGEN)
```

Uses of these arrays are changed in the following way:

```
      GTnBUF (NSTAGTnB(JGL)+KLENB*KM+IJ) = ...
      GTnBUF (IJ,KM,JGL) = ...
```

The arrays are used and defined in the following subroutines:

- GT0BUF is defined in ASRE1 and ASRE1B (last step in the inverse Legendre Transform) and used in SC2RDF (before the inverse Fourier Transform).
- GT1BUF is defined in SC2WRF (after the Fourier Transform) and read in PRFI2 and PRFI2B (first step the Legendre Transform).
- GT9BUF is defined in SC2WRT9 and used in SC2RDT9 (within grid point calculations).

Some attention has to be paid to the subroutine calls where the spaces are dummy parameters.

For the parallelization the arrays GT0BUF and GT9BUF will be redistributed to change the distribution over latitudes to a distribution of wave number m and vice versa.

5.2 Restructuring the Spectral Space

The following data structures are used in the spectral space

```
    DOUBLE PRECISION SPVOR (JP_NFLSUR, JP_NSEFRE)
    DOUBLE PRECISION SPDIV (JP_NFLSUR, JP_NSEFRE)
    DOUBLE PRECISION SPSP  (JP_NSEFRE)
    DOUBLE PRECISION SPOR  (JP_NSEFRE)
```

Logically, these arrays have the following structure:

```
    DOUBLE PRECISION SPVOR (JP_NFLSUR, 2, 0:JP_NSMAX, 0:JP_NSMAX)
    DOUBLE PRECISION SPDIV (JP_NFLSUR, 2, 0:JP_NSMAX, 0:JP_NSMAX)
    DOUBLE PRECISION SPSP  (2, 0:JP_NSMAX, 0:JP_NSMAX)
    DOUBLE PRECISION SPOR  (2, 0:JP_NSMAX, 0:JP_NSMAX)
```

The indexing is changed in the following way:

```
    SPxyz (L, NASNO (JN) + JM) = ...
    SPxyz (L, 1, JN, JM) = ...          ! also for JM=0
c  or
    SPxyz (L, NASNO (JN) - JM) = ...   ! imaginary part
    SPxyz (L, 2, JN, JM) = ...

c  JM <= JN <= NSMAX
c  note: NASNO(JN)=JN*JN+JN+1
c    0   1   2   3   4   5   6
c    1   3   7  13  21  31  43
```

This reorganization of the spectral space did require many essential code changes. Especially the indexing from $-m$ to m makes the transformation rather complicated.

5.3 Data Structures for the Legendre Polynomials

The following data structure for the legendre polynomials

```
    DOUBLE PRECISION RPNM(JP_NLEI3,JP_NSPOLEGP)
    INTEGER NPM(0:JP_NSMAX)
c
c   order is p(0,max+1), ..., p(0,0), p(1,max+1), ..., p(1,1),
c   p(2,max+1), .., p(2,2), ..., p(max,max+1), p(max,max)
```

will be transformed by hand to

```
    DOUBLE PRECISION RPNM(JP_NLEI3,0:JP_NSMAX+1,0:JP_NSMAX)
c   RPNM (JGL, JN, JM), JM <= JN <= NSMAX+1, 0 <= JM <= NSMAX
c                       1 <= JGL <= NDGNH
```

The indexing is changed in the following way:

```
    RPNM (JL, NPM(JM) + JN) = ...
    RPNM (JL, NSMAX+2-JN, JM)      = ...
```

The structure is defined within the subroutine SULEG. It is used within PRLE1 and PRLE2. The definition or computation should be done in parallel with respect to latitudes as the computation is most efficient when using the recursion over n and m. In this case the distribution is over the first index.

When using the polynomials they will be used for a given m. Therefore a redistribution or transposition after the setup should be done. In this case the partitioning should be over the last dimension.

6 Data Partitioning

After changing the arrays there is the possibility to define the correct distribution with layout or align/distribute directives within Adaptor.

```
!hpf$  spectral_template (0:JP_NSMAX)
c      actually use will be from 0 to NSMAX
!hpf$  legendre_template (1:JP_NLEI3)
c      actually use will be from 1 to NDGNH   ! (NDGL+1)/2
!hpf$  fourier_template (JP_NDGSA:JP_NDGEN)   ! e.g. 1:16
c      actually use will be from 1 to NDGL
```

The following alignment is appropriate for the chosen partitioning strategy:

```
!hpf$ align RPNM  (:,*,*) with legendre_template
!hpf$ align RPNMT (*,*,:) with spectral_template

!hpf$ align GTOBUF (*,*,JLEV) with fourier_template (JLEV)
!hpf$ align GTOBUFS (*,M,*) with spectral_template (M)
```

```
!hpf$ align GT1BUF (*,*,JLEV) with fourier_template (JLEV)
!hpf$ align GT1BUFS (*,M,*) with spectral_template (M)

!hpf$ align SPVOR (*,*,*,M) with spectral_template (M)
!hpf$ align SPDIV (*,*,*,M) with spectral_template (M)
!hpf$ align SPSP  (*,*,M) with spectral_template (M)
!hpf$ align SPOR  (*,*,M) with spectral_template (M)
```

There are many other data structures for the grid point space that should be aligned in the same way:

The following arrays are of the common block `LI_YOMGC` defined in the include file `YOMGC`.

```
      DOUBLE PRECISION RCORI (0:JP_NDLSM,1,JP_NDGSA:JP_NDGEN)
      ...
      DOUBLE PRECISION OROGM (0:JP_NDLSM,1,JP_NDGSA:JP_NDGEN)

!HPF$ ALIGN (*,*,:) WITH fourier_template :: RCORI
      ...
!HPF$ ALIGN (*,*,:) WITH fourier_template :: OROGM
```

The following arrays are of the common block `LI_YOMGEM` defined in the include file `YOMGEM`.

```
      DOUBLE PRECISION RCOLON(0:JP_NDLSM,JP_NDGSA:JP_NDGEN),
     &    RSILON(0:JP_NDLSM,JP_NDGSA:JP_NDGEN)
      DOUBLE PRECISION RATATH(0:JP_NDLSM,JP_NDGSA:JP_NDGEN),
     &    RATATX(0:JP_NDLSM,JP_NDGSA:JP_NDGEN)

!HPF$ ALIGN (*,:) WITH fourier_template :: RCOLON, RSILON
!HPF$ ALIGN (*,:) WITH fourier_template :: RATATH, RATATX
```

It should be noted that there are also many alignments for dummy parameters required, in all cases where the subroutines are called with the above data structures as actual arguments.

- the Fourier buffer GT0BUF is used in subroutines (PGPB in ASRE1 and ASRE1B is aligned with the spectral_template, PGPB ins SC2RDF and SC2RDFB is aligned with the fourier_template)
- Fourier buffer GT1BUF is used in subroutines (PGPB in PRFI2B is aligned with the spectral_template, PGPB in SC2WRF and SC2RDFB is aligned with the fourier_template)
- Spectral arrays can be dummy arrays in the subroutines PRFI1, PRFI1B, READINI, UPDSPB, SUSPEC, SUSPECB and new subroutines SPNORM2, SPNORM3.

At the moment only block distribution is supported. This results in some problems for the spectral space where the data structures are triangular. The proposed data distribution from [GJS93] as shown in figure 3 cannot be achieved with Adaptor. The block distribution will lead to a poor load balancing, better results would be possible with a cyclic distribution (is planned for a future version of Adaptor).

Figure 5: Distribution of the Spectral Space

7 Translation with Adaptor

After the partitioning of the data structures there is a good chance for the automatic partitioning with Adaptor. The following problems have also to be solved:

- Finding the parallel loops
- Transposing the data in the Fourier space (at the moment, this can only be done by defining a second data structure, redistribution is not supported within Adaptor)

7.1 Parallel loops

The following essential parallel loops can be identified within the code:

- loop over latitudes in `SULEG` (setup legendre polynomials)
- loop over m in `LTINVM` (inverse Legendre Transform)
- loop over m in `LTDIRM` (Legendre Transform)
- loop over m in `SPC2M` (spectral space calculations)
- loop over latitudes in `SCAN2M` (inverse Fourier, grid point calculations and direct Fourier Transform)

In these loops one or more subroutines are called that work on local data corresponding to the wave number or to the latitude. Furthermore, there exist many local data for one call.

```
do I = 1, M
   call P (I)       ! works on local data for I
end do
```

Adaptor can handle parallel independent loops if it knows how the iterations are distributed. Therefore such a loop will be transformed to:

```
!HPF$ INDEPENDENT, LOCAL_ACCESS, ON_HOME SPVOR(:,:,I)
      do I = 1, M
         call P (I)           ! works on local data for I
      end do
```

The distribution of the variable SPVOR decides how the iterations are distributed.

Furthermore, the subroutine P has to be something like a local subroutine (this feature was not available in Adaptor, but has now been realized). Otherwise there would be communication inserted to make replicated data consistent and there would be many conditional statements inserted.

```
      subroutine P (i)
!hpf$ local P
      integer i
      common /yom/ a
      real a(100)
!hpf$ distribute a(block)
      real X
      X = a(i) + 1.0      ! no broadcast of a(i) required
      a(i) = X            ! a(i) is local data by assertion
      end
```

It should be mentioned that such a subroutine can be handled by a compiler in exactly the same way as a pure subroutine. But it is not a pure subroutine in the sense of High Performance Fortran, as it has side effects on global data in the common block.

In the IFS code now the following loop computes the legendre polynomial in parallel, where SUPOL and all called subroutines in SUPOL have to be pure subroutines.

```
!HPF$ INDEPENDENT, LOCAL_ACCESS
      DO JGL=1,NDGNH
         RPNM (JGL,0,0) = RPNM (JGL,0,0)   ! execute on home of RPNM(..,0)
         CALL SUPOL(NSMAX+1,DLMU(JGL),DLPOL,DLA,DLB,DLC,DLD,DLE
     S                                    ,DLF,DLG,DLH,DLI)
         DO JM=0,NSMAX
            RPNM(JGL,JM:NSMAX+1,JM) = DLPOL(JM:NSMAX+1,JM)
         END DO
      END DO    ! JGL loop
```

When the spaces are correctly distributed, all these loops will only work on their local data and no communication is necessary. Adaptor replicates all other data structures so that every node will have its own work space within these parallel loops.

7.2 Array Syntax

Adaptor supports array syntax. Usually, this array syntax will be translated back to sequential loops, in some routines this array syntax is used to get parallelism.

```
         DO 334 JN=0,NSMAX+1
```

```
         DO 334 JM=0,NSMAX+1
         DLPOL(JM,JN)=0.0D0
 334     CONTINUE

c        becomes
         DLPOL (0:NSMAX+1,0:NSMAX+1) = 0.0D0
```

Especially in the subroutine `SPNORM` array syntax is necessary. Otherwise Adaptor would not be able to identify the reductions (`SUM`). The following loop shows a typical restructuring for the computation of the kinetic energy.

```
C*       1.6      KINETIC ENERGY
C
 160  CONTINUE
C
      DO JLEV=1,ILEVX
         DO JN=0,NSMAX
            JMUP = NASRN(JN)
            HSP_SPACE1 (1:JMUP)=-RLAPIN(JN)*
     1         (SPVOR(JLEV,1,JN,1:JMUP)*SPVOR(JLEV,1,JN,1:JMUP)
     2         +SPVOR(JLEV,2,JN,1:JMUP)*SPVOR(JLEV,2,JN,1:JMUP)
     3         +SPDIV(JLEV,1,JN,1:JMUP)*SPDIV(JLEV,1,JN,1:JMUP)
     4         +SPDIV(JLEV,2,JN,1:JMUP)*SPDIV(JLEV,2,JN,1:JMUP))

            HSP_SPACE1 (0) = (SPVOR(JLEV,1,JN,0)*SPVOR(JLEV,1,JN,0)
     $                     +SPDIV(JLEV,1,JN,0)*SPDIV(JLEV,1,JN,0))

            SPALL1 = HSP_SPACE1(0)
            SPALL2 = sum (HSP_SPACE1(1:JMUP))
            ZXVKE(JN,JLEV)= SPALL2 - 0.5D0 * RLAPIN(JN) * SPALL1
         END DO
         ZENORM(JLEV)=SUM(ZXVKE(0:NSMAX,JLEV))
      END DO

      ZENORA = SUM (ZENORM(1:ILEVX)) * ZUSLEVX
```

7.3 Redistributions or Transposition

At the moment Adaptor does not support redistributions. But it is possible to define the same data structure twice with different distributions. By an assignment between these two structures the transpose is done implicitly. The following code demonstrates it for the Legendre polynomials.

```
    DOUBLE PRECISION RPNM  (JP_NLEI3,0:NCSMAX,0:NCSMAX)
!hpf$ distribute RPNM (BLOCK,*,*)
    DOUBLE PRECISION RPNMT (JP_NLEI3,0:NCSMAX,0:NCSMAX)
!hpf$ distribute RPNMT (*,*,BLOCK)

    RPNM = RPNMT
```

The redistribution for the fourier spaces `GT1BUF` and `GT0BUF` is done in the same way.

7.4 Array Bounds

In most sequential codes array-sizes are "hard-wired", but at run-time it can be possible that only a part of the array is used.

```
common /YOM/ A
dimension A(MAXN)
...
read *, N
do I = 1, N
   A(I) = ...
end do
```

The block distribution of such an array can mean poor utilization of processing resources if N is smaller than MAXN. Therefore the "hard-wired" size should be the actual size as far as possible.

The IFS code has the same problems, but for some other reasons it is absolutely necessary that the declared array size must be the same as the actual array size.

When using automatic arrays or allocatable arrays this problem can be avoided. By the way, when Adaptor would know the real problem size, it can generate an equal distribution. But this feature is not supported in the available version. It should be mentioned that CMAX can handle these situations.

8 Results of the Adaptor Code

In this section many results of the generated message passing programs are presented. The message passing programs have been generated by Adaptor form the transformed serial program. Due to the DALIB as a common run time system it is possible to run the programs on many different architectures and machines.

8.1 Information about the Measurements

The following routines have been measured:

- `SCAN2M` is the routine for the updates in the Fourier and Grid Point space; here is the distribution along the latitudes.
- `LTINVM`, `LTDIRM` and `SPC2M` are measured separately. Here is the distribution along the wave number m.
- Transposition of the Fourier buffers

For the computations along the latitudes there is a linear speed up. For the computations in the Spectral space the speed up is approximately $(p + 1)/2$ where p is the number of processors. The transpositions or redistributions show no or only poor speed-up.

Alliant T21	NP = 1	NP = 2	NP = 3	NP = 4	NP = 6	NP = 8
LTINVM	9.2	5.5	3.8	3.2	2.7	2.0
SCAN2M	36.4	19.2	13.7	11.2	8.6	7.6
LTDIRM	4.3	2.9	2.0	1.8	1.4	1.2
SPC2M	1.3	1.0	0.8	0.6	0.5	0.4
Transpose1	1.5	3.0	3.4	3.1	2.8	2.9
Transpose2	0.6	1.1	1.2	1.1	1.4	1.5
Adaptor	5.4	3.3	2.5	2.1	1.8	1.6

Table 1: Results for T21 for Alliant FX/2800

Alliant T32	NP = 1	NP = 2	NP = 4	NP = 6	NP = 8
LTINVM	18.7	11.0	6.3	4.5	3.9
SCAN2M	79.2	44.0	25.3	19.7	17.3
LTDIRM	9.0	5.2	3.1	2.5	2.1
SPC2M	2.6	2.0	1.2	0.8	0.7
Transpose1	3.1	5.9	5.9	5.7	5.0
Transpose2	1.5	2.2	2.0	1.9	2.0
Adaptor	11.4	7.1	4.4	3.5	3.1

Table 2: Results for T32 for Alliant FX/2800

For many architectures the results of the hand-coded message passing program can be compared with the automatically generated message passing program. The following results are all times in seconds. While the results for the different routines are measured about 100 iterations, the summary results are based on results for 10 iterations and two norm calculations.

8.2 Results on Alliant FX/2800

The Alliant FX/2800 is a multiprocessor system with a shared memory. The message passing of the DALIB is realized by using a shared memory section that is shared by the different processes.

For running the programs the 16 processor machine installed at the GMD has been chosen, but only up to 8 processors have been used.

Alliant T63	NP = 1	NP = 2	NP = 3	NP = 4	NP = 6	NP = 8
LTINVM	77.3	50.5	35.5	30.4	22.1	19.8
SCAN2M	300.3	164.4	114.9	103.6	75.8	64.9
LTDIRM	38.5	23.8	17.1	15.2	11.2	10.2
SPC2M	10.3	7.9	5.7	4.5	3.1	2.5
Transpose1	14.0	26.1	27.3	25.8	21.3	20.0
Transpose2	6.5	8.4	8.6	8.8	6.7	5.6
Adaptor	44.7	28.1	20.9	18.9	14.1	12.3

Table 3: Results for T63 for Alliant FX/2800

CM 5, T10	NP = 1	NP = 2	NP = 4	NP = 8
LTINVM	2.7	1.6	1.0	0.8
SCAN2M	11.1	5.8	3.0	1.7
LTDIRM	1.3	0.8	0.5	0.4
SPC2M	0.5	0.4	0.3	0.3
Transpose1	1.1	1.4	1.1	1.4
Transpose2	0.5	0.5	0.5	0.6
Adaptor	1.7	1.1	0.6	0.5
Hand-Coded	1.4	0.8	0.5	0.3

Table 4: Results for T10 on CM 5 (no vector units)

CM 5, T21	NP = 1	NP = 2	NP = 4	NP = 8	NP = 16
LTINVM	11.8	7.6	3.9	2.4	1.8
SCAN2M	51.1	25.5	12.9	6.8	3.7
LTDIRM	6.6	4.3	2.3	1.4	1.0
SPC2M	1.9	1.4	0.8	0.6	0.5
Transpose1	4.5	4.8	3.5	2.9	2.8
Transpose2	1.7	1.8	1.5	1.1	1.6
Adaptor	7.8	4.6	2.4	1.5	1.1
Hand-Coded	5.7	3.1	1.7	0.9	0.6

Table 5: Results for T21 on CM 5 (no vector units)

It is planned to compare these results with a shared memory parallelization of the original code. Such a parallel program will not require any redistribution and will have a better load balancing as any free processor can execute every iteration of the parallel loops.

8.3 Results on CM-5

There have been no problems moving the code to the Connection Machine CM-5. The vector units are not utilized because there is no vectorization for a single node.

As one node of the CM-5 has about 32 MByte memory, bigger problem sizes can run on this machine.

8.4 Results on the KSR 1

The following problems arised when porting the Adaptor generated code to the KSR 1 machine:

- The `REWIND` statements on the logical unit `NULNAM` must be removed
- The computation of `RSIGMA` in the subroutine `SUCST` causes an overflow error. This computation must be rearrranged.

```
RPI=2.D0*ASIN(1.D0)
RCLUM=299792458.D0
```

CM 5, T42	NP = 1	NP = 2	NP = 4	NP = 8	NP = 16	NP = 32
LTINVM	48.9	32.7	17.6	9.2	5.8	4.0
SCAN2M	170.1	85.2	43.1	21.6	11.1	5.8
LTDIRM	30.9	19.6	10.3	5.7	3.4	2.4
SPC2M	5.9	4.3	2.4	1.5	1.0	0.8
Transpose1	17.5	19.1	15.7	10.2	10.1	6.9
Transpose2	7.0	8.0	5.2	4.2	3.3	3.4
Adaptor	28.0	16.9	9.4	5.3	3.5	2.3
Hand-Coded	25.3	13.5	7.1	3.9	2.0	1.2

Table 6: Results for T42 on CM 5 (no vector units)

CM 5, T63	NP = 1	NP = 2	NP = 4	NP = 8	NP = 16	NP = 32
LTINVM	134.7	92.0	52.2	27.4	14.0	7.4
SCAN2M	422.7	266.3	112.3	51.9	26.3	13.6
LTDIRM	86.0	56.7	31.6	16.4	8.5	4.5
SPC2M	14.3	10.6	6.1	3.5	2.0	1.1
Transpose1	39.0	41.3	34.5	37.6	18.0	16.2
Transpose2	16.3	15.5	11.9	6.7	9.3	5.4
Adaptor	71.3	48.2	24.9	14.4	7.8	4.8
Hand-Coded	64.5	34.0	17.5	9.0	4.6	2.6

Table 7: Results for T63 on CM 5 (no vector units)

CM 5, T106	NP = 4	NP = 8	NP = 16	NP = 32
LTINVM	214.7	111.7	61.1	35.3
SCAN2M	282.7	140.6	70.9	35.4
LTDIRM	135.6	71.1	38.4	22.4
SPC2M	16.0	8.8	5.3	3.1
Transpose1	46.2	39.7	22.9	16.2
Transpose2	18.3	13.9	10.9	10.5
Adaptor	71.4	38.6	21.0	12.3
Hand-Coded	57.4	29.1	15.0	8.0

Table 8: Results for T106 on CM 5 (no vector units)

CM 5, T169	NP = 32	NP = 48	NP = 64
LTINVM	133.6	95.2	72.0
SCAN2M	93.4	66.8	46.9
LTDIRM	107.4	76.0	57.5
SPC2M	7.0	4.8	3.7
Transpose1	40.6	24.4	29.2
Transpose2	14.8	16.1	24.7
Adaptor	39.7	29.2	23.4
Hand-Coded	26.8	19.5	n.a.

Table 9: Results for T169 on CM 5 (no vector units)

CM 5, T203	NP = 32	NP = 48	NP = 64
LTINVM	297.3	n.a.	168.7
SCAN2M	151.2	n.a.	78.9
LTDIRM	238.6	n.a.	135.3
SPC2M	10.7	n.a.	6.1
Transpose1	111.5	n.a.	74.2
Transpose2	38.0	n.a.	40.8
Adaptor	84.8	n.a.	50.4
Hand-Coded	50.1	36.5	n.a.

Table 10: Results for T203 on CM 5 (no vector units)

KSR 1, T63	NP = 1	NP = 2	NP = 4
LTINVM	60.8	42.0	25.5
SCAN2M	123.3	60.5	33.3
LTDIRM	39.1	26.1	15.8
SPC2M	13.5	8.2	6.1
Transpose1	19.2	30.0	27.7
Transpose2	7.4	11.7	10.8
Adaptor	26.3		

Table 11: Results for T63 on KSR 1

```
      RHPLA=6.6260755D-34
      RKBOL=1.380658D-23
      ...
C     old computation
C     RSIGMA=2.D0 * RPI**5 * RKBOL**4 /(15.D0* RCLUM**2 * RHPLA**3)
C     new computation :
      RSIGMA=2.D0 * RPI**5 * RKBOL**2 /(15.D0* RCLUM**2 * RHPLA)
      RSIGMA=RSIGMA * RKBOL ** 2 / RPHPLA**2
```

8.5 Results on the iPSC/860

The iPSC requires that the logical unit `NULNAM` is explicitly opened. The follwing `OPEN` statement must be inserted in the subroutine `SURIP`:

```
OPEN (UNIT=NULNAM,FILE='fort.4',STATUS='OLD', FORM='formatted')
```

It is not possible to run the problem T63 on one node, but on two nodes.

8.6 Other Results

The generated parallel program runs also on a net of workstation where the communication is realized by PVM [Sun90].

There have been no results on the Parsytec GCel machine because the input with namelists is not supported. The compiler failed to translate the generated code.

iPSC/860, T42	NP = 1	NP = 2	NP = 4
LTINVM	29.9	18.3	9.5
SCAN2M	83.4	42.1	21.2
LTDIRM	13.3	8.9	4.9
SPC2M	3.9	2.9	1.6
Transpose1	2.8	18.3	18.0
Transpose2	1.4	4.2	5.9
Adaptor	13.5	9.5	6.1
Hand-Coded	20.2	11.2	6.0

Table 12: Results for T42 on iPSC/860

iPSC/860, T63	NP = 2	NP = 4	NP = 8
LTINVM	53.8	30.3	15.8
SCAN2M	97.1	48.3	24.3
LTDIRM	29.9	17.0	8.8
SPC2M	5.9	3.4	1.8
Transpose1	36.0	41.1	29.0
Transpose2	10.4	13.2	9.7
Adaptor	23.3	15.3	9.0
Hand-Coded	n.a.	13.8	7.4

Table 13: Results for T63 on iPSC/860

Sun 10, T63	NP = 1	NP = 2	NP = 3
LTINVM	95.1	67.7	49.5
SCAN2M	201.5	102.7	70.3
LTDIRM	69.1	47.2	34.8
SPC2M	9.1	6.1	5.6
Transpose1	12.9	100.2	641.9
Transpose2	7.3	38.3	495.3
Adaptor	39.5	36.7	129.5

Table 14: Results for T63 on a net of Sun Sparc10 workstations (PVM 2.4)

9 Translation with CMAX

The parallelism in the IFS code is coarse-grained and cannot be used for the fine-grain parallelism that is needed for the CM and at that the CMAX tools targets.

To use the existing parallelism for the CM it is necessary to get the parallel loops within the subroutines. This is not done automatically, especially it would require the expansion of many scalar data.

As there was at the moment no general parallelization strategy to use fine-grained parallelism, no more work has been investigated in using CMAX. This does not imply that CMAX is of no use for this code but it would require more understanding and more restructuring of the code.

10 Comparison of hand-coded with automatically generated parallel programs

It should be mentioned here that in both cases of parallelization a deep understanding of the given program is necessary. It is also important that the code contains enough potential for parallel execution and data locality.

The advantages of the automatic partitioning can be summarized as follows

- faster development,
- more readable code (especially for maintenance),
- use of a parallel programming model with global name space,
- less changes to the sequential program,
- no host/node and sequential version of the same code as Adaptor can generate sequential version, host-node programs and hostless parallel programs.

But in the case of the 2D IFS code one has the following disadvantages:

- more space for data structures is required especially by the restructuring to multi-dimensional arrays,
- poor processor utilization when running small problems with large array sizes,
- communication is better optimized in the hand-coded version, especially for the transposition of the arrays,
- the hand-coded version has less communications, e.g. for sending initial values from the host, but also for the reductions many optimizations are used in the hand-coded version.

Furthermore, in contrary to the hand-coded message passing program, it was not possible to keep the main numerical components of the sequential algorithm unchanged.

All the disadvantages can be more harmless if automatic partitioning tools are improved. Especially by optimizing data transpositions or redistributions not only this code will profit but also many other codes.

11 Using High Performance Fortran

Adaptor is a tool that supports many features of CM Fortran and some of High Performance Fortran (HPF). If efficient HPF compilers would be available they could deal with the code in the same manner as Adaptor, but hopefully better as more features are supported.

In the case of the IFS code, one could benefit from High Performance Fortran [Hig93a] due to the more flexibility of data distributions. Especially the cyclic distribution of the Spectral space would improve the load balancing dramatically.

For the loop parallelization the current features of HPF would not be sufficient. But the `EXECUTE-ON-HOME` directive and the `LOCAL-ACCESS` directive that are proposed in the journal of development [Hig93b] are just the right directives for this code. But with some tricky handling it could be possible to make the parallelization possible with an existing compiler.

But there will be no support for the sparse representation of the data structures as it is done in the sequential code. Furthermore, the performance of the redistribution will depend on the realization on the different systems.

12 Alternative Parallelization Strategies

The Adaptor version of the code has required some changes of the serial code. But this code allows to make some experiments with other parallelization strategies or better other distribution strategies.

An alternative approach is not to redistribute the data between the various stages. This requires that one step itself will have a lot of communication. As this communication is generated automatically, it can be tested in a narrow time frame. But the code has to be changed to make the other kind of parallelism feasible.

The considered algorithm contains an Eulerian treatment of advection. If a Semi-Lagrangian scheme is used, the parallelization strategy will have to be extended in order to ingerate additional data exchanges in grid point space. This change can easily be done with the Adaptor tool as it supports overlap areas.

13 Summary

The experiences with the ECMWF code show that tools can help in automatic parallelization and partitioning of parallel codes. But it is very important to have a clean code and a good parallelization strategy, this means, the user has to know how to distribute the data and the work corresponding to the data. This will require changing of data structures and operations on them.

So tools can save much time in doing straightforward work more reliable. This time can be used to concentrate more on improving the algorithm and the parallelization itself.

Of course tools will generate less efficient programs than hand-coded ones. But there is still a high potential for optimizations, so that there is good chance that they will come in a state that they are really competitive.

References

[Bra93] T. Brandes. Autmatic 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.

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

[GJS93] U. Gärtel, W. Joppich, and A. Schüller. Parallelizing the ECMWF's Weather Forecast Program: The 2D Case, Technical Documentation and Results for the IFS-2D Model. Arbeitspapiere der GMD 740, Gesellschaft für Mathematik und Datenverarbeitung mbH, March 1993.

[Hig93a] High Perforamnce Fortran Forum. High Performance Fortran Language Specification. Final Version 1.0, Department of Computer Science, Rice University, May 1993.

[Hig93b] High Performance Fortran Forum. High Performance Fortran: Journal of Development. Version 1.0, Department of Computer Science, Rice University, May 1993.

[Sun90] V. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 3(10), December 1990.