Решение вопросов балансировки нагрузки и обеспечения отказоустойчивости в распределённой системе сопряжения секторных моделей

Н.Д. Гибаев[1], М.О. Кашинцев[1], С.С. Самборецкий[1]

Тюменский государственный университет[1]

Рассматриваются вопросы обеспечения балансировки нагрузки и отказоустойчивости вычислений в распределённой системе, реализующей алгоритм сопряжения секторных гидродинамических моделей. Предлагается постановка задачи параллельных вычислений в ходе алгоритма; рассмотрены три алгоритма распределения заданий между узлами системы, показана программная реализация, результаты экспериментов и некоторые выводы по их результатам; рассмотрены различные способы обеспечения отказоустойчивости, обоснован выбор одной из стратегий и показана её реализация для рассматриваемой системы. Получаемое приложение может быть использовано для различных задач, решение которых находится через вычисление частей и сопряжение результатов.

Ключевые слова: секторное моделирование, программный комплекс, нефтегазовые месторождения, распределенные вычисления, параллельное программирование, балансировка нагрузки, отказоустойчивость распределённых систем.

1. Введение

Недавние исследования [1-4] показали возможность организации сопряжения смежных секторных гидродинамических моделей пласта с помощью распределённой вычислительной системы. В данных работах вопросы организации оптимальной балансировки нагрузки на вычислительные узлы и обеспечения отказоустойчивости были только озвучены, и они остались открытыми. Данное исследование призвано продемонстрировать подходы к организации балансировки нагрузки и обеспечению отказоустойчивости для распределенных систем, решающих подобные задачи.

Для данного исследования были выделены ограничения применимости. Вычисления балансируются в гомогенной системе, это позволяет упростить модель балансировки и не учитывать производительность аппаратного обеспечения. Система разработана на базе MPI.

2. Постановка задачи

2.1. Организация вычислений

Переход к более абстрактному представлению задачи позволяет увидеть следующую ситуацию: имеются наборы параллельных вычислительных заданий A и B, такие, что задание из множества B будет передано на выполнение тогда и только тогда, когда выполнено некоторое подмножество заданий из множества A, определенное для данного задания. Математически данные множества представимы в следующем виде:

$$A = \{a; a_i || a_j, i, j \in 1 \dots N\},$$
$$B = \{b; b_i || b_j, i, j \in 1 \dots N, \forall b_i \; \exists A'_i \subset A \; \}$$

Выражение $a_i || a_j$ означает, что задания могут выполняться параллельно. Выражение $\forall b_i \; \exists A'_i \subset A$ указывает, что любое задание из множества B связано с некоторым подмножеством A'_i заданий из множества A. Задание b_i будет выполняться тогда, когда предикат $P(A'_i)$, проверяющий, выполнены ли все задания из множества A'_i, вернёт значение «истина».

Наглядным представлением такой организации вычислений может служить схема, представленная на рисунке 1. Это представление отражает прикладную задачу секторного моделирования. В данной задаче выделяются два типа заданий: 1) гидродинамическое моделирование на секторной модели и 2) сопряжение смежных секторных моделей.

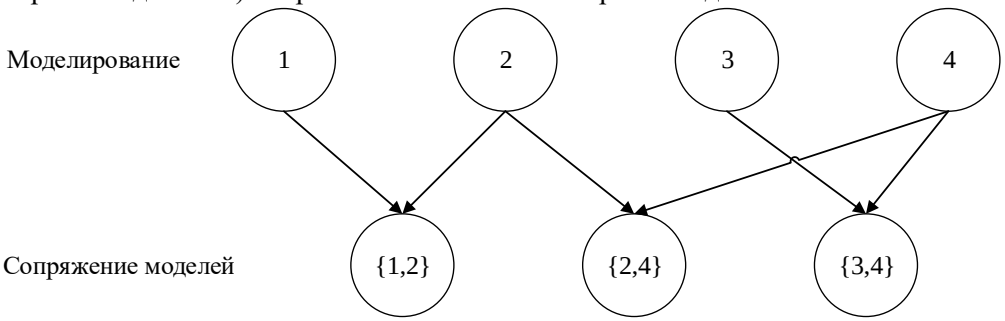

Рис. 1. Схема организации вычислений

Сопряжение производится между двумя смежными моделями тогда, когда на этих моделях уже произведены вычисления. Стоит указать, что вычислительная сложность сопряжения моделей низкая [3], поэтому к этим заданиям мы не применяем балансировку, в отличие от заданий на моделирование, вычислительная сложность которых на несколько порядков выше.

2.2. Проектирование распределённой системы

В работе [2] была предложена архитектура распределённого приложения для решения задач секторного моделирования. Для данной работы были изменены нефункциональные требования: программное обеспечение разрабатывается для гомогенного вычислительного кластера.

Принципы построения остались прежними. В основе системы лежит топология «звезда», рабочие процессы, участвующие в вычислениях, делятся на два типа: менеджер, координирующий работу системы, и рабочий, выполняющий вычисления.

Во время проведения исследования стало понятно, что проектируемая система может использоваться не только для задач секторного моделирования пласта. Данную систему можно использовать в дальнейшем для вычислений в любой предметной области, которые проходят через этапы разбиения на части (декомпозиции), параллельного вычисления частей и их сопряжения.

В ходе работы над приложением были рассмотрены различные варианты готовых решений для построения параллельных и распределённых систем. Был произведен сравнительный анализ двух наиболее подходящих технологий для данного решения: MPI и Windows Communication Foundation (далее WCF).

Изначально прототип разрабатывался на WCF. В пользу данного фреймворка говорили следующие его достоинства: он позволяет ускорить процесс разработки сетевого взаимодействия между процессами и предоставляет возможность отделить клиентское приложение, необходимое для формирования задания на запуск алгоритма, от приложения, производящего вычисления. Программирование производится на языке высокого уровня C#. Однако работа с данной технологией была ограничена созданием прототипа приложения по следующей причине: реализация WCF на отличных от Microsoft Windows операционных системах является неполной и нестабильной, что ставит под угрозу возможность организации вычислений на других системах. Ситуация может измениться в скором времени, поскольку были открыты исходные коды платформы .Net, в состав которой входит WCF, но временные рамки, наложенные на данное исследование, не позволяли ждать появления стабильных реализаций.

Поэтому было принято решение использовать уже достаточно хорошо изученный и апробированный на множестве задач пакет MPI. Его преимущества и недостатки исследованы во множестве работ. Для данного исследования виделись следующие проблемы: недостаточно высокий уровень абстракции при создании коммуникаций; отсутствие обеспечения отказоустойчивости. Однако при дальнейшем рассмотрении были сделаны выводы, что последнюю проблему можно решить несколькими способами, которые описаны ниже в данной работе. Это подтолкнуло к дальнейшей разработке с применением этой технологии.

3. Исследование алгоритмов распределения нагрузки

Пусть имеется M независимых задач и N процессов. Определим некоторую меру трудоемкости и построим вектор $T = (t_1, t_2, \dots, t_M)$ где t_i – трудоемкость i-ой задачи.

Требуется найти матрицу $A_{N\times M}$, удовлетворяющую следующим условиям:

$$\text{a) } a_{ij} = \{0,1\},$$
$$\text{b) } \sum_{i=1}^{N} a_{ij} = 1 \ \forall j.$$

При этом значение $|TA|$является минимальным среди всех $|TA'|$, где A' – матрица, удовлетворяющая условиям a) и b).

Иными словами, A – матрица распределения задач между процессами, TA – вектор загруженности процессов, i-ый элемент которого характеризует суммарную трудоемкость задач, назначенных i-ому процессу. Требуется, подобрать матрицу A так, чтобы минимизировать максимальный элемент TA, и, соответственно, минимизировать $|TA|$.

В системе были реализованы два алгоритма распределения заданий: обычный эвристический и улучшенный эвристический. Цель: определить, насколько оправдано усложнение эвристического алгоритма с точки зрения выигрыша во времени счёта и эффективности распределения.

3.1. Эвристический алгоритм

Для решения поставленной задачи предложен эвристический алгоритм, основной идеей которого является поочередное назначение задач наименее загруженному процессу. При этом на каждом шаге выбирается наиболее трудоемкая задача из оставшихся.

Пусть $T = \{T_1, \dots, T_M\}$ – множество заданий, $P = \{P_1, \dots, P_N\}$ – множество процессов. Алгоритм состоит из следующих этапов:

Шаг 1. Из T выбирается задача с максимальной трудоемкостью T_{max}.

Шаг 2. Из P выбирается минимально загруженный процесс P_{min}.

Шаг 3. Задача T_{max} назначается процессу P_{min}.

Шаг 4. Если остались нераспределенные задачи, то переходим к 1), иначе алгоритм завершается.

Таким образом назначая i-ую задачу процессу мы минимизируем шанс увеличения загрузки на максимально загруженном процессе.

Верхняя оценка времени работы алгоритма (при хранении частичных суммарных загруженностей) – $O(NM + S(M))$, где $S(M)$ – время, затраченное на сортировку M заданий по убыванию трудоемкости.

3.2. Улучшенный эвристический алгоритм

Данный алгоритм предложен К. Н. Ефимкиным [5] и основан на перестановке заданий между процессами, основанной на двух эвристических правилах.

Правило 1 переназначает задание менее загруженному процессу, если при этом разница в суммарной загруженности уменьшается.

Правило 2 переназначает два задания между двумя процессами, если при этом разница в суммарной загруженности уменьшается.

Для проведения данного исследования был реализован алгоритм для распределения M задач между M процессами ($M > 2$).

Шаг 1. Разобьём произвольным образом множество задач на непересекающиеся подмножества $T_1, \dots, T_N$ задач, выполняемых, соответственно, процессами $P_1, \dots, P_N$.

Шаг 2. Введем множество $MIN = \emptyset$ на множестве натуральных чисел $1 \dots N$.

Шаг 3. Выберем из $T_1, \dots, T_N$ множества T_{min} и T_{max} с минимальной и максимальной суммарной трудоемкостью соответственно.

Шаг 4. Применим к T_{min} и T_{max} правило 1, а если невозможно – правило 2. Если хотя бы одно правило применимо, переходим к 2), иначе – переходим к 5).

Шаг 5. Добавим к множеству MIN новый элемент $IND(T_{\min})$, где операция $IND(T_k)$ определяет номер $k \in \{1, \dots, N\}$ данного множества.

Шаг 6. Если $|MIN| = N - 1$ то алгоритм завершается, иначе – переходим к 3).

Также в работе [5] было показано, что итоговая верхняя оценка сложности алгоритма составляет $O(M + N + \frac{M^2}{2})(N - 1)$.

3.3. Реализация

Для обеспечения работы библиотеки с любыми типами задач, трудоёмкость которых поддаётся оценке, была введена система классов (рисунок 5). Ниже представлены классы `job` и `array_processor`. Оба данных класса являются виртуальными.

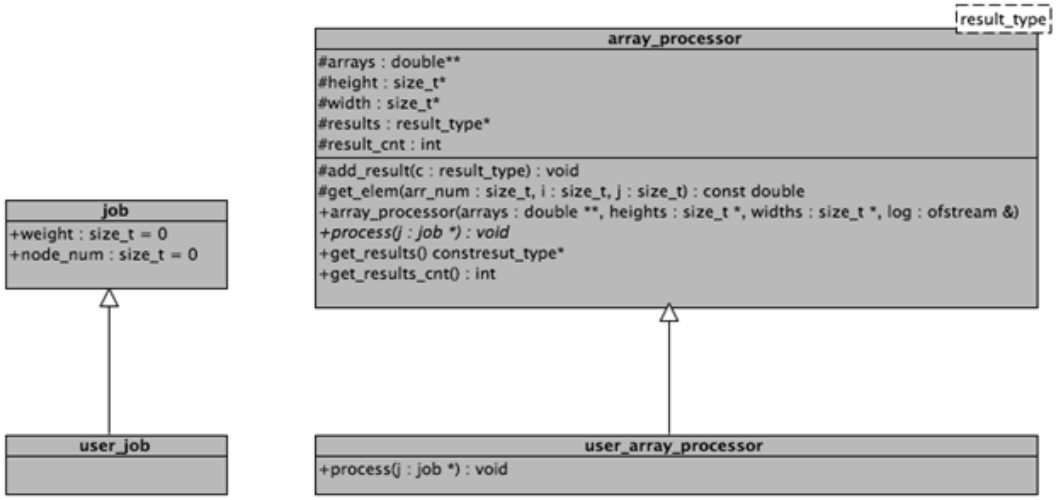

Рис. 2. Классы job и array_processor

Класс `job` представляет собой виртуальный класс задачи, содержащий трудоёмкость и номер процесса, выполняющего данную задачу. Для корректной работы алгоритмов класс реальной задачи должен быть унаследован от класса `job`.

Класс `array_processor` представляет собой виртуальный класс обработчика задач. Он содержит поля, указывающие на результаты выполнения, и поля, указывающие на входные данные. При использовании библиотеки данный класс также должен быть унаследован с указанием типа результата выполнения задачи в параметрах шаблона. Смысл состоит в реализации функции `process`, обрабатывающей задачу и добавляющей новый результат.

В библиотеке используется 2 функции распределения задач, каждая из которых соответствует одному из выше представленных алгоритмов:

- `heuristic_distribution` – распределяет задачи между процессами, руководствуясь эвристическим алгоритмом.

- `advanced_heuristic_distribution` – распределяет задачи между процессами, руководствуясь улучшенным эвристическим алгоритмом.

Выбор функции зависит от количества и типа задач и, на данный момент, определяется пользователем при создании объекта dispatcher, описанного ниже.

3.4. Результаты экспериментов

На таблице 1 приведены результаты тестирования эвристического алгоритма на разных объемах входных данных. В качестве оценочного критерия выступает отношение $\frac{\sigma}{AVG}$, где $AVG = \frac{\sum_{i=1}^{M} t_i}{N}$ – средняя (идеализированная) загруженность процесса, $\sigma = \sqrt{\frac{\sum_{j=1}^{N}(S_j - AVG)^2}{N}}$ – средняя разница между суммарной загруженностью процесса и AVG, S_j – суммарная загруженность процесса после распределения.

Таблица 1. Результаты тестирования эвристического и улучшенного алгоритма

Количество процессов	**Количество заданий**	$\frac{\sigma}{AVG}$**(%)**	$\frac{\sigma}{AVG}$**(%)**
2	30	3.2795	2.8207
3	20	7.5499	7.1889
4	15	9.3780	8.6258
5	15	15.5053	15.3769
2	300	0.3314	0.2743
3	200	0.8616	0.7546
4	150	1.2916	1.1493
5	150	1.6054	1.4289
2	3000	0.0332	0.0274
3	2000	0.0887	0.0799
4	1500	0.0946	0.0796
5	1500	0.1813	0.1628

Существенной разницы с точки зрения выбранного критерия в алгоритмах нет, однако стоит учитывать, что в 49,92% случаев решение, полученное улучшенным алгоритмом, в точности совпадает с оптимальным, полученным методом полного перебора. В 24,22% случаев решение, полученное улучшенным алгоритмом, отличается от оптимального, полученного методом полного перебора, не более, чем на 1%, в 25,5% случаев отклонение находится в пределах от 1 до 10%, и лишь в 0,35% случаев отличается более, чем на 10% [5].

4. Обеспечение отказоустойчивости распределённой системы

4.1 Возможные подходы реализации

Прежде всего было необходимо определиться с существующими подходами обеспечения отказоустойчивости с учетом специфики MPI, поскольку каждый подход требует не только различных способов проектирования самой MPI программы, но и вообще другого принципа построения оборудования вычислительной системы (например, различных топологий сети распределенной системы).

1. Использование наблюдателей (Supervisors, супервизоры) или Fail-Fast [7], суть которого заключается в том, чтобы отказы больше не вызывали остановку всех вычислительных MPI процессов, а только одного из них или лишь малой группы этих процессов. Использование

данной методики в проектировании распределенных вычислительных систем предполагает использовать определенный род вычислительных узлов (супервизоров), на которые ляжет ответственность в реагировании на ошибки выполнения. Если наблюдаемый узел не выполняет вычисления по той или иной причине, супервизор может иметь возможность либо перезапустить вычисления на данном узле, либо изменить политику работы и продолжить работать с оставшимися (но чаще предполагается использование первого подхода, ибо он более выгодный).

2. Сегментирование и выживание, когда при разработке распределенной системы предполагается низкий уровень связности. Это требует сбора избыточной информации перед коммуникационными сессиями. Выживание предполагает, что при отказе одной из компонент системы остальные процессы смогут продолжить дальнейшие вычисления. Данный подход значительно проще, чем предыдущий, поскольку тактика игнорирования отказов позволяет не решать такие проблемы, как перезапуск отказанных компонент распределенной системы. Для реализации данного подхода лучше всего подходит и звездообразная топология сети, и древовидная. Главное учитывать тот факт, что в случае отказа узла, его дочерние узлы также остановятся, поэтому лучше централизовать распределенную систему, скажем, при использовании звездообразной топологии, узел в центре звезды выделить под менеджмент вычислительных задач.

3. Модификация непосредственно MPI. Эта задача может быть простой, только если используется уже готовое решение. Сложность заключается в том, чтобы изменить правила работы MPI – изменяются эти правила так, чтобы некоторые исключительные ситуации больше не были исключительными. Подобная модификация сценариев, конечно, может рассматриваться как отступление от стандарта, тем не менее применяется не так уж и редко, если судить по количеству модификаций MPI [8].

4. Использование контрольных точек (Checkpointing). Контрольной точкой (Checkpoint) называют некоторый набор информации, хранящий состояние распределенной системы в конкретный момент времени. Данный подход продробно описан в [9].

В ходе проектирования приложения было решено использовать подход выживания, который позволяет сохранить баланс между сложностью своей реализации и предполагаемым уровнем надежности. После определения поведения программы в случае отказа, были сформулированы конкретные пути достижения отказоустойчивости:

1. Использование интеркоммуникаторов вместо стандартного MPI_Comm_World.
2. Переопределение каждому интеркоммуникатору обработчика ошибок.
3. Запрет использования стандартных групповых операций и их собственная реализация.
4. Периодическая проверка канала по результатам пересылки сообщения.

4.2. Архитектура отказоустойчивого приложения

В стандарте MPI определены два рода коммуникаторов: это интракоммуникаторы (IntraCommunicators) и интеркоммуникаторы (InterCommunicators). *Интракоммуникатор* – это коллекция MPI процессов, которые могут пересылать друг другу сообщения и участвовать в стандартных групповых MPI операциях. Для него в противовес существует *интеркоммуникатор*, который устанавливает канал связи "точка-точка" между двумя MPI процессами. Как можно видеть, коммуникатор по умолчанию MPI_Comm_World относится к первому типу коммуникаторов.

Для разработки отказоустойчивой системы можно предложить на выбор две возможных стратегии применения коммуникаторов:

1. Поделить систему на области, каждой области предоставить интракоммуникатор. Можно установить каждому интракоммуникатору обработчик ошибок MPI_ERRORS_RETURN и в случае отказа одного MPI процесса, его интракоммуникатор будет инвалидирован и вся область будет исключена из системы, что ведет к отключению всех смежных в данном интракоммуникаторе процессов.

2. Для всех имеющихся MPI процессов создать интеркоммуникатор(-ы) и так же установить каждому обработчик ошибок MPI_ERRORS_RETURN. В таком случае, в случае отказа будет инвалидирован только один интеркоммуникатор, связанный с отказавшим MPI процес-

сором. Пример подобного поведения показан на рисунке 3, где отключенный у правого узла интеркоммуникатор ведет только к исключению данного узла из топологии.

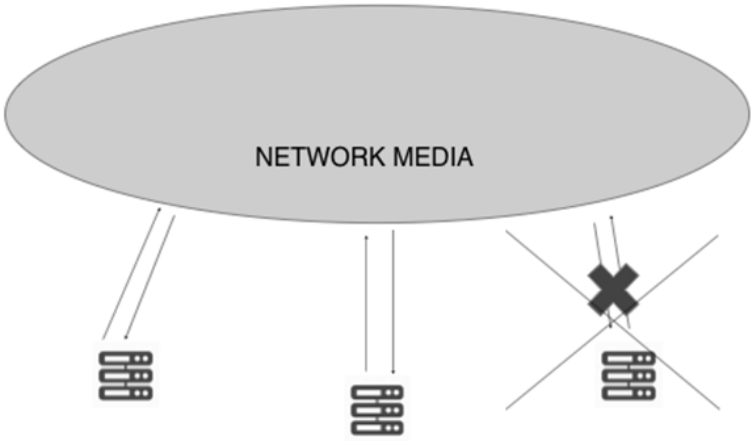

Рис. 3. Отключение интеркоммуникатора не затрагивает работы остальных MPI процессов

Предпочтение было отдано интеркоммуникаторам, поскольку они обладают следующими преимуществами:

1. В случае отказа одного MPI процесса, инвалидируется только его интеркоммуникатор, остальные продолжают работать без изменений в поведении.

2. Более тонкая конфигурация реагирования на отказ. Поскольку каждый интеркоммуникатор однозначно определяет канал связи между двумя определенными MPI процессами, имеется достаточно информации, чтобы гораздо более сложно реагировать на отказ, например, если определить свой собственный обработчик ошибок.

В данной работе было предложено использование логической топологии "звезда" для MPI программы. Централизация системы обусловлена тем, что необходимо иметь определенный MPI процесс, который будет заниматься менеджментом остальных MPI процессов. Конечно, еще остается уязвимость в виде самого центрального узла, но в рамках данного исследования было сделано допущение о его стабильной работе. В дальнейшем MPI процесс в центре будем называть Manager, а узлы на концах "звезды" – Workers. Задача Manager-процесса будет заключаться в инициализации фронта работ, инициализации Worker-процессов, пересылке указаний к заданию для Worker-узла и проверке их статусов. Иллюстрация полученной логической топологии узлов показана на рисунке 4.

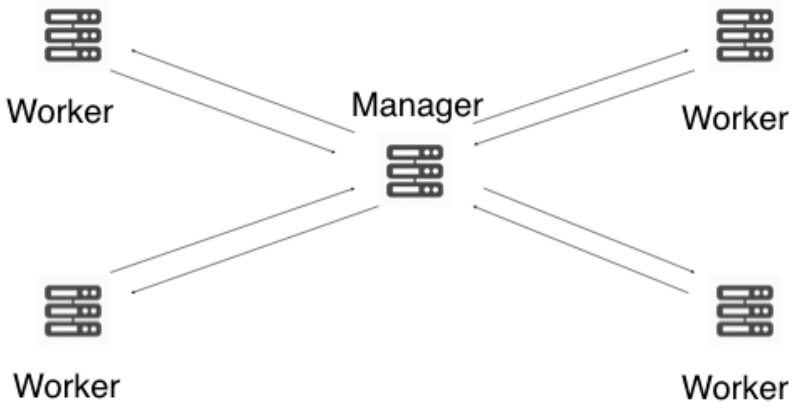

Рис. 4. Логическая топология процессов модели Manager-Worker

4.3. Программная реализация

Для обеспечения отказоустойчивости были разработаны два класса: Communicator и Dispatcher.

Класс Communicator предназначен для инкапсуляции основных MPI операций. Если запретить пользователю прямое управление коммуникационными процессами, это увеличит гарантию надежности.

Таблица 2. Некоторые члены класса communicator

Название члена, уровень доступа, аргументы, возвращаемый тип	**Описание члена**
Workers, private, N/A, MPI_Comm*	Массив интеркоммуникаторов, связанных с каждым Worker-процессом. *Прим.* : Не равен Null только у Manager-процесса
Manager, private, N/A, MPI_Comm	Интеркоммуникатор, связанный с Manager-процессом. *Прим.* : Не равен Null только у Worker-процессов
Maxworkers, private, N/A, int	Максимальное число Worker-процессов.
myrank, private, N/A, int	Ранг текущего процесса.
send_to_manager, public, (data: const void*, size: int), void	Выполняет отправку данных от текущего Worker-процесса до Manager-процесса. Если будет вызван на Manager-процессе, то будет выброшено исключение.
send_to_worker, public, (data: const void*, size: int, worker_num: int), void	Выполняет отправку данных от текущего (Manager) процесса до указанного Worker-процесса. Если будет вызван на Worker-процессе, то будет выброшено исключение
manager_recv, public, (data: void*, size: int, worker_num: int), &MPI_Status	Выполняет прием данных на Manager-процессе. Если будет вызвано на Worker-процесса, то будет выброшено исключение
worker_recv, public, (data: void*, size: int), &MPI_Status	Выполняет прием данных на Worker-процессе. Если будет вызвано на Manager-процесса, то будет выброшено исключение
barrier, public, –, void	Устанавливает барьер для всех синхронизаций процессов

Класс Dispatcher предназначен для инкапсуляции групповых операций MPI и для контроля состояния каждого Worker-процесса. Также данный класс содержит метод для запуска вычислительного процесса с учетом оговоренных концепций.

Таблица 3. Некоторые члены класса dispatcher

Название члена, уровень доступа, аргументы, возвращаемый тип	**Описание члена**
workers_state, private, N/A, int*	Массив, хранящий состояния интеркоммуникаторов
dispatcher, public, (comm: communicator), N/A	Конструктор, параметризуется типом шаблона result_type – тип результата вычислений и экземпляром класса communicator
init_worker_supervisor, public, –, void	Инициализация супервизора, инициализация массива состояний Worker-процессов
safe_gather, public, (data_from_worker: const void*, size_of_element: int, count: int, data_manager: void*), void	Отказоустойчивая реализация коллективной операции MPI_Gather
safe_bcast, public, (data_from_manager: const void*, size: int, data_worker: void*), void	Отказоустойчивая реализация коллективной операции MPI_Bcast
safe_scatter, public, (data_from_manager : const void*, size: int, count: int, data_worker: void*), void	Отказоустойчивая реализация коллективной операции MPI_Scatter
start_working_cycle, public, (jobs_array : void*, size_of_job_type: size_t, jobs_cnt: int, processor: array_processor<result_T>&,	Процедура для начала вычислений. Процедура параметризуется объектом-обработчиком данных и указателем на функцию, которая будет осуществлять распределение задач

Название члена, уровень доступа, аргументы, возвращаемый тип	Описание члена
distrib_f_ptr: (void)(jobs_array: job*, jobs_cnt: int, workers_state: const int *, nodes_cnt: size_t)), void	
clean_jobs, private, (job_array : void*, cnt: int, job_size: size_t), void	Процедура для очистки назначений задач

5. Заключение

Как было показано, решение проблем балансировки нагрузки и обеспечения отказоустойчивости возможно несколькими путями, которые должны соотносится с задачей. В данном исследовании продемонстрированы некоторые подходы и их реализации.

Для исследования вопроса load balancing были рассмотрены и реализованы два алгоритма: собственный эвристический алгоритм и улучшенный эвристический алгоритм Ефимкина. Были произведены эксперименты, показывающие различия в эффективности распределения нагрузки.

Далее были рассмотрены различные подходы к организации отказоустойчивости. После сравнительного анализа был выбран подход выживания, реализованный в соответствующем модуле распределённой системы.

На текущий проведено тестирование на наборах данных, имитирующих распределённую работу гидродинамических симуляторов и сопряжения соответствующих секторных моделей с областями счёта разного размера в условиях имитации отказов отдельных узлов.

Литература

1. Костюченко С.В. и др. Алгоритм параллельного моделирования разработки гигантских нефтегазовых месторождений с сопряжением секторных моделей // Материалы V научно-практической конференции «Суперкомпьютерные технологии в нефтегазовой отрасли. Математические методы, программное и аппаратное обеспечение». Москва, 2015.
2. Самборецкий С.С. и др. О распределенной вычислительной системе для компьютерного моделирования нефтегазовых месторождений на основе итерационного сопряжения секторных моделей. // Вестник тюменского государственного университета. Физико-математическое моделирование. Нефть, газ, энергетика. – 2015. – Т.1. – №2(2). – С. 204-213
3. Самборецкий С.С. Проектирование и разработка распределенной системы для итерационного сопряжения секторных гидродинамических моделей. // Суперкомпьютерные дни в России: Труды международной конференции (28-29 сентября 2015г., г. Москва). –М.: Изд-во МГУ, 2015. – С. 641-646
4. Самборецкий С.С. Программная реализация параллельного алгоритма итерационного сопряжения секторных моделей. // Параллельные вычислительные технологии (ПаВТ'2016): труды международной научной конференции (28 марта – 1 апреля 2016 г., г. Архангельск). Челябинск: Издательский центр ЮУрГУ, 2016. 816 с.
5. Ефимкин К.Н. Эвристический алгоритм распределения заданий // Препринты ИПМ им. М.В.Келдыша. 2009. No 42. 16 с.
6. Chien-chung Shen, Wen-hsiang Tsai «A Graph Matching Approach to Optimal task assignment in Distributed computing systems using a Minimax Criterion» // IEEE Transactions on Computers, vol. C- 34,No.3, 1985.
7. B. Daecker. Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction. 2000.
8. Outlook: Fault Tolerance in MPI programs, Barcelona Supercomputing center, Janko Strassburg.

9. Бондаренко А. А., Якобовский М. В. Обеспечение отказоустойчивости высокопроизводительных вычислений с помощью локальных контрольных точек //Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2014. – Т. 3. – №. 3.

The solution of load balancing and ensuring fault tolerance problems in the sector models conjugation distributed system.

N.D. Gibaev[1], M.O. Kashintsev[1], S.S. Samboretskiy[1]

Tyumen State University[1]

Questions of ensuring load balancing and ensuring fault tolerance of calculations in the distributed system realizing algorithm sector hydrodynamic models' conjugation of are considered. The problem definition of parallel calculations is offered during algorithm; two algorithms of distribution of tasks between clusters of system are considered, program realization, results of experiments and some conclusions by their results is shown; various ways of ensuring fault tolerance are considered, the choice of one of strategy is reasonable and its realization for the considered system is shown. The received application can be used for various tasks which solution is found through calculation of parts and conjugation of results.

Keywords: sector modeling, software system, oil and gas deposits, distributed computing, parallel programming, load balancing, fault tolerance.

References

1. Kostyuchenko, S.V., etc. Algorithm of parallel simulation of development of large oil and gas fields with conjugation of sector models//Materials V of scientific and practical conference "Supercomputer technologies in oil and gas branch. Mathematical methods, program and hardware". Moscow, 2015.

2. Samboretskiy S.S., etc. On the distributed computing system for computer model operation of oil and gas fields on the basis of iterative conjugation of sector models. // Bulletin of the Tyumen state university. Physical and mathematical simulation. Oil, gas, power engineering. – 2015. – T.1. – No. 2(2). – Page 204-213

3. Samboretskiy S. S. Design and development of distributed system for iterative conjugation of sector hydrodynamic models.//Russian Supercomputing Days: Proceedings of the International conference (September 28-29, 2015, Moscow, Russia). Moscow State University, 2015. P. 641-646

4. Samboretskiy S.S. Program implementation of parallel algorithm of iterative conjugation of large-scale deposits's sector models. // Parallel computational technologies (PCT) 2016: materials of international scientific conference (March 28th – April 1st, 2016). Chelyabinsk, 2016. 816 p.

5. Efimkin K.N. Heuristic algorithms for jobs scheduling //M.V. Keldysh Institute preprints. 2009. No 42. 16 pages.

6. Chien-chung Shen, Wen-hsiang Tsai «A Graph Matching Approach to Optimal task assignment in Distributed computing systems using a Minimax Criterion» // IEEE Transactions on Computers, vol. C- 34,No.3, 1985.

7. B. Daecker. Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction. 2000.

8. Outlook: Fault Tolerance in MPI programs, Barcelona Supercomputing center, Janko Strassburg.

9. Bondarenko A. A., Yakobovsky M. V. Ensuring fault tolerance of high-performance calculations by means of local reference points//Bulletin of the Southern Ural state university. Series: Calculus mathematics and informatics. – 2014. – T. 3. – No. 3.