A parallel data assimilation model for oceanographic observations

Fons van Hees & Aad J. van der Steen
{vanhees,steen}@phys.uu.nl
Computational Physics, Utrecht University
P.O. Box 80195, 3508 TD Utrecht
The Netherlands

Peter Jan van Leeuwen
leeuwen@phys.uu.nl
Institute for Marine and Atmospheric research Utrecht
Utrecht University
P.O. Box 80005, 3508 TA Utrecht
The Netherlands

June 26, 2001

Abstract

In this paper we describe the development of a program that aims at the optimal integration of observed data in an oceanographic model describing the water transport phenomena in the Agulhas area at the tip of South Africa. Two parallel implementations, MPI and OpenMP, are described and experiments with respect to speed and scalability on a Compaq AlphaServer SC and an SGI Origin3000 are reported.

1 Introduction

When one wants to study the dynamics in an area where numerical models have serious shortcomings a possible step forward is to use data assimilation. In data assimilation all information that is present in observations of the system is combined with all information we have on the dynamical evolution of the system, i.e., a numerical implementation of the physical laws. This can be done by using *ensemble models* that do not calculate the evolution of a single state but rather a large amount of different states. All the states in this collection, the ensemble, should correctly represent the partially known current state of the system under consideration. In our case we apply this data assimilation technique on water transport phenomena in the Agulhas area at the tip of South Africa.

The art of *ensemble smoothing* is all about finding that evolution under consideration (i.e., the ocean) that remains close to the observations while still obeying the numerical implementation of the dynamical laws relatively closely. The assimilation of the observed data is done by adjusting the ensemble members using these observations to ensure that their quality in representing the mean and their spread around the mean do not deteriorate. The principles of smoothing, filtering, and assimilation of observations are for instance treated in [1, 6].

As each of the ensemble members evolves independently of the others during the time between two data adjustments this is a particularly good starting point for parallelisation. This is indeed how the parallelisation is realised in the program with considerable success and we conducted timing experiments with two parallel implementations, MPI and OpenMP, on an SGI Origin3000 and, to a lesser extent, on a Compaq AlphaServer SC.

The structure of this paper is the following: in section 2, we briefly discuss the model and the numerical methods as implemented in the program, the restructuring of the code, and the parallelisation strategy. In section 3 we present the results of the timing experiments with a test problem of suitable size, and in section 4 we analyse these results with respect to scalability and possible sources of further improvement.

2 Model and methods

2.1 The data assimilation model

The data assimilation method used is the Ensemble Kalman smoother, as described by Evensen and Van Leeuwen in [4]. We provide a brief description of the idea behind data assimilation and present the resulting equations. Details and derivations can be found in [4], and [3].

At the heart of nonlinear data assimilation lies the notion of combining probability densities of model and observations. By expressing the problem in terms of probability density functions a Bayesian estimation problem can be formulated. In Bayesian statistics the unknown model evolution ψ is viewed as the value of a random variable $\underline{\psi}$.

The unknown probability density $f(\psi)$ of evolution of states ψ is related to the model being used. It is called the prior probability density in contrast to the posterior probability density $f(\boldsymbol{D}|\psi)$ that, according to Bayes' Theorem is given by

$$f(\psi|\boldsymbol{D}) = \frac{f(\boldsymbol{D}|\psi)f(\psi)}{\int f(\boldsymbol{D}|\psi)f(\psi)dL\psi}. \tag{1}$$

The first factor in the numerator, the density $f(\boldsymbol{D}|\psi)$, is the probability density of observations $\boldsymbol{D}$ given that the model is in state ψ. The second factor is the prior model density $f(\psi)$. The denominator is the probability density of the observations, expressed as an integral over all model states of the joint probability density of observations and model. We assume that the probability density of the observations $f(\boldsymbol{D}|\psi)$ is known, for instance a Gaussian. Because the model variable ψ is given, the mean of the Gaussian density will be the measurement of the optimal model state, while its variance is the measurement error.

The prior probability density $f(\psi)$ of the model evolution is more difficult to obtain. In principle the Kolmogorov equation describes its evolution. But since the probability density for the model state has a huge amount of variables, it is computationally not feasible for real oceanographic or meteorological applications to determine its evolution. The model evolution of the density could in principle be determined from ensemble integrations. In simulated annealing and related methods this probability density is generated approximately. However, these methods need a huge amount of storage and iterations due to the random nature of the probing. On the other hand, knowledge of the complete density is too much information. One is interested only in its first few moments, e.g., a best estimator of the truth and its error variance. In that case ensemble or Monte-Carlo experiments can be extremely useful. The most-used estimator is the minimum-variance estimator, in which only integrated properties of the density are needed. The ensemble smoother is an example of an approximate variance-minimizing estimator [4]

The approximation made is that the prior, or model evolution probability density is Gaussian distributed in state space at all times. As is explained in [4] and [5], the optimal model evolution is given by

$$\hat{\psi} = \psi_F + R^T \boldsymbol{B}, \tag{2}$$

Here, ψ_F is the best estimate of the model evolution without data assimilation. $\boldsymbol{B}$ are the representer coefficients, which can be determined from

$$(\boldsymbol{R} + \boldsymbol{W}^{-1})\boldsymbol{B} = \boldsymbol{D} - \mathrm{L}[\psi_F], \tag{3}$$

where $\boldsymbol{W}^{-1}$ is the error covariance of the observations, the representer matrix $\boldsymbol{R}$ is the measurement-operator covariance, given by

$$\boldsymbol{R} = E\left[\mathrm{L}\left[\psi - \psi_F\right] \mathrm{L}^T \left[\psi - \psi_F\right]\right]. \tag{4}$$

where $E[..]$ is the expectation operator, and in which $\mathrm{L}[..]$ is the measurement operator, assumed to be linear. The representers R, which are the model field-measurement operator covariances, given by

$$R = E\left[(\psi - \psi_F)\mathrm{L}\left[\psi - \psi_F\right]\right]. \tag{5}$$

The representers are crucial in the data-assimilation problem. A representer describes how the information of the corresponding measurement influences the solution at all space-time points. Bennett [1] gives an excellent treatment on the meaning of representers. The representer coefficients determine how strong each representer should be counted in the final solution. They depend on the model-observation misfit and on their error covariances. Clearly, an important ingredient in the success of the inversion procedure is the conditioning of the sum of the representer matrix and the error covariance of the observations, as given in (3).

The error covariance of the optimal estimate is given by

$$Q_{\hat{\psi}\hat{\psi}} = Q_{\psi_F \psi_F} - R^T (\boldsymbol{R} + \boldsymbol{W}^{-1})^{-1} R \tag{6}$$

Van Leeuwen has shown [5] that the assumption of Gaussianity leads to an error in proportional to the error covariance of the optimal state. So, the smaller this error, the better the ensemble smoother performs. Note that this will be one of the reasons that data assimilation methods that use the Kalman filter update equation work at all in (strongly) nonlinear situations: as long as the filter does not diverge the optimal estimate will be relatively close to the minimum-variance estimate.

2.2 Physical domain and assimilation data

Although the techniques disscussed above are generally valid, in the code in which the model is implemented a region of particular interest was chosen, the tip of South Africa with the Agulhas Stream in the surrounding ocean. This amounts to a region 30–45° South and 10–35° East. By considering 5 vertical layers in the ocean the region of interest is represented by a $251 \times 151 \times 5$ 3-D grid. The data to be assimilated within this computational domain are available from the TOPEX/Poseidon satellite. The satellite covers the domain approximately every 10 days. The analysis phase in the model is therefore chosen to span the same time period. As remarked, although the methodology used is general, the topographical details (land points, depth information, etc.) are explicit in the code and do not allow a simple generalisation.

2.3 Model implementation

At the highest level the model performs two computational intensive tasks: generation of the ensemble members and the computational flow part that describes the evolution of the stream function ψ in time. In addition, there is a non-trivial initialisation phase and every 240 hourly time steps an analysis of the ensemble members is done to obtain an optimal estimate from the past period. This information is used to adjust the ensemble that forms a basis for the evolution in the next period.
In the original Fortran 77 code the generation of the ensemble members and the flow part were not clearly separated. To obtain a better maintainability of the code we decided to separate these two parts leading to a Fortran 90 part for ensemble generation and analysis and a Fortran 77 part for the evolution of the flow. The serial program thus obtained has a very simple structure:

```
Program EnsFlow
Use      Field           !   Contains the e.g. psi (stream function)
                         !   psiRandom, psiEns
Implicit None
Integer  i, steps, tStart, tEnd

write *, 'Give number of time-steps'
read  *, steps

tStart = 0               !   Usually one starts at t = 0.
tEnd   = tStart + steps  !   tEnd is time in units of 'steps'.
```

```
amplitude = prescribedAmplitude
Call InitializeModel (psi)

!Initiate ensemble
Do i = 1, n_e            ! Loop over number of ensemble members.
   Call GenerateRandomField (psiRandom)
   psiEns(i) = psi + psiRandom
End Do

! Start of a possible assimilation loop.

! Time evolve ensemble
Do i = 1, n_e            ! loop over number of ensemble members
   psi = psiEns(i)
   Call TimeEvolve (tStart, tEnd, psi)
   psiEns(i) = psi
End Do

! Analyse and update Ensemble.
Call AnalyseEnsemble (psiEns)

! Random perturbation of ensemble.
Do i = 1, n_e            ! loop over number of ensemble members
   Call GenerateRandomField (psiRandom)
   psiEns(i) = psiEns(i) + amplitude * psiRandom
End Do

! End of a possible analysis loop.

End Program EnsFlow
```

In this program the routine `InitializeModel` reads various data files like the bottom topography and initial state data. This takes a negligible amount of time. The main time in this routine is spent in preprocessing the data such that a first initial state is generated that is stored in array `psi`.
Routine `TimeEvolve` does the time evolution of the flow, `GenerateRandomField` generates a random field that is added to an ensemble member, and `AnalyseEnsemble` performs the analysis phase every after 240 time steps. The routines `InitializeModel` and `TimeEvolve` are both black boxes for the user but they are not independent due to the data that are shared through `common` blocks between the routines.

The routine `GenerateRandomField` generates a random field that (possibly multiplied with an amplitude) is added to the state ψ. These random fields have zero mean and a prescribed variance. The correlation between 2 values at different grid points in the horizontal direction is set at e^{-d^2/α^2}, where α is a prescribed correlation length and d is the distance between the grid points. The correlation between the vertical layers is obtained by by combining two 2-D fields with appropriate weighting coefficients.

As explained before, every 240 time steps (10 days) an analysis of the ensemble is performed. We briefly mention the numerical steps and data stuructures used according to the data assimilation model described in section 2.1 following [3]. We first determine for each of the n_e ensemble members $\vec{s}$ the value that corresponds to the n_d observations leading to a matrix $L = \mathrm{L}[\psi]$ of size $n_e \times n_d$. The matrix δL is obtained by subtracting the ensemble average $\overline{L}$ form L and we arrive at the computational equivalent of formula (4) by computing

$$P = \frac{1}{n_e}(\delta L)^T(\delta L) + {q_d}^2 I, \tag{7}$$

In this equation ${q_d}^2 I$ is an $n_d \times n_d$ matrix describing the error covariances of the observations. Because these are assumed to be uncorrelated we just have a diagonal matrix.
Next the matrix X is calculated from

$$X = \frac{1}{n_e}\delta L\, B. \tag{8}$$

where B is a $n_d \times n_e$ matrix obtained by means of a Singular Value Decomposition from

$$PB = \boldsymbol{D} - L^T, \tag{9}$$

as given in formula (3). The matrix $\boldsymbol{D}$ is formed by perturbing the observation vector for each of the ensemble members:

$$\boldsymbol{D}_{i,j} = d_i + q_d \lambda_{i,j}, \tag{10}$$

where the λ's are drawn from the Normal distribution $N(0, 1)$. Lastly, the ensemble is updated according to

$$\vec{s}_{\mathrm{new}} = \vec{s} + \vec{s}X. \tag{11}$$

in which $\vec{s}$ contains the states of all the ensemble members.

As the ensemble generation, the analysis, and the flow calculations in the revised implementation are all self-contained, each of these parts can be adjusted or replaced at will by the user as long as the existing data structures used are heeded.[1]

2.4 Parallel implementation

Because of the structure of the serial implementation the parallelisation strategy could be rather simple: after initialisation, the generation of the ensemble members and all associated flow calculations within the ensemble members are divided evenly over the available processors using the SPMD programming model. We mainly discuss the MPI version of the program but the OpenMP implementation is structured similarly. We strictly used the standard MPI and OpenMP facilities as given in [7, 8] and [2, 9].

[1]Alternatively, instead of (9), matrix $\boldsymbol{B}$ could also be obtained from the system of equations $L^T\boldsymbol{B} = \boldsymbol{D} - L^T$. This system is of the much smaller size of $n_d \times n_e$ instead of $n_d \times n_d$ and can be solved in the Least Square sense by QR factorisation. The numerical stability of this approach is however to be evaluated more thoroughly.

The number of ensemble members n_e is an input parameter of the program and the number of ensemble members assigned to a processor is given by:

$$n_{e,r} = \lfloor \frac{n_e + r}{n_p} \rfloor,$$

where n_p is the number of processors and r is the rank of the process within the range $0, \ldots, n_p - 1$. Obviously, to avoid processors to be idle n_e should preferably be an exact multiple of n_p.
Each process performs the same ensemble initialization. They all read the (same) data that are required to specify all details of the time integration (the numerical flow calculation) including the initial state data. Each process creates a unique seed that depends on its rank r. From these seeds each process can generate independent sequences of pseudo-random numbers that are used to generate the appropriate pseudo-random fields. The complete intial ensemble (that is distributed over the various processes) is now generated by adding the initial state to n_e different pseudo-random fields.
Each process integrates all its members until a "time to analyse" is reached, i.e., when a certain number of time-steps has been made (in our case 240). This number is the same for all the processes, since it is read from the same data file.
When the analysis phase is reached each process sends all its members as calculated for the requested time, the time to analyse, to a single master process. The master knows exactly how many messages it has to receive from all the working processes including itself (n_e if members are sent one at a time). Once the master has received all the messages it goes on with the calculations to analyse und update the ensemble. All the other workers are now idle waiting to receive updated ensemble members. When the master is ready, process r receives $n_{e,r}$ messages, one (updated) member per message. Each such message is about 1.5 MB in size. The processes then continue to integrate their ensemble members until a next "time to analyse" is reached. As stated, only one processor performs the analysis and update of the ensemble. Although the most time consuming part of this procedure, the Singular Value Decomposition, can be done in parallel we refrained from that as the distribution of the ensembles provided a cleaner way of parallelisation. The n_e ensemble members together form a $n_d \times n_d$ size matrix, see Eq. (7). Hence, for large ensembles, data sets and numbers of processes it may be worthwhile to consider parallelisation of the SVD routine (having $\mathcal{O}(n_d^3)$ floating-point operations) as well as of the matrix operations that appear in Equations (7), (8), (9), and (11).
The code of the analysis part in the MPI implementation is almost identical to that in the serial and OpenMP implementation. The difference lies in the array `Ensemble`. This array is local to each of the processes in the MPI version and has to be collected in an array `GrandEnsemble` in the master process before proceeding with the analysis. Obviously, this distinction is not necessary in the serial and OpenMP version of the program.

2.5 Implementation considerations

In the MPI implementation the only thing we have to communicate is a field, a three-dimensional array with numbers. We have developed some generic Fortran

90 routines to facilitate this communication. Throughout the Fortran 90 code the datatype `Real` is used, without specifying what a `Real` is. A compiler option e.g., `-r8` guarantees that all floating-point numbers are 8 bytes long. Within the MPI communication routines one has to specify the type of object one intends to communicate. It is incorrect to specify `MPI_REAL` for this type, since it is not affected by the compiler option. Hence, in the communication routines it is necessary to specify `MPI_DOUBLE_PRECISION`, which is 8 bytes long, as the datatype.

In the OpenMP implementation one needs only a few directives to create a parallel program. The program is structured such that the integration of ensemble members is clearly recognizable. Hence this can be done in parallel easily. One only has to guarantee that some data is private. These private data are stored in Fortran 77 `common` blocks. Actually there are only 2 such named blocks present in the code, i.e., `/worksp/` and `/worksp2/`. It is important to include in every routine where these blocks occur an OpenMP directive like

```
!$OMP THREADPRIVATE(/worksp/)
```

In the NAG library routines used similar changes have to be applied for a correct working of the routines unless an OpenMP enabled NAG library is available for the computer platform of choice. Presently, this is not guaranteed for all platforms.

Instabilities in the evolution process are detected internally by monitoring the growth of the kinetic energy in the top layer. When this quantity rises to an unacceptable level, the program terminates. The instabilities may occur within the flow calculation but this is very unlikely to happen. It is more likely that instabilities arise immediately after an analysis step or after a random field addition, indicating that these operations introduce “unphysical” components. There are several mechanisms in order to prevent this from occuring:

- Smear the solution so much, inside the algorithm, that the introduced diffusion (either as a numerical artefact or deliberately put in) prevents this from occuring.
- Reduce the amplitude of the random field and perform such random field additions more frequently. The system may more easily adjust to a smaller, more frequent (random) forcing than to less frequent but larger boosts.

3 Results

In a production environment with OpenMP one may notice that during execution the situation changes dynamically. For example a job with an ensemble of 6 members and with 3 parallel threads may start with all 3 threads active, each one taking 2 ensemble members to calculate. However, when the load of the system, due to other activity becomes high, it might occur that only one thread remains active, and processes all ensemble members. This guarantees at least that the performance does not become very much worse than in the sequential program. With MPI however, one statically fixes the processes to work on a specific load. Therefore, if the operating system decides for whatever reason not to work on a particular process, this single process may halt the progress of the complete application. This may

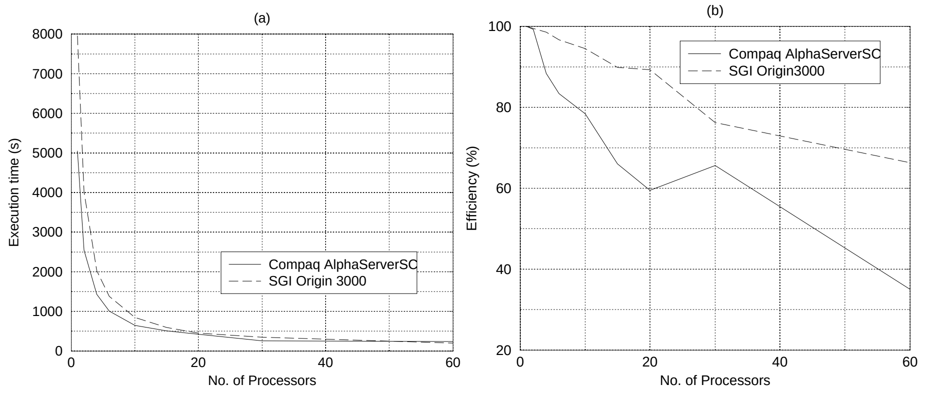

Figure 1: *Execution times for the MPI implementation of the program for 1–60 processors with 60 ensemble members,* $n_e = 60$ *(a), and the efficiency obtained (b) on the Compaq AlphaServerSC and the SGI Origin3000.*

eventually lead to a considerable slowdown.
The results presented hereafter did not suffer from these phenomema as we were able to acquire separate partitions of TERAS, an SGI Origin3000 system with 1024 MIPS R12000, 400 MHz processors and 1 GB of memory per processor. The maximum partition used in our experiments had a size of 256 processors. For the Origin3000 system we present two results: one while using $n_e = 60$ ensemble members and an increasing number of processors. A second experiment had one ensemble member per process and the number of processes ranged from 1–252. Those experiments were conducted both for the MPI and the OpenMP implementation.

3.1 MPI results

The timing of the code with 60 ensemble members and 1–60 processors is displayed in Figure 1. On 1 processor of the SGI Origin3000 the execution time for the integration of two 10-day periods was 7995 seconds decreasing to 201 seconds on 60 processors. This amounts to an efficiency of 66% on 60 processors. This loss of efficiency is due to the irreducible scalar part in the program, viz. 14.0 seconds for initialisation and 36.6 seconds for the analysis phases. This will be treated further in section 4.

The results for the experiments with 1 ensemble member per processor ($n_{e,r} = 1$) are shown in Figure 2. Note that irrespective of how the ensemble members are distributed over the processors the amount of messages is virtually constant as each ensemble member is sent individually to the master processor.

The total execution time as shown in Figure 2 consists of the time spent in the initialisation, 2 flow calculations, 2 analysis calculations and 2 times the collect and distribute communication phases necessary to compute the ensemble update centrally in the analysis phase. It turns out that the total communication time for $n_p \neq 1$ ranges from 0.6–5.4% of the total execution time. It is rather the analysis

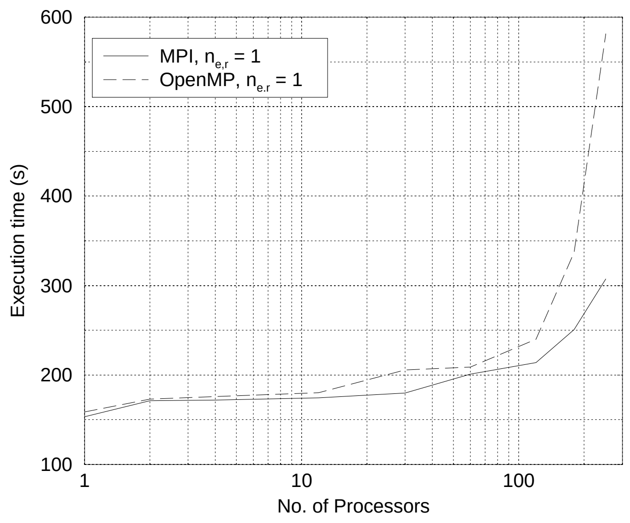

Figure 2: *Execution times for the MPI and the OpenMP implementation of the program for 1–252 processors with 1 ensemble member per processor ($n_{e,r} = 1$) on the SGI Origin3000.*

Table 1: *Timings and MPI-OpenMP speed ratio on the Compaq AlphaServer SC and the AlphaServer GS160, corrected for the clock speed.*

Processors	MPI Time (s)	OpenMP Time (s)	Ratio OpenMP/MPI
4	1430	2206	1.54
6	1008	1652	1.64
10	644	1272	1.97

time that contributes most to the growth of the execution time: from 10.3 seconds for 1 ensemble member to 134.1 seconds for 252 ensemble members. This is not surprising as the amount of work to update the ensemble scales with $\mathcal{O}(n_e^2)$ and eventually becomes the dominating term in the analysis, even more important than the SVD itself, see Eq. (10). However, the SVD as it appears in the present formulation is proportional to n_d^3 and thus easily becomes the computational bottleneck for larger observational data sets than used at present.

3.2 OpenMP results

With respect to MPI the SGI Origin3000 and the Compaq AlphaServer SC are equivalent because of the distributed memory model that ignores the amount of processors that can address a common memory. This is not true for OpenMP: the Origin3000 regards up to 512 processors all memory to be shared where in

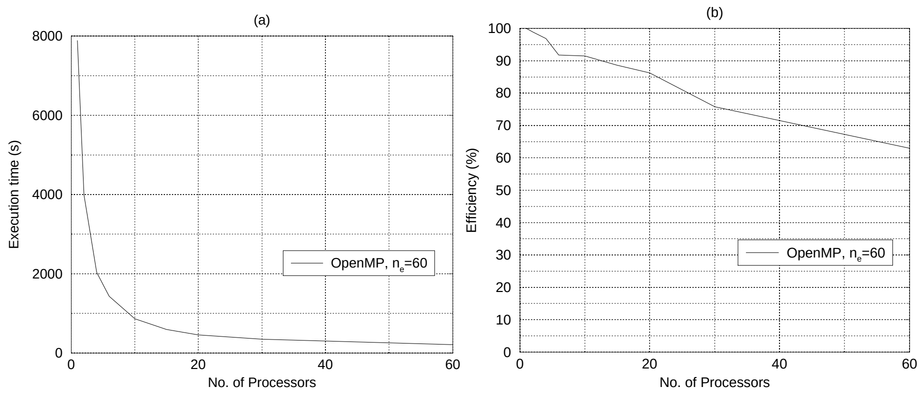

Figure 3: *Execution times for the OpenMP implementation of the program with $n_e = 60$ on 1–60 processors on the SGI Origin3000 (a) and the efficiency as a function of the number of processors (b).*

the AlphaServer SC 4 processors in a node share the memory local to the node in which they reside and, consequently, OpenMP can only be used up to 4 processors on this system. Instead, of the 833 MHz AlpaServer SC we were able to use an AlphaServer GS160 with 16 processors with 731 MHz EV67 Alpha processors to get an impression of OpenMP on a higher number of processors. In the latter system all memory is globally accessible, so tests with up to 16 parallel threads are possible.
In the following we first present the results for the SGI Origin3000. Where relevant and available we also discuss the additional results obtained from the AlphaServer GS160. Results are given in Table 1 for 4–10 processors.
As with MPI we measured the execution times of the program with 60 ensembles on 1–60 processors. The results are displayed in Figure 3.
Up till 60 processors the OpenMP implementation is marginally less efficient than the MPI implementation: on 60 processors the efficiency for OpenMP is 63.0% against 66.3% for the MPI version. The speedup is then still a factor 38.

From the limited results obtained with the AlphaServer GS160 it was evident that the OpenMP version was less efficient on this machine than the MPI version on the AlphaServer SC as can be seen from Table 1. Not only is the OpenMP version slower in the absolute sense, it also relatively decreases in speed with an increasing amount of processors. As far as presently known, this should be ascribed to the increasing synchronisation and thread handling overhead in the current implementation of OpenMP.

The scaling behaviour of the program with 1 ensemble member per processor for 1–252 processors on the SGI Origin3000 is displayed in Figure 2. It shows that for runs with more than about 100 processors the scaling is clearly worse than for the MPI implementation. This is due to the flow part in the computation. The analysis part takes roughly the same time in both the MPI and the OpenMP version but the flow calculation which is distributed over the CPUs suffers to a large extent from synchronisation and thread handling overhead for larger amounts of processors. In this case the MPI version is therefore to be preferred although a larger amount of memory is used in this case.

4 Analysis of results

In this section we want to consider the timing experiments in a little more detail. This should enable us to make an estimate on a preferred number of processors to use with a certain implementation and machine.
In the MPI implementation the following significant parts can be identified:

Setup In this phase the input data are read by all processors, values are initialised and an initial states are computed by all processors. The time spent in this phase is weakly dependent on the number of processors.

Flow calculation This part is parallel and very scalable.

Communication This part comprises the sending of the ensemble members to the master for analysis and sending back the updated states to the other processors after analysis.

Analysis Analysis of the ensemble and assimilation of data to create updated states for the next integration period. This part is completely scalar.

Final update At the end of the computation sequence final states are distributed to all processors. This process is very scalable.

We start with discussing the MPI results.

We can express the execution time in the MPI implementation as

$$T_p = T_{\text{init}} + T_{\text{flow}} + T_{\text{com}} + T_{\text{an}} + T_{\text{fin}} \tag{12}$$

In expression 12 the analysis time, T_{an}, is completely scalar. Furthermore, the setup time T_{init} contains a part that is not scalable at best: the reading of input files by the processors. As with the use of MPI-1 there are no facilities for parallel I/O each processor has to read all of the input data. Depending on the I/O configuration, this may be a constant amount of time or, because of I/O contention, increase with the number or processors that have to access the data.
Comparing the Compaq AlphaServer SC and the SGI Origin3000 in this respect we found:

$$\begin{array}{ll} T_{\text{init}} = 9.25 + 0.40P \text{ seconds} & \text{(AlphaServer SC)} \\ T_{\text{init}} = 14.00 + 0.01P \text{ seconds} & \text{(Origin3000)} \end{array}$$

We have to stress that these findings are very dependent on the I/O configuration and may be different even on systems of similar size and architecture.
The communication time T_{com} as seen by the master process is on both systems highly erratic: it ranges from 2.5–14.5 seconds on the Origin3000 and from 2.5–23

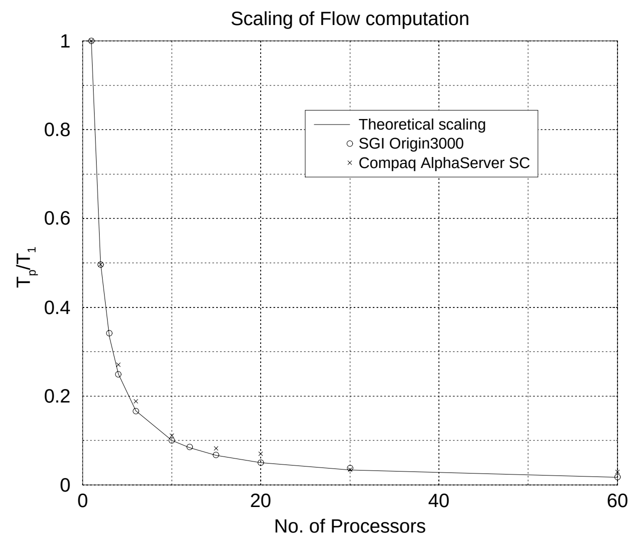

Figure 4: *Theoretical and observed scaling behaviour for the flow part of the program on the Compaq AlphaServer SC and the SGI Origin3000.*

seconds on the AlphaServer and bears virtually no relationship with the amount of processors used. This variability is in fact caused not by the communication itself, but time differences in the flow calculations on each of the processors. Although these computational part should take exactly the same amount of time on all processors, in practice the processors get slightly out of phase for no tractable reason. As the master only can begin its analysis when all results are gathered, reception of the last message determines the "communication time" although it is rather synchronisation time. In any case, T_{com} contributes only a small percentage to the total execution time. E.g., in the case of 60 ensemble member running on 60 processors T_{com} takes 2.5 and 5% of the execution time on the AlphaServer and the Origin3000, respectively.
It turns out that the main part of the total time is consumed by the flow calculations. As these are carried out completely independent in each of the ensemble members, this part is ideally scalable as is evident from Figure 4. The observed values of T_p/T_1 for both systems almost exactly agree with the theoretical curve.
T_{an}, the analysis time is constant on both systems for a fixed number of ensemble members as might be expected. Interestingly, for 60 ensemble members ($n_e = 60$), the time required for this part takes about 56.5 seconds on the AlphaServer while the same part take roughly 36 seconds on the Origin3000. For the flow part the situation is reversed: the Compaq system is about 1.5 times faster in this part. This is more in line with their respective clock frequencies: 400 MHz versus 833 MHz. The overall effect is that for the total execution time the Compaq system starts off faster because the flow computation dominates the total time. The cross-over point lies somewhere near 50 processors. On 60 processors, the Compaq system spends 40% in the scalar part and 60% in the parallel part. For the SGI system

these fractions are 31% and 69% respectively. This means that in both systems the scaling potential is still significant.
Finally, the time for the final update turns out to be insignificant with respect to the total execution time never exceeding 0.3% of this total time. As, in addition, this part scales perfectly it has no impact on the scaling behaviour of code as such.

5 Closing remarks

The results as presented in the former sections show that data assimilation models of this type can be parallelised very successfully. The model considered here contains a scalar part that sets bounds to the maximal speedup, but the experiments show that for both the SGI Origin3000 and the Compaq AlphaServer SC this bound lies clearly above 100 processors when MPI is used and the same holds for the OpenMP implementation on the Origin3000. Still, the program probably can be improved in several minor respects:

1. The initialisation phase may, except for the reading of input files, for a significant part be parallelised.

2. Also in the analysis part, which is now done in scalar mode, there is the potential of parallelising either the SVD routine presently used, or the QR factorisation routine that could alternatively be used.

3. Further evaluation should make clear whether using QR factorisation yields numerical results that have acceptable stability. If so, this could reduce the analysis time by more than 50% because the QR algorithm requires less floating-point operations than the SVD algorithm and because the linear system to be solved is of a signifcantly smaller size.

The first two items positively affect the overall percentage of the code that is parallelisable and hence the scalability of the code. The last item also indirectly influences the scalability because it would reduce the scalar part present in the program.

Apart from these points where our existing model could be improved there is a matter of more general importance that came up in the analysis of our timing experiments: the I/O configuration of the system used. When significant I/O has to be done the scalability of a program can be seriously affected by a non-optimal I/O configuration, whether this is only due to the hardware configuration or also may have an Operating System component. In such cases it may be difficult to project the performance of a program even on systems of the same vendor because when the I/O configuration is different this has an impact on the scalability of the program at hand.
In our particular case, the behaviour in the initialisation phase is less important: for testing purposes we only considered two evolution cycles following the initialisation. In realistic modelling runs many more of such cycles will be computed which makes the initialisation relatively insignificant.

Acknowledgments

We would like to thank the Dutch National Computer Facility Foundation (**Stichting NCF**) for its National Research Grant, NRG 99-03, which made this work possible. Also we thank Compaq for providing the opportunity to work on their AlphaServer SC and AlphaServer GS160. In particular we are grateful to Michael Lough at Compaq in Galway, Ireland for his help in porting and running the code on their machines. Furthermore, we are grateful for the Oceanographic work we could do under the Dutch National Research Program, grant 013001237 and the EEC DIADEM project, no. MAS3-CT98-0167.

References

[1] A.F. Bennett, *Inverse Methods in Physical Oceanography*, Cambridge University Press, 1992, 346pp.

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers Inc., January 2001.

[3] G. Evensen, P.J. van Leeuwen, *An ensemble Kalman smoother for nonlinear dynamics*, Mon. Wether Rev., **128**, 1852–1867.

[4] P.J. van Leeuwen, G. Evensen, *Data assimilation and inverse methods in terms of a probabilistic formulation*, Mon. Weather Rev., **124**, 2898–2913.

[5] P.J. Van Leeuwen, *An ensemble smoother with error estimates*, Mon. Weather Rev., **129**, 709-728, 2001.

[6] R.N. Miller, M. Ghil, F. Gauthiez, *Advanced data assimilation in strongly nonlinear dynamical systems*, J. Atmos. Sci., **51**, 1037–1056.

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI: The Complete Reference Vol. 1, The MPI Core*, MIT Press, Boston, 1998.

[8] W. Gropp, S. Huss-Ledermann, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir *MPI: The Complete Reference, Vol. 2, The MPI Extensions*, MIT Press, Boston, 1998.

[9] OpenMP Forum, *Fortran Language Specification, version 1.0*, Web page: `www.openmp.org/`, October 1997.