Advanced user support: On the parallelization of a non hydrostatic, sigma co-ordinate ocean model

Helge Avlesen, Para//ab,
Bergen Center for Computational Science, University of Bergen,
Bergen

November 2004

Abstract

This report describes the preliminary results from the parallelization of a non-hydrostatic, sigma co-ordinate ocean model. The work builds upon a previous project in optimizing the non hydrostatic parts of the Bergen Ocean Model (BOM). The model has been parallelised through calls to a high level library, targeted at finite difference codes, implemented on top of MPI. The library is optimized for clusters with inexpensive, high latency interconnects, by offering means to accumulate message exchange operations. We also aimed to explore the use of a variant of the iterative scheme with reduced number of inner products in order to even further improve the scalability of the iterative algorithm on linux clusters.

1 Introduction

Numerical ocean modeling is steadily becoming more important in ocean engineering and management of marine resources. Most modern numerical ocean models still rely on physical simplifications of the model equations such as the hydrostatic shallow water approximation, in which the pressure of the model only depends on the depth. As computers have become faster, scientists steadily increase the resolution of their models and start to see limitations of such simplifications. Many models are therefore these days modernized by also including non-hydrostatic physics. The major downside to this enhancement is that the computational complexity increases dramatically. The great advantage of hydrostatic ocean models has been that it is possible to obtain a 3D solution, without the need to solve elliptic equations as part of the solution process. Since this is not the case for non-hydrostatic models, the need for methods to easily parallelize models arise.

This report follows up on a previous advanced user support project, where the non-hydrostatic part of the Bergen Ocean Model (BOM) was optimized and parallelized using shared memory parallelization (OpenMP directives). The conclusion of that effort was that a transition into message passing parallelisation was necessary to get better parallel efficiency and to tackle larger problems. This transition would also enable the use of the (non)hydrostatic model on the increasingly popular clusters.

This report describes how the code has been parallelized with calls to a high level library using MPI as the primitive message passing layer.

2 Parallelization

The library taking care of the communication, previously developed at Para//ab, is implemented on top of the Message Passing Interface library (MPI), which is the de facto standard for message passing parallelization these days. This parallelization technology is often referred to as Single Program Multiple Data (SPMD), which means that the same program is executed on all participating processors, but each processor works on its private and separate data. The conversion from a sequential program to SPMD is achieved by distributing the data of the program during initialization, and from then on exchange data/information as seldom as possible as the program executes. This is conceptually quite different from the previously used shared memory type of parallelization, where we instead of distributing data, distribute the work, and let each processor work on the same pool of data.

The library is written while having systems with high latency interconnects in mind, by trying to reduce the number of sends and receives wherever possible. Care has also been taken to make it possible to parallelize the code with as few changes to the sequential code as possible. Another advantage is that the use of preprocessing is avoided, improving readability of the code.

The library consists of a set of subroutines, of which most work on objects of the same type; a distributed 3D array. The distributed array is equivalent to a 3D array with global shape (IM,JM,KB), and is scattered onto a 2D topology with NUMBLI×NUMBLJ blocks. See Figure 1. Blocks can be marked inactive, and turned off, in ocean mod-

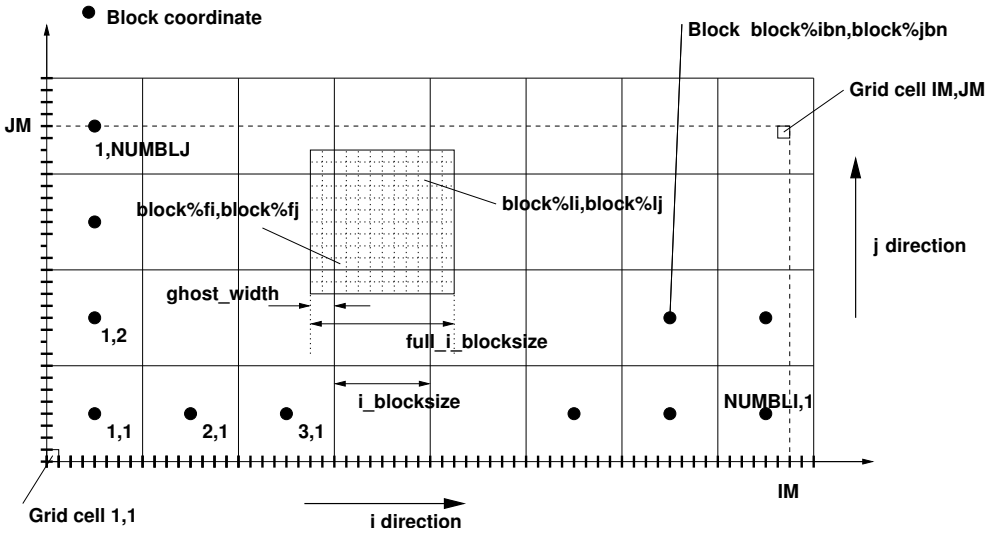

Figure 1: Partitioning the domain into blocks

els this will typically be areas with only land cells. All static library data structures are initialized by passing a 2D array, equal in shape to the first two dimensions of the distributed 3D arrays, as well as the desired number of blocks in the I and J direction to the subroutine BLOCK_SETUP. The 2D array will generally define the geometry in the first two dimensions of the problem, for ocean models the two first dimensions normally represent the horizontal directions. The sub arrays, or blocks, each have a data structure defining the location of the block in the global problem, with all necessary information of index ranges etc.

The most used operation in the SPMD parallel code is the exchange of halo data between processors. We observe in Figure 1 that all blocks also include a piece from each of the neighbour processes. The width of this stripe of data, often called "ghost zone" or "halo" is GHOST_WIDTH cells, and it is configurable. The number of blocks in each direction can be chosen at runtime, as all data is dynamically allocated. The local piece of a distributed array is allocated on each processor as e.g. F(I1:I2,J1:J2,KB), where I1,I2,J1,J2 are constants precomputed in BLOCK_SETUP so that e.g. I2-I1+1=FULL_I_BLOCKSIZE.

The exchange of data is best illustrated with an example; we want to parallelize the following loop

```
DO J=2,JM-1
  DO I=2,IM-1
    IF (DUM(I,J).GT.0.0) THEN
        NETFLUX = XFLUX(I,J) - XFLUX(I-1,J) + &
    &             YFLUX(I,J) - YFLUX(I,J-1)
        C2U(I,J) =   NETFLUX/AREA
    ENDIF
  END DO
END DO
```

The first thing to modify is the loop ranges in the I and J directions. Each block has its local index range system. In order for all blocks combined to cover exactly the same domain as the global array, the numbering for each blocks index ranges starts with 1 and ends with e.g. I_EQ_IM for the I direction. I_EQ_IM is a block local constant, equal to I_BLOCKSIZE for all blocks, except the last block where its local value corresponds to the global index I=IM.

We look at an example loop range in the I direction; the loop from I=3 to I=IM-1 will start at I=3 on the first block, while it must start at I=1 at all other blocks. It must end at I=I_BLOCKSIZE at all blocks, except the last block where it must end at I=IM_MI_1. IM_MI_1 is a block local constant with a value corresponding to the global index IM-1 at the last block. The computation of constants used to transform loop ranges is done automatically in the subroutine BLOCK_SETUP.

When looking at the loop kernel, we observe that the array XFLUX is accessed at index (I-1,J), which is in the west ghost zone of the block, while YFLUX is accessed in (I,J-1) which is in the "southern" ghost zone. We therefore need to update the data of these two ghost zones. We only need a one cell wide update of the data, since no kernel stencils reach beyond I-1 or J-1. The complete transformed code looks like this

```
CALL PACK2D(XFLUX, UPDATE_WEST, UPDATE_NONE, 1)
CALL PACK2D(YFLUX, UPDATE_NONE, UPDATE_SOUTH,1)
CALL UPDATE_GHOST

DO J=J_EQ_2,JM_MI_1
  DO I=I_EQ_2,IM_MI_1
    IF (DUM(I,J).GT.0.0) THEN
        NETFLUX = XFLUX(I,J) - XFLUX(I-1,J) + &
    &             YFLUX(I,J) - YFLUX(I,J-1)
        C2U(I,J) =   NETFLUX/AREA
    ENDIF
  END DO
END DO
```

The two PACK2D subroutine calls build up a list of ghost zone updates to be performed. The first call says update 2D array XFLUX in a 1 cell wide ghost zone at the western side of all blocks, the second call adds a command to update YFLUX at the southern zone for all blocks. The call to UPDATE_GHOST performs the actual update by going through the list of updates to be done, assembling data from the given arrays and packing it into one message for each of the sides where updates are requested, then initiates sends and receives on all involved processes. After a successful exchange, the arrays are updated. Corner data are propagated correctly in the diagonal directions by using the well known trick of first completing a full update in one direction, followed by a full update in the other direction. A full dependency analysis for all variables has been done so that it is possible to skip the update of a variable's ghost zone if we know it is not needed.

With this technique, the number of points with exchange of ghost data in the code is reduced from around 100 to ca 23 per 3D step + 3 updates per 2D iteration. The number of update points are commonly one order of magnitude higher than this in comparable models that do not do dependency analysis.

There are several different strategies for keeping the data in the ghost zone up-to-date. The perhaps most common approach is to skip the dependency analysis and update the ghost zone just before, and every time a variable is accessed there. Another way is to ensure the data in the ghost zones are identical to the neighbour values at all times. This approach implies the need to update the zone every time a variable changes. Both of these techniques work well on machines with low latency interconnects and a fast message passing library, and has the clear advantage that code can be inserted anywhere without having to care for dependencies.

The library contains subroutines for doing most necessary routine tasks. The subroutines GATHERFIELD and SCATTERFIELD take a global array as an argument on one processor and fetch/distribute it from/to the other processors:

```
REAL FLOCAL(I1:I2,J1:J2,KB), FGLOBAL(IM,JM,KB)
...
CALL SCATTERFIELD(FLOCAL, FGLOBAL, IM,JM,KB)
...
```

The subroutine FIND_MAX_REALS uses a single call to a collective operation to

find the maximum value and its location, of a distributed array. The following code

```
REAL F(IM,JM,KB)
REAL VAL
INTEGER LOC(3)
...
VAL=MAXVAL(F)
LOC=MAXLOC(F)
...
```

can therefore be parallelized as follows

```
REAL F(I1:I2,J1:J2,KB)
REAL VAL
INTEGER KB,LOC(3),ROOT
...
CALL FIND_MAX_REALS(KB, F, VAL, LOC, ROOT)
...
```

The subroutine COPY_REGION copies a rectangular region from one given global location to another, using asynchronous sends and receives. Fortran statements like

```
F(10:11,1:10)=F(1:2,1:10)
F2(10:11,1:10)=F2(1:2,1:10)
```

are translated (also accumulated) into

```
...
FROM=(/1,2,1,10/)
TO  =(/10,11,1,10/)
CALL COPY_REGION( FROM, TO, 1, F, F2)
...
```

There are routines for broadcasting scalars out to the other processors, e.g. broadcast scalar A and B from processor 2 out to all others, the logicals EN,TO,TRE from processor 0 and to the others, same for the integer NUM;

```
REAL A,B
LOGICAL EN,TO,TRE
INTEGER NUM
...
CALL BROADCAST_REALS(A,B,ROOT=2)
CALL BROADCAST_LOGICALS(EN,TO,TRE)
CALL BROADCAST_INTEGERS(NUM)
...
```

There is also a set of reduction subroutines that can find e.g. the sum or product of a scalar (optionally up to 6) from all processors. This example sums up MASS and VOLUME on each processor, then finds and returns the sum over all processors;

```
  DO J=1,J_EQ_JM
    DO I=1,I_EQ_IM
       VOLUME=VOLUME + D(I,J)
    END DO
  END DO
  DO K = 1,KB-1
    DO J=J_EQ_2,J_EQ_JM
      DO I=I_EQ_2,I_EQ_IM
        MASS = MASS + RHO(I,J,K)*D(I,J)*DZ(K)
      END DO
    END DO
  END DO
  CALL REDUCTION_ADD_DOUBLES(MASS,VOLUME)
...
```

The examples demonstrate that the only modification necessary to the sequential code is a change of the loop ranges, and one call to the high level routine. The number of modified lines in the code is comparable to OpenMP based parallelization. Occasionally, simple statements may require more logic to give identical results, e.g. the assignment

```
...
ETA(:,:) = ETATFRS(2,13)
...
```

can in the MPI world look like

```
...
I=2 ; J=13
CALL GLOBALPOINT2LOCAL_NG(I,J,INBLOCK)
TEMPVAL=1.0
IF(INBLOCK) TEMPVAL = ETATFRS(I,J)
CALL REDUCTION_MULTIPLY_REALS(TEMPVAL)
ETA(:,:) = TEMPVAL
...
```

The first subroutine call converts I,J from global indices to local, and returns INBLOCK=.FALSE. if the address is not on the current block. The reduction distributes the scalar to all other processors.

3 Testing

The model has been set up for and validated on several test cases:

- The Seamount case [1] with grid dimensions 50x50x21, which is very small.
- Banten bay of Indonesia, a case with dimensions 140x100x80, also on the small side.
- North Sea (hydrostatic only) 800x590x42, very large case.

The model gives the same result to the last digit for the hydrostatic runs, while the result will depend on the number of CPUs for the non hydrostatic case, due to computation of inner products in BiCGStab. It is possible to compute sums and get bit exact results on varying number of CPUs, but this requires an extra collective call (broadcast).

3.1 Performance

The hydrostatic model now scales reasonably well. The large test case of the North Sea, which with OpenMP parallelization uses 100-120s per time step on 4 CPUs on the IBM Regatta (with high load), has the following performance on the Pentium III cluster in Bergen (1255MHz processors and Fast Ethernet interconnect, MPICH 1.2.6)

processors	s per time-step
4	1942.
6	64.5
9	46.
20	21.
30	18.
42	15.2
56	11.5

We have to use at least 4 CPUs for this case due to the memory requirements, but we observe that the MPI model on the inexpensive linux cluster outperforms the OpenMP code on 4 CPUs of the IBM Regatta already on 6 CPUs, and it scales well on as many CPUs as we have, considering the slow interconnect. The 4 processor time is reproducible, and probably related to swapping.

On the smallish/medium sized Banten bay benchmark we here report speedup numbers for LAM MPI as well as MPICH

processors	LAM	MPICH	NONHYD
1	1	1	1
4	2.2	1.6	1.4
9	4.3	3.2	2.3
12	5.8	3.9	3.1
16	7.4	5.3	3.2
20	8.7	7.8	
25	10.4	8.7	
42		12.0	
49		12.7	

We observe that the scalability is not as impressive, and that LAM consistently performs better than MPICH. The non hydrostatic solver has performance problems on the linux cluster for such a small case, most likely due to the high latency. On the Regatta the speedup of the non hydrostatic model on a fully loaded machine (a load of approximately 30 when the jobs run) looks like this:

processors	OpenMP	MPI
1	1	1
2	1.8	1.76
4	2.5	2.75
6	2.9	4.27
9		5.78

These numbers are from the smallest test problem. The numbers demonstrate the improvement over the OpenMP model, taking into consideration the small size of the experiment. Further testing and tuning are being worked on, but the results look very promising. We will implement a BiCGStab solver with reduced number of inner products, in order to reduce the models clear sensitivity to the latency of the interconnect. For larger problems, these tricks will give substantial improvement in scalability.

4 Conclusions and further work

The non hydrostatic BOM has been parallelized and verified using a MPI based framework for parallelization of finite difference models.

The performance of the model can be further improved upon, but the low hanging fruit has been exploited. There is always a compromise on how easy it should be to modify the code in the future, improving the parallel performance further would require modifications that would make it necessary to do a more thorough dependency analysis every time new code is inserted into the model.

The hydrostatic model scales linearly up to the number of CPUs we currently have available, even with a very slow interconnect. Due to heavy load we have not been able to obtain scalability numbers from a full 32 node IBM system in Bergen, but on a loaded machine the scalability looks promising.

Work is underway to implement a BiCGStab solver with reduced number of inner products, which will improve the performance of the non hydrostatic code further.

Another issue that very likely affects performance, is load balancing. Currently the parallelization builds on the assumption that the model performance to a large extent depend on memory bandwidth, so each processor therefore sit on identically sized blocks. Each block, however, sit on a different number of "wet" cells, so the number of flops performed on each processor is not balanced. It is likely that the optimal distribution is somewhere in between an even distribution of wet cells and the even distribution of memory use. This may be investigated later.

References

[1] A. Beckmann and D.B. Haidvogel. Numerical simulation of flow around a tall isolated seamount. part I: Problem formulation and model accuracy. *Journal of Physical Oceanography*, 23:1736–1753, August 1993.