F90 OR HPF FOR THE ORIGIN 2000?

TOR SøREVIK AND ARILD DYRSETH
Para//ab, UNIVERSITY OF BERGEN *

Abstract.

In this paper we present a preliminary study of the new datadistribution directives for the SGI compilers on the Origin 2000. We discuss their relationship with the corresponding HPF directives and try to assess their value by testing them on a set of applications available to us. The testsuite was originally developed for HPF. For comparison we also compare efficiency under PGI's HPF compiler.

1. Introduction.

1.1. The issue. On state-of-the-art RISK based supercomputer the system tends to spend more time moving data than doing arithmetic. Good datalocality is the key word to minimize this overhead. With the Non-Uniform Memory Access (NUMA) architecture of the Cray Origin 2000, an extra layer of complexity was introduced in optimizing the datalayout. To aid the user in his optimization effort new tools have been introduced. In particular new datadistribution directives have been added to the languages, and some environment variables guiding the system have been introduced.

In this paper we report on some early experience with the new directives, in the context of Fortran90 (F90), where similar directives allready exists in High Performance Fortran (HPF). A dataparallel extension to F90.

1.2. Data parallel programming vs. SMP-programming. The data parallel programming model became a popular programming model on distributed memory machines a decade ago. Its main advantages is the ease of programming (as opposed to message passing) and good locality of data which minimize the need for communication.

One problem for dataparallel programming was the lack of standards. With the appearance of HPF this shortcoming has been repaired. In short, HPF is F90 + directives, with the addition of a small number of intrinsic functions. The key to efficient parallelization is the directives. In the spirit of the dataparallel programming model the key directives in HPF is the directives for distributing the data. One distributes the data and gives the different processors ownership to a chunk of data. The computation is than carried out by *the owner computes rule.* i.e. the processor owning the data to be updated computes the new values.

The typical SMP-directives, as we known them from SGI and Cray systems, are there to help the compiler identifying data independence, making it possible to create multiple threads to be executed in parallel. With a shared ownership of the data the programmer focus on distributing the work, the data are fetched by the processors as needed. For complicated data structures and codes where the data access pattern changes repeatedly, this is a more flexible programming model than the dataparallel one.

On a UMA-machine, distributing the work to processors is really all that make sense, as data (in theory) is equally accessible by all processors. (Yes, we know about caches

* http://www.ii.uib.no/$\sim$ tors [or $\sim$ arild]

and false sharing, limited bandwidth of the interconnecting bus and so forth, but for a while we retreat to the perfect world of theory). On a NUMA architecture as the Origin 2000, with data access time depending on location, the need for controlling data layout in memory becomes apparent in SMP-programming too. This is acknowledged by SGI which have introduced data distribution directives to their parallelizing compilers with their new S2MP architecture. These directives are, in spirit, similar to their HPF counterparts, but differences do exists.

In Section 2 of this paper we discuss the difference and similarity between the data distribution directive of SGI's Fortran and HPF. In Section 3 we describe the default data layout on the Origin 2000 and report on some problems related to F90 on the Origin 2000. While we in Section 4 provide some experimental evidence to the effect of different strategies for data placement on some examples.

2. Distributing data and work in HPF and Pf90.

2.1. High Performance Fortran(HPF). In HPF [Hig94, KLS+94] the data structures which are considered parallel objects, and thereby subject for distribution, are arrays.

Directives are inserted in the code to advise the compiler on how to allocate arrays onto (abstract) machine memories. The simplest possible way of distributing an array is given in Example 2.1.

EXAMPLE 2.1. *Example of a simple distribution statement*

```
      INTEGER, DIMENSION :: FOO
!HPF$ DISTRIBUTE FOO(BLOCK)
```

For large, complex code a two step procedure is recommanded. First arrays are aligned relative to each other and next distributed onto a template. If needed, the logical layout of the processors may also be specified in an `ONTO` clause.

EXAMPLE 2.2. *Example of a more advanced mapping*

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) ONTO (NX,NY) ::  MYGRID
!HPF$ ALIGN X, Y WITH MYGRID
```

It is also possible to postpone the distribution until execution, using the `DYNAMIC` directive, and redistributing data at runtime using the `REDISTRIBUTE` directive.

Two different partitioning of data are possible. Either contigous block of data (`BLOCK`) or a round robin, element by element distribution (`CYCLIC`) (The two can be combined in `CYCLIC<k>` which tells the compiler to distribute contigous blocks of size `k` cyclic). For multidimensional arrays one might mix `BLOCK`, `CYCLIC` or no distrubition in the different dimensions.

The data parallel operations in HPF are the array assignments, intrinsics operating on arrays and the `FORALL` construct with the `PURE` attribute. Further parallelism might be obtained by adding the `INDEPENDENT` directive, which is an assertion about the parallelism in `DO` loops.

Building on the rich set of intrinsics in Fortran 90, HPF defines a library module containing 51 new intrinsic procedures operating on distributed data. Should the user need to go beyond HPF to another language or programming model, HPF defines a labeling mechanism called *extrinsic procedures*, which gives the user a possibilty to define a sound interface to other programming languages.

Mapping objects in HPF becomes slightly more complex when taking procedures into account. The directives for mapping dummy arguments have the same functionality as for ordinary objects, but with additional features, giving the user some flexibility in expressing the mapping and helping the compiler to create more efficient code. (Some would also say that this flexibilty aids to the confusion of the compiler.)

We note in passing that HPF has a mechanism for letting dummy argument of a procedure inherit the distribution of the actual argument. The Power Pro compilers also allow this, but the mechanism doing it is quite different. We have not tested this features, but believe that the HPF mechanism is the most flexible and the most expensive one (as it requires runtime handling of the datalayout).

2.2. SGI MIPSpro POWER Fortran 90 (Pf90). The Fortran Compilers from SGI has the KAP optimizer [Ass] as an instrumental part of the compiler. (The KAP optimizer is a automatic code restructuring tool, focusing heavily on Fortran (77 not 90) based loop i optimization techniques. In the SGI vocabular it is called *The Power Fortran Accelarator(PFA)* [SG]). The KAP optimizer is found on other platforms, such as DEC Alpha, SUN Ultra and IBM. In addition the SGI compiler has a switch, `-mp`, enabling the user to explicitly control the parallelization by using directives.

Whether the programmer takes control himself or leaves it to the PFA, the focus is on loop-parallelism. Do-loops are analysed for datadependency and, if no such exist, multiple threads are created and the loop are executed in parallel. The key directive is `DOACROSS`. A number of useful clauses to this directive exists. `LOCAL` which allows each thread to produce a private copy of otherwise shared data. `RECURSIVE` tells the compiler that the recursion in the following loop is a global operation like sum or max of all elements in an array, which might be parallelized if properly restructured. (These loops can nowadays be replace with call to F90 intrinsic, which PFA parallelize nicely).

A number of other directives also exist in PFA, some of them are meant to help the compiler optimizing sequential code, while other, suchs as `CONCURRENTIZE, ASSERT PERMUTATION(`*name*`)` and `ASSERT CONCURRENT CALL` aims at assisting the compiler with parallelization.

Together these directives gives richer possibilities to uncover and create parallelism than the `INDEPENDENT` directive and the parallel execution of array statement of HPF do.

On the Origin 2000 the system does its best to allocate data close to where it will be used. Without the full information of the problem it is impossible to guarantee an optimal data layout, and some approximation algorithm have to be used. The Origin applies the "*first touch*" policy. Under this policy the process which first touches a page of memory will allocate the page to the node where the process is running. This policy works fine for sequential programs and parallel programs using message passing. In the case where initialization is done sequential it might create a serious memory bottleneck as all processors might have to access the memory of only one node.

Reasontly datadistribution directives are also being supplied by the Mips PRO compilers. These are:

```
DISTRIBUTE (block, cyclic(k) or *)
REDISTRIBUTE (with the DYNAMIC attribute)
DISTRIBUTE_RESHAPE (data distribution with reshaping)
PAGE_PLACE (address - explicit placement of data in a node)
```

The `DISTRIBUTE` and `REDISTRIBUTE` directives has almost the same functionality and syntax as the corresponding directives in HPF. As in HPF they also have the `ONTO` clause.

An important difference is however that while in HPF `DISTRIBUTE` works on element by element, the smallest garnularity in Pf90 is a page (16KB). This means that a page containg blocks belonging to different processors will be stored in the meory of a single node. Thus nonlocal access to some of the data may be required. This problem might be mitigated using the `DISTRIBUTE_RESHAPE` directives which does not have the page granularity constraint. `DISTRIBUTE_RESHAPE` does however have a more restricted use than `DISTRIBUTE`.

`ALIGN` and `TEMPLATE` is missing in Pf90, making it more laborous and error prone to change the mapping of arrays which need an internal coherent mapping.

On the Origin 2000 we are still in the shared memory world. Allthough the data might be physically distributed there is shared ownership to them and no such thing as "*the owner computes rule*". Consequently, with distributed data and multiple threads there is no garanty that the CPU closest to the data runs the thread assosiated with the data. The user might however control this by creating an affinity between thread and data. This is done by adding a new `AFFINITY` clause to the `DOACROSS` directive.

```
!$distribute a(*,block)
!$doacross, affinity (j) = data(a(i,j))
            do i = 1,n
               do j = 1,n
                  a(i,j) = b(i+1,j) + b(i-1,j)
               enddo
            enddo
```

Note that this example is written in F77-style. When rewriting it in F90-array style, the do-loops disappear. Seeing no do-loop the compiler tells you that you have a syntax error when seeing the `DOACROSS`-directive (sic!)

3. Some POWER Pro F90 features.

3.1. Parallelizing array statement. The basic parallel statements in Fortran 90 are array statements like

```
A = B+C
A = sin (B)
etc, etc
```

where `A, B` and `C` are arrays. These are obvious parallel statements and are treated as such by `pfa`. If, however, one try to give further assistance to the compiler by using, for example, the `AFFINITY` clause. The syntax require a `DOACROSS AFFINITY(i) = ...` This is not accepted by the compiler as it doesn't find a `DO-loop`.

Explicite parallelization with `-mp` does for the same reason not work at all on nicely written F90 code where array statements are extensively used.

3.2. A Case of datadependency. The following piece of code did not parallelize with the F90 compiler.

```
do k=1,ITERATIONS
  new = mod(k,2)
  old = mod(k+1,2)
  a(1:N,1:N,new)= -0.25d0*(CONST(1:N,1:N)-a(1:N,0:N-1,old)  &
        -a(0:N-1,1:N,old)-a(2:N+1,1:N,old)-a(1:N,2:N+1,old))
enddo
```

With the array statement replaced by a double do-loop and compiled with `f77 -pfa` it did parallelize.

We unrolled the loop and set `new = 0; old =1`, then it not only parallelize, but run faster in all cases.

3.3. The `NEST` clause. With multidimensional nesting it often make sence to parallelize over more than just the outer loop. This can be achieved adding the `NEST` clause to the `DOACROSS` directive, as in

```
!$distribute a(block,block)
!$doacross, nest(i,j), affinity (i,j) = data(a(i,j))
            do i = 1,n
               do j = 1,n
                  a(i,j) = b(i+1,j) + b(i-1,j)
               enddo
            enddo
```

On the betaversion of the F90 compiler we used this did not work. We where allowed to add the `NEST` clause, but not together with the `AFFINITY`, which is what it takes to get the affinity between datalayout and threads.

4. Experiments.

4.1. Our test example. As a testproblem we discretize the Poisson's equation with the standard 5 point centrale differences stensil, and solving it by the Jacobi iteration. The data are stored in the matrix A(0:N+1,0:N+1,0:1), where the 3'rd index represent the values of two consequetive iteration step, alternating between 0 and 1 being the previous and the latest iteration step, respectively. Boundary values are stored at A(0,:,:), A(N+1,:,:), A(:,0,:) and A(:,N+1,:) while the unknowns are stored in the interior of the array.

In the Jacobi iteration the unknowns are in each step updated by the neighboring points. As always when communication is local, block distribution should be prefered before cyclic. To get good cache performance on Fortran code one needs to distribute the data column wise. Thus the obvious choice for datadistribution is:

```
!$distribute a(*,block)        (in Pf90)
!HPF$distribute a(*,block)        (in HPF)
```

Before the iteration starts the boundary values are computed by:

```
do i = 0,N+1
   A(0,i) = g(0,i*h)
   A(N+1,i) = g(1,i*h)
   A(i,0) = g(i*h,0)
   A(i,N+1) = g(i*h,1)
enddo
```

If no datadistribution is used, the *first-touch* rule is used and cause the initialization to create a none optimal data layout.

We have carried out some experiments on our Origin 2000. Running Irix 6.4. As the F90 compiler we used a beta release of the new MIPSpro 7.2 compiler. The HPF compiler we used was PGI's HPF 2.2 for SGI-platforms.

4.2. Comparing the different Compilers. Below we show the effect of the `DISTRIBUTE` directive on F77, F90 and HPF compiler. For code written in F77-style, we also had the opportunity of adding the `AFFINITY` clause.

	F90-style code		F77-style code		
Nproc	No dir.	`DISTRIBUTE`	No dir.	`DISTRIBUTE`	`DISTRIBUTE + AFFINITY`
2	30.3	28.9	35.2	39.5	22.6
4	15.3	14.7	16.6	18.4	11.4
8	7.7	7.5	8.3	8.8	5.8?
16	3.5	3.4	4.2	4.3	3.4
32	2.8	2.0	2.2	2.3	1.8

FIG. 1. ***MIPSpro F90 Compiler, version 7.20.*** *Time (in sec) for evaluating 100 Jacobi iteration of* 2048×2048 *unknown. Only -O3 -mips4 -R10000 -IPA has been used for optimization.*

The F90 compiler seems to prefer F90-array syntax to the old fashioned F77 double do-loop. Thus it is really unfortunate that the `AFFINITY` clause is prohibited while using array syntax.

Nproc	No dir.	`DISTRIBUTE`	`DISTRIBUTE + AFFINITY`
2	31.7	31.6	19.4
4	14.4	13.8	9.7
8	7.2	7.0	5.9
16	3.8	3.6	2.6
32	1.9	1.9	1.3

FIG. 2. ***MIPSpro F77 Compiler, version 7.20.*** *Time (in sec) for evaluating 100 Jacobi iteration of* 2048×2048 *unknown. Only -O3 -mips4 -R10000 -IPA has been used for optimization*

These tests provides a convinsing argument for the use of `DISTRIBUTE + AFFINITY`. Comparing the F77 with the F90 results we see that the F90 compiler does a much better job in automatically placing the data correctly when dealing with array syntax.

PGI's HPF compiler is a source to source compiler. The version we are using translate the user code into F77+message passing. The message passing was done using SGI's own MP_library. We found surpricingly great differences in speed depending on the F77-compiler used, and unfortunately the fastest execution was obtained when compiling with the old F77-compiler. For comparision we also display the speed of the same computation done using MPI [SOH+96, GL94].

Nproc	HPF		MPI	
	F77 7.20	F77 7.1(6.2)	F77 7.2	F77 7.1(6.2)
2	70.0	35.0	59.6	55.9
4	35.0	18.0	21.8	27.9
8	17.6	8.7	13.7	12.7
16	8.6	4.2	6.2	5.8
32	4.4	1.7	2.1	2.4

FIG. 3. ***PGI's HPF Compiler for Origin 2000.*** *Time (in sec) for evaluating 100 Jacobi iteration of* 2048×2048 *unknown. Only -O3 -mips4 -R10000 -IPA has been used for optimization*

4.3. The effect of Blocking Strategy. Slicing the data in strips forces all processors, working on interior blocks, to fetch 2 vectors of length N from non-local memory. If the data was distributed `DISTRIBUTE A(BLOCK,BLOCK)` onto a $P_1 \times P_2$ array of processors, a processor holding an interior block would only have to fetch $2N/P_1 + 2N/P_2$ data from its 4 neighbors. For large number of processors this might be significantly less than the $2N$ need in the `DISTRIBUTE A(*,BLOCK)` distribution. On the other hand, `(BLOCK,BLOCK)` will force us to access row in the matrix an a potenial bad cache behavior. In the next table we display some timings for different BLOCKING-schemes.

Blocking strategy	HPF	F90
`(BLOCK,BLOCK)` (default)	1.08	
`(BLOCK,BLOCK) ONTO (1,48)`	2.60	1.24
`(BLOCK,BLOCK) ONTO (2,24)`	0.96	1.24
`(BLOCK,BLOCK) ONTO (3,16)`	1.64	1.24
`(BLOCK,BLOCK) ONTO (4,12)`	1.29	1.18
`(BLOCK,BLOCK) ONTO (6,8)`	1.55	1.14
`(BLOCK,BLOCK) ONTO (8,6)`	1.71	1.22
`(BLOCK,BLOCK) ONTO (12,4)`	1.58	1.22
`(BLOCK,BLOCK) ONTO (16,3)`	2.03	1.19
`(BLOCK,BLOCK) ONTO (24,2)`	0.96	1.24
`(BLOCK,BLOCK) ONTO (48,1)`	1.49	1.19
`(*,BLOCK) ONTO (48)`	1.74	1.21
`(*,BLOCK)`	1.76	1.18
`(BLOCK,*)`	1.71	1.18

FIG. 4. ***Comparing the effect of ONTO.*** *Time (in sec) for evaluating 100 Jacobi iteration of* 2048×2048 *unknown. Only -O3 -mips4 -R10000 -IPA has been used for optimization*

4.4. System Scheduling. An alternative to explicitely distribute the data by using directives in the code is assisting the system by setting environment variables. Two such exists, `_DSM_ROUND_ROBIN` and `_DSM_MIGRATION_ON`.

`_DSM_ROUND_ROBIN` tells the system to void its *first-touch* policy and instead distribute the data, page-by-page, in a round-robin fashion This correspond to the `CYCLIC` distribution described above.

_DSM_MIGRATION_ON enables automatic page migration. This might be of great help in cases where the same computation is carried out over and over again in a long iteration loop. _DSM_MIGRATION_ON comes in different flavors. One might specify whether or not all data is subject to migration or if only those not explicitly distributed by directives should migrate.

Migration is of course not free of cost. Thus one will always have to balance the benifite against the cost. For this reason the user is allowed to set the the migration level. A high level means "Do whatever it takes to get the dataplacement right, and do it NOW!". Level = 0 means no datamigration.

Here is the effect we obtained on our F90-Jacobi code

Environment setting	Time (in sec)
No env. variables	4.06
`%setenv _DSM_ROUND_ROBIN`	5.60
`%setenv _DSM_MIGRATION_ON`	3.42
both of the above	5.35
`%setenv _DSM_MIGRATION_LEVEL 10`	3.26
`%setenv _DSM_MIGRATION_LEVEL 50`	3.44
`%setenv _DSM_MIGRATION_LEVEL 90`	4.29

FIG. 5. ***The effect of system environment variables.*** *Time (in sec) for evaluating 100 Jacobi iteration of* 2048×2048 *unknown on 16 processors.*

As might be expected a round-robin distribution of data is not the best datadistribution for this problem. Being an iteration with exactly the same access pattern in each iteration we do expect the migration to help, and it does. It does, however, not pay off to do agressively migration. This seems reasonable. The access pattern is simple and exactly the same for each iteration. Thus the system get it rigth in the first place. Trying harder is not necessary and does only incure system overhead.

5. Conclusions. Allthough the extra latency for fetching data from remote memory is low on the Origin 2000, there is a lot to gain from good datalocality. If all processors reading and writing to local memory, we not only get the lowest latency, we also do not have to worry about congestions in the network.

To achieve the nearness of process and data locality, the data distribution is not enough, we need the affinity between process and data. There must be possible to express this F90. This also goes for the nested do.

For the ease of use SGI should aim at offering only one FORTRAN-compiler. That compiler should be F95 (which will incorporate F77) with the necessary extensions for HPF.

REFERENCES

[Ass] Kuck & Assoc. *http://www.kai.com.*

[GL94] William Gropp and Rusty Lusk. *Using MPI.* The MIT Press, 1994.

[Hig94] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.1. Technical Report CRPC-TR9222X, Center for Research on Parallel Computation, Rice University, 1994.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[SG] Inc. Silicon Graphics. *MIPSpro POWER Fortran 90 Programmer's Guide.* Silicon Graphics, Inc., mipspro power fortran 90 version 6.2 edition.

[SOH+96] Marc Snir, Steve W. Otto, Steven Huss-Ledermann, David W. Walker, and Jack Dongarra. *MPI: The Complete reference.* The MIT Press, 1996.