COSTA
common set of tools for the assimilation of data

MEMO	CTA memo200606
Date	01-12-2006
Author(s)	Nils van Velzen
Subject	The COSTA parallel modelbuilder

Document information

Version	Author	Date	Description	Review
1.0	CvV	2006-12-01	Initial version	
File location:		<COSTA_DIR>/doc/model_builders/m06006_modbuild_par		

Table of contents

1 Introduction

The COSTA environment makes a number of building blocks available for creating data assimilation and calibration systems. Combining and creating building new building blocks should be possible with a minimum of effort.

COSTA contains tools for rapidly creating COSTA model components. These tools are called modelbuilders. This document describes the parallel model builder. This modelbuilder will create a mode-parallel model from an arbitrary COSTA model.

This memo describes the first version of the parallel modelbuilder. Section 2 describes the form of parallelization implemented by the parallel modelbuilder. The usage of the modelbuilder and how to adjust your existing sequential data assimilation system is described in Section 3. The design of the parallel modelbuilder is described in Section 4. The modelbuilder is tested with a number of models and data assimilation methods. The results of these tests are presented in Section 5. This document describes the initial version. The testing and development stages yielded some ideas for future improvement of the modelbuilder. These ideas are presented in Section 6.

2 Mode-parallel

Data assimilation and model calibration algorithms need to perform a large number of model computations for a given timespan, called model propagations. These propagations are often very computational expensive and dominate the total time needed to do the assimilation or calibration run.

Not all propagations depend on each other, therefore it is possible to perform them in

arbitrary order and in parallel. For example:

- The propagations of all ensemble members of an ensemble Kalman filter;
- The propagation of the L-matrix of the RRSQRT-filter and the 'central' state.
- Computation of the gradient of the model with respect to some model parameters using a finite difference approach.

The COSTA parallel modelbuilder makes it possible to perform the propagations of different model instances in parallel. Significantly decreasing the computational time for most data assimilation applications.

3 Using the COSTA modelbuilder

3.1 Adjusting the code

In order to use the parallel modelbuilder it is necessary to make a small extension to the code of an (existing) COSTA data assimilation system. After the initialization of the model the function `CTA_MODBUILD_PAR_CREATECLASS` must be called. From that point the parallel modelbuilder is initialized and the available processes are divided into a master process and several worker processes.

`CTA_MODBUILD_PAR_CREATECLASS(modelcls)`
`OUT modelcls` Class handle of the parallel modelbuilder

```
void CTA_Modbuild_par_CreateClass (CTA_ModelClass *modelcls)

CTA_MODBUILD_PAR_CREATECLASS (MODELCLS)
   INTEGER MODELCLS
```

Note that the worker processes will not execute any code after this function call. As a consequence, the modelbuilder must be initialized after the class initialization of the model. The parallel model class handle must be used for the creation of all models. The XML-configuration of the modelbuilder will realize the link with the existing simulation model. This is explained in Section 3.2.

The First lines of a typical application using the parallel modelbuilder are similar to the code in Table 1.

```
call cta_initialise(retval)
!
!    Initialise model (oscillation model)
!
call oscill_model_createfunc()
!
! Initialise parallel modelbuilder and start workers
!
call cta_modbuild_par_createclass(CTA_MODBUILD_PAR)
!
!    Process input and call method
!
   :
call cta_finalise(retval)
```

Table 1: *Main program of a data assimilation system using the parallel modelbuilder.*

3.2 input-file

The input of the modelbuilder is quite simple. The input of a simulation system using the parallel modelbuilder is given in Table 2. In the current version only two things need to be specified:

- name (tag) of the modelclass of the model,
- the name of the input-file of the model.

```
<modelbuild_par modelclass="modelbuild_sp">
   <model>oscill.xml</model>
</modelbuild_par>
```

Table 2: *The input-file of the parallel model-builder*

As we can see there is one limitation compared to arbitrary models. The model-input can only consist of a single string, in most cases the name of a file containing the model configuration. Most models can however be quickly adjusted, when necessary. COSTA provides a special function `CTA_Model_Util_InputTree` for handling the input and configuration of model instances. If a COSTA tree is given at model creation nothing is done. When however a string with the name of XML-configuration file is passed, the file is parsed and the corresponding configuration tree is created.

`CTA_MODEL_UTIL_INPUTTREE(hinput, tinput, cleanup)`

IN	`hinput`	Models configuration; a string of the configuration file or a COSTA tree
OUT	`tinput`	COSTA tree with model configuration. When `hinput` is a COSTA tree than `tinput` is equal to `hinput`.
OUT	`cleanup`	Flag `CTA_TRUE`/`CTA_FALSE`. When `tinput` is a filename, a COSTA tree is created and this tree must be cleaned/freed by the caller of this function.

```
int CTA_Model_Util_InputTree(CTA_Handle hinput, CTA_Tree *tinput,
                             int *cleanup)

CTA_MODEL_UTIL_INPUTTREE(HINPUT, TINPUT, CLEANUP)
   INTEGER HINPUT, TINPUT, CLEANUP
```

3.3 Axpy model between two model instances

The Axpy operation between two models is supported in the parallel model builder. There is a limitation. In order to perform the Axpy for models that are on different workers a copy of the model on the remote worker must be created. Therefore it is necessary that Export and Import methods of the model are implemented.

3.4 Random numbers

Stochastic models will use a random generator. When working with multiple processes we must be careful. There is a change that all processes will generate the same sequence of random numbers leading to undesired results.

The modelbuilder handles this issue. Therefore nothing needs to be done for models that use the COSTA random generator `CTA_RAND_N` or a random generator that is based on the `rand` function of C and did not set the random seed (`srand`).

Models that use a different random generator must be checked and optionally adjusted.

3.5 Running

The parallel modelbuilder is uses the MPI system for starting up the processes and handling the communication between these processes. The program `mpirun` or `mpiexec` must be used to start the application. It is far behind the scope of this document to describe all features

of MPI, `mpirun` and `mpiexec`. We will only give some instructions on how to start up your program in a way that will work for most MPI distributions.

In order to startup your program `myfilter.exe` using 5 processes type on the command line:

```
% mpiexec -n 5 myfilter.exe
```

or

```
% mpirun -np 5 myfilter.exe
```

In this case we will have 1 master and 4 workers.

Some MPI distribution allow to start a sequential simulation without the use of `mpirun` or `mpiexec`.

```
% myfilter.exe
```

However this does not always work. If this does not work, start the sequential version of the application using:

```
% mpiexec -n 1 myfilter.exe
```

Note that the master process is idle for most assimilation methods when the workers are performing their computations. Therefore it is possible to start $n + 1$ processes on n processors.

The model instances are equally distributed over the available workers. For that reason it is wise to select the number of model instances e.g. ensemble members as a multiple of the number of worker processes. When this is not done not all resources are optimally used. The propagation of 11 modes on 5 processors will take as much time as the propagation of 15 modes on the same number of processes for example.

4 Technical aspects of the modelbuilder

4.1 Master worker

The goal of the parallel modelbuilder is to provide parallel computing and decrease computational (wall) time of computations on clusters of workstations or advanced multiprocessor machines. The modelbuilder must be easy to use with no or minimum adjustments to existing models and assimilation method implementations.

COSTA models have no idea about the context they are used in. Therefore they have no clue on what methods in what sequence are called. Configuration files describing all

algorithmic steps can be used to tackle this. This approach will result in an optimal performance and is used in [**?**]. The creation of a detailed configuration file it quite complicated and configuration files have to be written for all assimilation methods. To overcome this problem a master worker approach is chosen for the parallel modelbuilder.

In the master worker approach we use a single master process. In our case this process will run the assimilation algorithm like the sequential case. The remaining processes are worker processes executing all model component related computations. Depending on the number of model instances a worker holds one or more model instances.

The necessary information, the header variables of the call, will be send to the worker process when the master executes a method of a COSTA model. Whenever the method returns information, the master will wait for the worker to send the result back. Some optimization can be performed using non-blocking communication but this issue will be discussed in more detail in Section 6.7. The advantage of this approach is that the assimilation code does not need to be adjusted. All communication is handled by the parallel modelbuilder. The disadvantage is however that the parallelization will not be optimal.

4.2 Sequential bottleneck

The parallel modelbuilder will speedup most data assimilation systems but the improvement is limited by the sequential parts in the assimilation method. We will illustrate this using the ensemble Kalman filter.

A time step of the ensemble Kalman filter consists of the following steps:

1. Propagate all ensemble members
2. Assimilate observations and adjust state of all modes

The propagation part 1 is performed in parallel. In the most optimal situation it will take the time of a single mode propagation. The assimilation part 2 is performed by the master process. The usage of the modelbuilder will not improve the computational time of this part[1].

The wall time of the computations in the sequential run is approximately defined by

$$t_{wall} = nt_{prop} + t_{assim} \tag{1}$$

Where n denotes the number of members in the ensemble. Increasing the number of workers will only influence the wall time of the propagation. The lower bound on the computational time using the parallel modelbuilder is therefore

$$t_{wall} = t_{prop} + t_{assim} \tag{2}$$

The simulation results that are presented in Section 5 will illustrate this behavior.

[1]Due to some communication with the model for performing interpolation and setting of states it is likely that this step will be more expensive than in the sequential case

4.3 Sequential runs

A data assimilation system that uses the parallel modelbuilder should be equally efficient as the sequential implementation when running on one process. The parallel modelbuilder recognizes when it is initialized in a sequential run. In that case all methods of the model are directly connected to the parallel modelbuilder.

4.4 Communications

The master will send information to the worker when a method of the model is called and will wait for the results. In this section some insight is given on the actual implementation.

4.4.1 Local administration

For each model instance the master holds a limited administration currently only containing the rank of the worker that holds the model instance and the (integer) handle of the model instance at the worker.

4.4.2 Packing of data

In order to send information between the processes it is possible to pack and unpack COSTA objects like vectors state vectors and observation description instances. When an object is packed all information is copied into a continuous block of memory. The unpack operation will reconstruct the object from the packed information.

The COSTA pack component is used to hold packed data. Multiple components can be packed and unpacked into a single pack component. The export and import methods of components will perform the actual packing and unpacking.

4.4.3 Communication sequence

The parallel modelbuilder will perform the following actions when the worker executes a method of the model.

The master performs the following steps:

1. Send the name of the action including the local model handle to the worker handling the model instance,
2. (optional) pack all data; input arguments of method,
3. (optional) send packed input arguments to method,

4. (optional) receive packed results from worker.
5. (optional) unpack results from worker.

The worker performs the following steps:

1. receive the name of the action and local model handle
2. (optional) Receive packed input arguments,
3. execute method,
4. (optional) pack returned data; output arguments of method,
5. (optional) Send packed output arguments to method.

Some of the steps are marked as optional. It depends on the method that is called whether these steps are performed or not.

5 Tests and performance

5.1 Small test-models

The parallel modelbuilder is tested using two existing COSTA test models;

1. the heat model with the COSTA RRSQRT filter,
2. the oscillation model with the COSTA ensemble Kalman filter.

The purpose of these tests was to find errors in the modelbuilder. Since these models are very small we don not expect a lot of improvement in performance. These tests are therefore not run on a cluster of computers or a large multiprocessor machine.

Figure 1 gives the simulation results of a parallel and sequential run of the Ensemble Kalman filter for the oscillation model. The small differences in the results are due to different random seeds.

5.2 Lotos-Euros

Lotos-Euros is an operational used simulation model for air pollution in Europe. The model computations are computational expensive compared to the small models like the oscillation and heat model. The COSTA model component of this model was already available and therefore we have selected this model to look at the performance of the parallel model builder.

The tests have been performed on two different machines.

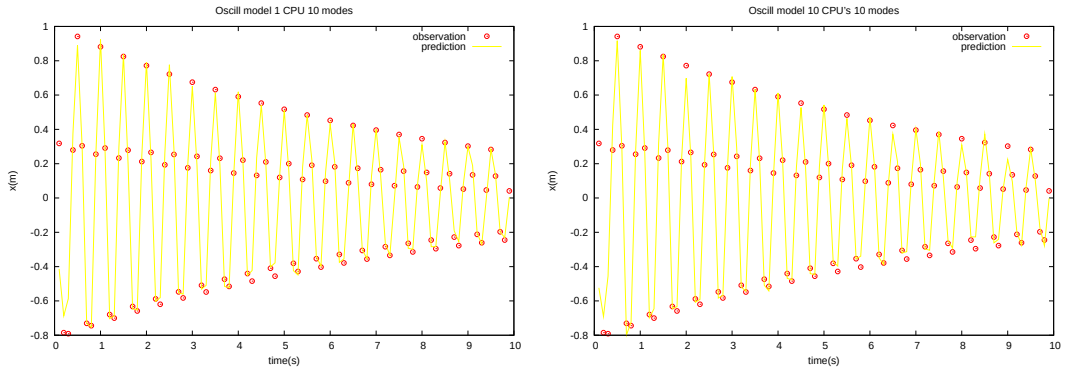

Figure 1: *Results of the Ensemble Kalman filter with 10 modes for the oscillation model. The left graph presents the results of a sequential run and the right graph the results of a parallel run with 10 workers. Differences are the result of different random seeds.*

- Laptop with Intel centrino duo, 1.66GHz CPU,
- Aster, an SGI Altix 3700 system with 416 Intel Itanium 2, 1.3GHz CPU's

The tests on the Intel centrino duo processor will illustrate that parallel computing is also useful on a single machine with a dual core processor. The simulation results are given in Table 3.

Gridsize	1 process	3 processes	Speedup
30x30x4	43.9 (s)	29.9 (s)	1.5
60x60x4	234.7 (s)	164.8 (s)	1.4

Table 3: Sequential and parallel run of the Lotos Euros model on a dual core processor. The run is an Ensemble Kalman filter with 24 modes for a simulation period of 2 hours.

The same simulation is also performed on the Aster on a number of CPU's varying from 1 to 12. The results are presented in Figure 2 and Figure 3. The speedup of the small model is much better than of the big model. We have not yet investigated this behavior. A possibility is that the sending of the states to worker is nonblocking for the small model as a result of an optimization in the MPI implementation and blocking for the large model.

5.3 Observation description

The default observations handling in COSTA is currently based on an SQLITE-database. Testing with the Lotos-Euros model revealed a performance issue. The time needed for reading from the database is not constant and requires a lot of wall-time, degenerating the performance of the parallel implementation.

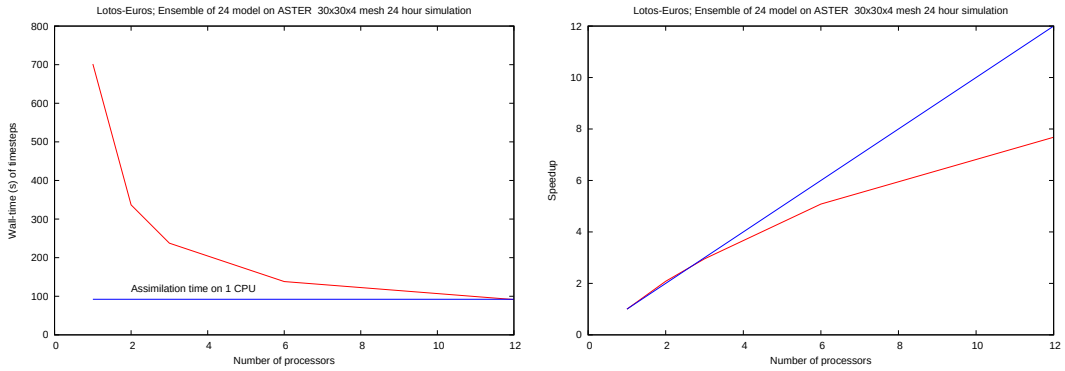

Figure 2: *Computational time and speedup. Lotos Euros simulation of 24 hours grid size 30x30x4 on ASTER. The ensemble Kalman filter used 24 model instances (23 members for covariance approximation and 1 background run)*

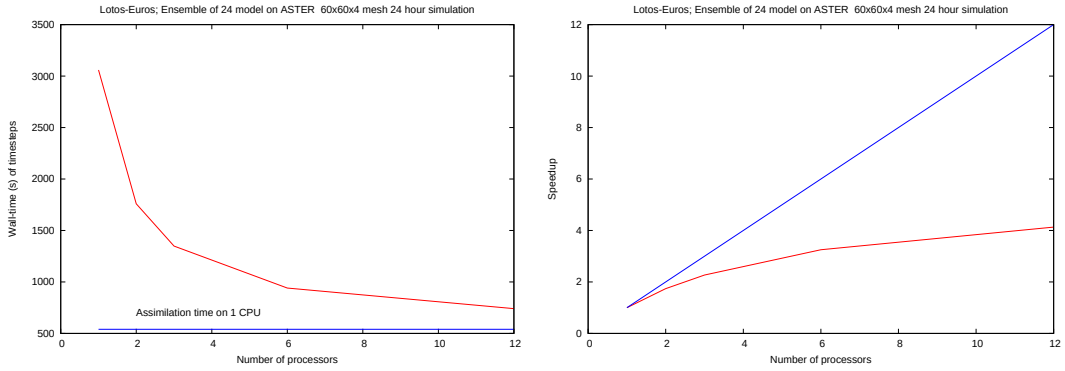

Figure 3: *Computational time and speedup. Lotos Euros simulation of 24 hours grid size 60x60x4 on ASTER. The ensemble Kalman filter used 24 model instances (23 members for covariance approximation and 1 background run)*

The parallel modelbuilder will will remember the data from the last send observation description. This will save some time because the information only needs to be read from the database once and not for all workers. This performance issue needs to be investigated in more detail in the future.

6 Future improvements

6.1 MPI initialization

The current implementation uses the `MPI_COMM_WORD` communicator and initializes MPI. For creating combinations with models that use MPI by them selves, this does not work. The initialization of the parallel model builder must be extended in order to create separate communication groups for the COSTA master and worker processes and the used model implementation.

6.2 Model state on master

The sequential bottleneck of the getstate method can be overcome by holding a model state of each model instance at the master. The value of this state is set using a nonblocking communication after every propagation.

This option can be switched on in the configuration file.

6.3 States on workers

The worker processes are now limited to holding model instances. The implementation of the workers is general and not limited to handling models. An extension to the parallel modelbuilder is the possibility to remotely hold state-vectors. This has two advantages:

1. part of the linear algebra computations with state-vectors in the assimilation part of the data assimilation method can be performed in parallel resulting in a better performance,
2. when the computations are performed on a non-shared memory machine the memory consumption is better distributed over the various machines and larger models and more model instances can be handled.

6.4 Constant obsselect

When the obsselect method results a constant selection criterion during the whole simulation. It can be specified in the input, saving communications.

6.5 Column selection for observation description

When the model must provide values that correspond to the observation description component, the whole observation description component is packed and send. In some cases not all data is needed by the model to perform the interpolation. Some of the data is only relevant for post processing etc. The column selection option in the configuration file of the parallel modelbuilder will specify what columns of the observation description component need to be send to the model.

6.6 Mode on all workers

In some situations like RRSQRT we have a central mode that is used in a large number of computations. A lot of communication can be saved when the value of this central state resides on all workers. This option is specified in the input-file of the modelbuilder.

6.7 Nonblocking communication

All receiving of data is currently blocking. This means that the Master must wait until the worker has send the requested data. MPI offers the possibility of a nonblocking receive. In the case of a non blocking receive the master will indicate at what position in memory it expects the data to put and continues. At the point the Master is actually going to use the data it has to wait until it is received.

The usage of the non-blocking receive can eliminate some of the sequential parts of the filter algorithm. For example; the assimilation method wants to have a copy of the states of all model instances using the following code:

```
do imode = 1,nmode
   call cta_model_getstate(hmodel(imode),sl(imode),ierr)
enddo
```

In the current implementation this procedure is sequential. Because no new state is requested before a state is received and unpacked.

Using non-blocking receive (what will probably need an extension of the state-vector) this operation can be performed parallel.