ADAPTOR : a public domain

HPF compilation system

Dr. Thomas Brandes

**German National Research Center
for Information Technology (GMD)**

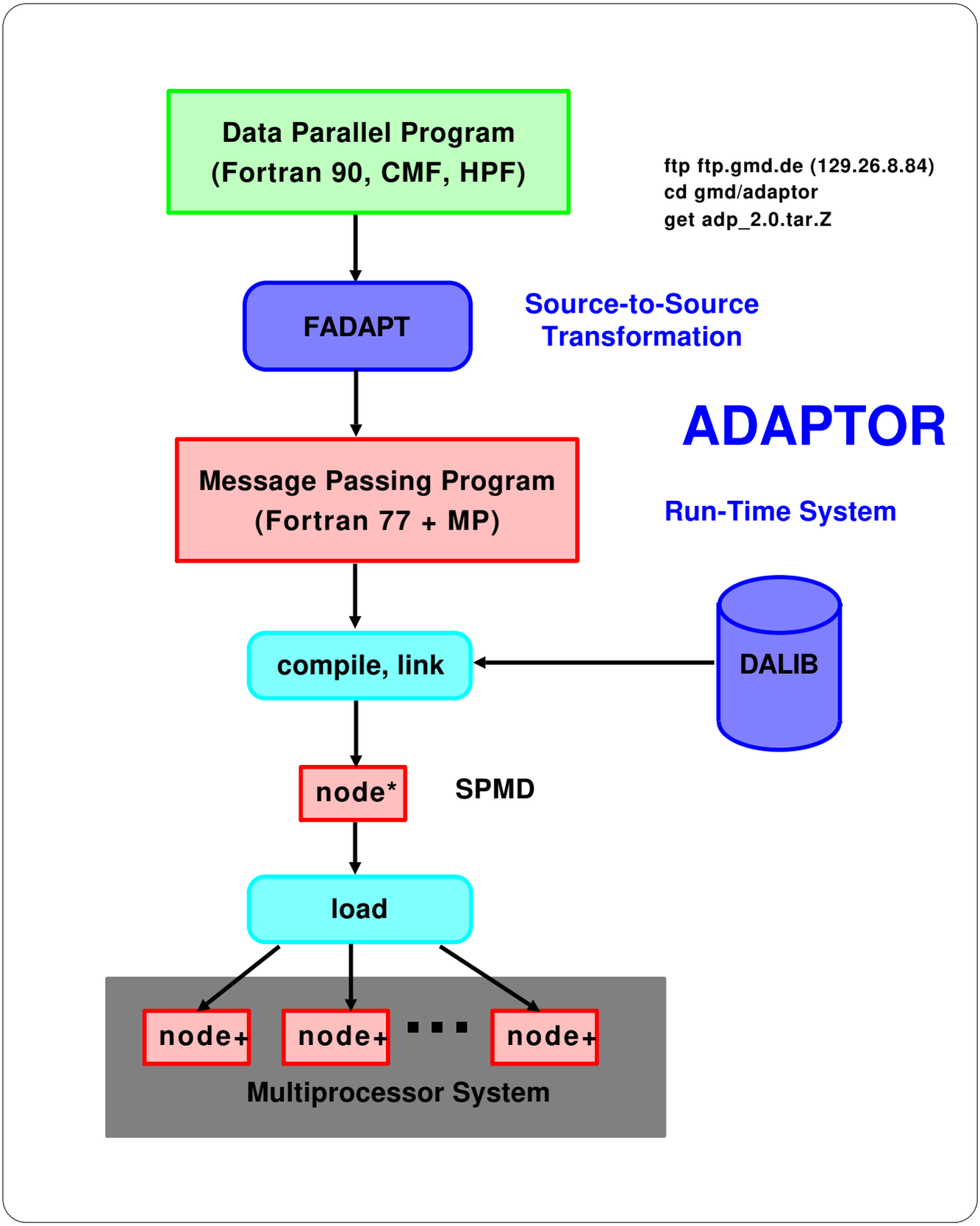
**Institute for Algorithms and
Scientific Computing (SCAI)**

May 1995

ADAPTOR: HPF compilation system

1. ADAPTOR : System Overview
2. Generation of SPMD code
3. Temporary data
4. Compiler optimizations
5. Example
6. Evaluation / Results

Content



1. ADAPTOR : System Overview

ADAPTOR available on:

- Intel Paragon
- Meiko CS-2
- IBM SP-2
- Power PC
- CM-5
- HP, IBM, SUN workstations
- NEC Cenju-3

Supported communication protocols:

- PVM
- MPI
- PARMACS
- Intel NX
- Meiko ELAN
- P4
- IBM EUI

Versions:

1.0	March 1993
2.0	March 1994
3.0	June 1995

More Information:

http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html

Supported Machines / Communication Protocols

Version 3.0 (June ’95) is available via anonymous ftp:

```
ftp ftp.gmd.de (129.26.8.84)
cd gmd/adaptor
bin
get adp_3.0.tar.Z
quit

uncompress adp_3.0.tar.Z
tar xvf adp_3.0.tar
```

bin/*	Executables
src/*	Source Files
man/*	Man Pages
doc/*	Documentation

Installation of ADAPTOR

Let $PHOME be the installation directory and $HOME the user home directory.

a) set link to the ADAPTOR directory

```
ln -s $PHOME $HOME/adaptor
```

b) set correct pathes (.login)

```
setenv PATH $HOME/adaptor/bin:$PATH
setenv MANPATH $HOME/adaptor/man:$MANPATH
```

c) first test

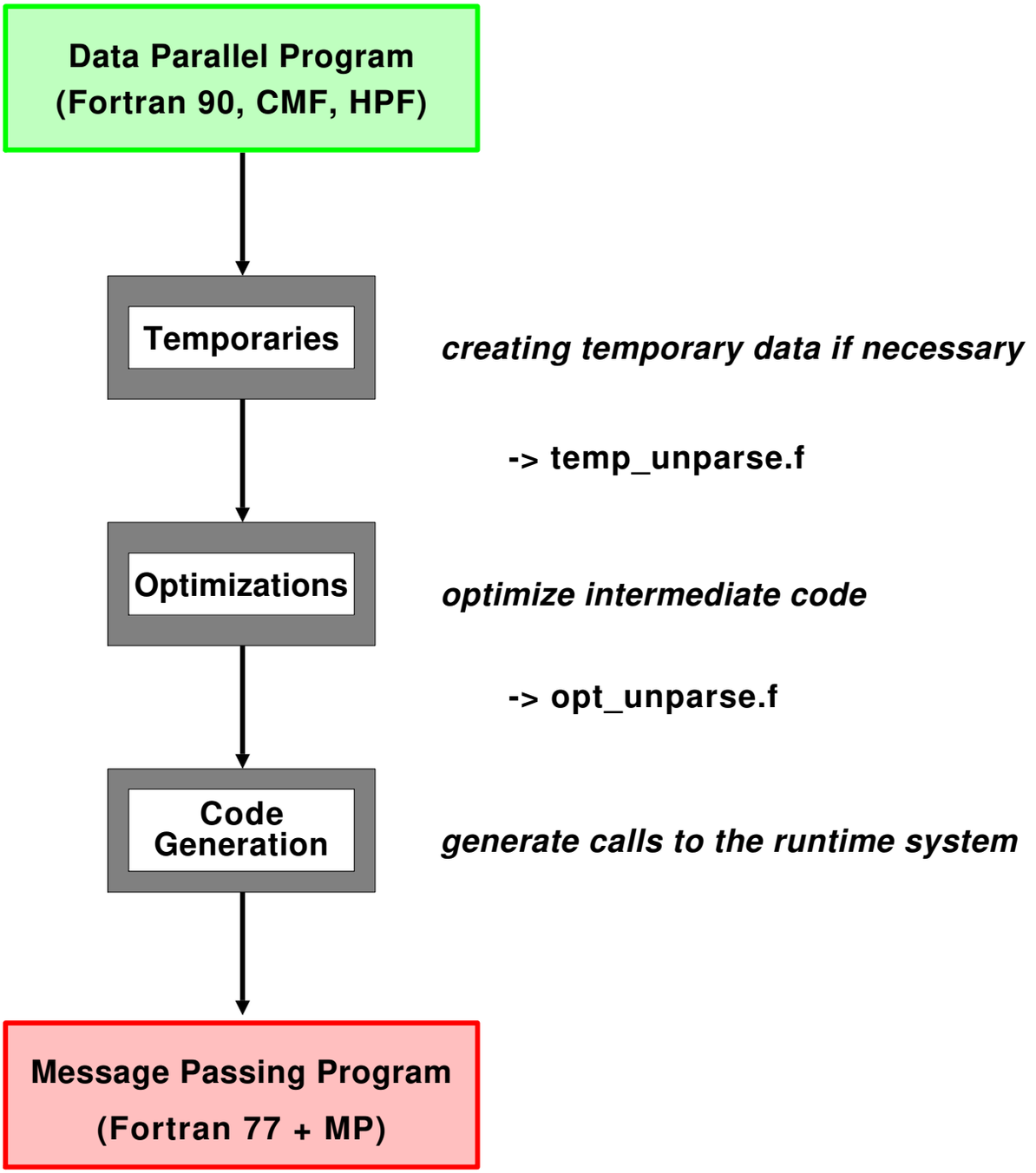
```
man fadapt
fadapt -help
```

d) first example

```
mkdir example
cd example
cp $HOME/adaptor/hpf_examples/simple/pi.hpf .
fadapt pi.hpf
make
```

run executable cube, e.g.: **cube 4, prun -n4 cube**

Using ADAPTOR



ADAPTOR : Phases of Translation

Code Generation of ADAPTOR

- generation of SPMD code
- replication of data and control flow
- array descriptors
- handling of distributed arrays
- processor arrays
- pseudo-dynamic arrays
- basic translation
- translation of array movements

Input source program (data parallel):

test.hpf:

```
PROGRAM TEST
CALL SUB ()
END

SUBROUTINE SUB ()
END
```

fadapt test.hpf

Output source program (SPMD, F77 + MP):

cube.f:

```
PROGRAM TEST
CALL dalib_init ()
CALL dalib_start_subroutine ('TEST', ...)
CALL SUB ()
CALL dalib_end_subroutine ()
CALL dalib_exit ()
END

SUBROUTINE SUB ()
CALL dalib_start_subroutine ('SUB', ...)
CALL dalib_end_subroutine ()
END
```

Generation of SPMD Code

Input source program (data parallel):

```
PROGRAM TEST
REAL A, B
IF (A .gt. B) THEN
    A = B
END IF
DO WHILE (A .le. B)
    A = A + B
END DO
....
```

Output source program (SPMD):

```
PROGRAM TEST
REAL A, B
CALL dalib_init ()
...
IF (A .gt. B) THEN
    A = B
END IF
DO WHILE (A .le. B)
    A = A + B
END DO
....
```

- all scalar variables are replicated
- the control flow is replicated

Replication of data and control flow

Info about one dimension:

```
typedef struct                          P1          P2
  { int global_size [2];              (1:50)      (1:50)
    int local_size [3];               (1:25:1)    (26:50:1)
    int overlap [2];                  (1:1)       (1:1)
  } DimInfo;
```

Note: values of global_size and overlap will be the same on all processors, but local_size might be different.

Info about a distributed array:

```
typedef struct
  { int size;        /* bytes for one single element      */
    int rank;        /* rank of the array, 1 to 7         */
    char *data;      /* pointer to locally allocated data */
    Distribution DistributeInfo;
    Alignment   AlignInfo;
    DimInfo dimensions [rank];
  } ArrayRecord;
```

Info about distribution:

```
typedef struct
  { int topology;    /* pointer to processor array */
    DistDimDescriptor [rank];
  } DistributionRecord;

typedef DistributionRecord *Distribution;
```

Array descriptors

Input source program (data parallel):

```
          PROGRAM TEST
          INTEGER N1, N2, N3
           PARAMETER ( N1=100, N2=50, N3=10 )
          REAL A ( N1, N2, N3 )
!HPF$ DISTRIBUTE A ( *, BLOCK, CYCLIC)
          ...
```

Output source program (SPMD):

```
PROGRAM TEST
INTEGER N1, N2, N3
PARAMETER (N1=100,N2=50,N3=10)
INTEGER A_DSP
INTEGER A_LOW2, A_HIGH2, A_STR2
INTEGER A_LOW3, A_HIGH3, A_STR3
REAL A(...)
...
!  create array descriptor of rank 3 with 4 bytes for one element
CALL dalib_array_create (A_DSP, 3, 4, 1, N1, 1, N2, 1, N3)
! distribute onto two-dimensional default topology
! 0 : serial, 1 : block, 2 : cyclic
CALL dalib_distribute (A_DSP, 2, 0, ..., 1, ..., 2, ...)
CALL dalib_array_query (A_DSP, 2, A_LOW2, A_HIGH2, A_STR2)
CALL dalib_array_query (A_DSP, 3, A_LOW3, A_HIGH3, A_STR3)
CALL dalib_array_allocate (A_DSP, A, ...)
...
CALL dalib_array_free (A_DSP)
```

Handling of distributed arrays (1)

Array Descriptors:

```
      PARAMETER ( N1=100, N2=50, N3=10 )
     REAL A (N1, N2, N3)
!HPF$ DISTRIBUTE A (*, BLOCK, CYCLIC)
```

Values on a 2 x 3 processor topology:

A(100,1:25,1:10:3)

A_LOW2	1
A_HIGH2	25
A_STR2	1
A_LOW3	1
A_HIGH3	10
A_STR3	3

A(100,1:25,2:8:3)

A_LOW2	1
A_HIGH2	25
A_STR2	1
A_LOW3	2
A_HIGH3	8
A_STR3	3

A(100,1:25,3:9:3)

A_LOW2	1
A_HIGH2	25
A_STR2	1
A_LOW3	3
A_HIGH3	9
A_STR3	3

A(100,26:50,1:10:3)

A_LOW2	26
A_HIGH2	50
A_STR2	1
A_LOW3	1
A_HIGH3	10
A_STR3	3

A(100,26:50,2:8:3)

A_LOW2	26
A_HIGH2	50
A_STR2	1
A_LOW3	2
A_HIGH3	8
A_STR3	3

A(100,26:50,3:9:3)

A_LOW2	26
A_HIGH2	50
A_STR2	1
A_LOW3	3
A_HIGH3	9
A_STR3	3

Handling of distributed arrays (2)

Input source program (data parallel):

```
!HPF$ PROCESSORS P (3,4)
      REAL A ( N1, N2, N3 )
!HPF$ DISTRIBUTE A ( *, BLOCK, CYCLIC) ONTO P
```

Output source program (SPMD):

```
---
CALL dalib_top_create 2 (P_TOPID, 1, 3, 4)
CALL dalib_array_create (A_DSP, 3, 4, 1, N1, 1, N2, 1, N3)
CALL dalib_distribute (A_DSP, P_TOPID, 0, 0, 1, 0, 2, 0)
...
```

Note:

- there are a one-, a two- and a three-dimensional topology availabe by default (identified by integer values 1, 2 and 3)

- the size of the default topologies is given by the number of available processors at runtime.

- the one-dimensional default topology will contain all available node processors

- default topologies can be chosen explicitly by command arguments

  cube 2x3
  cube 3x2x4

Processor arrays / topologies

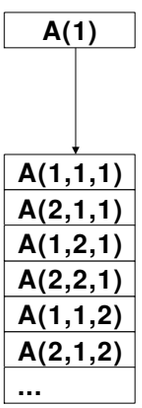
Pseudo-dynamic arrays:

```
      PARAMETER ( N1=100, N2=50, N3=10)
     REAL A (N1, N2, N3)
!HPF$ DISTRIBUTE A (*, BLOCK, CYCLIC)
```

Memory will be allocated on the heap:

```
REAL A(1)
INTEGER A_DIM1, A_DIM2, A_DIM3
INTEGER A_ZERO
...
CALL dalib_array_allocate (A_DSP, A, A_ZERO,
                           A_DIM1, A_DIM2, A_DIM3)
```

A (I, J, K) -> A (A_ZERO + I + J * A_DIM1 + K * A_DIM2)



FORTRAN array

HEAP

! out of boundaries addressing !

Input source program (data parallel):

```
        REAL A (N), B
!HPF$ DISTRIBUTE A (BLOCK)
        A(10) = B      ! update of local data with replicated data
        B = A(10)      ! update of replicated data with local data
         A = 0.0
         A = A + 1.0
        B = sum (A)
```

Output source program (SPMD):

```
....
IF (dalib_is_local (A_DSP, 1, 10)) THEN
     A(A_ZERO+10) = B
END IF
IF (dalib_is_local (A_DSP, 1, 10)) THEN
     B = A(A_ZERO+10)
END IF
CALL dalib_array_broadcast (A_DSP, 1, 10, B, 4)
DO I1 = A_LOW1, A_HIGH1
     A(A_ZERO+I1) = 0.0
END DO
DO I1 = A_LOW1, A_HIGH1
     A(A_ZERO+I1) = A(A_ZERO+I1) + 1.0
END DO
B = 0.0
DO I1 = A_LOW1, A_HIGH1
     B= B+ A(A_ZERO+I1)
END DO
CALL dalib_reduction (B, 8)    ! 8 stands for sum of reals
...
```

Basic Translation

Input source program (data parallel):

```
          REAL A(N,N), B(N,N)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) :: A, B
          A = cshift (B, dim = 1, shift = -1)
          A ( 1:N-1:2 , : ) = B ( 2:N:2 , : )
          A = transpose (B)
```

Output source program (SPMD):

```
....
CALL dalib_cshift (A_DSP, B_DSP, -1, 1)
...
CALL dalib_section_create ( ISEC_DSP1, A_DSP,
                                    1, 1, N-1, 2,  1, 1, N, 1)
CALL dalib_section_create ( ISEC_DSP2, B_DSP,
                                    1, 2, N, 2,  1, 1, N, 1)
CALL dalib_assign ( ISEC_DSP1, ISEC_DSP2)
CALL dalib_section_free (ISEC_DSP2)
CALL dalib_section_free (ISEC_DSP1)
...
CALL dalib_assign_permute (A_DSP, B_DSP, 2, 1)
...
```

Translation of array movements

Problems:

- distributed data in control flow expressions
- complex array assignments
- distributed data in IO
- distributed arrays in argument lists of subroutine calls

Solution:

- creation of temporary data (intermediate source)

General Idea:

Create temporaries in such a way that there are only

- basic data movements
- local array statements (without communication)

Note:

The capability of creating correct temporary data
has dramatically improved with version 3.0.

---

3. Temporary data

Input source program (data parallel):

```
      REAL A(N), B(N)
!HPF$ DISTRIBUTE A(BLOCK) :: A, B
      IF (A(10) .gt. B(10)) THEN
      ....
      IF (A(10) .gt. B(11)) THEN
      ....
```

Intermediate source:

```
      ....
      LOGICAL TMP1
      REAL TMP2
      REAL TMP3
      LOGICAL TMP4
      ...
      TMP1 = A(10) .gt. B(10)
      IF (TMP1) THEN
      ...
      TMP2 = A(10)
      TMP3 = B(11)
      TMP4 = TMP2 .gt. TMP3
      IF (TMP4) THEN
      ...
```

Creating temporary scalars

Input source program (data parallel):

```
          REAL A(N), B(N)
!HPF$ DISTRIBUTE A(BLOCK) :: A, B
          A (2:N-1) = ( B (2:N-1) + B (3:N) ) * 0.5
          ....
          A (2:N-1) = ( A (1:N-2) + A (3:N) ) * 0.5
```

Intermediate source:

```
          ....
          REAL*4 TMP1 (:)
!HPF$ ALIGN TMP1 (I) WITH A(I)
          REAL*4 TMP2 (:)
!HPF$ ALIGN TMP2 (I) WITH A(I)
          REAL*4 TMP3(:)
!HPF$ ALIGN TMP3 (I) WITH A(I)
          ....
          ALLOCATE (TMP1(1:N))
          TMP1(2:N-1) = B(3:N)
          A(2:N-1) = ( B(2:N-1) + TMP1(2:N-1) ) * 0.5
          DEALLOCATE (TMP1)
          ...
          ALLOCATE (TMP2(1:N))
          TMP2 (2:N-1) = A(1:N-2)
          ALLOCATE (TMP3(1:N))
          TMP3 (2:N-1) = A(3:N)
          A(2:N-1) = ( (TMP2(2:N-1) + TMP3(2:N-1) ) * 0.5
          DEALLOCATE (TMP3)
          DEALLOCATE (TMP2)
          ....
```

Creating temporary arrays

Optimization Issue	Effect
Overlap Area	reduces local memory transfers
Loop Fusion	better cache usage
Statement Reordering	allows other optimizations
Communication sets	reduces time for testing ownerships
Combining Messages	reduces message startup time
Reusing Comm. Pattern	reduces communication costs
Runtime System	general overhead
Overlapping Computation/ Communication	hides communication time

fadapt -O <test.hpf>

4. Compiler Optimizations

Example: Computation of PI

High Performance Fortran Code

```
program  picalc
integer n, i
double precision w, gsum
double precision, allocatable :: v(:)
print *, 'Input number of stripes : '
read *, n
w = 1.0 / n
allocate (v(1:n))
v = ([1..n] - 0.5) * w
v = 4.0 / (1.0 + v*v)
gsum = sum(v)
print *, 'Pi is approximately : ', gsum*w
deallocate (v)
end
```

Note: default distribution of v is (block)

5. Example

generated SPMD code

```
program  picalc
integer n, i
real*8 w, gsum
integer V_DSP, V_LOW1, V_HIGH1, V_DIM1, V_ZERO
real*8 v(1)
integer I1
integer dalib_pid
external dalib_pid
call dalib_init ()
IF (dalib_pid() .eq. 1) THEN
   print *, 'Input number of stripes : '
   read *, n
END IF
call dalib_broadcast (N, 4, 1)
w = 1.0 / n
call dalib_array_create (V_DSP, 1, 8, 1, N)
call dalib_distribute (V_DSP, 1, 1, 0)
call dalib_array_query (V_DSP, 1, V_LOW1, V_HIGH1)
dall dalib_array_allocate (V_DSP, V, V_ZERO, V_DIM1)
gsum = 0.0d0
DO I1 = v_low1, v_high1
      V(V_ZERO+I1) = (I1-0.5)*W
      V(V_ZERO+I1) = 4.0/(1.0+V(V_ZERO+I1)*V(V_ZERO+I1))
      gsum = gsum + V(V_ZERO+I1)
END DO
call dalib_reduction (gsum, 9)      ! 9 stands for sum of doulbes
if (dalib_pid() .eq. 1) THEN
     print *, 'Pi is approximately : ', gsum*w
END IF
call dalib_array_free (V_DSP)
call dalib_exit ()
END
```

Example: generated SPMD code

Shallow Water Equation

Hydrodynamics Code

Ocean Model

AVL FIRE Code

ECMWF Integrated Forecasting System

6. Results / Experiences

Results: Shallow Water Equation

Machine: Meiko CS-2 (University of Vienna)

Problem size : 1024 x 1024

Distribution : (block, block)

ADAPTOR: version 3.0

F77: Fortran 3.0.1 from SUN

	1	2	4	8	16	32
not optimized		124.70 s	58.37 s	25.81 s	11.55 s	6.41 s
optimized	66.71 s	33.59 s	16.70 s	8.17 s	3.90 s	1.93 s
MFlops	12.8	25.4	51.0	104.3	218.3	440.4

HYDFLO

~ 1500 lines of code

three-dimensional compressible hydrodynamics code based on an explicit two-step Lax-Wendroff finite difference scheme
regular Eulerian mesh in a regular domain (3D, (*,block,block))
using stencil operations

Meiko CS-2	1	2	4
not optimized	12.32 s	5.74 s	2.93 s
optimized	8.49 s	4.30 s	2.27 s

MOMA

~ 5500 lines of code

Free surface ocean model, based on the Bryan-Cox-Shutner code
D.J. Webb, Institute of Oceanographic Sciences, U.K.
CM Fortran version by Mike Ashworth, NERC

Meiko CS-2	1	2	4
not optimized	11.14 s	5.82 s	3.61 s
optimized	8.36 s	4.80 s	2.89 s

Results for some larger data parallel codes

AVL FIRE code

- interactive fluid dynamics package
- 3D, 2D unsteady and steady simulations
- complex geometries
- indirect addressing

Benchmark solver

- GCCG : conjugate gradients, orthomin
  with diagonal scaling
- about 400 lines of code
- main loop is iteration over the geometry
  using indirect addressing

About the parallelization

- special runtime support for indirect addressing
- communication patterns will be reused
- overlap areas for irregular grids will be used

Comparison of runtimes:

Results are runtimes (in seconds) on an IBM SP-1

47312 cells	1	2	4	8
Adaptor , overlap	31.3 s	17.9 s	11.5 s	6.1 s
F77 + Mess.Pass.	27.5 s	17.2 s	15.5 s	10.4 s

Results for the AVL FIRE benchmark solver

ECMWF: IFS Code (2D version)

ECMWF: European Centre for Medium Weather Forecasting, Reading

IFS : Integrated Forecasting System

- **uses Spectral Transform Method**
- **about 22.000 Lines of Code**
- **contains all relevant data structures**

Message Passing Paralleliization

- **Parallelization of IFS (PARMACS) done by U. Gärtel, W. Joppich, A. Schüller (GMD)**

- **Efficiencies about 80 - 95 %**

HPF Parallelization

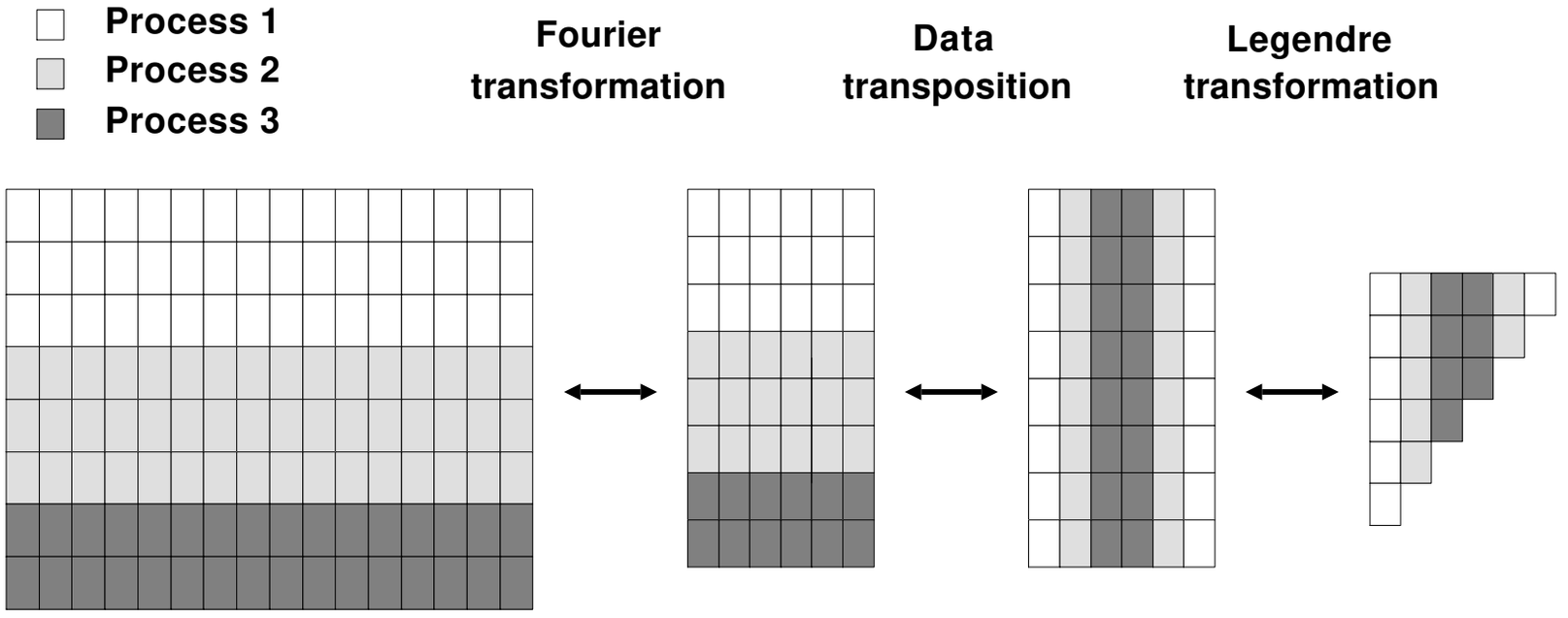
- **Problem : Usage of workspace arrays**
- **required restructuring of data structures and cleaning up**
- **needs now double space of memory**

Results:

T63	1	4	8	16	32
HPF, ADAPTOR	71.0 s	19.9 s	10.0 s	5.5 s	3.2 s
F77 + Mess. P.	64.5 s	17.5 s	9.0 s	4.6 s	2.6 s

*** results are measured on a CM-5 without using vector units**
*** results are measured with HPF compilation system 'ADAPTOR'**

Results : Integrated Forecasting System



Parallelization Strategy for Spectral Transform Method:

- distribution of grid point space along the latitudes
- distribution of spectral space along the wave numbers
- Fourier / Legendre transformations do not require communication (local operations)
- updates in grid point space and spectral space without any communication
- only the data transposition in the Fourier space requires communication

Parallelization Strategy for IFS