Parallelization of a Local Area Ocean Model

Luka Onesti and Tor Sørevik

Para//ab, University of Bergen

Abstract. In this paper we describe the parallelization of a local area ocean circulation model using HPF. We find that this particular problem, leading to a typical data parallel algorithm is easily expressed in HPF. The problems which occurs is mainly due to restrictions in the current HPF-compilers.
We have run our program on 3 different platforms, using different HPF-compilers. and will report on the status of these compilers as measured by our problem.

1 Introduction

Parallel computing has proved to offer the computational power it claimed a decade ago, and today all computers on the TOP500 list are parallel computers. In spite of the enormous performance gains offered by parallel computers there is still some reluctance among application programmers to use these. The reason for this is the huge effort needed to port large application programs to parallel platforms. In particular this is true when using the message passing programming paradigm for distributed memory computers.

High performance Fortran (HPF) [Hig] [KLS$^+$94] is an attempt to meet the need for easier parallel programming, removing the burden of figuring out the communication pattern of data from the programmer to the compiler. HPF is built upon FORTRAN 90 (F90) [ofSC91] by adding directives and a few new language features and intrinsics to F90. Being close to sequential F90 programming an additional advantage of HPF/F90 would be the need to maintain only one code for sequential as well as parallel programming.

While HPF in theory offer an easy way to parallel programming., life is somewhat different i practice. In particular, the present stage the HPF-compilers available to us has certain shortcomings. In part they do not implement the full HPF (or F90) standard, nor do they agree on what not to implement, which make portability difficult.

Being a much richer language than F77, F90 gives, in many cases, multiple choices for how to express complex statements. Although being semantic equivalent the different syntactical variants sometimes gave huge and unexpected performance differences. Moreover, we found that although an HPF code is portable across a variety of platforms, it is not necessarily performance portable. We contribute these obstacles to the shortcomings of current HPF (and F90) compilers, and expect these to improve in the future.

In this paper we describe the parallelization, using HPF, of a local area ocean model, developed jointly between the Univ. of Bergen and the Norwegian Marine Research Institute [BES96].

After a brief description of the model in section 2, we introduce and discuss the F90 and HPF features necessary for parallel implementation of this application in section 3. In section 4 we discuss the data mapping. In section 5 we present our numerical result and discuss them.

2 The Ocean Model

The governing equations of the model is the standard equations for continuity, momentum and conservations, with the modifying hydrostatic assumption that the weight of the fluid identically balances the pressure, and the Boussinesq approximation that density differences are neglected unless the differences are multiplied by gravity. Motions induced by small scale processes are parametrized by horizontal and vertical eddy viscosity/diffusivity terms.

Even with these assumptions the equations constitute a complicated coupled set of non-linear time dependent PDEs in three spatial dimensions.

The model is a local area model as opposed to a global model. Thus the vertical boundaries are in part fixed (over land) and open (over sea). Fluxes at the open boundaries are taken from a global model. The horizontal boundaries impose von Neuman boundary conditions on the vertical velocity.

The spatial variables is discretized by central finite differences, using a staggered C-grid. To take care of the irregularity imposed by bottom topography the σ-coordinate system is used. This is a linear transformation of the depth such that all gridpoints, with the same vertical coordinate, have the same relative position between the surface and the bottom. In each horizontal layer a cartezian grid with rectangular shape was used. The irregular shape of the coastline implies that we include some onshore area by this discretization. All variables are defined here too, but have, of course, no meaningful physical interpretation. Computation over these gridpoints can easily be omitted by masking.

With this spatial discretization all data is stored in regular 2 or 3-dimensional arrays, corresponding to their physical displacement.

To advance the solution in time an operator splitting technique is used. The effect of each term in the governing equations is computed separately and added up to give the total contribution. Gravity waves are computed by an spatial split implicit method in each horizontal layer. Thus one have to solve multiple, independent tridiagonal linear system in each timestep. All other terms are propagated in time by explicit timestepping. At boundary points data for central differences is not available, thus these points are taken care of by special code segments.

3 The HPF implementation

3.1 Platforms used

Available to us for this project were three different HPF-compilers, running on different hardware platforms. Below we give a brief description of the compilers and their related hardware.

The bulk of the porting and the initial testing was done on Digital's proprietary F90/HPF compiler, version 4.0. This is an integrated compiler where the user determine by a switch whether he wants to compile for sequential execution (=F90) or parallel execution (=HPF). Digital has been a forerunner in developing a full F90-compiler, and that part of the compiler appear to have reached a mature state. This software was run on a cluster of 8 DEC workstations, each equiped with a 233 Mhz alpha processor, connected with an fddi-switch, DEC's own Gigaswitch. The programs were run under Digital's PSE (Parallel Software Environment), which enables you to use the cluster with the same ease as any "all-in-one-box" MPP system.

We've also used Portland Group (PGI)'s portable compiler. This compiler compiles HPF code into f77 + message passing. The message passing layered used can be adapted to the hardware and software platform available. We have run version 2.0 of the compiler on a 100 processors Intel Paragon using Intel's nx as the message passing layer.

The third compiler we have used is IBM's proprietary HPF compiler version 1.0 running on an IBM SP2. This compiler is, from the user perspective, quite similar to DEC's. The HPF parts are the parallel extension to the F90-compiler.

3.2 Language features used

F90 provides an array syntax for matrix and vector operations. In addition to its beauty and convenience, this notation also provides compiler writers with good opportunities for optimization and parallelization. Whenever possible this notation has been used in the present code.

Other features, new to FORTRAN with the F90-standard, used in this code are: `MODULE` used to implement global variables in place of `COMMON`, `INTERFACE` used to assist the compiler with interprocedural analysis, `WHERE` used to mask out array elements, not needed to be updated in array statements. We also experimented with dynamic memory allocation, but this lead to a serious performance degradation, and was for that reason abandoned. The `FORALL` statement of HPF was also used.

A complicating feature of F90 is that it does not allow nested `WHERE`. This forced us to rewrite the code, throwing complicated, concatenated, logical statement into one `WHERE` construct. This did not only make the code pretty ugly, but turned out to be a tough case for the compiler. It tended to perform badly in these cases. An alternative to `WHERE` is to use `FORALL` with a masking statement, but on the DEC-compiler this had even worse performance characteristics. The reason seems to be that when a masking array is found within a `FORALL` statement the compiler by default choose to replicate the masking array on all processors. If the masking array is aligned with the other distributed arrays this impose a enormous data traffic and we experience slow down when using more than one processor.

Communication imposed when distributing data to multiple processors is only nearest neighbor. Consequently the data was distributed `BLOCK`-wise. A `TEMPLATE` was used to defined the data distribution and all 2- or 3- dimensional arrays was align with this. Global arrays, defined in `MODULE`, were given their distribution here. When the MODULE was incorporated in a program segment with the `USE` statement, the data distribution automaticly followed. This strategy works fine on the DEC-platform, but although the `MODULE` feature is supposed to be supported in the last version of the Intel HPF compiler, we were not able to use it as indicated above. On the Paragon we therefore arranged the global arrays in traditional common statement and collected these in an `INCLUDE`-file, which also contained the HPF-attribute statements and the template declaration as well.

Local arrays were always aligned with a global array. When a local array was passed to a subroutine through a dummy argument, we intended to use the `INHERIT` statement to make sure the already defined data distribution was inherit. This approach gave, however, bad performance on both platforms. We found the most efficient implementation to be using the descriptive `ALIGN` directive, in which we declared the dimension of the arrays explicitly and then aligned them to a template and passed it to the subroutine using module (or common in the case of the Intel Paragon). We believe the difference to be that the runtime calculation of indexes is no longer necessary.

We also tried using `ASSUMED`-sized array. This works, but slow, and was for that reason abandoned in the final version.

We also took advantage of some of the new array inquiry function of F90, like `MAXVAL, ANY` and `MAX`. Using these was necessary to get good parallel performance on multiple processors. The standard F77-code with an `IF`-test within a set of nested `DO`-loops, was always executed sequential.

Features that is not yet implemented or does not yet work properly is the `INDEPENDENT` statement of `HPF`. This is the directive needed to tell the compiler that `F77 DO`-loops are parallel. Without this directive one are forced to change all code into `F90`-array syntax and/or using `FORALL`. We also had problems with the PURE-attribute. Either it didn't work (on the PGI-compiler) or it was not used to parallelize the code.

3.3 Code rewriting

The procedure we followed in the porting procedure was the obvious one. First rewriting the code to F90 and testing this version for correctness and efficiency on a sequential platform, and than introducing the HPF-compiler directives for parallelization. The great advantage of this is the incremental nature of this step by step procedure. As expected the bulk of the work is related to doing the translation to F90. The sometimes unexpected performance behavior also created some headache.

Some times substantial rewriting was needed to better express the true data parallel nature of the problem. In particular code like:

```
DO I = 1, N
   TMP1 = ....
   TMP2 = ....
   CALL FOO (A(I), TMP1, TMP2)
END DO
```

was expanded to

```
FORALL (I = 1:N)
       TMP1(I) = ...
       TMP2(I) = ...
END FORALL
CALL FOO (A, TMP1, TMP2)
```

where FOO was rewritten to operate on arrays instead of scalars. In particular this was important for solving multiple tridiagonal systems in parallel.

Note that this rewriting expands temporary variables to arrays instead of scalars and thereby increasing the memory requirements. We experience that the compilers we used also introduced this kind of temporary arrays in their intermediate code.

4 Data distribution

The scheme for computing the vertical mixing leads to $N_x \times N_y$ independent tridiagonal linear equation systems in each horizontal grid point. For this part of the code to be solved efficiently in parallel, all vertical gridpoints in the same horizontal cell have to remain at the same processor. This gives a straightforward parallelization of the vertical mixing.

Except for the gravity waves all timestepping are done by a simple explicit scheme, independently for the different vertical layers. Thus a simple block partition in one or two of the x- and y- direction imposes a simple and regular nearest neighbor communication, which has proved efficient in many similar cases.

On a $N_x \times N_y \times N_z$ grid the amount of data needed to be sent in each step, if divided in the y-direction is:

$$Comm_{1d} = 2\alpha + 2\beta N_x N_z \tag{1}$$

similar if divided in the x-direction. A 2d- partition on $P \times P$ processors gives

$$Comm_{2d} = 4\alpha + 2\beta(N_x N_z + N_y N_z)/P \tag{2}$$

Here α is the latency and β represents the inverse of the bandwidth for a give system.

As is evident from these formulas a 2d partitions is always best for a large number of processors. For less than 4 processors a 1d partion parallel to the axis with fewest gridpoint is the best. Exactly where the crossover point is depends on the system (α and β) and the problem size.

The simple model above does not take into account the cost of data access. In FORTRAN data in multidimensional arrays is stored in column major order. Thus to send a column of a matrix, one need to retrieve and store data which is stored consecutively in memory. While accessing data in an array segment imposes stride in memory. For this reason we do expect a higher crossover point between 1-d and 2-d distribution, than predicted by the formula above.

The only nonobvious data distribution occurs when solving the multiple tridiagonal systems of equations for the implicit timestepping of the gravity waves. The operations are by themselves "embarassingly parallel", but the data needed to construct the coefficient matrix of each individual tridiagonal system is distributed across the processors. The coefficients for each tridiagonal system is, as usually, stored in 4 vectors The coefficients for all the systems is kept in 4 2-d arrays. To solve one specific tridiagonal system for splitting in "X-direction" one has to traverse the corresponding columns of these matrices. This can be done without communication if the coefficient matrices are distributed `(*,BLOCK)`. When solving in "Y-direction" one need to traverse rows of the matrices, consequently the matrices should be distributed `(BLOCK,*)`. These matrices are constructed using data involved in other parts of the computation which might, as described above, be distributed in either of these ways. The important thing is, however, that for the reminding computation to execute efficiently all these arrays have to be aligned. Thus at least one of the two sets of coefficients are not aligned with the data it is constructed from, and thereby imposing communication.

5 Numerical Experiments

All experiments reported here are based on the full application code or part of it. The grid size we used was $125 \times 113 \times 12$. The work associated with a gridpoint depends on whether or not it is over land or sea. When distributing the gridpoints the current HPF standard provides no mechanism for taking the differences in workload of the gridpoints into account.

5.1 Syntax dependent Performance

In the explicit timestepping procedure a number of array assignment is used. Typically we need to handle boundary points separately. Moreover for efficiency reasons we will like to mask out "on-shore" gridpoints. F90 provides different ways of expressing this and we tried some of these.

All these variants have the option of replacing the masking with an appropriate placed multiplication with zero. This does, however, require careful implementation to avoid division by zero. On the Digital platform we found that the `WHERE` construct got serious performance degradation when the masking statement become complicated. The `FORALL` statement was for all practical purposes impossible to use on more than one processor, due to the need for replicating masking arrays. Wherever possible we therefore replaced masking with multiplication with zero in Figure 1 This was not possible everywhere, hence variant 3 outperformed the two other constructs.

In table 2 we summerize how the different flavors of the code perform sequential on the DEC alpha and on the i860 processor of the Paragon

On the DEC-cluster all versions suffer a severe loss in performance as soon as we move to parallel execution, (See Figure **??**). In particular this is true for the `FORALL`-version. This is due to the (unnecessary) replication of the masking array. The relatively poor performance of the interconnecting network is also a part of the explanation.

All versions show a very good speed-up on the Paragon up to 32 nodes, (See Figure **??**). Only the communication free, tridiagonal solver scales beyond this. With a 1-d data distribution and the current resolution this is as expected. We believe the superlinear speed-up when moving from 1 to 2 processor is due to memory access pattern.

On the SP2 the `WHERE`-construct shows decent speed-up, while the `FORALL`-construct should be avoid.

5.2 Which Blocking to choose?

As discussed in section 4 the choice between 1-dimensional and 2-dimensional distribution is not obvious for moderate number of processor. Moreover when

1. `FORALL` with masking as in:

   ```
   FORALL (I = 2:IM,J = 2:JM-1,K = 1:KB-1,DUM(I,J).NE.0.)
          XFLUXUP(I,J,K) = AREATU1(I,J,K)*FIELD(I-1,J,K)
          XFLUXLW(I,J,K) = ....
          .........
   END FORALL
   ```

2. Array-syntax and the `WHERE`-statement as in the included example. In this case the masking array must have the same shape as the arrays it applies to. Thus one either have to expand `DUM` with a third dimension on the expenses of extra storage or wrapping the `WHERE`-construct in a `DO`-loop keeping the third index of `XFLUXUP`, `XFLUXLW, ...` fixed.

   ```
   DO K = 1, KB-1
   WHERE (DUM(2:IM,2:JM-1).NE.0.)
         XFLUXUP(2:IM,2:JM-1,K) = AREATU1(2:IM,2:JM-1,K)) &
                *FIELD(1:IM-1,2:JM-1,K))
         XFLUXLW(2:IM,2:JM-1,K) = ....
         .........
   END WHERE
   END DO
   ```

3. `FORALL` without masking, but call to a pure function as in:

   ```
   PURE FUNCTION SUPERBEEF_X(...,SCALAR1, SCALAR2, DUM,...)
   ..................
   FORALL (I = 2:IM,J = 2:JM-1,K=1:KB-1)
          XFLUX(I,J,K)=SUPERBEEF_X(...,FIELD(I,J,K),&
                       FIELD(I-1,J,K),DUM(I,J),...)
   ```

   In this case the masking is done with a traditional `IF`-statement inside SUPERBEEF_X which only takes scalar variables. Here each processor runs a copy of the sequential routine `SUPERBEEF_X` with the data associated with the grid point it is responsible for.

Fig. 1. Different syntax for expressing masking on arrays

	DEC's HPF alpha cluster	PGI's HPF Paragon	IBM's HPF SP-2
Forall+pure function	.52	Doesn't work	4.83
Forall with masking	1.58	6.27	2.73
Where with 2d masking	1.57	10.02	1.82
Where with 3d masking	-	10.22	1.50

Fig. 2. *Single node performance in seconds of the different version of the code-segment displayed in Figure 1*

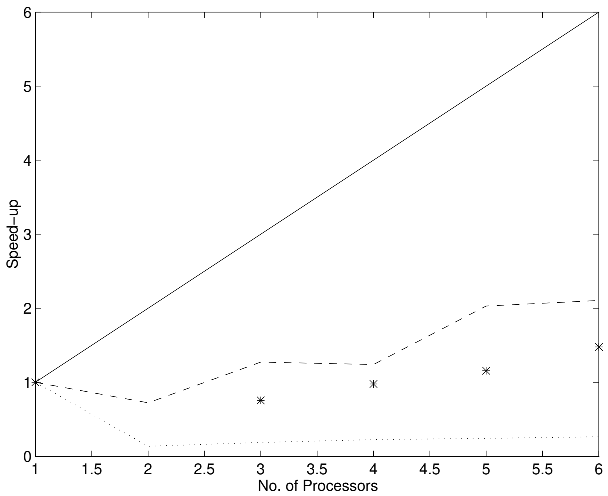

Fig. 3. *The straight line represent linear speed-up. The '*' is `FORALL` with `PURE`-function, the dashed line is `WHERE` and dotted line is the `FORALL`-version*

the prime factorization of the processor number contains more than 2 primes there is multiple choices for 2-dimensional distribution. This choice could be left to the compiler, or made explicit by the programmer by using the following form of the distribution directive:

```
!HPF$ DISTRIBUTE A (BLOCK,BLOCK) ONTO (P,Q)
```

where (P,Q) are given parameters. To give an idea of how this effect the performance, we have run the entire code on 36 processor of the Intel Paragon with all possible `BLOCK` distribution. Parameters one expect to influence the performance is load imbalance and ratio between computation an communication. In our example these can be measured as the maximal number of grid point on one processor ($\lceil 125/P \rceil \times \lceil 113/Q \rceil \times 12$) and the amount of data to be communicated. This is proportional to the vertical surfaces of the "box" for individual processor. This should be $((\lceil 125/P \rceil + \lceil 113/Q \rceil) \times 2 \times 12)$ The result is given in figure 6.

As expected `(*,BLOCK)` is better than `(BLOCK,*)`, since passing a column instead of row means accessing consecutive data in memory. This effect seems to dominate the communication performance. The amount of data to be communicated and the load imbalance for computation indicated seems to have only minor effect on the efficiency.

When `ONTO` is not used the compiler does not have the number of processor available at compile time, thus some index calculation have to be deferred to runtime. Whether this explains the non-optimal performance of `(BLOCK,BLOCK,*)` without `ONTO` or the compiler simply choose a non-optimal factorization we don't know.

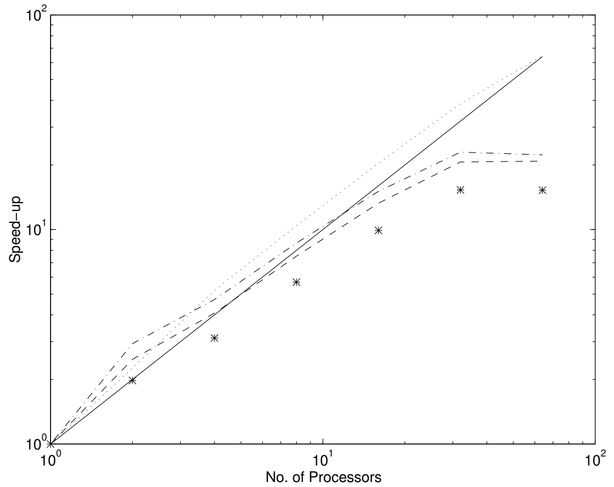

Fig. 4. *The straight line represent linear speed-up. The dash-dotted line is the forall-version, * is `WHERE` with 2d-masking and dash-dotted line is `WHERE` with 3d masking. The dotted line is the speed-up of the multiple tridiagonal solver.*

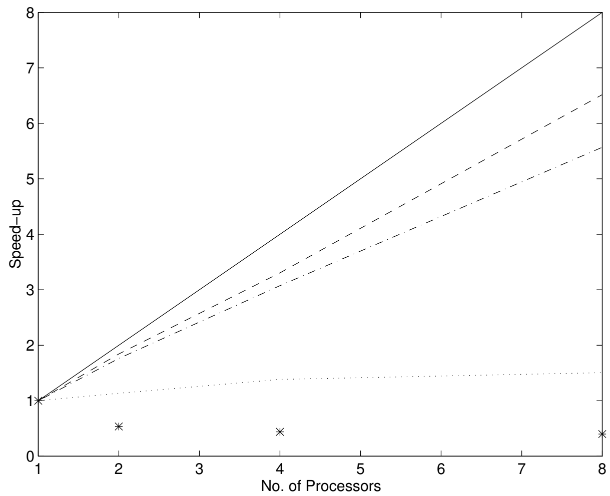

Fig. 5. *The straight line represent linear speed-up. The '*' is using `FORALL` with a `PURE`-function, The dashed line is `WHERE` with 2d-masking. The dotted line is the `FORALL`-version with masking. The dash-dotted line is `WHERE` with 3d-masking*

.

distribution	Max gridpoint	Data to send	Time in sec.
(*,BLOCK,*)	6000	3000	3.39
(BLOCK,*,*)	5425	2712	17.83
(BLOCK,BLOCK,*) ONTO (18,2)	4788	1536	7.03
(BLOCK,BLOCK,*) ONTO (12,3)	5016	1176	8.54
(BLOCK,BLOCK,*) ONTO (9,4)	4872	1032	7.01
(BLOCK,BLOCK,*) ONTO (6,6)	4536	960	3.99
(BLOCK,BLOCK,*) ONTO (4,9)	4992	1080	7.11
(BLOCK,BLOCK,*) ONTO (3,12)	5040	1248	8.65
(BLOCK,BLOCK,*) ONTO (2,18)	5292	1680	6.29
(BLOCK,BLOCK,*)			7.09

Fig. 6. *The elapsed time in seconds pr. timestep as a function of the distribution*

5.3 Scalability and Data distribution

On the basis of the above experiment we have picked the most efficient versions of the different subroutines and run the full scale application program on different systems, with different number of processor and with the (BLOCK,BLOCK,*) and (*,BLOCK,*) distribution. The result is presented in figure 7 below.

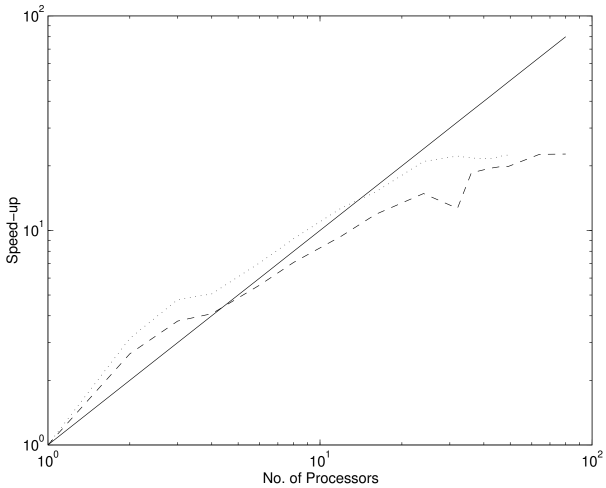

Fig. 7. *The Straight line represent linear speed-up. The dashed line is the BLOCK,BLOCK distribution while the dotted line is the (*,BLOCK) distribution*

As expected the 1-d distribution performs best on low to moderate number of processors, but it doesn't scale well. The 2-d distribution scales somewhat better.

6 Conclusions

This exercise has convinced us that HPF is a much easier road to parallel programming than message passing. The incremental way of porting large application code makes the debugging much easier. The advantage of having only one code to maintain is also huge. It does make meaningful to do almost all the developing work on your (sequential) local workstation, and only when finished move your code to the central (parallel) HPC-platform.

The reasons for the problems we occurred appears to be immature compilers. Bugs and missing features cost us lot time and frustrations. The quality of the compilers are however, rapidly increasing and we have seen great improvements over the months we worked on this project.

Also confusing and frustrating is the inconsistent performance behavior of the different compilers. Since performance is a key issue it is important that the code not only works when moved to another platform, but still runs efficiently. This is not the case now. Choosing the most efficient HPF-programming style for the platform you do the development work on, is by no means any guarantee that you have an efficient code when moving to another platform.

We do believe these differences will diminish as the vendors put more effort into optimizing their compilers. Some of the differences are due to the underlying sequential Fortran compilers and although FORTRAN 90 has been around for some years, not all vendors have put enough effort into optimizing their F90-compiler.

References

[BES96] J. Berntsen, T. O. Espelid, and M. D. Skogen. Description of a σ-coordinate ocean model. Technical Report ??, Department of Mathematics, University of Bergen, 1996.

[Hig] High Performance Fortran Forum Home Page. http://www.crpc.rice.edu/HPFF/home.html.

[KLS+94] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Perofrmance Fortran Handbook.* The MIT Press, Cambridge Massachusetts, 1994.

[ofSC91] International organization for Stadarization and International Electrotechnical Commission. Fortran 90, iso/iec 1539: 1991], 1991.

This article was processed using the LaTeX macro package with LLNCS style