

Система совместного моделирования 2.0

В. В. Калмыков

2011-2013

Содержание

1	Введение	2
2	Общие принципы и обозначения	3
3	Схема работы совместной системы	5
4	Подробные шаги по подключению модели	8
5	Описание интерфейса	13
5.1	comp_reg_model()	13
5.2	comp_reg_array()	15
5.3	comp_send_intgrls()	16
5.4	comp_halo_updateV<4,8>()	17
5.5	comp_halo_updateT<4,8>()	17
5.6	shared_error()	18
5.7	shared_timer()	18
5.8	shared_memory_usage()	19
6	Дополнительные блоки	20
6.1	offline-блок	20
6.1.1	Интерфейс	22
6.1.2	Замечания	23
6.2	Блок визуализации	24
6.2.1	Скрипт scl	24
6.2.2	Скрипт vct	25
6.2.3	Скрипт cmb	26
6.2.4	Скрипт itg	26
6.2.5	Замечания и примеры	26
7	Соглашения архитектуры системы	29
8	Соглашения по кодированию	31
9	Общие важные замечания	36
10	Изменения версий	42

1 Введение

Проблемой объединения нескольких физических компонент в одну совместную модель начали заниматься еще в 90-х годах. На сегодняшний день наиболее известными и используемыми исследователями являются совместные модели CCSM/CESM(*NCAR*), OASIS(*CERFACS*), FMS(*GFDL*). К сожалению, до сих пор не существует хорошей канонической системы, про которую будет точно известна ее эффективность, работоспособность и простота для произвольной модели компоненты.

Например, с 2004 года распространяется версия каплера OASIS3, признанная во всем мире как надежный инструмент для совместного моделирования. Тем не менее, эта версия содержит однопроцессорный каплер, что в условиях современных размеров сеток и количеств процессоров приводит к крайне низкой скорости вычислений. Полностью параллельная версия OASIS4 так и не завоевала популярности своей предшественницы, к тому же, была закрыта в сентябре 2012 года. Данных по эффективности последней системы OASIS-MCT пока (2013 год) нет.

Далее, в своей недавней работе (2011 год), группа создателей каплера CCSM сообщила, что теперь версия модели содержит только локальные массивы и готова к запуску на *ultrahigh resolution* сетках, что, как выяснилось, означает всего лишь $1/10^\circ$ океана и $1/2^\circ$ градуса атмосферы. То есть, казалось бы, естественное требование локальности данных (необходимое для больших задач) было выполнено совсем недавно. В 2012 году та же группа CCSM/CESM проводила проверку качества своей модели. Известно, что для долгосрочных симуляций существует неофициальная планка – 5 модельных лет должны быть просчитаны за 1 день астрономического времени. На 8000 ядрах компьютера Cray авторы достигли лишь отметки 2,5 года – это максимальный результат на всевозможных компьютерах и числах процессоров. Стоит отметить, что для тестов использовалась как родная система MCT, так и новая отдельно разрабатываемая ESMF. Наверное, ESMF сегодня является наиболее совершенной системой, но имеет один существенный недостаток – размер кода каплера составляет около 500000 строк кода на C/Fortran.

Отсутствие отработанной и общепризнанной эффективной системы, нежелание зависеть от чужого кода, сложность внедрения и стремление вести проект со своей спецификой вылилось в решение о создании собственной системы. За основу взяты общепризнанные стратегии построения совместных моделей, а именно: единый исполняемый файл, модульность компонент, параллельный каплер, параллельная система интерполяции на базе SCRIP, использование параллельной библиотеки работы с файловой системой HDF5/netCDF.

2 Общие принципы и обозначения

Каплер (от англ. *coupler*) – это компонента системы, объединяющая неограниченное число физических моделей, которые далее будут называться *физическими компонентами*. Схема работы - клиент-серверная, где каплер – параллельный, запускаемый на N процессорах сервера, а компоненты – параллельные, запускаемые на P , Q и т.д. процессорах физические модели-клиенты. Такая структура означает концентрацию сервисной нагрузки на сервере-каплере и связь компонент между собой только через него. Таким образом, структура всей системы имеет вид модели со *звездообразной топологией* (*hub-and-spoke model*) (Рис. 1).

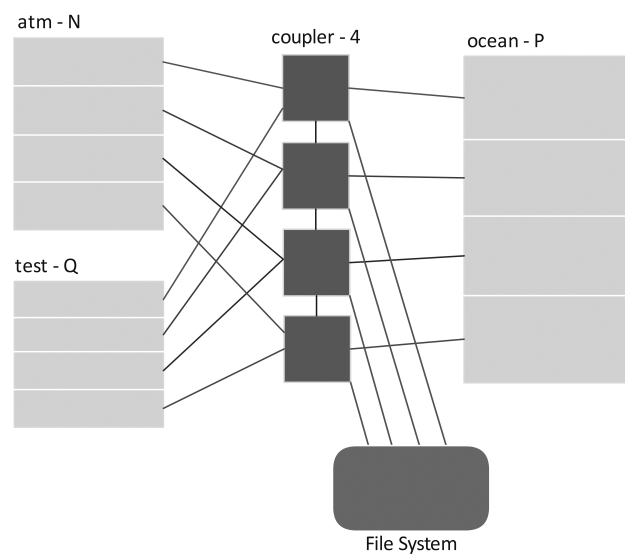


Рис. 1: Общая схема работы совместной модели.

Функции каплера (префикс " $c_$ ") являются скрытыми и ненужными для пользователя. Напротив, интерфейс функций компоненты (префикс " $comp_$ ") используется для связи компонент друг с другом и для выполнения других, общих для компонент задач (например, для обмена приграничными ячейками). Таким образом, под термином *совместная модель* понимается:

- несколько физических компонент, например, океан и атмосфера
- сервисная компонента-каплер (" $c_$ "), которая является сервером и производит, например, интерполяцию данных и работу с файловой системой
- оболочка интерфейсов (" $comp_$ "), которые позволяют компонентам с этим сервером связываться. Кроме того, в этот блок удобно инкапсулировать все похо-

жие функции компонент, не относящиеся непосредственно к работе с каплером, например, обмен приграничными ячейками

В результате, совместная система представляет собой единый исполняемый файл, в процессе выполнения которого на компьютере, несколько физических моделей работают параллельно и общаются друг с другом с помощью predetermined интерфейсов, которые связывают их либо с каплером, либо друг с другом.

Важно отметить, что работа каплера никак не зависит от работы компонент, то есть при разработке, например, модели океана гарантируется, что не потребуются изменять код в какой-либо папке, кроме океанологической. Верно и обратное – при расширении и изменении компоненты каплера не потребуются никакие изменения в коде физических компонент. В этом смысле, каплер представляет собой некий черный ящик с определенными для него интерфейсами вида “послать данные”, “записать данные”, и т.д. и конкретная реализация скрыта от пользователя, а физическая компонента – черный ящик с командами “посчитать шаг физики”, “проинициализироваться” и т.д. и конкретная реализация скрыта от разработчика каплера.

Руководство построено следующим образом: в разделе 3 описана общая структура системы, в части 4 объясняется, как внедрить свою модель, в части 5 приведено подробное описание интерфейса системы, в части 6 рассмотрены дополнительные блоки, в следующих главах содержатся необходимые соглашения, важные замечания и дополнительные сведения. По мере расширения модели руководство будет изменяться. Все изменения текущей версии будут фиксироваться в части 10.

Обязательно ознакомьтесь с разделом важных замечаний. В нем собраны основные вопросы об архитектуре системы и специфических моментах запуска на суперкомпьютерах.

3 Схема работы совместной системы

Одна из задач системы совместного моделирования – синхронизация компонент, то есть объединение отдельно разрабатываемых моделей в единый ансамбль. При наличии нескольких компонент, имеющих различную внутреннюю структуру, поддержка кода совместной модели становится крайне сложной задачей. В результате, чтобы ограничить пользователя от *любой* изменений за пределами его программы, но при этом позволить ему участвовать в совместной системе, реализована идея с производными классами.

В блоке *comp* определен базовый класс, который является некоторой абстрактной компонентой, содержащей в себе общие для нее функции, которые будут у любого экземпляра данного класса (например, функция отправки данных одинаковая для океана, атмосферы, льда и т.д.). Кроме того, класс содержит несколько *абстрактных* интерфейсов, которые не определены для базового класса и обязаны быть реализованы пользователем, так как зависят от конкретной модели. Примером может быть функция регистрации, которая, очевидно, различается для разных компонент.

Таким образом, для работы в совместной системе, пользователь должен создать класс, например, *test*, который наследует базовый класс *comp* и реализовать абстрактные методы. Теперь он может скомпилировать *библиотеку* своей модели. В результате, имеется некоторый черный ящик (именно так он выглядит для капера), с несколькими определенными функциями (например, послать данные, выполнить инициализацию). Что конкретно будет происходить внутри ящика, какие именно пользовательские функции будут вызываться, с точки зрения системы неважно.

После того, как каждая модель будет приведена к библиотечному виду, можно собрать единый исполняемый файл. Все черные ящики линкуются к главной программе системы *c_main()* – она является входной точкой и именно в ней происходит непосредственный вызов методов компонент (например, проинициализировать окна, выполнить шаг физики атмосферы, отправить диагностику льда и т.д.). Каждый процесс содержит свою ссылку *comp_p* на объект (черный ящик) компоненты. В результате, несмотря на единый код, каждое ядро вызывает именно свою специфическую функцию. Например, блок предварительной инициализации выглядит так:

```
call comp_p % ini_params
call comp_p % ini_reg_comp
call comp_p % ini_allocate
call comp_p % ini_reg_data
call comp_p % reg_sendrecv
```

Головной файл *c_main.f90* делится на: блок совместной инициализации (зайти

всем компонентам, получить коммуникаторы), блок предварительной инициализации (считать свои параметры, зарегистрироваться у каплера, разместить массивы, зарегистрировать данные у каплера), блок начальных данных (начать с контрольной точки или задать начальные условия), блок окончательной инициализации (доделать все, что нужно до цикла, с учетом считанных или заданных начальных условий), блок главного цикла (основная физика), блок построения (освобождение ресурсов, обработка результатов). Теперь подробнее:

- **блок совместной инициализации:** этот блок выполняется всеми запущенными процессами. В нем происходит чтение параметров командной строки, инициализация MPI и деление общего коммуникатора на локальные коммуникаторы компонент. Вообще, общий блок(префикс *shared_*) содержит функции и данные используемые как компонентами, так и каплером. Сюда входит модуль с определением констант (*shared_const_module.f90*), модуль общих данных (*shared_module.f90*) и т.д.
- **блок предварительной инициализации:** определяются данные, нужные для регистрации компоненты в системе каплера и задания начальных условий (или старта с контрольной точки). Сначала вызывается функция *ini_params()*, которая считывает и определяет главные параметры физической модели. Далее, процедура *reg_comp()*, регистрирующая компоненту в системе и выполняющая внутренние инициализации, с Далее, вы размещаете данные (*ini_allocate()*) и помечаете те их них, которые будут участвовать в обмене с каплером (*ini_reg_data()*).
- **блок начальных данных:** работу блока регулирует логический флаг *comp_read_cp*: пользователь сам задает начальные условия (*.FALSE.*) или стартует с контрольной точки (*.TRUE.*).
- **блок окончательной инициализации:** здесь вызывается процедура *ini_main()*, где происходит любая пользовательская инициализации, полностью подготавливающую компоненту к главному вычислительному циклу(инициализация физики, первые обмены приграничными ячейками и т.д.) Далее происходит нулевая отправка потоков для переинтерполяции другим компонентам.
- **блок главного цикла:** здесь определяется главный цикл по шагам(шаги различны для разных компонент и определяются при регистрации). Внутри цикла происходит:

при наступлении определенного шага подкачка файловых данных (ACTION_READ_FD);

вызов подпрограммы `main_step()` содержащей всю физику одного шага компоненты и внутренних обменов(например, halo-обмены);

при наступлении определенного шага посылку граничных условий для переинтерполяции внешней компоненте процедурой (ACTION_MAPPING);

при наступлении определенного шага прием переинтерполированных граничных условий от внешней компоненты процедурой (ACTION_MAPPING);

при наступлении определенного шага сохранение контрольной точки процедурой (ACTION_SAVE_CP);

при наступлении определенного шага сохранение диагностических полей (ACTION_SAVE_DG);

Все перечисленные процедуры приема и посылки данных выполняются автоматически на основе переданных при регистрации периодов обмена и адресов массивов.

- **блок построцессинга:** освобождение памяти и закрытие файлов. Здесь в будущем возможна обработка результатов.

4 Подробные шаги по подключению модели

- базовое изменение структуры внедряемой модели

Предположим, что перед вами модель *atm_inm*. Все время до этого, она работала суперкомпьютере в параллельном или последовательном режиме. Инициализация (например, `MPI_INIT()`), работа с файловой системой и т.д. выполнялись средствами этой же модели. Поскольку теперь компонента *atm_inm* станет частью сложного ансамбля, необходимо придать ее структуре более интерфейсный вид, выделив некоторые общие функции. Мы создадим единственный модуль, который будет промежуточным звеном между этой моделью и остальной системой: система будет вызывать процедуры этого модуля, а он, в свою очередь, внутренние процедуры модели атмосферы.

Для того, чтобы пользователь не забыл определить все необходимые процедуры, они реализованы в виде *абстрактных интерфейсов*. Это означает, что в системе определен общий для всех компонент тип данных `comp`, который содержит как уже готовые функции (например, послать данные), так и процедуры которые обязан определить пользователь в зависимости от своих целей (например, регистрация, отдельная для каждой модели). При отсутствии реализации одного из абстрактных методов, компилятор выдаст сообщение об ошибке.

В итоге, сначала пользователь должен создать модуль, например, `cpl_atm_inm_module` и определить в нем тип, например, `cpl_atm_inm`, который *наследует* базовый класс `comp`. Теперь, он может пользоваться всеми процедурами базового типа и должен определить несколько своих, реализовав тем самым абстрактные интерфейсы базового типа. Например, модуль компоненты океана `cpl_ocn_module` начинается так:

```
TYPE, EXTENDS (comp) :: cpl_ocn

CONTAINS
    PROCEDURE :: ini_params => o_ini_params
    PROCEDURE :: ini_reg_comp => o_ini_reg_comp
    PROCEDURE :: ini_allocate => o_ini_allocate
    PROCEDURE :: ini_reg_data => o_ini_reg_data
    PROCEDURE :: ini_main => o_ini_main
    PROCEDURE :: main_step => o_main_step
END TYPE cpl_ocn

CONTAINS

subroutine o_ini_params(this)
```

...

То есть модель океана обязуется обеспечить конкретную реализацию следующих методов:

- `ini_params()`: считывание параметров из входящих именованных списков и определения ваших часто изменяемых величин
- `ini_reg_comp()`: регистрация модели в системе и получение некоторых глобальных переменных с помощью вызова процедуры `reg_model` (Раздел 5.1).
- `ini_allocate()`: размещение всех динамических массивов в виртуальной памяти
- `ini_reg_data()`: регистрация массивов, которые будут участвовать в межкомпонентных обменах с помощью процедуры `reg_array` (Раздел 5.2).
- `ini_main()`: вся инициализация, необходимая вам для счета
- `main_step()`: вся физика модели за один шаг по времени (сам временной цикл контролируется системой на основе входных параметров)

Обратите внимание – несмотря на то, что вы наследовали базовый класс `comp`, главная программа `s_main()` ничего об этом не знает, так как к ней не подключен модуль новой компоненты `cpl_atm_inm_module` и не создан экземпляр класса `cpl_atm_inm`. Это будет сделано автоматически, с помощью *макроподстановок* на этапе компиляции главной программы – для всех объединяемых библиотек будут добавлены модули, созданы объекты класса и ссылки на них будут переданы системе. Автоматическая работа накладывает небольшие ограничения на именование файлов компоненты:

- название папки модели **должно** иметь вид `comp_atm_inm`
- имя производного класса **должно** иметь вид `cpl_atm_inm`
- имя содержащего его модуля **должно** иметь вид `cpl_atm_inm_module`
- имя модели **должно** иметь вид `<название климатической компоненты>_<уточнение>`, то есть например, `atm_inm`, `atm_ncar`, `atm_test`.

Соблюдая эти соглашения, пользователь полностью ограничивает себя от работы вне своей папки модели – все необходимые структуры будут созданы автоматически.

- **компиляция модели**

Теперь пришло время скомпилировать библиотеку вашей модели. Вы можете использовать любые приемы (Makefile, bash-скрипт) и флаги компиляции. Для того, чтобы процедуры каплера и внешних библиотек netCDF были найдены компилятором, не забудьте подключить опцию `-I../coupler/include` `-I../software/include`. Результатом работы вашего компиляционного файла должно быть следующее:

- в папке `./comp_atm_inm/lib` лежат все необходимые библиотеки, требуемые для работы вашей модели, то есть по крайней мере один файл – библиотека компоненты `atm_inm`. Если вы используете внешние библиотеки, например, *mk1* – они также должны располагаться здесь. Кроме того, в данной папке необходимо создать файл `lib-flags`, содержащий все необходимые ключи для подключения библиотек. Например, он может выглядеть так:

```
#!/bin/bash
echo "-latm_inm -lmkl"
```

То есть при создании исполняемого файла, система зайдет в вашу папку, выполнит данный скрипт и получит описание того, какие библиотеки вы хотите подключить. Такой подход позволяет создать черные ящики уже не на уровне кода, а на уровне папок, чтобы инкапсулировать все частности внутри директории и не нагружать остальных пользователей знанием того, что именно должна подключить модель *atm_inm*.

- в папке `./comp_atm_inm/include` лежат все необходимые `.mod`-файлы, требуемые компилятором.

- **сборка совместной системы**

Теперь, когда библиотеки отдельных компонент скомпилированы (или получены от других разработчиков), можно приступить к созданию единого исполняемого файла совместной системы. Для этого, в корне папке должен быть выполнен скрипт `make` со списком компонент, которые необходимо объединить, например:

```
bash make ocn atm_inm
```

Данная команда попытается найти в папках `comp_ocn` и `comp_atm_inm` папки `lib`, `include`, подключить все описанные в файле `/lib/lib-flags` библиотеки и создать единый исполняемый файл для совместной модели океан-атмосфера.

- **задать параметры запуска**

Во входящем именованном списке *run_list.in* можно определить необходимые входные параметры запуска системы.

```
&RUN_INF
time_start_year = 1988,
time_start_month = 3,
time_start_day = 15,

time_save_cp = 12,
time_append_cp = .TRUE.,
time_read_cp = '01021988',
time_sync_fs = 2,
run_save_log = .TRUE.
run_log_file_num = 6
/
```

time_start_*: начальные год, месяц, день счета

time_save_cp: период в часах сохранения общесистемной контрольной точки (0 если сохранение не требуется)

time_append_cp: режим записи контрольной точки – делать последовательность точек(.TRUE., текущее время отображается в имени файла) или каждый раз перезаписывать(.FALSE., используется файл с постфиксом REWR)

time_read_cp: временная строка для имени файла, из которого будут считаны данные в случае старта с контрольной точки. В случае задания искусственных начальных условий данное поле игнорируется. Например, для старта с 1 февраля 1988 года укажите **time_read_cp = '01021988'**, а для перезаписываемой – **time_read_cp = 'REWR'**. Система выполнит поиск указанного файла и выдаст сообщение об ошибке в случае его отсутствия. Подробности в разделе 9.

time_sync_fs: период в контрольных точках реального сброса данных на диск(при небольшом значении даже в случае аварийной остановки кластера вы не потеряете диагностические данные)

run_save_log: флаг ведения каплером log-файла, фиксирующего все события системы и информацию о текущем запуске

run_log_file_num: номер дескриптора log-файла. Значение 6 соответствует стандартному потоку. Если вы хотите, чтобы вывод системы направлялся в отдельный файл, измените это число.

- **создать файлы интерполяционных весов**

Для связи с остальными компонентами необходимо создать файлы маппинга, с помощью которых данные с сетки модели *atm_inm* интерполируются на другие сетки. Для этого предусмотрен *offline*-блок, полное описание которого приведено в разделе 6.1.

- **запустить задачу**

Теперь можно запустить программу, передав ей в качестве параметров размеры коммуникаторов запущенных компонент, строковое имя эксперимента (3 символа) и длительность запуска (годов, месяцев, дней). Например, запуск модели на 1 год, 4 месяца и 2 дня для эксперимента *'tst'* с 2 ядрами на каплер, 8 на океан и 2 на атмосферу на компьютере *Ломоносов* выглядит так:

```
sbatch -p test -n 62 impi ./model.exe CPL 2 OCN 40 ATM 20 tst 1 4 2
```

Обратите внимание, что в отличие от компиляции, при запуске вы уже оперируете не с конкретными моделями (*ocn*, *atm_inm*), а названиями компонент земной системы: *OCN*, *ATM*). Например, в качестве модели атмосферы будет выполняться та, которая была скомпилирована шагом ранее, в данном случае, *atm_inm*.

Если вы еще не создали файлы интерполяционных весов, то компоненты не могут обмениваться полями, а это значит, что пока вы можете запустить модель только в монопольном режиме, например, так:

```
sbatch -p test -n 22 impi ./model.exe CPL 2 ATM 20 tst 1 4 2
```

5 Описание интерфейса

В данном разделе приведено описание существующего на данный момент интерфейса системы.

5.1 `comp_reg_model()`

Регистрирует вашу компоненту в системе, создает и размещает все необходимые структуры и определяет некоторые удобные глобальные переменные. Процедура должна быть вызвана в интерфейсе `reg_comp`.

Входные параметры:

`integer reg_comp_np`: число процессоров, запущенных для данной компоненты.

`integer reg_comp_i_size`: глобальный размер сетки по долготе(без учета дополнительных обрамляющих ячеек).

`integer reg_comp_j_size`: глобальной размер сетки по широте(без учета дополнительных обрамляющих ячеек).

`integer reg_comp_k_size`: вертикальный размер сетки.

`integer reg_comp_m_size`: количество трехмерных массивов в четырехмерном массиве. Если вы не используете такие массивы, сделайте это измерение равным 1.

`logical reg_comp_decomp_type_2D`: при делении горизонтальной сетки использовать двумерную декомпозицию и стараться сделать расчетные области процессоров максимально похожими на квадраты (`.TRUE.`) или использовать одномерную широтную декомпозицию (`.FALSE.`). При невозможности системой удовлетворить всем условиям разбиения буде выдано сообщение об ошибке.

`logical reg_comp_i_cycle`: наличие цикличности по широте. При значении `.TRUE.` обменные функции добавляют в ячейки западнее первой и восточнее последней циклические данные. При значении `.FALSE.` таких обменов производится не будет – такой вариант используется при счете на замкнутой области (например, море).

`logical reg_comp_bsc_grid`: наличие у сетки биполярной шапки. При значении `.TRUE.` обменные функции будут циклически обменивать массивы на биполярном Гринвиче этой сетки. Значение `.FALSE.` означает отсутствие биполярной части (только широтно-долготная сетка)

`integer reg_time_steps_day`: количество модельных временных шагов в день.

`integer reg_time_save_it`: период в минутах сохранения интегральной диагностики. Период 0 означает, что никаких действий производить не требуется.

`logical reg_comp_read_cp`: флаг, означающий старт с контрольной точки (`.TRUE.`) или задание собственных начальных условий (`.FALSE.`).

`integer reg_comp_halo_stencil`: максимальная ширина обычного обмена для `comp_halo_update()`-функций. Например, в большинстве случаев обмен происходит для ширины 1 и иногда (в океане это сейчас массив *ssl*) – для ширины 2. Тогда значение параметра должно быть равно 2. Система автоматически сгенерирует `MPI_INIT()` вызовы для всех ширин от 1 до `reg_comp_halo_stencil`. Вы можете обменивать массивы с любыми ширинами из этого промежутка.

`integer reg_comp_halo_wide`: дополнительная опция для возможных в вашей компоненте быстрых схем решения явных систем уравнений. Параметр определяет величину широкого обмена для таких схем. Сейчас используется в океане в блоке решения системы уравнений мелкой воды. Если вы не используете широкий обмен, передайте значение 0.

Результат работы:

Помимо регистрации компоненты в системе, появляется возможность использовать некоторые глобальные переменные. Какие-то из них общие для всех процессоров компоненты (например, `comp_np`), какие-то – разные для каждого процессора (например, `comp_west`). Во-первых, определяются все входные переменные без префикса `reg_` (то есть, `comp_i_size`, `comp_name` и т.д.). Во-вторых, дополнительно рассчитываются некоторые удобные параметры. Их список приведен ниже. Все переменные можно использовать, просто подключив модуль *comp_module* к вашей функции.

`character comp_name`: строковое имя компоненты.

`character comp_names`: строковые имена всех компонент совместной системы.

`integer comp_inp`: количество процессоров по долготе.

`integer comp_jnp`: количество процессоров по широте.

`integer comp_i_loc_size`: размер локальной процессорной области по долготе.

`integer ncomp_j_loc_size`: размер локальной процессорной области по широте.

`integer comp_west, comp_east, comp_south, comp_north`: координаты границ локальной области данного процессора. Изменяются от 1 до `comp_i_size` и от 1 до `comp_j_size` соответственно.

Кроме того, при подключении *shared_module* становятся доступными следующие глобальные переменные:

`integer rank_local`: локальный ранг процессора в данной компоненте.

`integer rank_world`: глобальный ранг процессора во всей системе.

`integer comm_local`: локальный коммуникатор для обменов внутри данной компоненты.

`integer run_num_of_comp, run_np_coupler, run_np_ocean, run_np_atm:` явные размеры системы.

`integer time_run_steps:` количество шагов во временном цикле модели.

`integer time_*:` а так же остальные переменные времени, рассчитываемые таймером. Их описание приведено в *shared_timer.f*.

Пример использования:

```
call comp_register(  
  reg_comp_np = run_np(COMP_OCN),      &  
  reg_comp_i_size = 360,                &  
  reg_comp_j_size = 180,                &  
  reg_comp_k_size = 49,                 &  
  reg_comp_m_size = 2,                  &  
  reg_comp_decomp_type_2D = .TRUE.,     &  
  reg_comp_i_cycle = .TRUE.,            &  
  reg_comp_bsc_grid = .TRUE.,           &  
  reg_time_steps_day = 96,               &  
  reg_time_save_it = 0,                  &  
  reg_comp_ic = ocean_ic,                &  
  reg_comp_halo_stencil = 2,             &  
  reg_comp_halo_wide = 1                 )
```

Листинг 1: Пример функции регистрации для компоненты океана

5.2 comp_reg_array()

Единый интерфейс для регистрации одного массива внешнего обмена в системе. Данная функция должны быть вызвана в процедуре `reg_data()` для каждого массива, который будет участвовать в послыке или приеме данных.

Входные параметры:

Все параметры являются необязательными для обеспечения единого интерфейса для разных вариантов событий.

`integer send_to:` идентификатор компоненты-получателя.

`integer recv_from:` идентификатор компоненты-источника.

`integer action:` идентификатор действия. Возможные значения: `ACTION_READ_IC`, `ACTION_READ_CP`, `ACTION_READ_FD`, `ACTION_MAPPING`, `ACTION_SAVE_DG`, `ACTION_SAVE_CP`.

`real(kind=4) DIM4_arr`, `real(kind=4) DIM3_arr`, `real(kind=4) DIM2_arr`, `real(kind=8) DIM2_kind8_arr:` массив данных, который будет участвовать в обмене. **Обратите внимание**, что данные к моменту вызова процедуры уже должны

быть размещены в памяти вызовом `allocate`. Система сохранит именно адрес массива, и в будущем будет просто брать или класть данные по этому адресу.

`character(*) name, long_name, units`: имя, расширенное имя и единицы измерения переменной.

`integer period`: период события в минутах.

`map_type`: тип маппинга для событий `ACTION_MAPPING`.

`filename`: имя файла для событий `ACTION_READ_FD`.

Пример использования:

В данном примере компонента помечает на начальное считывание массив `kv_real`, на прием от атмосферы массив `U_10_MOD_atm` и тот же массив `U_10_MOD_atm` на диагностику.

```
call this% reg_array(recv_from=COUPLER, action=ACTION_READ_IC ,
    DIM2_arr=kv_real, name="kv")
call this % reg_array(recv_from=COMP_ATM, action=ACTION_MAPPING ,
    DIM2_arr=U_10_MOD_atm, period = 60)
call this % reg_array(send_to=COUPLER, action=ACTION_SAVE_DG ,
    DIM2_arr=U_10_MOD_atm, name = 'U_10_MOD_atm', period = 120 )
```

5.3 comp_send_intgrls()

Выполняет сбор и посылку диагностических интегралов каплеру для последующей записи в файл. Интегралы должны быть уже рассчитаны для локальных областей – перед записью они будут только осреднены в виде взвешенной суммы по площади. Период в минутах передается через параметр `reg_time_save_it` в процедуре регистрации. Сама функция должна быть вызвана, например, в конце процедуры `main_step()`.

Входные параметры:

`real(kind=8) area`: площадь области, на которой присутствует характеристика.

`real(kind=8) ints_vector(:)`: интегралы, которые вы хотите сохранить.

`character(LEN=NAME_LEN) names_vector(:)`: соответствующие имена.

Пример использования:

```
call comp_send_intgrls(area, [int1,int2],[ 'int1','int2'])
```

5.4 comp_halo_updateV<4,8>()

Получает прилежащие к границе ячейки для функций, определенных в V-ячейках. Обменивает сразу 2 массива, так как функции, определенные в V-ячейках, часто идут парами (например, горизонтальные скорости).

Входные параметры:

`real(kind = 4,8) DIM4_u, DIM4_v, DIM3_u, DIM3_v, DIM2_u, DIM2_v`: необязательные аргументы в том смысле, что в вызове требуется указать только одну пару в зависимости от того, какие массивы вы обмениваете – 4D, 3D или 2D. Ожидается, что массивы определены на области, на которой вы хотите получить приграничные ячейки. То есть, например, если вы обмениваете массив `u_c` с шириной 4, то он должен быть определен (в начале программы оператором `allocate`) как `[iwest-4, ieast+4] × [jsouth-4, jnorth+4]`.

`integer update_width`: ширина обмена. Это значение должно быть из промежутка от 1 до `comp_halo_stencil` или равным `comp_halo_wide`, то есть определенным при старте максимальной шириной апдейта для обычных обменов и широких.

`integer change_sign_on_bipolar`: необязательный аргумент. Передайте -1, если хотите поменять знак при обмене на биполярном Гринвиче, 1 – если нет. При отсутствии биполярной сетки (этому случаю соответствует значение флага `comp_bsc_grid` = 0 в функции регистрации) параметр можно опустить.

Пример использования:

Обмен трехмерных массивов скоростей (обычной точности) с шириной обмена 1 и изменением знака на биполярном Гринвиче:

```
call comp_halo_update_V4(DIM3_u = u_f, DIM3_v = v_f, &
                        update_width = 1, change_sign_on_bipolar = -1 )
```

5.5 comp_halo_updateT<4,8>()

Получает прилежащие к границе ячейки для одной функции, определенной в T-ячейках.

Входные параметры:

`real(kind = 4,8) DIM4_t, DIM3_t, DIM2_t`: необязательные аргументы в том смысле, что в вызове требуется указать только один массив в зависимости от размерности – 4D, 3D или 2D. Ожидается, что массивы определены на области, на которой вы хотите получить приграничные ячейки. То есть, например, если вы обмениваете

массив `w_surf_c` с шириной 1, то он должен быть определен(в начале программы оператором `allocate`) как `[iwest-1, ieast+1]×[jsouth-1, jnorth+1]`.

integer update_width: ширина обмена. Это значение должно быть из промежутка от 1 до `comp_halo_stencil` или равным `comp_halo_wide`, то есть определенным при старте максимальной шириной апдейта для обычных обменов и широких.

integer change_sign_on_bipolar: необязательный аргумент. Передайте -1, если хотите поменять знак при обмене на биполярном Гринвиче, 1 – если нет. При отсутствии биполярной сетки(этому случаю соответствует значение флага `comp_bsc_grid` = 0 в функции регистрации) параметр можно опустить.

Пример использования:

Обмен четырехмерного массива свойств одинарной точности:

```
call comp_halo_update_T4(DIM4_t=t_f, update_width=1,           &
                        change_sign_on_bipolar=1               )
```

Обмен с шириной 2 двумерного массива уровня двойной точности:

```
call comp_halo_update_T8(DIM2_t=ssl_c, update_width=2,         &
                        change_sign_on_bipolar=1               )
```

5.6 shared_error()

Выводит сообщение об ошибке и завершает программу.

Входные параметры:

Единственный аргумент – строка об ошибке, подробно описывающая фатальную ситуацию.

Пример использования:

Вызов функции при фатальном нарушении условий в одной из функций океана:

```
call shared_error(TRIM(comp_name)//":_error_in_solar_function.")
```

5.7 shared_timer()

Вычисляет ряд текущих временных характеристик.

Входные параметры:

Неявно использует год начала счета `time_start_year`, количество шагов в модельном дне `time_steps_day` (определяются во входящем именованном списке), и глобальный временной шаг `time_l` для расчета ряда временных величин: текущего

года, месяца, числа, месяца в году, дня в году и т.д. Полный перечень рассчитываемых переменных приведен в комментариях к функции в файле *shared_timer.f90*. В дальнейшем возможно добавление необходимых вам временных величин.

Пример использования:

Вычисляет ряд временных величин, для использования которых требуется подключить *shared_module*:

```
call shared_timer
```

5.8 shared_memory_usage()

Анализирует системный файл процесса для определения текущих реальных(со всеми внутренними структурами, например, MPI) значений использования виртуальной памяти.

Входные параметры:

`character(*) compName`: имя проверяемой компоненты

`character(*) strMessage`: необязательный аргумент, содержащий информационное сообщение пользователя.

Пример использования:

Проверка изменений размера виртуальной памяти первого процессора океана до и после процедуры размещения массивов:

```
if(rank_local == 0) call shared_memory_usage(comp_name, "before_ini")
call o_ini_allocate
if(rank_local == 0) call shared_memory_usage(comp_name, "after_ini")
```

Наиболее важные параметры: `VmSize` – текущий размер виртуальной памяти, `VmPeak` – его пиковое значение, `VmRss` – размер данных, к которым уже происходило обращение(если вы еще не читали или изменяли аллокированные данные, то это число может быть небольшим), `VmHWM` – его пиковое значение, `VmRss` – размер статических данных. Описание остальных параметров можно найти в мануале к системному файлу `/proc/self/status`.

Всегда инициализируйте определяемые массивы, например, нулями, так как `VmSize` является "честным" значением виртуальной памяти только при условии, что к памяти происходило обращение. Подробности в разделе 7.

6 Дополнительные блоки

6.1 offline-блок

Данный блок предназначен для однопроцессорной довычислительной подготовки модели к запуску: построения сеток, файлов начальных условий и матриц интерполяционных весов. Для создания файлов интерполяционных весов компонента должна наследовать класс `GridConstructor`, определяющим все инструменты для работы с сеткой. Как и в случае с базовым классом компоненты `comp`, пользователь наследует как общие методы (например, добавления поля в файл начальных условий), так и несколько абстрактных интерфейсов, нереализованных в базовом классе, и требующих задания специфических для его модели действий. Сейчас, абстрактных методов три: построения сетки, добавления маски и добавления произвольных начальных условий в стартовый файл. Разделение первых двух функций обусловлено тем, что, например, для океана требуется сначала построить модельную сетку, переинтерполировать на нее топографию дна ЕТОРО и только потом, проанализировав данный файл и проведя чистку глубин, вычислить маску. От пользователя требуется заполнить поля экземпляра своего производного `center_lat`, `center_lon`, `corner_lat`, `corner_lon`, `imask` за счет вызова своих функций построения сеток. Методы добавления маски и начальных условий теоретически могут быть оставлены пустыми.

Например, в папке океана создаем файл `off_module.f90` в котором содержится модуль с производным классом `offline`-блока океана `off_ocn` (неважные строки пропущены):

```
module off_ocn_module

TYPE, EXTENDS(GridConstructor) :: off_ocn

CONTAINS

PROCEDURE :: make_grid => o_make_grid
PROCEDURE :: make_mask => o_make_mask
PROCEDURE :: make_ic => o_make_ic

END TYPE off_ocn

CONTAINS

! =====

subroutine o_make_grid(this)
```

```

call o_ini_set_grid

grid % z(1:grid % nz) = z(1:grid % nz)

do j = 2, jl-1
do i = 2, il-1
    grid % center_lat(i-1,j-1) = x2_t(i,j)
    grid % corner_lat(1,i-1,j-1) = x2_v(i,j)
    grid % corner_lat(2,i-1,j-1) = x2_v(i-1,j)
    grid % corner_lat(3,i-1,j-1) = x2_v(i-1,j-1)
    grid % corner_lat(4,i-1,j-1) = x2_v(i,j-1)

    grid % center_lon(i-1,j-1) = x1_t(i,j)
    grid % corner_lon(1,i-1,j-1) = x1_v(i,j)
    grid % corner_lon(2,i-1,j-1) = x1_v(i-1,j)
    grid % corner_lon(3,i-1,j-1) = x1_v(i-1,j-1)
    grid % corner_lon(4,i-1,j-1) = x1_v(i,j-1)
end do
end do

```

То есть, как и в случае с `cpl_ocn_module`, описанным в начале руководства, мы наследуем базовый класс с некоторыми предопределенными инструментами и, кроме того, обязуемся реализовать несколько абстрактных интерфейсов (в примере показан только сокращенный метод построения океанской сетки).

Правила именования остаются прежними: имена модуля и производного класса фиксированы, теперь с префиксом `off_`. Для работы с `off`-блоком, библиотека вашей компоненты должна содержать производный класс, наследующий базовый класс `GridConstructor`.

Работу блока регулирует скрипт `make` в папке `/off`:

```
bash make <команда> <имя компоненты> <аргументы>
```

Он попытается зайти в папку компоненты и подключить ее библиотеку к главному файлу программы `off_main`. Все необходимые подключения модулей и создание экземпляра компоненты выполнится автоматически (для этого и требуются соглашения в именованиях). Таким образом, и подготовительные работы с сеткой не требуют от пользователя изменений вне его папки – все необходимые действия контролируются за счет собственной реализации абстрактных интерфейсов.

На данный момент доступно четыре вида команд:

- **grd** – создание файла сетки и попытка добавления к нему маски. Если маска пустая, пользователь получает предупреждение.

```
bash make grd ocn 3600 1800 50 T
```

- **ics** – чтение входных файлов, их анализ и обработка, и добавление данных в стартовый файл.

```
bash make ics ocn 3600 1800 50 T
```

- **wts** – построение интерполяционных коэффициентов между двумя модельными сетками с помощью пакета CDO и SCRIP. Доступны все методы построения, определенные в SCRIP, например, консервативный, билинейный, бикубический, и т.д.

```
bash make wts bil ./data/src_cdo.nc ./data/dst_cdo.nc
```

- **int** – CDO-интерполяция двумерных и трехмерных полей. Необходима, например, для создания файлов начальных условий Левитуса. Данная опция пока работает в тестовом режиме.

```
bash make int ./data/ETOP05.nc topo ./data/OCN_720x360_V_cdo.nc.
```

6.1.1 Интерфейс

Вся работа блока выполняется над объектом класса, наследуемого от `GridConstructor`. Переданные из командной строки аргументы и тип сетки записываются в поля `nx`, `ny`, `nz`, `gType`. Процедура построения сетки дополняет объект полями `center_lat`, `center_lon`, `corner_lat`, `corner_lon`. Наконец, поле `imask` заполняется в процедуре добавления маски.

Приведенные ниже методы помогают пользователю работать с файловой системой полями, сетки которых определены объектами класса `GridConstructor`.

- `off_get_var()`

Считывает переменную из netCDF-файла.

Входные параметры:

`character(*) :: var`: имя переменной

`character(*) :: filename`: имя файла

`real(kind=4) :: arr_1D(:), arr_2D(:, :), arr_3D(:, :, :)`: необязательные аргументы, в которые будет считана переменная.

`real(kind=8) :: arr_2D_8(:, :), arr_3D_8(:, :, :)`: необязательные аргументы, в которые будет считана переменная.

`real(kind=4) :: misVal`: необязательный аргумент, при наличии которого будет попытка считывания потерянного значения (атрибут “_FillValue”).

Пример использования:

Считывание файла топографии в функции построения маски океана:

```
call this % get_var(filename, "topo",
                  arr_2D = hv(iwest(myrank):ieast(myrank),
                             jsouth(myrank):jnorth(myrank)))
```

- `off_put_var()`

Записывает поле во входной IC-файл начальных условий.

Входные параметры:

`character(*) :: var, long_name, units`: имя, расширенное имя и единицы измерения переменной

`character(*) :: filename`: имя файла

`real(kind=4) :: arr_1D(:), arr_2D(:, :), arr_3D(:, :, :)`: необязательные аргументы, из которых будет считана переменная.

`real(kind=8) :: arr_2D_8(:, :), arr_3D_8(:, :, :)`: необязательные аргументы, из которых будет считана переменная.

Пример использования:

Добавление в IC-файл океана массива топографии kv:

```
call this % put_var(var = "kv", long_name = "kv-array",
                  units = "unitless",
                  arr_2D = REAL(kv(iwest(myrank):ieast(myrank),
                                   jsouth(myrank):jnorth(myrank))))
```

6.1.2 Замечания

Работа блока требует установленного пакета CDO. В настоящее время рекомендуется работать с ним на локальной машине под *Ubuntu*. Установка:

```
sudo apt-get install cdo
```

6.2 Блок визуализации

Одна из проблем визуализации данных эксперимента моделей высокого разрешения – огромные размеры массивов, и связанные с ними временные затраты на перенос данных на локальную машину. Встроенная система рисования позволяет визуализировать массивы непосредственно в рабочей папке на суперкомпьютере.

Блок `/draw` использует один из самых качественных инструментов в данной области – питоновский модуль `PyNGL`. С помощью нескольких скриптов появляется возможность создавать *print quality*-изображения и скачивать уже не гигабайты массивов, а небольшой графический файл.

Сейчас для использования доступны 4 питоновских скрипта: для скалярных, векторных, комбинированных полей и интегральной диагностики. Их вызывает `bash-скрипт draw`, первый параметр которого и определяет то, что вы хотите сделать. Также вы можете создавать `gif`-анимацию.

Для отрисовки доступны любые двумерные поля, расположенные в любом временном трехмерном или двумерном `netCDF`-файле. Какие именно поля вы хотите сохранить в этом файле – температуру на уровне 17, поверхностные течения, вертикальную скорость, соленость у дна, атмосферные осадки – решать вам. Просто зарегистрируйте данные массивы в виде диагностических(`ACTION_SAVE_DG`) и передайте имена переменных программе рисования.

На легенде рисунка будут отображены данные эксперимента, а также информация, которую вы передавали при регистрации массива работы с файловой системой – подробное имя функции и единицы ее измерения.

Ниже представлен список доступных программ – пример вызова, входные параметры и тонкая настройка. Дело в том, что значения ресурсов и цветовая схема, используемая по умолчанию, уже позволяют создавать высококачественные рисунки. Тем не менее, исходя из ваших пожеланий, вы можете регулировать некоторые расширенные параметры отображения. Для этого необходимо заменить определения в первой(пользовательской) функции файлов `draw_scl.py`, `draw_vct.py`.

6.2.1 Скрипт `scl`

```
bash draw scl ../data/OCN_60x30_IC.nc ../data/OCN_60x30_000_DG.nc
          4 4 h_bt Orthographic 1 -90 90 0 360 100 True
```

Входные параметры: файл сеточной информации, файл диагностической информации, начальный и конечный временные отсечки, имя переменной изображаемого поля, тип проекции(`Orthographic`, `Stereographic`, `Mercator`, `Robinson`, `LambertEqualArea`, `Gnomonic`, `AzimuthalEquidistant`, `Satellite`, `Mollweide`,

CylindricalEquidistant, CylindricalEqualArea), уровень, прямоугольник вида [minLat, maxLat][minLon, maxLon], центральная долгота, флаг заливки материков.

Тонкая настройка:

- `colorMap` – одна из цветовых схем, названия которых можно посмотреть здесь: http://www.pyngl.ucar.edu/Graphics/color_table_gallery.shtml. Значение по умолчанию: "posneg_1".
- `minValue`, `maxValue` – минимальные и максимальные значения на легенде. По умолчанию (если оба параметра равны 0), данные величины вычисляются как минимум и максимум по массиву данных первой временной отсечки с 20-процентным зазором (на случай изменения максимума при построении видео). Вы можете задать собственные градации – цвета будут равномерно распределены между ними.
- `nglSpreadColorStart`, `nglSpreadColorEnd` – начальные и конечные индексы цветов, отображаемых на легенде. Индексы ссылаются на выбранную цветовую схему `colorMap`. Значения по умолчанию: 4 и 20.
- `cnFillMode` – способ заливки скалярного поля. Возможные значения: "AreaFill" "RasterFill" "CellFill". Последние два используются, если вы хотите показать, что данные именно модельные, то есть у каждой ячейки свой цвет. Значение по умолчанию: "AreaFill".
- `cnLinesOn` – флаг отображения изолиний. Значение по умолчанию: False.

6.2.2 Скрипт `vct`

```
bash draw vct ../data/OCN_60x30_IC.nc ../data/OCN_60x30_000_DG.nc  
1 4 u_bt v_bt Stereographic 1 -90 90 0 360 100
```

Входные параметры: те же, за исключением появления пары векторных функций и отсутствия флага заливки земли в конце.

Тонкая настройка:

- `vcRefMagnitudeF` – референс-вектор. Все вектора ваших полей, равные этому значению, будут нарисованы с длиной `vcRefLengthF`. По умолчанию, если это значение не задано, вычисляется максимум по обоим массивам векторных данных.
- `vcRefLengthF` – отображаемая длина референс-вектора. Значение по умолчанию: 0.03

- `vcMinFracLengthF` – задаётся в частях от вашего референс-вектора `vcRefMagnitudeF` и означает длину самого маленького вектора. Длины всех остальных векторов распределяются между длиной этого, самого маленького и длиной самого большого – референс-вектора. Значение по умолчанию: 0.3
- `vcMinDistanceF` – расстояние между стрелочками на рисунке. Этим параметром можно контролировать кучность течений. Значение по умолчанию: 0.005

6.2.3 Скрипт `cmb`

```
bash draw cmb ../data/OCN_60x30_IC.nc ../data/OCN_60x30_000_DG.nc
              3 4 h_bt u_bt v_bt Mercator 1 -90 90 0 360 100 True
```

Входные параметры: те же, за исключением наличия как скалярной, так и векторных функций. Вызывает скрипты `scl` и `vct`, так что все настройки соответствующих функций будут иметь место и на общем рисунке.

6.2.4 Скрипт `itg`

```
bash draw itg ../data/OCN_60x30_IT.nc 1 30 avTemp
```

Входные параметры: файл интегральной диагностики, начальная и конечная временные отсечки, имя переменной отображаемой диагностики.

6.2.5 Замечания и примеры

На суперкомпьютерах, возможно, придется запускать однопроцессорную задачу командой `mpirun`. Дело в том, что на больших сетках объем используемой оперативной памяти практически исчерпывает память управляющей машины, которая компилирует задачи всех пользователей. Поэтому, для детальных сеток запускайте скрипты не просто в `bash`, а с помощью системы очередей определенной на кластере. Например, для *Ломоносова*:

```
sbatch -p test -n 1 run ./draw <имя скрипта> <параметры>
```

Кроме того, так как на вычислительных узлах кластера обычно не установлены никакие внешние программы, объединять видео из отдельных кадров теперь требуется вручную. Для этого, после того, как программа создала необходимое число кадров, просто вызовите утилиту работы с изображениями:

```
convert -delay 70 'ls -v *.temp.png' my.gif
```

В этой команде с задержкой 70 сотых секунды создается gif-анимация `my.gif` из всех файлов, оканчивающихся на `.temp.png`, предварительно расположенных в

лексикографическом порядке. После создания нужных рисунков рекомендуется временные файлы командой:

```
rm -f *.temp.png
```

Ниже приведены несколько рисунков тестовых запусков с изображенными на них функциями `w_surf`, `h_bt`, `u_bt`, `v_bt` в разных проекциях. Данные массивы были заданы как диагностические(`ACTION_SAVE_DG`) в Листинге ??.

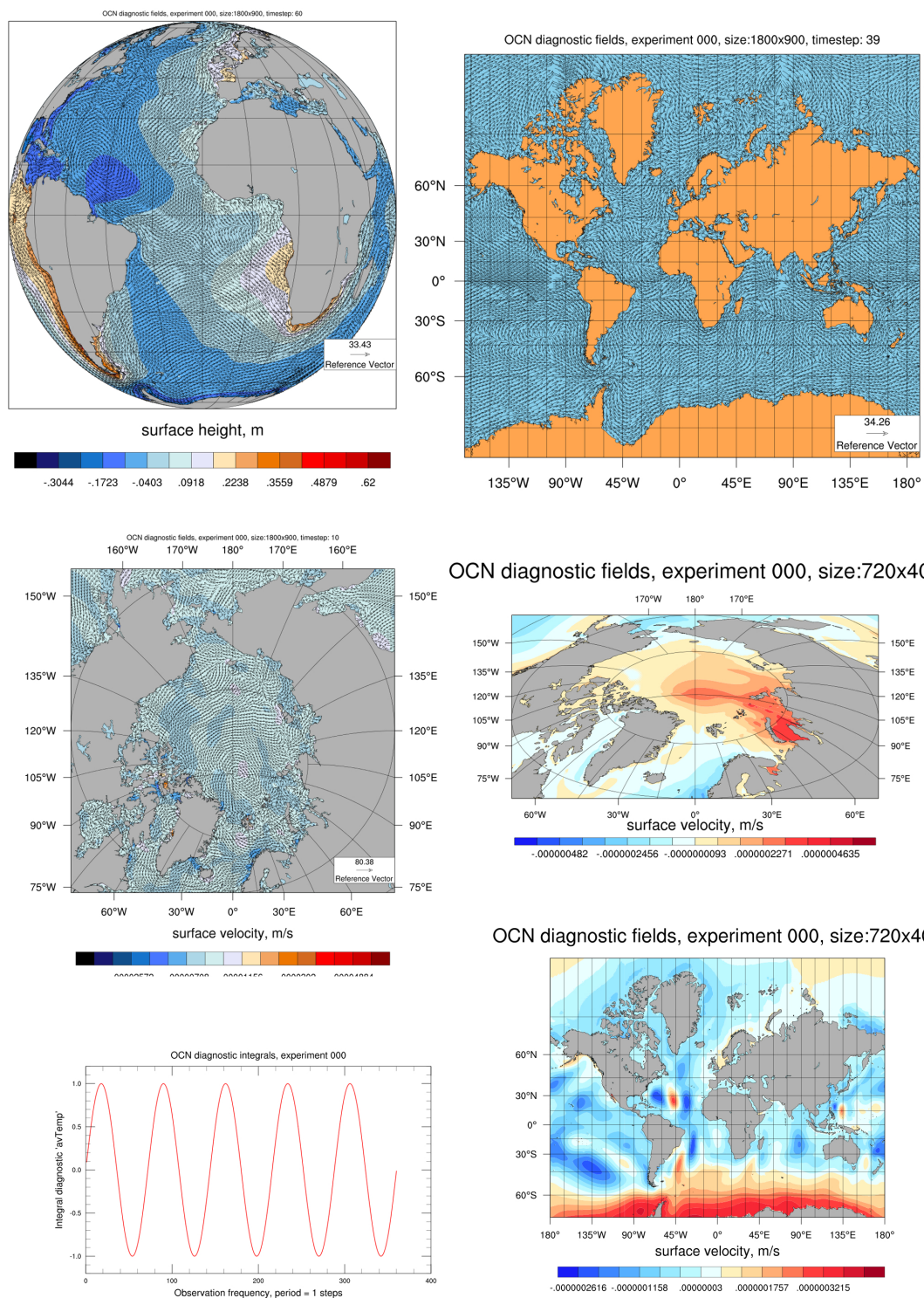


Рис. 2: Пример работы блока рисования

7 Соглашения архитектуры системы

- Не допускаются прямые обращения (не через каплер) к компонентам (прямая посылка другой компоненте данных через `MPI_SEND()` и к памяти (работа с файловой системой средствами `read/write`). Если все-таки вам нужен функционал, не определенный сейчас, мы обсудим его на семинаре и добавим в систему.
- В главном цикле не должно быть последовательных алгоритмов или таких, при которых процессоры ждут освобождения общего ресурса. Пример – ожидание окончания вычислений одного процессора с последующим распределением данных. По закону Амдала, ускорение всей системы не может быть выше обратного от доли последовательного алгоритма.
- Требуется соблюдать максимальную локальность данных. Запрещены глобальные трехмерные и четырехмерные массивы. Любой глобальный массив, участвующий в вычислениях, является участком кода, который не ускоряется при росте числа процессоров, так как количество вычислений не изменяется. Поэтому разрешены только глобальные двумерные массивы, не участвующие в вычислениях (например, на инициализации) и то, если они крайне необходимы. Память для этих массивов должна быть освобождена до начала цикла.
- Для передачи часто изменяемых параметров в вашу модель используйте именованные списки (*namelist*). Например, вы определяете переменную `run_days`, а в вашем *namelist*-е *input_params.in* пишете строчку `run_days = 10`. При считывании именованного списка переменной `run_days` будет присвоено значение 10. То есть для изменения значения переменной `run_days` перекомпиляция программы не нужна, требуется просто изменить *namelist*. В список имеет смысл выносить часто изменяемые параметры, например, размер задачи, или постоянно изменяемые физические характеристики форсинга. Все остальные, редко изменяемые переменные (например, гравитационную постоянную или положение биполярного полюса) логичнее определять константами внутри процедур.
- Одним из удобных соглашений является определение функции `ini_params()`, вызываемой самой первой в блоке инициализации компоненты и считывающей (из вашего или общего именованных списков) или определяющей все параметры модели. В моделях практически не используются статические (память под которые выделяется на этапе компиляции) массивы, и вся память выделяется на этапе выполнения в зависимости от числа процессоров. Поэтому старый метод задания констант в разных модулях становится бессмысленным с точки

зрения программы. Подход использования одной процедуры позволяет сосредоточить все объявления в одном месте.

Речь идет именно о константах(`parameter`) в модулях, которые требуется редко, но изменять. Разумеется, все физически параметризации и задания физических переменных это не касается и их следует определять в соответствующих модулях.

- **Всегда инициализируйте размещенные массивы**, например, нулями. Дело в том, что вызов `allocate()` на самом деле не выделяет память процессу, а только помечает ее в специальных системных таблицах. В результате, отработав, например, 50 лет и начав запись в файл, модель может упасть с ошибкой *insufficient virtual memory*, потому что только тогда к ней началось *реальное* обращение (например, запись в большой буфер послыки) и выяснилось, что памяти уже не хватает. Такие ошибки крайне сложно отследить, но гарантия того, что вы все контролируете – любое обращение к данным, например, инициализация массивов нулями. Для контроля использования памяти на этапе тестирования используйте функцию `shared_memory_usage()`. С помощью нее можно контролировать реальную загрузку всех видов памяти вычислительного ядра.
- Порядок индексов массивов для передачи каплеру: двумерных – (i, j) , трехмерных – (k, i, j) , четырехмерных – (k, i, j, m) . Гарантия того, что вы не забудете поменять индексы в каждом четырехмерном массиве программы (а их не так много) – флаги компиляции `-fpe0 -check bounds -traceback`. Они укажут вам на строчку и файл, где произошел выход за границы массива.
- Должна быть определена процедура финала финалайзинга – закрытие файлов и освобождение памяти. Пока это не обязательно, но, в будущем, модель должна быть освобождать все занятые ресурсы перед завершением. Так что, если вы пишете, например, новые модули, то аналогично процедуре `allocate_test_module()` рекомендуется(а в скором времени и будет требоваться) определение процедуры `deallocate_test_module()`. Подобное освобождение памяти может понадобится и на этапе постпроцессинга, который в будущем планируется разместить в конце программы.

8 Соглашения по кодированию

- Настоятельно рекомендуется использовать систему контроля версий(например, *TortoiseHg*). Это простой и удобный механизм отслеживания изменений, отката к предыдущим версиям и слиянием различных веток. Каждая версия дерева сопровождается вашими комментариями, которое позволяют кратко описать нововведения и следить за общим развитием проекта. Для каждой функции можно посмотреть все произведенные версии изменений и в случае неудовлетворительной работы быстро заменить новую версию на рабочую.
- Строго запрещены операторы `goto`, за исключением двух случаев, когда их использование оправданное: выход из нескольких вложенных циклов сразу и переход на обработку ошибки в случае возникновения таковой. Переход при этом разрешается только вперед. Для выхода из одной ветки цикла по условию используется удобный оператор `EXIT`(не функция `EXIT()`, завершающая программу, а именно оператор без скобок). Пример: `if(k > k_max) EXIT`.

Для перехода на следующую итерацию цикла используйте директиву `CYCLE`.

- Запрет переменных по умолчанию – директива `implicit none` в начале всех процедур. Эта директива запрещает использование неопределенных переменных, что очень часто является причиной незаметных ошибок. Например, в главном модуле вы переименовали константу `albedo` в `ocean_albedo`, а в одной процедуре внутри программы забыли это сделать. Без директивы `implicit none` компилятор не укажет вам на то, что переменная `albedo` больше не существует и в забытой процедуре она будет равна в лучшем случае нулю, а в худшем – неопределенному значению.
- Настоятельно рекомендуется переходить на формат кода стандарта `f90`. Это всего лишь означает избавление от неудобной фиксированной формы, оставшейся с 70 годов. Свободная форма позволяет делать строки сколь угодно длинными, что позволяет спокойно форматировать код без привязки к узкой полоске. Символ продолжения строки `&` при свободной форме ставится в конце строки, а не в начале.
- Рекомендуется использовать визуально более понятные операторы сравнения: например, `<=` вместо `.le.`, а также верхний регистр для констант и встроенных функций Фортрана(например, `MAX()`, `SIZE()`).

- Для множественного выбора рекомендуется использовать конструкцию CASE. То есть, например, код выбора схемы в зависимости от параметра мог бы выглядеть так:

```
select case(mix_mode)
  case(MIX_MODE_1)
    call sub1
  case(MIX_MODE_2)
    call sub2
  case default
    call shared_error(comp_me, "Unknown_scheme.")
end select
```

- Запрещаются так называемые *божественные значения* – численные параметры в тексте кода. Например, `arr(112) = coeff`. При изменении размера массива `arr` получим ошибку памяти, далеко не всегда сразу приводящую к завершению программы, которая остановится потом в произвольном месте. Кроме того, любые переменные для выбора (например, схемы по времени), следует обозначать понятными именами-константами, например `if(v_mix_mode == MIX_MODE_MUNK)` вместо `if(v_mix_mode == 1)`. Пример использования констант можно посмотреть в модуле `shared_const_module`. Аналогично, размеры массивов не должны задаваться числами – допускаются либо константы, либо переменные. Например, `real t_river(jl)` – верно, `real t_river(134)` – нет.

Кроме того, если размер некоторой таблицы задан числом, то при увеличении количества ее полей (например, при добавлении новых свойств) придется искать все вхождения этой таблицы и зависящих от ее размера значений. Проще определить в модуле констант ее размер, и при нововведениях делать только одну замену.

- Удобной возможностью является использование необязательных (ключевое слово `optional`) аргументов. Они позволят объединить в одной функции обработку сразу нескольких вариантов списка аргументов. Например, так работает интерфейс `comp_reg_array()`, функции которого имеют необязательные аргументы – двумерные, трехмерные и четырехмерные массивы. В зависимости от реально переданного массива, будут вызываться немного отличные части процедуры.
- Если вы создаете объект с некоторым набором свойств, то его удобно определять в виде производного типа. Например, характеристики сетки удобно хранить в некой структуре следующего вида:

```
type GridType
```

```

character(20) :: name
real (kind = 4), allocatable :: lat_t(:, :)
real (kind = 4), allocatable :: lon_t(:, :)
real (kind = 4), allocatable :: topo(:, :)
integer, allocatable :: imask(:)
end type GridType

```

Теперь можно создавать экземпляры нового типа так же, как и для обычных встроенных типов и обращаться к полям структуры с помощью символа %:

```

type (GridType) grid_ocn, gr_atm
grid_ocn % name = "Tripolar"

```

Структурный подход удобен как для создания экземпляров разного содержания, но одинаковой структуры, так и для логического объединения неких характеристик в одну общую группу (например, структура всех характеристик форсинга).

- При построении больших проектов становится очень важно отделять различные функции системы в отдельные независимые модули. Под модулем здесь понимается любой независимый (*инкапсулированный*) логический блок программы. Блочная структура позволяет многократно менять отдельные элементы, не затрагивая остальную систему. Например, блок каплера для сбора-распределения данных от одного процессора-каплера своим подчиненным работает с некоторыми массивами данных. При этом, он ничего не знает о том, какие данные попали в эти массивы: считанные из контрольной точки, подкачанные из файловой системы или полученные с помощью переинтерполяции. При этом, для изменения блока взаимодействия каплера с компонентами требуется изменять только этот блок – остальные функции от него не зависят.

При грамотной блочной структуре переход от одной версии программы к другой является естественным подключением нового блока. При этом отключение или подключение новой функции (например, находящейся еще в стадии тестирования) осуществляется закомментированием одной строки кода вызова этой функции, вместо постоянного изменения всей программы.

Блочная структура, например, позволяет существенно изменять физику модели (сосредоточенную в процедуре `make_step()`) без какого-либо влияния на оболочку каплера, и наоборот, изменять и расширять систему каплера, никак не затрагивая физические процедуры.

Если разрабатываемый вами блок данных является логически независимой единицей (например, динамика льда) – удобно выделить его в модуль. Он будет

содержать данные и ваши функции – способ работы с этими данными – и выглядеть, например, так:

```
module i_dynamics

use test_module
implicit none

PRIVATE
real, allocatable :: flow_south(:, :)
real, allocatable :: flow_west(:, :)
real, allocatable, PUBLIC :: ice_temp(:, :)
integer, allocatable :: mask(:, :)

PUBLIC sub1

CONTAINS

subroutine sub1()
    flow_south = 10.5
    write(*,*) "Hello_ice!"
end subroutine sub1

end module i_dynamics
```

Идея модуля – черный ящик со скрытыми данными и известными методами работы с ними (**contains**). Поэтому, когда какая-то часть программы решит использовать блок динамики льда, она подключит его директивой **use** и сможет считать массив **ice_temp** или вызвать процедуру **sub1**. При этом, при попытке обратиться к закрытой (**PRIVATE**) переменной **flow_south**, уже на этапе компиляции вы получите ошибку. Такой подход, а именно гарантия того, что на работу черного ящика нельзя повлиять извне, обеспечивает более надежную работу блока.

- Соглашения по стилю программирования: использовать отступы (например, следующий уровень – 8 пробелов, размеры Tab зависят от редактора и конфигурируются), пустые строки и символьные разделители для максимально понятного представления кода. Обязательны комментарии, понятно описывающие, что происходит внутри процедуры. Для всех процедур желательно указать, какие массивы являются входными и выходными. Использовать комментарии внутри кода, объясняющие ход тех или иных нетривиальных действий.
- Имя функции рекомендуется делать совпадающим с именем содержащего ее

файла. Например, главная функция динамического блока `o_d_main()` содержится в файле `o_d_main.f90`. Имя функции должно описывать ее назначение и происхождение. Например, `o_ini_set_grid_bsc()` – океанологическая функция из блока инициализации, задающая биполярную часть сетки. Удобные обозначения: `_main` для главной функции блока, `_ini` для инициализации, `_module` для модулей, `_halo_update` - для любых процедур получения приграничных ячеек и т.д. Общие функции, используемые в разных модулях (например, процедуры, реализующей прогонку) могут использовать префикс `shared_` (например, `o_shared_tridiag`).

- Рекомендуется использовать понятные имена. Если вы используете переменную для льда, то лучше назвать ее `ice_density`, чем `id` или `ds`. Если переменная используется только внутри процедуры, то называть ее можно как угодно, но если она идет через всю программу и используется другими функциями - имя должно понятно объяснять происхождение переменной. Кроме того, понятные имена (`counter_land`, `counter_depth` вместо `k12`, `i77` и т.д.) позволяют понимать, что происходит в функции, даже при первом просмотре.
- Пользуйтесь продвинутыми тестовыми редакторами. Современные программы позволяют настраивать ширину пробелов, подсвечивают код и выделяют встроенные операторы, ищут и заменяют текст во всей папке программы, имеют множество полезных комбинаций клавиш, переходов к файлам по результату поиска и т.д.
- Для компиляции системы рекомендуется использовать стандартную linux-утилиту `make`.

Удобство *makefile* в создании гибкого и общего алгоритма компиляции с простым подключением различных флагов и внутренним использованием каких-либо переменных окружения архитектуры. Грубо говоря, это отдельная программа для компиляции программы. Кроме того, если вы изменили один файл, перекомпилирован будет только он, что в случае 100-файловой системы сильно сокращает время.

9 Общие важные замечания

- **Готовые библиотеки**

Система предоставляется в виде *precompiled binaries*, что означает, что все необходимые программы уже скомпилированы для целевой системы и просто сложены в архив. Данный подход позволит избавить пользователя от нетривиальной установки библиотек (например, параллельного netCDF) из исходных файлов. Таким образом, скопировав архив в любое место на машине, через несколько шагов вы получите полностью рабочую систему. Папки для разных систем незначительно отличаются наличием флагов компиляции в установленных библиотеках. Если все-таки необходимо установить софт, то подробные инструкции приведены в *Руководстве по установке*.

На данный момент пакет содержит:

- саму систему cpl-1.3
- параллельную версию netcdf-4.1.3
- параллельную версию hdf5-1.8.9
- Python-модули визуализации PyNGL и PyNIO
- дистрибутив Enthought Python

Свой питоновский дистрибутив понадобился для независимости от плохо установленных версий языка Python (например, на *Ломоносове*). Обратите внимание, что для локальной машины требуется уже установленный компилятор и эмулятор параллельных процессов (mpif90 и mpirun). Система протестирована для дистрибутива *mpich2*, поэтому настоятельно рекомендуется использовать именно его. Для *Ubuntu* все пакеты можно установить либо через Менеджер Пакетов, либо через консоль командой `sudo apt-get install <имя пакета>`.

- **Запуск на разных системах**

Запуск для *Ubuntu64*

```
mpirun -np 12 ./model.exe CPL 2 OCN 8 ATM 2 tst 0 0 1
```

Запуск для *MBC*

```
mpirun -np 12 -maxtime 3 ./model.exe CPL 2 OCN 8 ATM 2 tst 0 0 1
```

Запуск для *Ломоносов*

```
sbatch -np 12 -p test -t 3 ./model.exe CPL 2 OCN 8 ATM 2 tst 0 0 1
```

Перед компиляцией на данной системе необходимо добавить системные модули компилятора в аше окружение, выполнив команды:

```
module load impi/4.1.0  
module load intel/13.1.0
```

Рекомендуется добавить эти 2 строчки в конец вашего корневого файла `.bashrc`, тогда данные модули будут импортироваться автоматически при каждом входе на кластер. Для тестов рекомендуется использовать раздел `test`, он состоит из 512 ядер и на нем практически всегда отсутствует очередь.

- **Передача массивов в качестве параметров**

Если вы хотите передать в обменную процедуру только некоторые поля массива, вы можете воспользоваться разделителем диапазонов `:`. Такой знак на месте одного из измерений массива означает, что вы хотите передать это измерение целиком. Например, вызов функции `comp_halo_update()` для 3D массива скоростей при условии, что мы хотим обменять только верхний слой $k = 1$ (обратите внимание, что поскольку одна координата фиксирована, то используется уже аргумент для 2D массива – DIM2):

```
call comp_halo_update_V4(DIM2_u=u_f(1,:,:), DIM2_v=v_f(1,:,:), &  
                        update_width=1, change_sign_on_bipolar=-1)
```

- **Возможности чтения-записи**

Система написана таким образом, что существует возможность чтения и записи контрольной точки для различного количества процессоров каплера и компонент. То есть, возможно, например, считать и сохранять контрольные точки на конфигурации каплер(1 процессор) - океан(4 процессора), а при следующем запуске стартовать уже на конфигурации каплер(4 процессора) - океан(16 процессоров) или каплер(1 процессор) - океан(200 процессоров) и т.д.

Не забывайте, что массивы распределяются локально (без приграничных ячеек), поэтому для корректного счета первого шага требуется выполнить на этапе инициализации halo-обмен для каждой считанной из файла функции решения.

Хранение последовательности контрольных точек при достаточно детальном разрешении модели требует огромного объема дискового пространства (например, один трехмерный массив океана одинарной точности при разрешении 3600x1800x100 занимает 2,6 гб). Поэтому будьте внимательны при запуске больших задач со значением параметра `time_append_cp = .TRUE..` Файл

контрольной точки имеет вид `OCN_720x400_000_01021988_CP.nc` для последовательности контрольных точек и `OCN_720x400_000_REWR_CP.nc` для перезаписываемых. Имя состоит из названия компоненты, размера задачи, текущего эксперимента и режима записи. Именно такой файл будет искать система при старте с контрольной точки, если имя эксперимента не изменится. Для старта с определенной контрольной точки, укажите ее имя для параметра `time_read_cp`. Например, для старта с 1 февраля 1988 года укажите `time_read_cp = '01021988'`, а для перезаписываемой – `time_read_cp = 'REWR'`. Система выполнит поиск указанного файла и выдаст сообщение об ошибке в случае его отсутствия.

Помимо массивов данных, контрольная точка содержит информацию об эксперименте: название, текущее время счета и т.д. При старте с нее, текущий шаг будет передан компоненте через параметр глобального временного шага `time_1` и модель начнет счет именно с него.

Все временные периоды сохранения/чтения данных построены таким образом, что при указании 0 в качестве одного из значений (например, `time_save_dg`) данные функции (в этом случае сохранения диагностических полей) выполняться не будут. Того же эффекта можно достичь, не указывая ни одного имени в массиве имен работы с файловой системой.

Обратите внимание, что разные компоненты могут по-разному начинать счет. Например, океан может стартовать с контрольной точки, а атмосфера с искусственных начальных условий.

- **Работа с netCDF файлами**

NetCdf – это формат данных с определенными для работы с ним библиотеками. Формат – самоописываемый, то есть отдельный файл инкапсулирует не только массивы, но и размеры их измерений и, например, единицы измерений данных массивов и некоторые ваши глобальные переменные, такие как название эксперимента и время, из которого данный массив был получен.

Для работы с netCDF файлами есть множество стандартных программ. Одна из них, `ncdump` – программа командной строки, что означает, что вы можете использовать ее как на кластере, так и на локальной машине. Со всеми опциями можно ознакомиться с помощью `ncdump -help`.

Наиболее употребляемая комбинация:

```
ncdump -h myfile.nc
```

которая выдает только заголовок файла(измерения, названия массивов данных и их атрибутов, глобальные переменные). В листинге представлен вывод данной команды для тестового файла океанской диагностики(на кластере команда лежит в папке **software**):

```
[vovo]$ ../software/bin/ncdump -h OCN_720x400_000_DG.nc
netcdf OCN_720x400_000_DG {
dimensions:
    t = UNLIMITED ; // (64 currently)
    i = 720 ;
    j = 400 ;
variables:
    float u_bt(t, j, i) ;
        u_bt:long_name = "barotropic_velocity" ;
        u_bt:units = "m/s" ;
    float v_bt(t, j, i) ;
        v_bt:long_name = "barotropic_velocity" ;
        v_bt:units = "m/s" ;
    float h_bt(t, j, i) ;
        h_bt:long_name = "surface_height" ;
        h_bt:units = "m" ;
// global attributes:
    :title = "OCN_diagnostic_fields" ;
    :experiment = "000" ;
```

Если вы работаете на локальной машине, можно использовать удобную графическую утилиту:

```
ncview myfile.nc
```

которая визуализирует ваш файл, позволяет просматривать различные измерения, строить графики по одной из координат и даже проигрывать анимацию в случае временных(диагностических) файлов.

- **Использование интерфейса системы и глобальных переменных**

Для использования интерфейса обмена приграничными ячейками требуется подключить модуль *comp_halo_update_module*. Для океана, например, эти процедуры удобно использовать при инициализации, для обмена приграничными ячейками сеточных массивов(например, *cv*, *hv* и т.д.) и в обычных ситуациях обмена на каждом шаге.

Если вам требуются глобальные переменные компоненты, требуется подключить модуль *comp_module*.

Наконец, в *shared_module* содержатся переменные `rank`, `rank_world`, `comm`, необходимые для коммуникаций. Переменные времени также определены в этом модуле. По умолчанию, временной цикл только увеличивает глобальный шаг `time_1` на единицу. Если вы хотите посчитать время в данном месте функции, просто вызовите процедуру `shared_timer()` и используйте посчитанные ей глобальные величины.

- **Деление областей и процессоров внутри системы**

Процедура `comp_register()` попытается разделить сетки между процессорами исходя из их размеров, количества процессоров каждой компоненты и каплера и флага метода декомпозиции (2D или 1D). Если разбиение невозможно, будет выведено соответствующее сообщение об ошибке.

При делении проверяются следующие условия:

Число процессоров компоненты можно представить в виде двух множителей (пусть `inr` и `jnr`), равным соответственно числу процессоров по широте и по долготе, $inr * jnr = comp_nr$. В случае 2D декомпозиции система попытается подобрать `inr` и `jnr` так, чтобы области были максимально приближены к квадратам, в случае 1D декомпозиции выбирается широтный тип, $inr = 1$, $jnr = comp_nr$.

Размеры сетки в каждом направлении должны быть кратны числу процессоров в данном направлении.

Число процессоров по широте `jnr` должно быть кратным числу процессоров каплера.

Число процессоров по долготе `inr` должно быть четным, или равным 1.

Пример корректных размеров задачи: *coupler*(4 процессора), *osn*(100x60 на 100 процессорах), *atm*(40x40 на 8 процессорах).

Пример некорректных размеров задачи: *coupler*(3 процессора), *osn*(102x61 на 97 процессорах), *atm*(40x47 на 11 процессорах). Невозможно разделить ни область внутри физических моделей, ни числа процессоров компонент для каплера.

- **Если что-то не работает**

сначала попробуйте компилировать вашу модель с флагами (для *Ло-моносова*: `-fpe0 -check bounds -traceback`, для *Ubuntu*: `-fbounds-check -g -fbacktrace -ffpe-trap=overflow,zero`). Эти опции ловят ошибки сегментации, деления на ноль, переполнения переменной и выход за пределы массива.

далее прокомментируйте все ваши вызовы физики, таким образом оставив только оболочку каплера и проверьте, работает ли она. Если да, то ошибка в физике, если нет, то

проверьте, что правильно ли вызывается интерфейс каплера (границы массивов, количество аргументов).

если вы **все** поверили, то присылайте на *vvk88@mail.ru* неработающую версию и используемые параметры запуска с описанием проблемы.

10 Изменения версий

В данном разделе будут отражены последние существенные изменения новых версий.

• Версия 1.0

Начальная сборка системы. Полная компонента-океан(лед как модуль океана и не вынесен в отдельную компоненту). Реализована компонента атмосферы в качестве функций-заглушек. На их место требуется вставить реальные процедуры.

Для первого теста совместной работы маппинга предлагается единственная сетка размеров 400×180 для океана и 240×100 для атмосферы. На данных размерах модель тестировалась на полгода счета на различном числе процессоров для компонент и каплера.

Запуск возможен только для старта с искусственных начальных условий(`*_ic = 1`). Запуск с контрольной точки пока недоступен, так как требуются некоторые изменения в океане.

Для начального тестирования океан представляет собой прямоугольную яму постоянной глубины.

Сетка атмосферы представляет собой равномерную широтно-долготную сетку, с вырезанным южным полюсом, то есть начинающаяся с точки `lon_ssc_west_south = -100.0625`, `lat_ssc_west_south = -80.0625`. Сетка океана такая же, за исключением биполярной шапки на севере.

Возможность изменения конфигурации сетки будет добавлена в следующих версиях вместе с руководством для *offline*-блока.

• Версия 1.1

Изменена схема работы с файловой системой. Помимо чтения-сохранения контрольной точки и маппинга появилась возможность считывать одноразовые начальные данные(например, топография дна) и подкачивать массивы из файла во время счета. Кроме того, теперь все необходимые имена массивов задаются в единой таблице при инициализации, которая и передается в функцию регистрации.

Убран параметр текущего времени(`cur_time`) при чтении-записи контрольной точки. Теперь вся временная информация передается компоненте на этапе регистрации. Это позволяет запустить одну модель в режиме старта с контрольной

точки, а другую в режиме искусственных начальных условий, задав например, при нормальном океане аномальный режим атмосферы.

Используя возможность чтения начальных условий, океан теперь может использовать реальную топографию ЕТОРО1. Для данной тестовой сетки файл топографии будет считан по умолчанию.

Для версии 1.1 используется сетка размеров 720×400 для океана и 360×160 для атмосферы. Размеры пока фиксированы (из-за требования наличия весовых файлов), в будущем появится возможность их произвольного регулирования (с помощью *offline*-блока).

• Версия 1.2

Изменена схема работы с файловой системой. Теперь все считываемые и создаваемые файлы используют параллельный формат Hdf5/NetCDF4 – самый распространенный на сегодняшний день формат работы с геофизическими данными.

Подключен *offline*-блок построения сеток интерполяционных весов. С помощью одного скрипта вы можете построить матрицы маппинга для любых размеров сеток с помощью пакета SCRIP. На данный момент построение сеток возможно только для компонент океана и атмосферы.

Подключен блок встроенного рисования. Он содержит скрипты на языке Python, которые, используя модули PyNGL и PyNIO позволяют разрисовать результаты эксперимента прямо на кластере без необходимости переносить достаточно большие файлы эксперимента на локальную машину.

Добавлена возможность сохранения интегральной диагностики ACTION_SAVE_IT.

Данная версия протестирована на локальной машине (Ubuntu), на суперкомпьютерах МВС-100К и Ломоносов.

• Версия 1.3

Добавлена возможность выбора стратегии создания контрольных точек (time_append_cp) – перезаписывать последнюю или создавать их последовательность. Также введен параметр для реального сброса внутренних буферов netCDF в память (time_sync_fs).

Добавлена возможность сохранения трехмерных массивов диагностики и разрисовки их срезов в блоке диагностики. Введена константа MISSING_VALUE, позволяющая маскировать массивы при визуализации.

Доработан offline-блок. Есть возможность тестового запуска экспериментов с форсингом NCAR.

Появилась функция анализа реальной загрузки памяти `shared_memory_usage()`.

Версия тестировалась на конфигурации "океан(3600x1800) 3 поля каждые 2 часа атмосфере, пустая атмосфера (720x400) 8 полей каждый 1 час океану" на 7000 процессорах. Тестовое время счета модельного года – 74 минуты – это время океана(с включенным льдом) с коммуникационной нагрузкой интерполяцией всей системы(обе компоненты принимают и посылают тестовые поля). В следующей версии планируется проверить масштабируемость системы уже с реальной атмосферой ПЛАВ.

• Версия 1.4

Произведены серьезные внутренние изменения и оптимизация памяти для запуска на сетках океана 5400x2700x49 и 7200x3600x49. Пройдены тесты для полной нагрузки памяти – с сохранением диагностики, контрольных точек и постоянным маппингом с атмосферой. Построение интреполяционных весов уже не может быть выполнено на обычных разделах суперкомпьютера(требуется разделы с более чем 12 гб ОП на узел), контрольная точка океана занимает 20 гб и требует 80 ядер каплера для более или менее оперативной работой с файловой системой(порядка 3-4 минут на контрольную точку). Предыдущие версии `netCDF` уже не могут быть использованы для записи из-за внутренних ограничений на размер полей.

Косметические изменения в интерфейсе: параметры `comp_i_cycle`, `comp_bsc_grid`, `comp_decomp_type_2D` стали логическими, имя эксперимента и длительность запуска вынесены в аргументы командной строки.

Кроме того, теперь для перезаписываемой контрольной точки создается файл вида `OCN_720x400_000_REWR_CP.nc` (для старта с нее требуется указать во входном параметре `time_read_cp = 'REWR'`). Для последовательности контрольных точек будут создаваться файлы с датой в названии вида `OCN_720x400_000_01021988_CP.nc` (для старта с одной из них требуется указать во входном параметре, например, `time_read_cp = '15031988'`). Текущее время эксперимента и остальная информация о нем теперь хранится в соответствующей контрольной точке.

Программный `makefile` теперь единый для всех используемых систем. Подключение новых компонент осуществляется в виде линковки библиотек при

компиляции и с использованием реализации абстрактных интерфейсов базового класса `comp` при кодировании.

Произведены первые тесты с моделью атмосферы ИВМ. Совместная система отрабатывает месяц на кластере ИВМ и компьютере *Ломоносов*, пока обмениваясь полями без их использования.

- **Версия 2.0**

Кардинальные изменения структуры системы. Модели теперь наследуют абстрактные интерфейсы на уровне кода и содержат свои библиотеки и их описания на уровне папок. Таким образом, пользователь полностью ограничен от любых изменений вне своей модели. Корневой скрипт производит все необходимые макроподстановки в головной программе и линкует к ней произвольное число библиотек.

Функции приема-посылки переписаны для общего вид: на регистрации они получают адреса массивов и всю необходимую информацию о событии. Приемы и посылки происходят автоматически без пользовательских вызовов.

Оффлайн блок теперь полностью абстрактный и его работу регулирует один скрипт, осуществляющий как линковку, так и выполнение команд. Кроме того, средства пакета SCRIP заменены на аналогичные из CDO.

Совместная система океан-атмосфера ИВМ стабильно функционирует для разрешений 720x360x49 и 640x400x50 соответственно.