Вихреразрешающая иерархическая трёхмерная исследовательская модель
ВИТИМ 3.1.1α
практическое руководство пользователя

Ушаков К.В., Кауркин М.Н., Ибраев Р.А., Калмыков В.В. и др.

16 октября 2018 г.

Содержание

1 Введение

Предлагаемое руководство представляет собой пилотную версию инструкции пользователя совместной геофизической модели ВИТИМ. Изучая его, вы сможете последовательно пройти все уровни разворачивания модели и погружения в работу, от установки операционной системы до работы с внутренним содержанием модулей физических моделей и каплера. В конце изучения начальных уровней предлагаются простые тесты для того, чтобы в первом приближении убедиться, что у вас всё работает. Все пути до папок и файлов в данном руководстве будут указаны от папки **vitim3.1**, либо файлы и папки будут приведены без путей.

1.1 Некоторые характеристики данного выпуска

— Поддерживаемые конфигурации: global20, WOM025_ice, WOM05_ice, arctic0125a, arctic025t, laptev0125(a,c) и WOM01.

— Поддерживается работа в средах CMF2.0 и CMF3.0.

— Возможность выбора льда: Schrum или CICE

— Возможность запуска CICE на редуцированной, т. е. урезанной относительно океана, сетке для экономии ядер (только под управлением CMF2.0).

— Интреполяции с учётом масок областей сделаны средствами SCRIP.

— Средства диагностики o_diagn: осреднение по времени и расчёт потоков тепла для исследования МПТ.

2 Подготовка рабочего места

2.1 Установка операционной системы и компилятора

Тонкости установки операционной системы находятся вне рассмотрения данного руководства, полностью ложась на ваши плечи и каналы связи со справочными ресурсами интернета. Текущая версия ВИТИМ для ПК устанавливается на операционную систему Ubuntu 14.04. С более новыми выпусками Ubuntu возникали сложности, которые, однако, скорее всего преодолеваются парой строчек в .bashrc. Поэтому желание разобраться с ними с вашей стороны горячо приветствуется.

Для работы вам понадобится компилятор Intel версии не ниже 15. Если вы собираетесь работать на кластере, то компилятор устанавливать не надо – ищите его в списках подгружаемых модулей. Для работы на ПК бесплатную годовую версию Intel® Parallel Studio XE для студентов можно скачать с сайта Intel, для этого нужно иметь университетский почтовый ящик, например, @phystech.edu.

Распаковываем, запускаем графический установщик:

```
tar -zxvf intel.xxx.tgz
cd inlet.xxx/
./install_GUI.sh
```

Со всем соглашаемся (в единственном нетривиальном моменте выбираем root using sudo), серийный номер должен был прийти вам на почту при скачивании компилятора. Когда установщик закончил, добавляем 2 строчки в /.bashrc (пример для версии Intel 17.0.4, для других версий пути могут чуть отличаться)

```
. /opt/intel/bin/iccvars.sh intel64
. /opt/intel/impi/5.1.3.210/bin64/mpivars.sh
```

Закрываем консоль и открываем заново, чтобы выполнился bashrc. Устанавливаем g++ (используется как препроцессор для icc):

```
sudo apt-get install g++
```

Проверяем, что всё работает

```
angarsk@angarsk:~$ ifort -v
ifort version 17.0.4
angarsk@angarsk:~$ icc -v
icc version 17.0.4 (gcc version 4.8.0 compatibility)
```

2.2 Настойка SSH

SSH – сетевой протокол, позволяющий удаленно работать с некоторой системой, например, с кластером. Чтобы подключиться к удаленной машине, например, выполняем:

```
ssh ivanov@mvs10p.jscc.ru
```

Как система поймет, что вы действительно ivanov? Вас либо попросят ввести пароль (выданный вам администратором машины), либо попросят ваш секретный ключ. Считается, что метод с ключом безопаснее. В чем его суть? Вы генерируете пару ключей командой *ssh – keygen*: открытый и закрытый. Открытый передаете администратору машины, а закрытый храните у вас в надежном месте. При подключении предъявляете закрытый ключ. Если пара совпала, вас пускают на машину. При подключении может возникнуть ошибка вида: WARNING: UNPROTECTED PRIVATE KEY FILE! Это значит, что ssh обнаружила, что ваши данные в папке $\sim$ /*.ssh* имеют слишком открытые права доступа и могут быть напрямую прочитаны без всяких взламываний. Поэтому установите нужные права на папку и её содержимое (почитайте о правах доступа)

```
chmod 700 ~/.ssh && chmod 600 ~/.ssh/*
```

Как упростить жизнь с ssh? Для каждого соединения мы создаем запись в конфигурационном файле. Всегда, когда это возможно, ssh будет обращаться за информацией в свой конфиг, сводя к минимуму количество вводимых пользователем параметров. Например, если вам для входа требуется предъявить закрытый ключ, то вносим в в файл /*.ssh*/*config* строчки:

```
Host inmio
    HostName ivm-nat.nc.mstn.ru
    Port 40222
    User ivanov
    IdentityFile ~/.ssh/ivanov_inmio_key
```

Именно так вам надо поступить, для того, чтобы получить доступ к серверу ГРОСМ, на котором хранятся коды модели, скрипты установки, данные атмосферного форсинга и т.д. Сгенерируйте пару ключей и попросите администратора зарегистрировать открытый ключ.

Проверяем:

```
angarsk@angarsk:~$ ssh inmio
Welcome to Ubuntu 16.04 LTS (GNU/Linux 4.4.0-21-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

739 packages can be updated.
384 updates are security updates.
```

```
Last login: Fri Mar  2 15:26:27 2018 from 91.225.112.
```

2.3 Установка геофизического программного обеспечения

Модель использует ряд библиотек ввода-вывода, межпроцессорного обмена, обработки данных и т.д. Для удобства написан скрипт, который их скачивает, устанавливает и прописывает переменные окружения. Скачиваем последнюю версию скрипта и вспомогательных файлов из удаленного репозитория на сервере ГРОСМ:

```
git clone ssh://inmio/git/scripts.git
```

Для конфигурирования скрипта установки достаточно установить несколько главных параметров (например, нужные версии компиляторов: mpif90 или mpiifort и т.д.) в параметрах скрипта. На данном этапе у вас должны быть установлены необходимые компиляторы Intel. На домашней машине это готовый к работе Intel Complier, а на кластере – подгруженные модули, занесенные в bashrc. Теперь можно запустить скрипт установки. В данном примере софт будет устанавливаться в папку $*HOME*/*House*/*software*

```
bash install_soft.sh --install-dir $HOME/software --all \
--mpifc [Fortran compiler] --mpicc [C compiler] --mpicxx [C++ compiler]
```

Обычно для компиляторов Intel работают следующие параметры:
[Fortran compiler] = mpiifort
[C compiler] = mpiicc
[C++ compiler] = mpiicpc

По умолчанию скрипт попытается скачать данные с удаленных серверов. Если интернет-соединение отсутствует, вручную перенесите на кластер архивы библиотек и повторите установку. Откуда взять архивы? Выполните на системе, где интернет есть, скрипт с параметрами `--all --only-download`. В целевой папке инсталяции появятся нужные архивы.

В конце установки автоматически выполняются тесты. Убедитесь, что они завершились без ошибок:

```
Testing NetCDF...SUCCESS
+ mpiifort -c -cpp test_ga.f90 -I/home/angarsk/House/software/include
+ mpiifort -o test_ga.exe test_ga.o -L/home/angarsk/House/software/lib
+ mpiexec.hydra -np 4 ./test_ga.exe
 Testing GlobalArrays...GA_INIT
 Testing GlobalArrays...MA_INIT
 Testing GlobalArrays...NGA_CREATE_IRREG
 Testing GlobalArrays...NGA_DISTR
 Testing GlobalArrays...GA_PUT
 Testing GlobalArrays...GA_SPD_INVERT           0
 Testing GlobalArrays...NGA_GATHER
SUCCESS
+ rm -f simple_xy_par.nc test_ga.exe test_netcdf.exe '*.mod' test_ga.o tes
+ set +x
================
ALL TESTS PASSED
================
```

В итоге программное обеспечение и вспомогательные скрипты установлены в папку $*HOME*/*House*/*software*. Закройте консоль и откройте её заново, чтобы необходимые пути прописались в вашей системе. Теперь, набрав, например, *which cdo*, вы обнаружите, что путь ведёт как раз в вашу папку софта. Проверим:

```
angarsk@angarsk:~$ which cdo
/home/angarsk/House/software/bin/cdo
```

```
angarsk@angarsk:~$ which h5dump
/home/angarsk/House/software/bin/h5dump
angarsk@angarsk:~$ which ncdump
/home/angarsk/House/software/bin/ncdump
angarsk@angarsk:~$ locate job_launcher.sh
/home/angarsk/House/software/bin/job_launcher.sh
/home/angarsk/Plots/scripts/launchers/job_launcher.sh
```

2.4 Скачивание модели и геоданных

Рабочее место пользователя представляет собой папку **vitim3.1**, в которой хранятся все модели и данные. Чтобы её получить, скачайте заготовку из репозитория:

```
git clone inmio:/git/vitim3.1
```

Скачайте из репозиториев актуальные версии моделей в папку **comps**:

```
cd vitim3.1/comps/ocn
git clone inmio:/git/inmio4.1.git
cd ../ice
git clone -b vitim2.1 inmio:/git/cice-5.1.git
```

Для минимального запуска в папке **vitim3.1/data_external** должны лежать топография ETOPO, данные Левитуса WOA2009 и форсинг CNYFv2. Если эти данные уже есть на вашем компьютере (кластере), то положите в **data_external** символьные ссылки на их папки. Если нет, то запустите скрипты скачивания (распаковка форсинга займёт несколько минут):

```
cd ../../data_external
bash get_IC_databases.sh
bash get_forcing_databases.sh
cd ../coupling
```

В дальнейшем, работая с более продвинутыми модельными конфигурациями, следите, чтобы в **data_external** лежал нужный форсинг или ссылки на него.

3 Как выбрать конфигурацию и начать работать

Модель поставляется с набором из нескольких стандартных конфигураций с разными расчётными областями, разрешением, форсингом и включёнными параметризациями. При первом знакомстве вам достаточно выбрать одну из них. Обращайте внимание, однако, на разрешение: сетки с размером больше 200×100, как правило, не влезают в оперативную память на ПК.

В файле **coupling/config** задайте полный путь до папки **vitim3.1** и выберите конфигурацию численного эксперимента (раскомментировав соответствующую строчку). Например, море Лаптевых с размером сетки 40×60:

```
export VITIM_PATH=~/VITIM3.1/vitim3.1
...
export RES="laptev0125c"; export GRID="40x60"; export INMIOCOUPLED="yes"
```

Теперь создайте символьные ссылки между файлами модели для данной конфигурации.

```
bash links_inmio set
```

Дальнейшие действия зависят от версии Системы совместного моделирования (каплера).

3.1 Сборка и запуск под управлением CMF2.0

Соберите совместную модель с неинтерактивными компонентами атмосферы и континентального стока:

```
cd coupling
./makeclean_all
```

Будет выполнен полный набор команд компиляции с предварительным удалением объектных файлов. В дальнейшем можно вызывать **./make_all** – система перекомпилирует только изменившиеся файлы (см., однако, примечание в разделе 4.1). Убедитесь, что появилось сообщение `"Coupled model for component set <ocn atm_ncar lnd_core ice_cice> compiled successfully."`

Теперь создайте файлы начальных условий и интерполяционных весов. Внимание: при выполнениии этой команды будут удалены все nc-файлы в папках-ссылках **coupling/data** и **coupling/off/data**.

```
cd configure
./generate_laptev0125c
```

В этом примере вызывается скрипт генерации сеток для конфигурации laptev0125c. Для других конфигураций скрипты называются аналогичным образом и лежат в своих экземплярах папки configure (ссылки на которые, как вы уже знаете, активируются при вызове `bash links_inmio set`). Если происходит ошибка библиотеки, то попробуйте сделать `make clean` в папке **off/SCRIP/source**.Проверьте, что в **coupling/data** появились 5 файлов с весами, начальные условия, а также есть активная ссылка на форсинг:

```
angarsk@angarsk:~/VITIM3.1/vitim3.1/coupling/data$ ls -1
ATM_NCAR_192x94_to_ICE_CICE_40x60.nc
ATM_NCAR_192x94_to_OCN_40x60.nc
CNYFv2
ICE_CICE_40x60_to_OCN_40x60.nc
LND_CORE_360x180_to_OCN_40x60.nc
OCN_40x60_to_ICE_CICE_40x60.nc
OCN_40x60x49_IC.nc
```

Создайте символьные ссылки для CICE. Внимание: если забыть этот шаг, то модель может запуститься без видимых проблем, но выдавать неправильные результаты!

```
bash links_cice set
```

После этого в папке **coupling** не должно остаться неработающих символьных ссылок.

Запуск производится из папки **coupling**. Пример команды запуска:

```
mpirun -np 7 ./model.exe CPL 1 OCN 2 ATM 1 LND 1 ICE 2 abc 0 0 3
```

Здесь abc – название эксперимента (обязательно 3 символа), 0 0 3 – продолжительность расчётов (модельных лет, месяцев и дней)

3.2 Сборка и запуск под управлением CMF3.0

Соберите CICE, а затем совместную модель океан-лёд с неинтерактивными компонентами атмосферы и континентального стока:

```
cd coupling/comp_ice_cice/cice
./comp_ice
cd ../..
bash make ocn atm_ncar lnd_core ice_cice --clean
```

Убедитесь, что появилось сообщение `"Executable <.//cpl.exe> was created successfully."`

Теперь создайте файлы начальных условий и интерполяционных весов. Внимание: при выполнениии этой команды будут удалены все nc-файлы в папках-ссылках **coupling/data** и **coupling/off/data**.

```
cd off
bash generate_ocean_all.sh 40 60 49 abc
```

В этом примере 40 и 60 – горизонтальные размеры сетки (должны быть те же, что в выбранной вами конфигурации в файле **coupling/config**), 49 – количество уровней по вертикали (пока доступен только этот вариант), abc – название вашего эксперимента (любое сочетание из 3 цифр или букв). Если происходит ошибка библиотеки, то попробуйте сделать `make clean` в папке **off/SCRIP/source**.Проверьте, что в **coupling/data** появились 5 файлов с весами, начальные условия, а также есть активная ссылка на форсинг:

```
angarsk@angarsk:~/VITIM3.1/vitim3.1/coupling/data$ ls -1
ATM_NCAR_192x94_to_ICE_CICE_40x60.nc
ATM_NCAR_192x94_to_OCN_40x60.nc
CNYFv2
ICE_CICE_40x60_to_OCN_40x60.nc
LND_CORE_360x180_to_OCN_40x60.nc
OCN_40x60_to_ICE_CICE_40x60.nc
OCN_abc_IC.nc
```

Создайте символьные ссылки для CICE. Внимание: если забыть этот шаг, то модель может запуститься без видимых проблем, но выдавать неправильные результаты!

```
bash links_cice set
```

После этого в папке **coupling** не должно остаться неработающих символьных ссылок.

В папку **coupling/configure** необходимо положить неймлист-файл для данного названия эксперимента (в нашем примере это **exp_abc.in**). Его содержание легко читаемо, примеры лежат в папках **coupling/configure** для всех базовых конфигураций.

Запуск производится из папки **coupling**. Пример команды запуска (в одну строчку):

```
bash job_launcher.sh --machine ubu --np 9 --exe ./cpl.exe
    --args "DTR 1 CPL 1 IOD 1 OCN 2 ATM 1 LND 1 ICE 2 abc"
```

4 Основные настройки

4.1 Запуск на разном количестве ядер

В приведённом выше примере запуска можно задавать другие числа ядер для компонентов OCN, ATM, LND, ICE. Допустимые числа ядер и соответствующие размеры подобластей подскажет утилита **coupling/comp_cpl/bin/test_decomp.exe**, создающаяся при компиляции системы под CMF2.0. Для работы CICE необходимо, чтобы кол-во ядер и размеры подобласти одного ядра льда были указаны в файле **coupling/comp_ice_cice/cice/comp_ice** для выбранной конфигурации, а также кол-во ядер было указано в **coupling/comp_ice_cice/cice/input_templates/имя_конфигурации/ice_in**.

После каждой такой перенастройки (затрагивающей CICE) нужно заново полностью пересобрать CICE и совместную модель. В параметр командной строки np при запуске не забудьте передать суммарное кол-во ядер на совместную модель. Если какую-то из компонент запускать не нужно, то её имя и число ядер в строке запуска не приводится. В частности, если не указать ICE, то будет работать встроенная в океан термодинамическая модель льда Шрум.

4.2 Выбор атмосферного и речного форсинга

В разделе 2.4 мы скачали атмосферные и речные данные CNYFv2 – нормальный годовой ход CORE-I. Дистрибутив модели может также работать с данными IAFv2 – реанализом и наблюдениями за 1948-2009 гг. из протокола CORE-II. Они занимают около 30 Гб и, как правило, доступны на кластерах, используемых ГрОСМ. Для выбора базы данных, которую будут читать ваши модели atm_ncar и

lnd_core, задайте соответственно параметры именованных списков `atm_forcing_type` и `lnd_rivers_type` в файлах **coupling/configure/atm_list.in** и **coupling/configure/lnd_list.in**. Список основных параметров именованных списков модели приведён в приложениии A.

4.3 Встроенная модель льда

Если в строке запуска не указать ICE и его кол-во ядер, то будет работать встроенная в океан термодинамическая модель льда Шрум. В этом случае для экономии памяти при генерации сеток (особенно в конфигурации WOM01) можно отключить генерацию сеток CICE: переключить `ice_grid` на .false. в **comps/ocn/inmio4.1/driver_cmf2.0/off_ocn_module.f90** (в среде CMF2.0) или **comps/ocn/inmio4.1/driver_cmf3.0/cmf_ocn_off_adapter.f90** (в среде CMF3.0). Если работаете с CICE, то убедитесь, что `ice_grid = .true.`

4.4 Работа на редуцированной ледовой сетке (только в среде CMF2.0)

Некоторые стандартные конфигурации для экономии ресурсов по умолчанию работают на редуцированной ледовой сетке CICE, покрывающей только северную полярную шапку. В остальной части расчётной области в этом случае работает встроенная модель льда Шрум. Включая стандартную редуцированную конфигурацию

- Проверьте значение флага `cice_and_schrum` в **comps/ocn/inmio4.1/driver_cmf2.0/o_par_module.f90**. Для редуцированной сетки должно быть `.true.`, для обычной `.false.`
- Проверьте значение `additional_ny` в начале файла **coupling/comp_ice_cice/off_ice_cice_module.f90**. Для обычной сетки должно быть 0, для редуцированной – см. нужные значения в комментариях.
- Для работы на урезанной сетке CICE нужно, чтобы декомпозиция океана по j состояла хотя бы из двух полос (чтобы в одной работал лёд CICE, а в другой Шрум). Проверьте это с помощью утилиты **test_decomp.exe**.

Если вы хотите включить редукцию сетки для конкретной конфигурации (например, взяли глобальную нередуцированную модель и хотите с её помощью исследовать Арктику), то сделайте следующее. Рекомендуется менять эти параметры осторожно, только если вы понимаете, что происходит. Для включения редукции:

- В **off_ice_cice_module.f90** задать `additional_ny` – отрицательное число, сколько отрезать линий по j с юга
- Задать новый размер по j в **config** и в **ice_list.in**, новые размеры блоков в **comp_ice**.
- Добавить строчки переименования размеров CICE в именах трёх интерполяционных файлов в конце скрипта генерации сеток (см. образцы в конфигурациях global20 и WOM025_ice)

Для выключения редукции всё вернуть: положить `additional_ny=0`, размеры ледового компонента совпадают с размерами океанского, интерполяционные файлы не переименовывать.

4.5 Работа в режиме оффлайн-анализа (только в среде CMF2.0)

Если Вы запускете модель в режиме оффлайн-анализа, то в файле **coupling/analysis_flag** значение ключа должно быть равно `.false.` Если в обычном режиме расчётов, то `.true.`

5 Система совместного моделирования

Система Совместного Моделирования (ССМ) выполняет две основные задачи:

— Поддержка сервисных операций отдельной модели (например, работы с файловой системой). ССМ позволяет четко разграничить код для вычисления физики (например, океан) и код, отвечающий за технологии (например, процедуру сохранения данных).Разделение, во-первых, упрощает архитектуру (каждый модуль занимается своим делом), а, во-вторых, дает возможность разработчику сервисных модулей модифицировать их внутренности с появлением новых технологий незаметно для физической модели.

— Поддержка совместной работы моделей (например, создание связки океан-атмосфера). Исторически модели являются отдельными программами, вычисляющими свою физику (модель океана моделирует океан). Как соединить две независимые модели, чтобы они работали совместно? Один из подходов – подключить их с помощью адаптера (аналогично электрическому адаптеру) к ССМ. В результате, внутри себя модель продолжает считать свою физику, а через адаптер общается с остальными участниками совместной модели.

5.1 Начало работы в системе CMF3.0

Чтобы понять, как работает система и что нужно для ее запуска, лучше использовать конкретные примеры. В примере 1 приведена последовательность действий, позволяющая запустить простой тест ССМ с нуля. В следующих примерах объясняется, как можно усложнить эту последовательность действий, чтобы использовать все возможности CMF. Последний пример показывает, как подключить вашу модель к системе.

Пример 1: запуск пустой совместной модели океан-атмосфера

В данном примере будет показано, как с нуля запустить простой тест CMF3.0, моделирующий запуск двух моделей (океана и атмосферы), которые ничего не делают.

Зайдите в папку модели и выполните:

```
cd test_suit
bash tester.sh --t empty_comps --clean --test
```

Что происходит в этом тесте? Запускается специальный скрипт, позволяющий совместить сборку модели и ее запуск (этот скрипт сделан для удобства, сейчас не нужно вникать, как он работает). Скрипт заходит в папку empty_comps, содержащую тестовую (но с точки зрения системы вполне полноценную) версию совместной модели океан-атмосфера. В папке лежит простой скрипт для описания теста (test_description.sh), который выглядит так:

```
COMPS_BUILD="ocn_test atm_ncar"
RUN_COMMAND[1]="-np 5 ./cpl.exe DTR 1 IOD 1 CPL 1 OCN 1 ATM 1 tst"
```

То есть скрипт **tester.sh** видит, что надо собрать совместную модель из папок **ocn_test** и **atm_ncar** и запустить их на 5 ядрах, дав каждому сервису (пока не надо об этом думать) и океану с атмосферой по 1 ядру. В консоли вы увидите, что система вывела данные эксперимента и перешла в фазу вычислительного цикла, которая в данном тесте состоит только из приема системой сигналов STOP (нормальное завершение) от моделей. Результат: вы только что запустили hello-world пример совместной модели. Пока модель ничего не делала, а только посылала сигнал о нормальном завершении. В следующих примерах мы добавим ей работы.

Пример 2: учим модель сохранять диагностику

В прошлом примере модель просто подключалась с помощью адаптера к CMF, пробегала без работы отведенное время эксперимента и завершалась. Где описываются эти действия? Логика любой модели описывается с помощью специального класса-адаптера, который, как и подобает адаптеру, знает,

как подключиться к системе и в то же время, имеет входы для физической модели. Для понимания дальнейшего процесса необходимо прочитать первые разделы руководства о модельном компоненте.

Теперь вы примерно понимаете логику работы системы и настало время взглянуть на код предыдущего примера. Откройте файл **/empty_comps/comp_ocn_test/cmf_ocn_test_cpl_adapter.f90**. Вы видите реализацию тех самых интерфейсов, которые описаны в руководстве. В данном примере они пустые (отсюда и название теста). Единственный непустой метод вызывает `ini_reg_comp` для регистрации модели в системе. Именно благодаря этой регистрации система знает, на сколько шагов запустить данную пустую модель и какие размеры ее массивов.

Теперь, давайте научим модель скидывать диагностику и для этого немного усложним код. Для этого создадим новый тест (уже создан для вас) и назовем его **/save_dg**. Логика новой модели не сильно сложнее и описана в файле **/empty_comps/comp_ocn_test/cmf_ocn_test_cpl_adapter.f90**. Помимо регистрации модели мы добавили 2 массива (2D и 3D), код, выделяющий для них память, и добавили регистрацию событий над этими массивами. Здесь появляется второе важное свойство CMF, а именно возможность сказать “хочу, чтобы этот массив каждые 2 часа сохранялся в файл”. Чтобы понять, о чем речь, обязательно прочитайте начала раздела о системных событиях. Теперь можно переходить к тому, что делает пример. В реализации интерфейса `ini_reg_data` для 2D массива:

```
call this % register_array(arr_name = "test_dg_2D", indexing = "ij",
       arr = save_dg_2D(iwest:ieast, jsouth:jnorth))
call this % register_periodic_event(arr_name = "test_dg_2D", act = "SAVE_DG", dh = 1)
```

Первая строчка регистрирует массив `save_dg_2D` под именем `"test_dg_2D"`в системе (по сути, система запоминает его адрес и индексацию). Вторая строчка регистрирует периодическое событие над этим массивом, говоря, что каждый 1 час необходимо создать событие сохранения диагностики (а именно, взять массив по адресу, отправить его соответствующему сервису для сохранения в диагностический файл). Можно запустить этот тест, для этого сначала заглянем в файл **test_description.sh**:

```
COMPS_BUILD="ocn_test"
RUN_COMMAND[1]="-np 4 ./cpl.exe DTR 1 OCN 1 IOD 1 CPL 1 tst"
....
```

Мы просим собрать нам только модель океана и запустить ее на 1 ядре (остальные ядра служебные). Ниже в файле описаны всякие условия проверки тестов (а именно, что для разных чисел ядер мы получаем одинаковый файл диагностики). Сейчас об этом можно не задумываться. Запускаем:

```
cd test_suit
bash tester.sh --t save_dg --clean --test
```

Скрипт прогонит 5 разных тестов, сравнив результаты с первым, и выдаст сообщения о результатах. Результат: мы разобрались с тем, как система понимает, где модель и как с ней работать, научились генерировать события и запустили первую адекватную модель, которая сбрасывает данные на диск.

Пример *: подключение вашей модели

До настоящего времени мы запускали готовые тесты. Теперь можно создать вашу модель (как основную модель в корне системы или пока в виде такого же теста). Для этого нужно:

Создать костяк производного класса

Для создания новой модели предусмотрен скрипт (он доступен, если вы правильно установили софт):

```
bash generate_comp.sh ocn_test
```

Скрипт создает папку модели, все необходимы подпапки, 2 шаблона производных классов (для компонента и офф-блока) и собирающий это все **makefile**. Можно сразу же выполнить `make -` будет собрана библиотека модели океана, которая ничего не делает и состоит из двух файлов. Собственно, все модели из предыдущих примеров начинались именно с вызова этого скрипта.

Заполнить производный класс
Заполните каждый метод, следуя требованиям вашей модели.

Подключить компонент к системе
Система компиляции построена таким образом, что если вы добавляете свою модель (например, океана), то достаточно при компиляции указать название папки, которое имеет определенное соглашением имя. При компиляции скрипт сборки принимает имя вашей версии компонента и понимает, что первые три буквы обозначают компонент Земной системы (`ocn, atm, ice, lnd`), а дальше следует ваша версия (`ncar, test, inmio`). Скрипт зайдет в нужную папку (например, **comp_ocn_test**), соберет там библиотеку и сообщит главной программе, что в данной компиляции будет участвовать модель океана, и ее версия (например, `test`). Все эти соглашения автоматически выполняются, если вы вызываете **generate_comp.sh**. Главный скрипт системы компиляции **make** принимает названия папок (моделей). Собственно, его и вызывает для сборки используемый во всех примерах скрипт **tester.sh**. Например, если мы создали новую модель океана `ocn_test` в корне системы, можно собрать и запустить такую модель:

```
bash make ocn_test
mpirun -np 4 ./exe OCN 1 CPL 1 DTR 1 IOD 1 exp
```

При вызове процедур океана будет использоваться именно та версия океана, которая была передана при компиляции скрипту сборки (а именно, `ocn_test`).

5.2 Модельные компоненты

Общая идея

Как вообще CMF может узнать о вашей модели (например, океана) и, соответственно, помогать ей выполнять сервисные действия и общаться с другими моделями?

Один из подходов состоит в том, чтобы определить специальный класс-адаптер, или, другими словами, модель общего вида (компонент). Такой компонент представляет собой скелет модели и определяет только ее интерфейс, но не реализацию. Система сможет вызывать методы класса-адаптера (поскольку знает интерфейс), которые представляют собой очень общие действия, например, выполнение всей инициализации или полного физического шага. При этом, система не знает, что конкретно будет выполнено внутри этих методов – она оставляет их реализацию на усмотрение модели (например, расчет термодинамики и динамики в главном шаге).

На практике описанный процесс превращается в наследование класса `Component` и предоставление реализации абстрактных интерфейсов. Каждая модель определяет их по своему усмотрению. Плюс, пользователь может вызывать удобные методы, определенные для него в классе `Component` (например, регистрацию события). В итоге, для работы в системе достаточно создать один класс-адаптер, который вызывает конкретные методы модели (ваши физические процедуры) и вспомогательные методы системы (определенные для удобства в классе `Component`). Логика работы класса будет определять логику работы вашей модели в CMF.

Интерфейс класса Component

Ниже представлены некоторые интерфейсы класса Component:

```

! Абстрактные методы, которые  должны быть реализованы в модели

! Абстрактный метод для регистрации модели в системе
! должен вызывать register_model()
procedure(I_ini_reg_comp), DEFERRED :: ini_reg_comp

! Абстрактный метод для осуществления всех allocate() в модели,
! так как дальше потребуется передавать адреса
procedure(I_ini_allocate), DEFERRED :: ini_allocate
```

```
! Абстрактный метод для регистрации всех данных и событий для обмена в системе
! должен вызывать register_array(), register_event()
procedure(I_ini_reg_data), DEFERRED :: ini_reg_data

! Абстрактный метод для любой пользовательской инициализации
procedure(I_ini_main), DEFERRED :: ini_main

! Абстрактный метод для выполнения одного шага физики
procedure(I_make_step), DEFERRED :: make_step

! Абстрактный метод для выполнения любых действий финалайзинга
procedure(I_finalize), DEFERRED :: finalize

! Вспомогательные методы базового класса, которые можно вызывать из модели

! Регистрирует данные модели в системе
procedure, public :: register_model

! Регистрирует определенный массив в системе
generic, public :: register_array => ...

! Создает генератор для равномерных событий
procedure, public :: register_periodic_event

! Создает генератор для привязки к временной оси netCDF файла
procedure, public :: register_synced_event

! Генерирует событие
procedure, public :: raise_event
```

Подробнее о вспомогательных методах

```
procedure, public :: register_model

 Описание:          Регистрирует данные модели в системе
 Параметры:
  *_size           - размеры массивов модели
  decomp_type      - тип желаемого разбиения ("1D" или "2D")
  timestep_sec     - временной шаг в секундах
```

```
generic, public :: register_array

 Описание:          Регистрирует массив в системе, сохраняя его данные
                    адрес(, атрибуты) под меткой <arr_name>
 Параметры:
  arr_name         - строковое имя массива для регистрации
  indexing         - индексы массива ("ijk", "kij")
  arr              - сам массив данных его( адрес)
```

```
generic, public :: register_periodic_event
 Описание:          Создает генератор для периодического события
```

```
 Параметры:
  arr_name        - строковое имя уже зарегистрированного массива
  act             - действие, которое нужно выполнить в момент наступления события
                    например, "SAVE_DG"
  src             - (optional) источник данных ( файл, другой компонент ),
                    например, "/data/ocn_test_data.nc"
  dst             - (optional) получатель данных
  info            - (optional) любая дополнительная информация
  dh, dm, ds      - (optional) период события. Если все периоды равны 0, то событие
                    произойдет только 1 раз в начале ( например, для "READ_CP").
```

```
procedure, public :: register_synced_event

 Описание:         Создает генератор для события привязки к временной
                   оси netCDF- файла
 Параметры:
  arr_name        - строковое имя уже зарегистрированного массива
  src             - (optional) источник файловых данных,
                    например, "/data/ncar_temp.nc"
  start_date      - (optional) с какой даты начинать привязку,
                    иначе взять дату начала эксперимента.
```

```
procedure, public :: raise_event

 Описание:         Генерирует пользовательское событие ( в обход генератора ).
 Параметры:
  arr_name        - строковое имя уже зарегистрированного массива
  src             - (optional) источник данных ( файл, другой компонент ),
                    например, "/data/ocn_test_data.nc"
  dst             - (optional) получатель данных
  dt_rec          - (optional) дата записи ( нужно, если хотим взять
                    данные из файла по определенной дате)
```

Для разработчиков системы

Как выглядит временной цикл модели

После всех инициализаций, компонент входит в главный временной цикл `model_cycle`:

```
subroutine model_cycle(this)
    ! Параметры пропущены для краткости

    ! Отправка всех массивов на регистрацию сервисам
    call this % ev_scheduler_ % get_all_events(new_events)
    do i = 1, new_events % length()
        ev = new_events % get(i)
        call this % send_request(ev)
        call this % try_register_comp_ga(ev)
    end do
    call this % raise_event(act = "STOP")
    call CompSplitter % barrier()

    do while (.TRUE.)
```

```
        ! Просим Scheduler собрать все события для данного шага
        call this % ev_scheduler_ % gather_events(this % model_time(), new_events)

        ! Обрабатываем их на стороне модели
        do i = 1, new_events % length()
            call this % handle_event(new_events % get(i))
        end do

        ! Пока таймер @model_clock_@ тикает, работаем в цикле
        if (this % model_clock_ % is_stopped()) EXIT

        ! Вызываем абстрактный метод физики одного шага модели
        call this % make_step()

        ! Тикаем 1 раз
        call this % model_clock_ % tick()
    end do

    ! В конце цикла оповещаем сервисы, что модель нормально завершила работу
    call this % raise_event(act = "STOP")

end subroutine
```

Как модель реагирует на события

Как описано в разделе События, после генерации события, оно обрабатывается в первую очередь моделью. Для этого в классе `Component` определен метод `handle_event()` (сокращенный код):

```
subroutine handle_event(this, ev)
    ...

    ! Реагируем на разные типы событий по разному
    select case (ev % action())

        ! События отправки: ждем, пока ga- массив освободится, кладем туда данные,
        ! синхронизируемся и помечаем массив как заполненные,
        ! отправляем запрос сервису
        case("SAVE_CP", "SAVE_DG", "SEND_MP")

            do while (TRIM(this % comm_ % get_info(ev % ga_name(), &
                COMM_GA_STATUS)) /= "free"); end do
            call this % put_to_ga(ev)
            call this % comm_ % sync(CompSplitter % i_am_id())
            if (CompSplitter % is_first_rank()) &
                call this % comm_ % put_info(ev % ga_name(), COMM_GA_STATUS, "full")
            call this % send_request(ev)

        ! Событие приема: отправляем запрос сервису, ждем, пока ga- массив
        ! не заполнится, берем данные, синхронизируемся и помечаем массив как пустой
        case("READ_FD")
            call this % send_request(ev)

            do while (TRIM(this % comm_ % get_info(ev % ga_name(), &
                COMM_GA_STATUS)) /= "full"); end do
            call this % get_from_ga(ev)
```

```
            call this % comm_ % sync(CompSplitter % i_am_id())
            if(CompSplitter % is_first_rank()) &
                call this % comm_ % put_info(ev % ga_name(), COMM_GA_STATUS, "free")

        ! Прием маппинга: нет запроса, просто ждем, пока ga- массив не заполнится,
        ! берем данные, помечаем массив как пустой
        case("RECV_MP")
            ! This is push-event, so no request:
            ! just register, wait, get, mark as free

            do while (TRIM(this % comm_ % get_info(ev % ga_name(), &
                COMM_GA_STATUS)) == "free")
                ! call Debugger % log_msg("Current GA status is: &
                ! "//TRIM(this % comm_ % get_info(ev % ga_name(),&
                ! COMM_GA_STATUS)))
            end do
            call this % get_from_ga(ev)

            call this % comm_ % sync(CompSplitter % i_am_id())
            if(CompSplitter % is_first_rank()) &
                call this % comm_ % put_info(ev % ga_name(), COMM_GA_STATUS, "free")

        ! Просто отправляем запрос и выходим
        case("STOP")
            call this % send_request(ev)

        ! Отправляем запрос и ожидаем, пока сервис все завершит,
        ! так как дальше работать нельзя
        case("ERROR")
            call this % send_request(ev)
            call CompSplitter % barrier()

        case default
            call this % fatal_error("raise event: &
                uknown action: <"//TRIM(ev % action())//">")
    end select
end subroutine
```

Как изменить реакцию модели на события

Внимание: в этом пункте вносятся изменения в код системы. Если вы не уверены в своих действиях, обратитесь к разработчику.Для изменения реакции или добавления нового поведения просто расширьте метод `handle_event()`.

Внимание: обратите внимание на вопросы синхронизации! Модель должна блокироваться, если ga-массив (представляющий собой буфер обмена) еще занят (для события вида “положить данные”) и, наоборот, свободен (для события вида “взять данные”). Для этого выполняется проверка статуса информационного массива. Если не устанавливать блокировку, нет гарантии, что модель получит или отправит цельные данные. После того, как модель положила или взяла данные, она должна поменять статус ga-массива соответствующим образом.

Замечание: добавление нового компонента Земной системы

Внимание: в этом пункте вносятся изменения в код системы. Если вы не уверены в своих действиях, обратитесь к разработчику. В файле `ComponentSplitter` (что это за класс читать тут) добавить необходимое имя компонента:

```
character(3), parameter :: COMPONENT_NAMES(9) = &
    (/ "DTR", "CPL", "IOD", "OCN", "ATM", "ICE", "LND", "SEA", "TST"/)
```

Замечания для разработчика системы

Если в будущем появится возможность избавиться от явной синхронизации через массив информации, это будет хорошо. Сейчас такой подход выбран, так как мы не можем "терять данные". То есть даже если какой-то компонент (модель океана), работает быстрее другого компонента (модели атмосферы или IOD, который медленно пишет данные в файл), мы не имеем право "потерять"массив. Накопление массивов в виде очереди тоже ни к чему не приведет, так как модели обычно работают с постоянной скоростью и в результате очередь просто исчерпает всю доступную память. Поэтому сейчас, если “быстрая” модель уже готова положить данные, но ga-буфер еще занят, она блокируется.

5.3 События в системе

Общая идея

События являются сообщениями о необходимости выполнить те или иные действия над массивом данных. Например, когда мы хотим послать данные модели на сохранение в файл диагностики, мы генерируем (raise) событие с типом `SAVE_DG` и некоторыми параметрами (например, файлом назначения). События рождаются на стороне модели, либо непредсказуемым вызовом `raise_event()` пользователем (например, при наступлении условия критического падения уровня в океане), либо запланированными заранее генератором (но, по сути, тем же самым `raise_event()`. Обратите внимание: и модель, и сервисы имеют свою реакцию на события (как на это повлиять, читайте в разделах про Модельный компонент и Сервисы).

Первой на событие реагирует модель – она смотрит на его тип и определяет, что делать (в случае `SAVE_DG` – положить данные в ga-массив, отправить запрос сервису и продолжить счет). Далее событие запаковывается в MPI-сообщение и улетает в виде запроса к сервисам (если модель решила отправить событие). Сервисы распаковывают событие, смотрят на его название, и либо обрабатывают событие, либо игнорируют.

5.3.1 Типы событий

Сейчас определены следующие типы событий (реакция на них приведена для понимания и задается не в самих событиях, а в их обработчиках в классах `Component` и `Service`):

- `READ_FD` – чтение из файла (его частными случаями являются `READ_IC`, `READ_CP`).
  Компонент: отправить запрос сервису, дождаться данных, взять данные, продолжить работу.
  Сервис IOD: получить запрос, взять данные из файла, положить в ga.

- `SEND_MP` – отправка данных на интерполяцию и потом другому компоненту
  Компонент: положить данные в ga, продолжить работу.
  Сервис CPL: взять данные из ga, переинтеполировать их на сетку получателя, положить в ga получателя.

- `RECV_MP` – прием данных от другого компонента
  Компонент: дождаться, пока данные появятся в ga, взять из, продолжить работу.
  Сервис CPL: ничего не делает (все уже сделано на `SEND_MP` шаге)

- `SAVE_CP` – сохранение контрольной точки
  Компонент: положить данные в ga, отправить запрос сервису
  Сервис IOD: получить запрос, взять данные из ga, записать в файл.

- `SAVE_DG` – сохранение диагностики (по сути, то же, что и `SAVE_CP`, но выделено из соображений производительности)

- `STOP` – нормальное завершения работы модели.
  Компонент: отправить запрос, продолжить работу
  Сервисы: Когда последняя модель разошлет сообщение `STOP`, сервисы нормально завершатся.

- `ERROR` – аварийное завершение работы модели.
  Компонент: отправить запрос, встать на ожидание, так как дальше работать нельзя.
  Сервисы: Сервис, получивший такое сообщение, должен аварийно завершиться.

Генераторы

Вообще, поскольку класс `Component` определяет метод `raise_event()`, то, теоретически, генераторы событий не нужны, так как в любой момент временного цикла можно отослать запрос сервису и он как-то отреагирует. Но на практике такой подход означает, что пользователь должен сам следить за временем и в нужные моменты отправлять запросы. Чтобы упростить жизнь пользователя, в системе определены несколько генераторов событий, то есть объектов, которые в зависимости от модельного времени выдают запрос или ничего не делают.

Примером генератора может служить генератор периодических событий, например, сохранющий диагностику каждые 2 дня. Такой генератор создает событие в 0 часов, 48 часов, 96 часов, ... модельного времени и ничего не создаёт в остальные временные промежутки (например, в 11 часов 12 минут по модельному времени)

Как зарегистрировать событие?

Для регистрации события достаточно передать его генератору. Для этого пользователь вызывает метод класса `Component`, которая сама передает его куда нужно (см. интерфейс `Component`).

Для разработчиков системы

В файле **VarEvent.f90** определен абстрактный класс `VarEvent`, представляющий интерфейс любого генератора:

```
! Абстрактный метод, обновляющий внутреннее состояние генератора
! и возвращающий или( нет) событие
procedure(I_update_ve), DEFERRED :: update

! Абстрактный деструктор
procedure(I_destroy_ve), DEFERRED :: destroy
```

Конкретные реализации генераторов наследуют базовый класс и определяют, что конкретно будет делать объект при вызове `update()`. Например, класс `VarEventNormal` просто проверяет условие кратности времени (передается в качестве аргумента) периоду, заданному при инициализации генератора.

Дальше, класс `EventScheduler` создает массив полиморфных ссылок на все такие генераторы и при каждом изменении таймера опрашивает все генераторы, не готовы ли они выдать запрос.

Как определить новый тип события

Если вы хотите добавить новое событие, то необходимо:

— В классе `Actions` добавить новый тип события, его приоритет и соответствующий сервис, который его будет обрабатывать.

— В классе `Event` добавить условие для создания вашего типа события. Эти условия позволяют проверять, что событие полное (например, событие маппинга должно иметь адресата, иначе оно не может быть обработано).

— Если вам необходимы дополнительные поля, отсутствующие в классе Event, добавьте их, убедившись, что вы реализовали их упаковку и распаковку в MPI-сообщение.

Теперь в системе определен новый тип события. События с этим типом можно построить и отправить. При этом, чтобы сообщение реально производило какое-то воздействие на систему, надо добавить обработку события в сервисы и компонент (как это сделать, читайте в соответствующих разделах, посвященных сервисам и компоненту).

Как сделать свой генератор

Для создания своего генератора необходимо наследовать базовый класс `VarEvent` и реализовать два абстрактных метода базового класса. Кроме того, поскольку для хранения всех генераторов передается указатель, вы должны предоставить именно указатель на объект генератора, для чего удобно сделать модульную функцию (аналог `new` в C++). Для примера приводится класс `VarEventNormal` для генерации периодических событий:

```
module var_event_normal_module

use var_event_module

type, extends(VarEvent) :: VarEventNormal
private
    integer :: period_sec_ = 0
    type(DateTime) :: start_date_
 contains
    procedure :: update
    procedure :: destroy
end type

 CONTAINS

! Аналог new: создаем динамический объект и возвращаем ссылку на него.
! И заодно выполняем обычные функции конструктора -
! инициализируем генератор начальной датой, событием и периодом генерации.
function new_VarEventNormal(ev, start_date, dd, dh, dm, ds) result (obj)
    type(Event), intent(in) :: ev
    type(DateTime), intent(in) :: start_date
    integer, intent(in) :: dd, dh, dm, ds
    class(VarEventNormal), pointer :: obj

    allocate(VarEventNormal::obj)

    obj % ev_ = ev
    obj % start_date_ = start_date
    obj % period_sec_ = dd*60*60*24 + dh*60*60 + dm*60 + ds
end function

! Главная функция генерации. В зависимости от текущего времени
! <cur_time> генерирует событие <ev>
! и возвращает true или же возвращает false.
! По сути просто проверяет кратность периода генерации разнице текущего
! времени и стартового.
logical function update(this, cur_time, ev)
    class(VarEventNormal) :: this
    type(DateTime), intent(in) :: cur_time
    type(Event), intent(inout) :: ev

    integer(8) :: sec_from_start
```

```
    sec_from_start = date2sec(cur_time) - date2sec(this % start_date_)
    update = sec_from_start >= 0 .AND. MOD(sec_from_start, this % period_sec_) == 0

    if (update) then
        ev = this % ev_
    end if
end function

! Объект не содержит внутренних динамических данных, поэтому деструктор пустой.
subroutine destroy(this)
    class(VarEventNormal) :: this
end subroutine

end module
```

Теперь в системе определен еще один тип генератора, но она пока не знает об этом. Чтобы подключить генератор к системе и облегчить пользователю жизнь, необходимо добавить простой враппер для создания нового генератора в класс `Component`:

```
subroutine register_periodic_event(this, arr_name, act, src, dst, dd, dh, dm, ds)
    class(Component) :: this
    character(*), optional, intent(in) :: src, dst
    character(*), intent(in) :: arr_name, act
    integer, intent(in), optional :: dd, dh, dm, ds
    type(Event) :: ev
    type(ArrayInfo) :: arr_info

    arr_info = this % get_array_info(arr_name)
    ev = Event(arr_info = arr_info, act = act, owner = CompSplitter % i_am(), &
        src = src, dst = dst, file_prefix = this % prefix())

    call this % ev_scheduler_ % add( &
        new_VarEventNormal( ev, ExpInfo % start_date(), &
        MERGE(dd,0,PRESENT(dd)), MERGE(dh,0,PRESENT(dh)), &
        MERGE(dm,0,PRESENT(dm)), MERGE(ds,0,PRESENT(ds))))
end subroutine
```

В итоге, пользователь пишет что-то вроде:

```
call this % register_periodic_event(arr_name = "test_dg_2D", act = "SAVE_DG", dh = 1)
```

и враппер `register_periodic_event()` сам строит объект события, определяет какие-то переменные по умолчанию, создает объект генератора в динамической памяти и дает указатель на него в объект `EventScheduler`.

Замечания для разработчика системы

Сейчас для типа события надо знать сервис, который его будет обрабатывать. Эта информация не используется нигде, за исключением момента регистрации ga-массива в компоненте, так как он должен явно знать, с кем синхронизироваться. Если вопрос явной синхронизации при создании массива в `CommunicatorGA % init_array()` будет решен, эту зависимость можно убрать вообще.
Возможно, имеет смысл сделать аналог JSON для фортрана (FSON).

5.4 Сервисы

Общая идея

Система совместного моделирования в каком-то виде реализует сервисно-ориентированную архитектуру (SOA). Идея в том, что на стороне клиента (модели) генерируются события и отправляются соответствующие запросы (control flow), которым соответствуют данные, укладываемые в Global arrays (data flow). На стороне сервера некоторые сервисы разбирают запросы из единой очереди и выполняют работу (аналог конвейера).

Сейчас, определены следующие сервисы:

- `DTR` (distributor) – подписан на все события, просто рассылает их всем остальным сервисам. Нужен для содержания единой очереди в параллельной среде (аналог мастера).
- `IOD` (I/O device) – подписан на события `READ_FD`, `SAVE_CP`, `SAVE_DG`. При получении сообщения он распаковывает его, понимает, что от него требуется (например, взять данные из GA с именем "`test_ga_ocn`" и записать в файл "`OCN_180x90_tst_DG`") и выполняет необходимые действия. Остальные типы сообщений просто игнорируются.
- `CPL` (coupler) – подписан на событие `SEND_MP`. Данное событие определяет, где взять данные, что с ними сделать, и куда положить (то есть это push-событие, так как события `RECV_MP` для его обработки не требуется).

Как это работает

Для упрощения создания нового сервиса реализован базовый абстрактный класс `Service`, имеющий следующий интерфейс:

```
! Конструктор базового класса
procedure, public :: init_base

! Деструктор базового класса
procedure, public :: destroy_base

! Главный цикл обработки событий
procedure, public :: request_cycle

! Виртуальный метод обработки одного события
procedure(I_handle_request), private, DEFERRED :: handle_request

! Виртуальный конструктор
procedure(I_init_service), public, DEFERRED :: init

! Виртуальный деструктор
procedure(I_destroy_service), public, DEFERRED :: destroy
```

Главная программа **cpl_main.f90** содержит такие строчки:

```
select case(CompSplitter % i_am())
    case("DTR")
        allocate(ServiceDTR :: service_p)
    case("CPL")
        allocate(ServiceCPL :: service_p)
    case("IOD")
        allocate(ServiceIOD :: service_p)
end select

! Start model cycle
```

```
if (CompSplitter % is_model()) then
    call comp_p % model_cycle()
else
    call service_p % init_base(comm)
    call service_p % init()
    call service_p % request_cycle()
end if
```

То есть каждый процесс, который принадлежит к группе процессов некоторого сервиса (определяется в `CompSplitter`), аллокирует свой полиморфный указатель и дальше вызывает конструктор и входит в цикл обработки событий `request_cycle()`. (В конце работы программы подобным же образом вызываются деструкторы сервисов)

Метод базового класса `request_cycle()` содержит два одинаковых цикла, один для регистрации массивов, второй для реальной обработки событий. Структура цикла проста: пока еще есть работающие модели, принимай запрос, вызывай виртуальный метод `handle_request(ev)` и отслеживай сигналы `STOP` от моделей.

```
do while (this % running_count_ > 0)

    ! Receive any request
    call this % receive_request(ev)
    call this % handle_request(ev)

    select case(ev % action())
        ! One component finish work
        case("STOP")
            this % running_count_ = this % running_count_ - 1
            CYCLE
    end select
end do
```

Как сделать свой сервис

- Создаем костяк производного класса

В результате, для создания сервиса, необходимо наследовать класс `Service` и определить там три виртуальных метода: `init`, `destroy`, `handle_request`.

```
module service_tst_module

use utils_module
use actions_module
use service_module
use event_module
use communicator_ga_module
use component_splitter_module

implicit none

type, extends(Service) :: ServiceTST
private

 contains
    !=======================
    ! ===== PUBLIC API =====
```

```
    procedure, public :: init => init_tst
    procedure, public :: destroy => destroy_tst
    procedure, public :: handle_request => handle_tst

    procedure, private :: handle_my_method1
    procedure, private :: handle_my_method2

    ! ===== PUBLIC API =====
    !======================
end type

 CONTAINS

subroutine init_tst(this)
    class(ServiceTST) :: this
    ! Ваш конструтор
end subroutine

subroutine destroy_tst(this)
    class(ServiceTST) :: this
    ! Ваш деструктор
end subroutine

subroutine handle_tst(this, ev)
    class(ServiceTST) :: this
    ! Код обработчика события ev читай( далее)
end subroutine

! Остальные методы
end module
```

- Заполняем класс

Заполняем методы `init_tst`, `destroy_tst` необходимыми действиями. Далее заполняем основной метод – `handle_tst`. По действующему сейчас соглашению, когда сервис "видит" массив первый раз, метод должен зарегистрировать данный канал связи (то есть ga-массив) в коммуникаторе. Для этого можно использовать такую конструкцию:

```
if (.NOT. this % comm_ % is_registered(ev % ga_name())) then
                call this % try_register_service_ga(ev)
...
! Другие действия, требуемые в первый раз при приеме сообщения данного типа
! ( то есть по этому ga- каналу)
end if
```

Если массив уже зарегистрирован, можно сразу заняться его обработкой. В качестве примера ниже приведен метод `handle_request` класса `ServceIOD`. Он обрабатывает только события, связанные с работой с файлами и сообщение об ошибке `ERROR` (которое просто ведет к аварийному завершению). Остальные события игнорируются. Во время первого приема выполняется регистрация массива, во время остальных – обработка события и вывод информации в stdout встроенным вспомогательным методом базового класса `report_handle()`. Методы `handle_put()`, `handle_get()` содержат реальную логику извлечения массива из ga и записи его в файл средствами `FileHandler_NC`.

```
subroutine handle_iod(this, ev)
    class(ServiceIOD) :: this
```

```
    type(Event), intent(in) :: ev

    select case (ev % action())
        case("SAVE_CP", "SAVE_DG")
            if (.NOT. this % comm_ % is_registered(ev % ga_name())) then
                call this % try_register_service_ga(ev)
            else
                call this % handle_put(ev)
                call this % report_handle(ev)
            end if

        case("READ_FD")
            if (.NOT. this % comm_ % is_registered(ev % ga_name())) then
                call this % try_register_service_ga(ev)
            else
                call this % handle_get(ev)
                call this % report_handle(ev)
            end if

        case("ERROR")
            call this % report_handle(ev)
            call exit(1)
    end select
end subroutine
```

Внимание: обратите внимание на вопросы синхронизации! Сервис должен блокироваться, если ga-массив (представляющий собой буфер обмена) еще занят (для события вида “положить данные”) и, наоборот, свободен (для события вида “взять данные”). Для этого выполняется проверка статуса информационного массива. Если не устанавливать блокировку, нет гарантии, что в результате переданные данные будут цельными. После того, как сервис положил или взял данные, он должен поменять статус ga-массива соответствующим образом.

Например, когда `ServiceIOD` получает запрос `SAVE_DG`, он знает (смотри соответствующий обработчик в `Component`), что данные уже полностью лежат в ga-массиве, поэтому дополнительных проверок не требуется. Когда же сервис готов освободить ga-массив (в методе `handle_put()` после того, как скопировал их в свою память, он синхронизируется и помечает массив как свободный:

```
    call this % comm_ % sync(service_id)
    if(CompSplitter % is_first_rank()) call this % comm_ % put_info(ev % ga_name(),&
        COMM_GA_STATUS, "free")
```

- Регистрируем сервис в системе

Внимание: в этом пункте вносятся изменения в код системы. Если вы не уверены в своих действиях, обратитесь к разработчику.

На данный момент система ничего не знает о новом сервисе (назовем его `TST`), поэтому нужно:

1) В файле `ComponentSplitter` (что это за класс читать тут) добавить необходимые имена сервисов в массивы компонент и сервисных компонент:

```
character(3), parameter :: COMPONENT_NAMES(9) = &
    (/ "DTR", "CPL", "IOD", "OCN", "ATM", "ICE", "LND", "SEA", "TST"/)
character(3), parameter :: SERVICE_COMPS(4) = &
    (/ "DTR", "CPL", "IOD", "TST"/)
```

Класс занимается делением на группы процессов в многопроцессорной среде, и теперь каждый процесс может узнать, принадлежит ли он группе, например, `CPL`.

2) Подключаем в `cpl_main` модуль сервиса и определяем создание реального объекта сервиса как раз с использованием предыдущего пункта:

```
select case(CompSplitter % i_am())
...
    case("TST")
        allocate(ServiceTST :: service_p)
...
end select
```

Теперь, если при запуске вы укажете в аргументах `TST 2`, то система запустит новый сервис на 2-х процессах.

- Посылаем правильные запросы из клиента

Теперь сервис полностью в рабочем состоянии – он запускается и принимает запросы от клиента (пока это просто уведомления об окончании счета `STOP`). Чтобы клиент гененировал правильные события, необходимо определить новое событие, сделать для него генератор, и описать необходимые с его стороны действия. Как это сделать, описано в разделе Модельный компонент .

Замечания для разработчика системы

Если в будущем появится возможность избавиться от явной синхронизации через массив информации, это будет хорошо. Сейчас такой подход выбран, так как мы не можем “терять данные”. То есть даже если какой-то компонент (модель океана), работает быстрее другого компонента (модели атмосферы или `IOD`, который медленно пишет данные в файл), мы не имеем право “потерять” массив. Накопление массивов в виде очереди тоже ни к чему не приведет, так как модели обычно работают с постоянной скоростью и в результате очередь просто исчерпает всю доступную память. Поэтому сейчас, если “быстрая” модель уже готова положить данные, но ga-буфер еще занят, она блокируется.

5.5 Работа с NetCDF-файлами

Общая идея

NetCDF представляет собой машиннонезависимый самоописываемый формат и набор библиотек для работы с ним. NetCDF является фактическим стандартом для хранения геофизических данных. В результате, чтобы сохранить, например, массив скоростей, не нужно придумывать свои процедуры с кучей read/write, а достаточно вызвать готовую функцию библиотеки NetCDF. Важным свойством процедур является то, что они могут выполняться как в последовательном, так и в параллельном режимах.

NetCDF имеет довольно высокоуровневый интерфейс с точки зрения операций над файлами, но довольно низкоуровневый с точки зрения пользователя, так как приходится разбираться с тонкостями тех или иных встроенных процедур. При этом контроль за правильностью всех операций полностью лежит на пользователе. Поскольку работать с NetCDF файлами приходится часто, возникает желание создать некий класс-помощник, который будет иметь высокоуровневый интерфейс, скрывая все сложности NetCDF внутри себя. Так появился класс `FileHandler_NC`. Он, во-первых, упрощает работу с NetCDF, во-вторых, добавляет некоторый функционал. Например, класс дает удобную возможность доступа к данным не только по индексу, но и по временной метке и возможность считывания временной оси файлов в разных форматах. Для работы с классом достаточно подключить модуль `file_handler_nc_module`, создать экземпляр класса и с помощью него управлять файлом. `FileHandler_NC` используется в:

— `Service_IOD` для параллельных put/get-операций

— `Component` для анализа файла с временной осью

— `Offline` для создания файлов начальных данных

— Service_CPL для чтения интерполяционных файлов

— в системе подкачки данных наблюдений и т.д.

В результате все операции с NetCDF всей системы делегируются классу-помощнику, что сильно упрощает код за счет инкапсуляции всей логики в одном месте.

Пример работы

Разные способы работы с классом можно посмотреть в тесте **Coupler/test/test_filehandler_nc.f90**. Например, стандартная схема работы такая: создать хэндлер, с помощью него создать файл, переменные, записать данные.

```
use file_handler_nc_module

type(FileHandler_NC) :: handler

call handler % create_file("test_2D_dt.nc", mpi_comm = tm % comm())
call handler % create_dim("i", il)
call handler % create_dim("j", jl)
call handler % create_dim("k", kl)
call handler % create_time_dim()

call handler % create_var("test_2D_dt", "real4", "i", "j", dimt_name = "TIME")

call handler % put(arr_2D, lo = decomp % lower_bound_2D(), &
    hi = decomp % upper_bound_2D(), dt = DateTime(1988, 03, 15, 0, 0, 0))

call handler % close_file()
```

Описание API

Создание/открытие/закрытие файла

```
create_file(filename, mpi_comm)

Description:      Create file or rewrite previous
Parameters:
  filename        - string name of file to create
  mpi_comm        - <optional> MPI mpi_comm if it is parallel run
```

```
open_file(filename, mpi_comm, status)

Description:      Try to open file
Parameters:
  filename        - string name of file to open
  mpi_comm        - <optional> MPI mpi_comm if it is parallel run
  status          - <optional> status of operation: 0 if ok, 1 is error
```

```
open_or_create_file(filename, mpi_comm)

Description:      Try open and then create file
Parameters:       Combination of parameters for open_file and create_file procedures
```

```
close_file()
Description:      Close current file
Parameters:
```

Создание размерностей

```
create_dim(name, length)

Description:      add NC-dimension to file
Parameters:
  name            - name of dimension
  length          - corresponding length of dimension
```

```
create_time_dim()

Description:      add time NC-dimension to file
Parameters:
```

Создание/открытие переменных на размерностях

```
create_var(var_name, var_type, dim1_name, dim2_name, dim3_name, dimt_name)

Description:      Try to create var. Error if this var is already exist.
Parameters:
  var_name        - string name of file to create
  var_type        - type of variable
  dim*_name       - create variable of these dimensions
```

```
open_var(var_name, status)

Description:      Try to open variable
Parameters:
  var_name        - string name of variable to open
  status          - <optional> status of operation: 0 if ok, 1 is error
```

```
open_or_create_var(this, varname, var_type, dim1_name, dim2_name, dim3_name, dimt_name)
Description:      try open and then create variable
Parameters:       combination of parameters for open_var and create_var procedures
```

Запись/чтение переменных

```
put/get (arr, lo, hi, dt)

Description:      put and get data.
Parameters:
  arr    - data array of supported type and dimension
           (int4, int8, real4, real8, 1D, 2D, 3D)
  lo, hi - lower and upper bounds of dimensions (e.g. (/1, 1/), (/ il, jl /) )
  dt     - <optional> DateTime corresponding to field. Necessary for time vars.
```

Различные операции

```
put_att/get_att(att_name, att_val, is_global)

Description:      put/get attribute to variable or whole file
```

```
Parameters:
  att_name        - name of attribute
  att_val         - value of attribute of supported type (int, character)
  is_global       - <optional> if .TRUE., this attribute is made NF90_GLOBAL
```

```
function get_dim_size(dim_name)
Description:      Return size of interested dimension
Parameters:
  dim_name       - dimension name
```

```
get_time_axis(time_axis)
Description:      Get time axis
Parameters:
  time_axis       - integer(8), allocatable :: time_axis(:) - where to put time axis
                    in seconds from DateTime % epoch_start()
```

```
logical is_time_var()
Description:      Check if current variable has a time axis
Parameters:
```

5.6 GA-коммуникатор

Общая идея состоит в том, чтобы предоставить пользователю возможность простого доступа к разным частям распределенного массива. Для этого используется абстракция PGAS (Partitioned Global Adress Space). PGAS предполагает, что есть некоторый виртуальный огромный массив, который доступен из любого процесса, участвовавшего в его создании. Конечно, на самом деле никакого глобального массива нет, а его части хранятся в памяти процессов, но пользователь об этом не знает – все тонкости берет на себя библиотека, за счет чего и достигается простота. Например, клиент на процессе 12 может попросить элемент с индексами [124, 97], как будто он имеет к нему прямой доступ. За кадром PGAS узнает, какому процессу именно принадлежит элемент (например, 18-му), выполнит MPI-запрос на него, получит результат и вернет клиенту.

Реализация абстракции PGAS – библиотека Global Arrays (GA) (она, кстати, тоже устанавливается скриптом установки ПО). Существуют и другие реализации. Наконец, класс Communicator_GA – это класс системы CMF, представляющий своего рода фасад для этой библиотеки, то есть он определяет еще более высокоуровневый интерфейс и прячет некоторые тонкости GA.

Интерфейс

```
subroutine init(max_index, proc_local_count)

Description:              construct communicator object for <num_of_sides> components
Parameters:
    max_index             - maximum index of component, which will be used for work
                            with object (normally equal to number of defined comps)
    proc_local_count      - size of local communicator, required for agile memory
                            allocation
```

```
subroutine init_group(src_id, src_ranks, dst_id, dst_ranks)

Description:              register processor group between two sides
Parameters:
   src_id, dst_id         - ids of sides
   src_ranks, dst_ranks   - ranks of all processes of sides
```

```
subroutine init_array(arr_name, datatype, dimnum, holder_id, holder_decomp, &
                      subscriber_id)

Description:               initialize array based on Decomposition object, it can be
                           accessed from <holder_id> and <subscriber_id> components,
                           but stored on <holder_id>. If array with such name already
                           exists - delete previous and create new.
Parameters:
    arr_name               - string name of array
    datatype               - supported datatype string: "real4", real8", "int4"
    dimnum                 - supported dimension string: "2D", "3D"
    holder_id              - id of source component who hold array in memory
    holder_decomp          - decomposition of holder side
    subscriber_id          - id of subscriber component
```

```
subroutine init_array(arr_name, datatype, dim1_len, dim2_len, dim3_len, holder_id, &
                      holder_size, subscriber_id)

Description:               initialize array based on dimension sizes, it can be
                           accessed from <holder_id> and <subscriber_id> components,
                           but stored on <holder_id>. If array with such name already
                           exists - delete previous and create new.
Parameters:
    arr_name               - string name of array
    datatype               - supported datatype string: "real4", real8", "int4"
    dimnum                 - supported dimension string: "2D", "3D"
    dim*_len,              - size of each dimension
    holder_id              - id of source component who hold array in memory
    holder_size            - how many processors owns the GA
    subscriber_id          - id of subscriber component
```

```
subroutine destroy_array(arr_name)

Description:      destroy global array (you should set appropriate group before this
                  call - the same as on init_array)
Parameters:
    arr_name        - string name of array
```

```
subroutine sync(src_id, dst_id)
Description:               barrier for src_id, dst_id
Parameters:
    src_id, dst_id         - indexes of groups
```

```
integer function id(arr_name)

Description:              return ga_id of array with given name.
                          Return -1 if no such array.
Parameters:
    arr_name              - string name of array
```

```
subroutine put (arr_name, lo, hi, arr)
subroutine get (arr_name, lo, hi, arr)
```

```
Description:            put/get data
Parameters:
    arr_name            - string name of array
    lo, hi              - arrays representing area (in global indexing) you
                          want to put/get
    arr                 - your buffer for data
```

Примеры использования

Более детально примеры использования можно посмотреть в тестах к классу (**coupler/test/ga_communicator**). Ниже приведены популярные примеры, взятые как раз оттуда.

Создание массива, разделяемого двумя компонентами

Класс позволяет создать массив, который будет распределенным на одном компоненте, но при этом еще и видимым другому компоненту. Это позволяет, например, создать массив температуры, который физически будет распределен по ядрам океана (и они могут класть и брать из него данные), но, кроме того, сервис каплера тоже может работать с этим массивом, хотя не хранит у себя никакую его часть.

```
! Просим у CompSplitter-а идентификаторы компонентов
ocn_id = CompSplitter % comp_id("OCN")
cpl_id = CompSplitter % comp_id("CPL")

! Инициализируем объект коммуникатора количеством компонентов и локальным размером
! коммуникатора каждой
call comm_ga % init(CompSplitter % comp_defined(), CompSplitter % comm_local_size())

! Просим у CompSplitter-а списки процессов в каждом компоненте
call CompSplitter % proc_list("OCN", list = proc_list_ocn)
call CompSplitter % proc_list("CPL", list = proc_list_cpl)

! С помощью них регистрируем группу [ocn_id, cpl_id]
call comm_ga % init_group(ocn_id, proc_list_ocn, cpl_id, proc_list_cpl)

! Океан создает свою декомпозицию и раздает всем - это нужно, чтобы все
! заинтересованные ядра группы вызвали регистрацию массива с одинаковыми параметрами.
! В реальной программе можно взять данные декомпозиции из
! глобального массива ModelInfo, который хранит информацию обо всех компонентах
if (CompSplitter % i_am() == "OCN") then
    ocn_decomp = Decomposition(il, jl, kl, "2D", CompSplitter % comm_local_size(), &
        CompSplitter % rank_local())
end if

call ocn_decomp % broadcast(CompSplitter % proc_first("OCN"), &
    CompSplitter % comm_world())

! Наконец, регистрируем массив, указав в качестве держателя океан, передав его
! декомпозицю чтобы( GA распределила массив именно так), а в качестве пользователя -
! каплер
call comm_ga % init_array(arr_name = "glob_2D", datatype = "real4", dimnum = "2D", &
    holder_id = ocn_id, holder_decomp = ocn_decomp, subscriber_id = cpl_id)

! Теперь можно положить в массив данные: например, пусть только океан кладет, причем
! каждое ядро кладет глобальный массив бессмысленно( в реальной программе)
```

```
if (CompSplitter % i_am() == "CPL") call comm_ga % put("glob_2D", (/ 1, 1 /), &
    (/ il, jl /), tgd % glob(:,:,1,1))

! Обязательно выполняем синхронизацию, то есть ждем, что все положили данные,
! так как вызовы put/get - неблокирующие
call comm_ga % sync(ocn_id, cpl_id)

! Теперь можно взять данные: все ядра обоих компонентов берут локальные куски и
! сравнивают их с предопределенными тестом
call comm_ga % get("glob_2D", (/ w, s /), (/ e, n /), tgd % loc_2D)
call tm % assert(tgd % is_correct(arr_type = "2D"), "all get 2D local patch &
    defined in parameters")
```

Создание массива, разделяемого только одним компонентом

Иногда необходимо создать массив для использования только внутри одного компонента. В этом примере мы создадим такой массив для компонента океана. Кроме того, вместо того, чтобы передавать процедуре декомпозицию океана, мы просто передадим размерности, чтобы класс сам поделил массив на части за нас. Большинство шагов повторяют предыдущий пример, за исключением того, что группа и массив теперь создается для одинаковых идентификаторов (держатель и подписчик одинаковые) и вызывается версия процедуры регистрации массива без указания декомпозиции.

```
! Просим у aCompSplitter- идентификаторы компонента
ocn_id = CompSplitter % comp_id("OCN")

! Инициализируем объект коммуникатора количеством компонентов и локальным
! размером коммуникатора каждой
call comm_ga % init(CompSplitter % comp_defined(), CompSplitter % comm_local_size())

! Просим у aCompSplitter- списки процессов в каждом компоненте
call CompSplitter % proc_list("OCN", list = proc_list_ocn)

! Регистрируем группу только для океана
call comm_ga % init_group(ocn_id, proc_list_ocn)

! Регистрируем массив, не указав подписчика - это будет сам компонент. Кроме того,
! передаем только размеры массива il, jl
call comm_ga % init_array(arr_name = "priv_2D", datatype = "real4", dim1_len = il, &
    dim2_len = jl, holder_id = ocn_id, &
    holder_size = CompSplitter % comm_local_size("OCN"))

! Кладем данные опять( глобальные)
call comm_ga % put("priv_2D", (/ 1, 1 /), (/ il, jl /), tgd % glob(:,:,1,1))

! Синхронизируем, чтобы убедиться, что все положили данные
call comm_ga % sync(ocn_id)

! Берем локальные данные
call comm_ga % get("priv_2D", (/ w, s /), (/ e, n /), tgd % loc_2D)
```

5.7 Дополнительные инструменты для модели

Перечисленные в этом разделе инструменты не являются логической частью CMF, а представляют собой некоторые внешние инструменты, которыми может пользоваться клиентский код. Например, несмотря на то, что хало-обмены или reduce-операции не нужны всем моделям, CMF содержит их виде

отдельных “удобных” функций, которые работают на коммуникаторе вызвавшей модели и не видны остальным компонентам системы.

Хало апдейтер

Общая идея

Во многих моделях возникает необходимость обмена приграничными ячейками локальной расчетной области данного процесса с процессами-соседями. Эту задачу решает класс `HaloUpdater`. Формально, он не является частью CMF, а представляет собой отдельный модуль, который любая модель может использовать. Сейчас функции обмена реализованы для широтно-долготной и биполярной сеток (T и V), для 2D-, 3D-массивов любого типа. Структурно модуль состоит из шаблонного (без конкретизации типов) класса `HaloUpdaterBase`, который выполняет основную работу, и класса `HaloUpdater`, который представляет высокоуровневый интерфейс клиенту и вызывает конкретные методы низкоуровневого класса.

С точки зрения пользователя при подключении модуля `halo_updater_module` становится доступен объект `HaloUpdateMaster`, который и используется для обменов.

Пример использования

Подробно с работой блока можно ознакомиться в тесте (**coupler/test/halo_update**). Ниже приведен типичный пример использования. Обратите внимание на необязательный параметр `change_size_on_bipolar`: если он равен истине, знак будет изменен. В обратном случае (или если он не указан) – знак будет сохранен. Кроме того, для удобства все процедуры имеют один вид для всех видов массивов (это называется перегрузкой функций).

```
use halo_updater_module

! Создаем объект декомпозиции
ocn_decomp = Decomposition(180, 90, 20, "2D", comm_local_size, rank_local, &
   is_icycle = .true., is_bsc = .true.)

! Инициализируем мастера, передавая ему декомпозицию модели и максимальную
! ширину обмена
call HaloUpdateMaster % init(decomp = ocn_decomp, max_halo_width = 2)

! Можно обменивать: в данном случае обмениваем трехмерный массив температуры
! с шириной обмена 1
call HaloUpdateMaster % update(t_c(:,:,:,1), update_width = 1, grid_type = 'T')

! А теперь двумерный массив u_c(1,:,:) с шириной 2
call HaloUpdateMaster % update(u_c(1,:,:), update_width = 2, grid_type = 'V', &
    change_sign_on_bipolar = .TRUE.)
```

Мелкие утилиты

UtilsAllReduce

Возвращает глобальную сумму переменной по коммуникатору или, при наличии весов, взвешенную сумму. Внимание: будьте осторожны с глобальными операциями – они могут привести к падению производительности.

```
use utils_module

real(kind=8) :: my_global_sum, my_global_int

! Вычисляем сумму по коммуникатору океана для переменной some_local_val
```

```
my_global_sum = UtilsAllReduce(local_val = some_local_val, comm = ocn_comm)

! Вычисляем взвешенную сумму по коммуникатору океана для переменной some_local_int
my_global_int = UtilsAllReduce(local_area = some_area, local_val = some_local_int, &
    comm = ocn_comm)
```

PointSaver

Сохраняет точку данных вместе с соответствующей временной меткой.

```
use point_saver_module

type(PointSaver) :: ps
real(8) :: some_val

! Инициализируем объект именем файла ( и таким же именем переменой внутри него)
! и периодом реального сброса данных на диск.
call ps % init(varname = "var1", flush_period = 100)

! Кладем данные в файл вместе с текущим временем ( в данном случае
! оно запрашивается у компонента через метод model_time())
call ps % put(some_val, cmp_ptr % model_time())

! Не забываем в конце вызвать деструктор закрыть( файл)
call ps % destroy()
```

6 Краткие инструкции

Здесь приводятся краткие списки, о чём надо не забыть при той или иной перенастройке модели.

6.1 Переключение конфигурации

— К-во ядер для льда задаётся в **ice_in** и **comp_ice**

— Шаг по времени для льда задаётся в **ice_in** и **ice_list.in**

— Базы данных форсинга выбираются в **atm_list.in** и **lnd_list.in**

— Формулу точки замерзания в **ice_module.f90** желательно вызывать ту же, что и в используемой термодинамике CICE (по умолчанию — пористая (mushy)).

— После каждой перенастройки, затрагивающей настройки CICE, полностью перекомпилировать совместную модель.

— Если используется урезанная сетка льда, то в **ice_list.in** и **config** должен быть указан именно урезанный размер по j.

— Частота обменов океан-лёд задаётся в двух местах — в **o_tf_module.f90** и **ice_cice_driver_module.f90**.

6.2 Перенастройка форсинга в среде CMF2.0

— Форсинг во всех компонентах (атмосфера, суша,...) должен начинаться с одной даты.

— Стартовая дата в **run_list.in** должна совпадать с начальной датой форсинга

— Проверить корректировку `time_start_min`

— Проверить конверсию в потенциальную температуру

— Проверить инициализацию уровня

— Проверить, включена ли релаксация ТПО

— При регистрации чтения из файлов (`ACTION_READ_FD`) в рамках каждой компоненты все периоды чтения должны быть кратны минимальному из периодов чтения этой компоненты. В частности, из-за этого либо периоды чтения среднемесячных величин (runoff, rain и др.) приходится делать 30.5 или даже 30 суток вместо 1440*365/12 минут, либо надо вводить дополнительную пробную величину (probe), читаемую с периодом 2 часа.

7 Элементы численной и программной реализации

7.1 Замечания по отличиям CMF2.0 и CMF3.0

В обеих версиях системы счётчики `time_l` и `time_l_in_run` инициализируются единицей при старте с файла начальных условий и инкрементируются вне `_driver_module`. Они представляют собой номер шага, который делается сейчас (т. е. для океана они означают сколько будет сделано шагов, когда завершится текущий шаг физической модели). Отличие в том, что для CMF2.0 весь календарь вычисляется по этим счётчикам и, соответсвенно, `time_min`, `time_hour`,... – это момент окончания текущего модельного шага. В CMF3.0 эти переменные (`time_min`, `time_hour`,...) не вычисляются непосредственно каплером и сделаны как добавка, для совместимости с CMF2.0. И при этом они соответствуют не концу, а началу текущего шага.

A Приложение: параметры именованных списков

Таблица 1: Основные параметры именованных списков ИВМИО

Файл	Переменная	Значения	Комментарии
atm_list.in	atm_forcing_type	1	«нормальный» годовой цикл CNYFv2 (CORE-I)
		2	«реальные» данные IAFv2 за 1948-2009 гг. (CORE-II)
lnd_list.in	atm_rivers_type	1	«нормальный» годовой цикл CNYFv2 (CORE-I)
		2	«реальные» данные IAFv2 за 1948-20011 гг. (CORE-II)

Таблица 2: Основные параметры именованных списков CICE

Файл	Переменная	Значения	Комментарии
ice_in	calc_Tsfc	T	рассчитывать поверхностные тепловые потоки и температуру
		F	не рассчитывать, получить от каплера (требуется $ktherm = 0$)
ice_in	calc_strair	T	рассчитывать напряжение трения ветра
		F	не рассчитывать, получить от каплера

B Приложение: стандартные конфигурации

laptev0125c

Тестовая конфигурация для ПК. Расчётная область 40×60 в районе моря Лаптевых с добавленным искусственным островом-кольцом. Сетка сферическая, разрешение около $0.125°$. Ледовая сетка полная.

Форсинг CNYFv2. Коэффициенты турбулентности небольшие, близкие к глобальным вихреразрешающим настройкам.

arctic025t

Арктика от 50° с.ш. с разрешением 0.25°. Расчётная область 1440×160, сетка трёхполярная. Ледовая сетка полная. Форсинг CNYFv2. Вязкость только бигармоническая, диффузия 300 – по типу NEMO. Неявный Кориолис.