ISBN 978-5-91601-111-1

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ НАУКИ
ВЫЧИСЛИТЕЛЬНЫЙ ЦЕНТР ИМ. А.А. ДОРОДНИЦЫНА
РОССИЙСКОЙ АКАДЕМИИ НАУК

А.И. ДОЛМАТОВА, Н.Н. ОЛЕНЕВ

МОДЕЛИРОВАНИЕ ДИНАМИКИ СОЦИАЛЬНОЙ СТРАТИФИКАЦИИ

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ И ПРОГРАММЫ

ВЫЧИСЛИТЕЛЬНЫЙ ЦЕНТР ИМ. А.А.ДОРОДНИЦЫНА
РОССИЙСКОЙ АКАДЕМИИ НАУК
МОСКВА 2014

УДК 519.86

Ответственный редактор

член-корр. РАН И.Г. Поспелов

При прогнозировании экономики России необходимо учитывать социальную стратификацию. Идентификация модели и численное исследование динамики количественного и качественного состава страт проведено с помощью параллельных алгоритмов и программ.

Ключевые слова: модель социальной стратификации экономики, параллельные алгоритмы, идентификация.

Dynamics of social stratification modeling: parallel algorithms and programs

A.I. Dolmatova, N.N. Olenev

It is necessary to take into account the social stratification for forecasting of Russian economy. Modeling the dynamics of quantitative and qualitative composition of strata is performed using different algorithms and methods for the most accurate to describe it.

Keywords: social stratification, modeling, parallel algorithms, identification

Работа выполнена при финансовой поддержке РФФИ (код проекта 13-07-01020).

Рецензенты: В.Г. Жадан,
А.В. Арутюнов

Научное издание

1. Введение

В настоящей работе рассмотрены задачи математического моделирования экономики страны и региона, основанные на разделении общества на социальные группы – страты. В процессе идентификации этих моделей социально-экономических систем используются параллельные алгоритмы и программы для высокоскоростных вычислений на кластерных суперкомпьютерах. На основе идентифицированных моделей могут строиться прогнозы развития экономики и общества для страны и региона.

Математическое моделирование социально-экономических систем является трудоемкой задачей из-за сложности предмета моделирования. Модели динамически развивающихся социально-экономических систем имеют немалое число внешних параметров, большую часть из которых обычно не удается найти напрямую по данным статистики. Использование высокопроизводительных вычислительных методов, в частности параллельных алгоритмов и программ, в косвенной идентификации параметров модели позволяет в значительной степени сократить временные затраты на процесс их расчета.

Постоянное увеличение производительности и мощности производимых компьютеров в настоящее время дает возможность осуществлять более трудоемкие расчеты, учитывать при моделировании большее количество факторов и, следовательно, использовать в практике анализа и прогонозирования более сложные модели. Многоядерные процессоры сейчас имеются практически у каждого пользователя, поэтому запуск вычислительных задач, разделяющихся на потоки, возможен практически в домашних условиях. Наличие доступа к высокопроизводительным кластерным суперкомпьютерным системам позволяет решать задачи грандиозного масштаба.

Для эффективного использования в научной работе суперкомпьютерных систем нужно знать основы параллельного программирования, стандарты и среду разработки параллельных приложений. В данной работе предполагается знакомство читателя с языками программирования C/C++ или Фортран, которые будут использоваться при создании параллельных программ.

Общепринятым стандартом в параллельном программировании на высокопроизводительных кластерных системах в настоящее время является интерфейс передачи сообщений MPI (Message Passing Interface) – библиотека функций, реализованная на языках программирования C/C++ и Фортран. Библиотека функций MPI служит для обмена данными и синхронизации задач между процессами параллельной программы с распределенной памятью [1]. Например, на кластерном суперкомпьютере ВятГУ установлена реализация MPI – LAM [2], осуществленная Ohio Supercomputing Center.

В настоящей работе дано подробное описание работы с библиотекой MPI, приведены практические задания и варианты их решения, даны задания для самостоятельной работы, позволяющие лучше усвоить пройденный материал и на практике закрепить использование тех или иных функций.

Одной из составляющих успеха при использовании параллельных вычислений для получения максимального ускорения является возможность декомпозиции модели на отдельные блоки при минимальном их взаимодействии в процессе моделирования. В используемых задачах математического моделирования экономики с социальной стратификацией процесс декомпозиции максимально прост, поскольку в них присутствует естественный параллелизм. Другими словами, мы имеем дело с системой, состоящей из относительно независимых процессов, расчет по каждому из которых (по каждой страте) можно производить независимо.

Параметры модели определяем параллельно по стратам. При этом большую часть параметров модели невозможно определить непосредственно из данных статистики. Эти параметры определяем верификацией модели по статистическим данным, то есть косвенным образом, сравнивая близость расчетных и статистических временных рядов для макропоказателей.

Современная российская экономика, оказавшаяся в точке бифуркации в результате собственных проблем и международного финансового кризиса, для развития нуждается в модернизации. Дальнейший рост за счет увеличения загрузки старых производственных мощностей уже невозможен, поскольку их

текущая загрузка близка к предельной и может только уменьшаться по мере старения мощностей. Значит, в дальнейшем рост может идти только за счет развития, которое основано на строительстве новых производственных мощностей, сопровождаемом демонтажем старых мощностей. Как осуществить модернизацию и кто ее будет проводить, зависит от сложившихся в обществе отношений между людьми.

В последнее время обсуждение произошедшей дифференциации доходов населения и его жесткой стратификации просочилась из специализированных социологических журналов в популярную прессу, чтобы стать предметом междисциплинарных обсуждений. В частности, указывается на невозможность решения назревших проблем с помощью социальной революции снизу в силу разобщенности экономических интересов страт, которые расщепляют общество поперек классов [3]. При утрате общего языка общество становится управляемым и коррумпированным, поскольку междисциплинарную оценку заменяет финансовый интерес. Также существует опасность самозащиты элиты: «бунт сытых – это война», конец которой вовсе не обязателен.

Для оздоровления общества требуется выработать междисциплинарные критерии оценки, новую общую эстетику, при этом применять принцип братства, который важнее принципа соревнования. Чтобы проверить насколько такое представление о сложившихся отношениях внутри общества соответствует экономической действительности, можно построить математическую модель экономики, основанную на явном описании социальной стратификации с учетом демографических процессов, попробовать идентифицировать эту модель, а в дальнейшем с помощью идентифицированной модели проводить сценарные расчеты.

Для исследования сложившихся производственных и социальных отношений между людьми в настоящей работе предложена модель стратификации российского общества применительно к стране в целом и к Кировской области. В модели страны общество поделено на десять страт, а в модели области на шесть страт, каждая из которых характеризуется собственными особенностями и выполняет определенные экономические функции. В каждой из рассматриваемых моделей страты

дифференцированы не только по уровню дохода, но также и по уровню образования, так что если их изобразить на плоскости с двумя осями, одна из которых показывает уровень дохода, а другая уровень образования, то они образуют своеобразные пирамиды. Описав экономические функции страт, демографическую динамику каждой страты, а также взаимодействие между ними, получим динамику изменения валового регионального продукта. В этой стратификации удалось привязать страты к конкретным отраслям региональной экономики, что подчеркивает несовпадение экономических функций страт и позволяет дать наглядную интерпретацию их взаимодействия. Построенную модель после ее идентификации можно будет использовать для прогноза развития экономики области.

Экономическая ситуация в Кировской области в последние годы складывалась неблагоприятно. На фоне общероссийских кризисов заметен сильный перекос, хотя попыток развития было предпринято довольно много. Успех их отнюдь не проявился с той отдачей, которая ожидалась. Почему это происходит и как именно нужно проводить антикризисные меры?

Какие факторы влияют на успех или провал той или иной экономической программы? Ответы на подобные вопросы получить не так-то просто.

Каждый человек по-своему реагирует на любые внешние воздействия. При этом одни могут бурно проявлять свои эмоции, другие останутся безразличными. Но редкий человек отнесется равнодушно, когда нововведение касается его лично. Причем в группе людей, имеющих примерно одинаковое положение в обществе, равный уровень доходов и образования, мнение на ту или иную проблему зачастую сходится. Деление общества на страты в зависимости от основных факторов, отвечающих за уровень жизни, называется стратификацией. Страта – объединенная группа людей – может быть описана всего несколькими параметрами. Таким образом, изучив общество и поделив его на страты, можно делать прогнозы относительно реакций и, следовательно, эффективности тех или иных методов экономического развития.

Математическая модель российской экономики, описанная в [4], основана непосредственно на учете социальной стратификации. Настоящая работа учитывает специфику региональной экономики. Кроме того, она дополнена прогнозами и сценариями развития экономики по полученным параметрам и начальными выводами по сделанным прогнозам.

Идентификация параметров может быть осуществлена благодаря параллельным вычислениям процессов каждого регионального блока модели экономики. Отдельно перебирая параметры каждого блока по сетке на заданных интервалах изменения параметров, последовательно их уменьшая, можно найти значения всех параметров.

Не определяемые напрямую из статистики параметры модели будем находить косвенным образом, сравнивая выходные временные ряды переменных модели (макропоказателей экономики) с доступными историческими временными рядами [5]. Временные ряды считаются похожими, если они близки как функции времени. В качестве критериев близости расчетного X_t и статистического Y_t временных рядов здесь из-за небольшой длины этих рядов достаточно будет использовать коэффициент корреляции Пирсона

$$P = \frac{\sum_t (X_t - \overline{X})(Y_t - \overline{Y})}{\sqrt{\sum_t (X_t - \overline{X})^2 \cdot \sum_t (Y_t - \overline{Y})^2}},$$

который показывает линейную связь рядов, а также индекс несовпадения Тэйла [6]

$$U = \frac{\sqrt{\sum_t (X_t - Y_t)^2}}{\sqrt{\sum_t (X_t^2 + Y_t^2)}},$$

который, в свою очередь, является аналогом среднеквадратического отклонения и удобен для экономических временных рядов, нередко растущих экспоненциально. Через $\overline{X}$ и

$\overline{Y}$ в [1] и [2] обозначены средние значения для соответствующих рядов. Коэффициент корреляции Пирсона $P \in [-1,1]$, а индекс несовпадения Тэйла $U \in [0,1]$.

Коэффициент корреляции P является мерой силы и направленности линейной связи между сравниваемыми временными рядами, и чем он ближе к +1, тем более схоже поведение этих рядов. При этом следует учитывать, что инфляционная составляющая может преувеличивать линейную связь рядов, поэтому при использовании коэффициента корреляции расчетные и статистические временные ряды показателя нужно брать в реальных величинах. Индекс Тэйла E измеряет несовпадение статистических и расчетных временных рядов Xt и Yt некоего макропоказателя, и чем ближе этот индекс к нулю, тем ближе сравниваемые ряды.

2. Модель динамики социальной стратификации

2.1. Общественное устройство

Организация большей части обществ такова, что их институты неодинаково распределяют блага и ответственность среди разных категорий людей и социальных групп. Поэтому социальный порядок не является нейтральным, а служит достижению целей и интересов одних людей и социальных групп в большей степени, чем других, в чем и проявляется неравенство.

Социальная стратификация — это деление общества на социальные слои (страты) путем объединения различных социальных позиций с примерно одинаковым социальным статусом. Стратификация отражает сложившееся в обществе представление о социальном неравенстве в виде социальной иерархии, где все общество выстроено по вертикали вдоль своей оси по одному или нескольким стратификационным критериям, являющимся показателями социального статуса. В социальной стратификации устанавливается определенная социальная дистанция между людьми (социальными позициями) и фиксируется неравный доступ членов общества к социально значимым дефицитным ресурсам путем установления на границах

разделяющих их социальных фильтров. Социологи используют многомерный подход, в котором классовая стратификация современных обществ проводится по четырем главным критериям:

1). Доход, измеряется в рублях или долларах, которые получает индивид или семья в течение определенного периода времени, скажем, одного месяца или года.

2). Образование, измеряется числом лет обучения в школе или вузе. Имеет значение также и престижность учебного заведения.

3). Власть, измеряется количеством людей, на которых распространяется решение, принимаемое индивидом.

4). Престиж профессии. В отличие от предыдущих показателей, это не объективный, а субъективный показатель. Это уважение статуса, сложившееся в общественном мнении. Измеряется в баллах по шкале профессионального престижа в результате опроса.

Современные исследования факторов, критериев и закономерностей стратификации российского общества позволяют выделить слои, различающиеся как социальным статусом, так и местом в процессе реформирования российского общества. Согласно гипотезе, выдвинутой академиком РАН Т.И. Заславской, российское общество состоит из четырех социальных слоев: верхнего, среднего, базового и нижнего, а также десоциализированного «социального дна» [7].

Верхний слой включает реально правящий слой, к которому относятся элитные и субэлитные группы, занимающие наиболее важные позиции в системе государственного управления, в экономических и силовых структурах. Средний слой является зародышем буржуазии в западном понимании этого термина. В будущем полноценный средний слой в России сформируется на основе социальных групп, образующих сегодня соответствующий протослой. Это мелкие предприниматели, менеджеры средних и небольших предприятий, среднее звено бюрократии, старшие офицеры, наиболее квалифицированные и дееспособные специалисты и рабочие. Базовый социальный слой охватывает более двух третей российского общества. Его представители обладают средним профессионально-квалификационным потенциалом и относительно ограниченным трудовым потенциалом. Сюда

относится основная часть интеллигенции (специалистов), полуинтеллигенция (помощники специалистов), технический персонал, работники массовых профессий торговли и сервиса, большая часть крестьянства. Хотя социальный статус, менталитет, интересы и поведение этих групп различны, их роль в переходном процессе достаточно сходна. Нижний слой замыкает основную, социализированную часть общества. Отличительными чертами его представителей являются низкий деятельностный потенциал и неспособность адаптироваться к жестким социально-экономическим условиям переходного периода. В основном этот слой состоит из пожилых малообразованных, не слишком здоровых и сильных людей, из тех, кто не имеет профессий, а нередко и постоянного занятия, места жительства, безработных, беженцев и вынужденных мигрантов из районов межнациональных конфликтов. Признаками представителей данного слоя являются очень низкий личный и семейный доход, низкий уровень образования, занятие неквалифицированным трудом или отсутствие постоянной работы. «Социальное дно» характеризуется главным образом изолированностью от социальных институтов большого общества, компенсируемой включенностью в специфические криминальные и полукриминальные институты. Представителями социального дна являются преступники и полупреступные элементы, а также опустившиеся люди.

Моделирование региональной экономики возможно на основе разделения общества на страты и выявления закономерностей взаимодействия этих страт путем перераспределения добавленной стоимости с целью выявления узких мест и слабых сторон отдельного региона.

Что касается Кировской области, то здесь можно выделить несколько социальных слоев – страт в зависимости от дохода, образования, власти и престижа профессии. Во-первых, это элита, имеющая наибольший доход и наивысшую власть в обществе. Сюда относятся высшие чиновники, руководители муниципальных и государственных предприятий, их заместители. Также это крупные бизнесмены и другие значимые в области фигуры, то есть, топ-менеджеры. Ко второму слою относятся все работники торговли, сервиса, в том числе финансового. Эта страта

отличается довольно высокими доходами и средним уровнем образованности. Их можно назвать средним классом. Следующий слой – работники промышленности, транспорта и строительства. Уровень дохода в данной страте относительно невысок, как и уровень образования. Это самая многочисленная страта, являющаяся базовой в обычных классификациях. К четвертому слою, интеллигенции, можно отнести работников культуры, образования и науки. Их уровень дохода достаточно низок, но зато в данной страте присутствует довольно высокий уровень образованности. И последний в социализированной части общества пятый слой – это работники сельского хозяйства. Их доходы очень низки, образование также находится на низком уровне. Последняя шестая страта включает социальное дно. Таким образом, данная классификация по уровню дохода в порядке снижения его уровня отображает основные классы общества, упрощенно выявленные на основе существующих реально в Кировской области.

Таким образом, описанное выше деление позволяет выделить шесть страт в социальной структуре Кировской области, пять из которых вовлечены в учитываемую статистикой экономическую деятельность. Что касается экономики, то здесь также можно выделить основные, ведущие отрасли. Это сельское хозяйство, промышленность (пищевая, обрабатывающая, лесная), торговля и сервис (включая финансовый), образовательная сфера (в нее объединены три сферы – наука, культура и образование), а также сфера управления.

Для моделирования региональной экономики сопоставим пять верхних социальных страт и пять выделенных экономических отраслей. В сфере управления действуют главным образом члены элитарной страты. Их труд не отличается какими-либо усилиями с физической стороны, основная деятельность умственная. При этом в данной сфере не создается продукт производства. Сферу торговли и сервиса представляет средний класс, основу которого составляют соответствующие работники. Здесь также не производится продукт, но присутствует высокая добавочная стоимость. Базовая страта, рабочие промышленной сферы, транспорта и строительства, является основным источником производственного продукта, то есть в ней создают продукт, на

который в более высоких по уровню дохода стратах наращивается добавленная стоимость. Эта страта преимущественно связана с пищевой и обрабатывающей промышленностью, а также с лесным хозяйством. Интеллигенция работает в образовательной сфере (в науке, культуре и образовании), где невысокий уровень дохода соответствует невысокой добавленной стоимости и нет производственного продукта. Труд работников сельского хозяйства характеризуется высокими физическими затратами и низкой отдачей в плане добавленной стоимости. Они также в процессе труда создают основной продукт, который в дальнейшем перераспределяется среди других классов и слоев населения Кировской области.

Между описанными стратами и сферами экономики существует движение материальных и финансовых потоков. Основной продукт создается в промышленности и сельском хозяйстве. Соответственно между базовой стратой и стратой сельских рабочих происходит взаимный обмен произведенными товарами и финансовыми средствами. Пищевая промышленность использует результаты сельскохозяйственной деятельности для дальнейшей переработки, а сельское хозяйство использует промышленные объекты и инструменты в процессе производства. При этом продукция промышленной сферы стоит дороже, чем сельскохозяйственной, вследствие чего у страты крестьян добавленная стоимость наименьшая, как, следовательно, и уровень дохода. Сфера промышленности дает свою продукцию также каждому из слоев как в виде продуктов питания (пищевая промышленность), так и в виде промышленного оборудования, в результате чего образуется добавочная стоимость, часть из которой уходит на заработную плату. Непосредственное использование продукции каждой сферы другими очевидно в виде потребления товаров и услуг соответствующих отраслей.

Учет в нашей работе добавленной стоимости требует описания дополнительных взаимодействий некоторых сфер экономики. Так, продукция страты крестьян поступает в сферу торговли, где и создается добавленная стоимость, часть которой идет на зарплату работников торговли. Услуги, оказываемые средним классом, потребляются каждой из страт, то есть каждая страта тратит часть дохода на данную сферу, которая также

создает добавочную стоимость. Это пример того, что взаимодействие страт двусторонне. Каждая из четырех страт нижнего уровня выплачивает налоги на существование элиты, а также прибыль и взятки для обеспечения существования собственной сферы. При этом каждая страта платит свою долю налогов. Финансовые потоки за товары промышленного производства идут из сферы торговли в промышленную сферу. Последние поступают в виде материальных потоков от базового класса среднему. Таким образом, средний класс получает в качестве вознаграждения за свой труд часть добавленной в торговле стоимости. Такова упрощенная схема экономического взаимодействия социальных страт.

2.2. Описание модели

Модель региональной экономики строится на основе выделения социальных страт, связанных с определенными секторами региональной экономики. Предполагается, что каждая страта производит добавленную стоимость в соответствующем секторе экономики, а взаимодействие страт и секторов экономики сводится в модели к перераспределению добавленной стоимости.

Для простоты предполагаем, что инфляция описывается заданным извне региона дефлятором валового регионального продукта p(t), а все остальные макропоказатели фиксированного года t описываем после дефлирования на этот единый в модели индекс цен.

Стоимость $y_i(t)$, добавленная в году t к региональному валовому продукту стратой i (i = 1,…,6), определяется численностью занятых в экономике людей страты $L_i^E(t)$ и их производительностью труда $u_i(t)$ – нормой выхода добавленной стоимости на единицу живого труда:

$$y_i(t) = u_i(t) L_i^E(t).$$

Производительность труда $u_i(t)$ в страте i зависит от среднего оборотного капитала $k_i(t)$ в ней и среднего уровня образования $o_i(t)$ [4]:

$$u_i(t) = u_i(k_i(t), o_i(t)).$$

Численность занятых в экономике людей $L_i^E(t)$ страты i определяется ее человеческим потенциалом – численностью людей трудоспособного возраста $L_i^T(t)$, принадлежащим данной страте:

$$L_i^E(t) = v_i(t) L_i^T(t),\ L_i^T(t) = \sum_{a=18}^{65} l_i(t, a).$$

Здесь $v_i(t) \in (0,1)$ – доля трудоспособных лиц, занятых в экономике, а $l_i(t, a)$ – численность населения возраста a в i-й страте.

Пусть q_i – доля теневых доходов в i-й страте; n_i – уровень налогообложения в ней (налогообложение доходов разных страт отличается в силу принятой в России регрессионной шкалы); m_i – уровень собираемости штрафов за уклонение от налогов, тогда $q_i y_i(t)$ – теневые доходы; $(1 - q_i)y_i(t)$ – легальные доходы; $n_i (1 - q_i)y_i(t)$ – налоговые отчисления, а $m_i q_i y_i(t)$ – штрафные санкции с i-й страты. Налоговые отчисления и штрафные санкции поступают в консолидированный бюджет региона, образуя доходы бюджета, которыми распоряжается элита – страта 1. Элита осуществляет расходы бюджета, осуществляя трансферты во все страты. Доходы консолидированного бюджета D(t) и его расходы R(t) определяются соотношениями

$$D(t) = \sum_{i=1}^{6} \left(n_i - (n_i - m_i) q_i\right) y_i(t),\ R(t) = \sum_{i=1}^{6} b_i D(t),$$

где b_i – доля доходов бюджета, идущая страте i. Заметим, что $\sum b_i = 1$ в редком случае сбалансированного консолидированного регионального бюджета.

Для замыкания модели формирования доходов страт считаем, что коррупционные доходы $C_1(t)$ части элиты пропорциональны численности занятых в экономике лиц первой

страты $L_1^E(t)$, а коррупционные расходы других страт ограничены их теневыми доходами.

$$C_1(t) = r_1(t)L_1^E(t) + \sum_{i=2}^{6} \min\left(r_i(t)L_1^E(t), (1-m_i)\, q_i y_i\right).$$

Здесь $r_i(t)$ – норма прибыли на одного занятого в экономике члена первого слоя, включая взятки, для страты i.

Теперь можно определить реальные располагаемые доходы страт $d_i(t)$ после налогообложения, штрафных санкций и бюджетных трансфертов:

$$d_1(t) = C_1(t) - r_1(t)L_1^E(t) + b_1 D(t) + \left(1 - n_1 + (n_1 - m_1)q_1\right)y_1(t),$$

$$d_i(t) = b_i D(t) + \left(1 - n_i + (n_i - m_i)q_i\right)y_i(t), \ (i = 2,...,6).$$

Средние доходы страт $\delta_i(t) = d_i(t)/L_i(t)$ определяют не только их положение в обществе, но также их демографические показатели [4] и показатели $k_i(t)$, $o_i(t)$ производительности труда. Последние представлены здесь соотношениями

$$k_i(t) = \kappa_i \delta_i(t), \ o_i(t) = \rho_i \sum_{a=0}^{A_i} \delta_i(t-a),$$

где κ_i, ρ_i – положительные константы, A_i – средний возраст обучения в данной i-й страте. При этом в качестве функции (2) возьмем производственную функцию леонтьевского типа: $u_i(k_i(t), o_i(t)) = \min\left(k_i(t), o_i(t)\right)$.

Динамику населения в каждой страте описываем подобно работе [4]. В каждый момент времени t численность населения

страты i, имеющей возраст a, определяется ее численностью в предыдущий момент времени и силой смертности $\mu_i(t,a)$.

$$l_i(t,a) = (1-\mu_i(t,a))l_i(t-1,a-1),$$

Зависимость силы смертности от социальных и биологических параметров будем описывать по формуле, подобной формуле Гомперца-Мейкема:

$$\mu_i(t,a) = \min(1, \sigma_i(t) + \eta\exp(\gamma a)),$$

где положительные параметры η,γ описывают биологическую составляющую смертности, а $\sigma_i(t)$ – социальную. Социальная составляющая смертности зависит от страты и уровня ее текущих доходов. Если уровень доходов превысит некий минимальный уровень δ_i^0, то социальная составляющая смертности начинает снижаться. Опишем здесь эту зависимость следующей функцией [4]:

$$\sigma_i(t) = \xi_i\exp\left(-\lambda_i\left(\delta_i(t)-\delta_i^0\right)_+\right),$$

где ξ_i – максимальный уровень социальной смертности. Здесь и далее используется обозначение $x_+ = \max(0,x)$. Введенное ограничение показывает, что социальная составляющая силы смертности падает только тогда, когда этот уровень превысит заданную величину δ_i^0.

Численность новорожденных в страте i определяется коэффициентами рождаемости, которые также зависят от уровня доходов. Здесь мы для простоты описания эту зависимость представим с помощью одного единственного коэффициента рождаемости и численностью населения в детородном возрасте:

$$l_i(t,0) = \beta_i(t) \sum_{a=15}^{50} l_i(t,a),$$

где функция рождаемости зависит от текущего среднего дохода и некоторого уровня дохода δ_i^1 :

$$\beta_i(t) = \chi_i \exp\left(\theta_i \left(\delta_i(t) - \delta_i^1\right)_+\right).$$

Перетоки населения между стратами, связанные с получением образования, в модели мы пока не учитываем. Рассматриваем такую крайнюю ситуацию, когда все социальные лифты закрыты вообще.

На основании полученных данных по имеющимся материально-сервисным и соответствующим им финансовым потокам определяем валовой региональный продукт (ВРП) Кировской области как сумму первоначальных доходов отдельных ее отраслей – страт, исключив при этом добавленную стоимость по нерыночным коллективным услугам (оборона, государственное управление):

$$Y(t) = \sum_{i=1}^{6} y_i(t).$$

В качестве основной задачи, поставленной в исследовании, выделим определение параметров модели и определение реального ВРП области, включающего не только официальные статистические данные, но и теневой доход каждой сферы.

2.3. Программные коды параллельной программы

Параметры каждой страты идентифицировались параллельно по одинаковому набору параметров. Перетоки

населения между стратами рассматривались как внешние заданные функции времени.

Каждый из параметров модели может изменяться заданное количество раз в определенном диапазоне. Нижнюю границу диапазонов изменения каждого параметра будем хранить в массиве limits1, верхнюю – в limits2, число изменений по каждому параметру будем хранить в массиве N. Таким образом, общее число итераций по всем параметрам для получения одного решения задачи (NN) вычисляется следующим программным кодом:

```
unsigned int NN = 1; // всего итераций
for (int p = 0; p < NUM_OF_PARAMS; p++) {
NN *= N[p];
step[p] = N[p] == 1 ? 0.0 : (limits2[p] - limits1[p])/ (N[p] - 1);
        }
```

Этот же программный код вычисляет величину шага изменения каждого параметра step.

Для удобства параллелизации процесса перебора параметров вместо пяти циклов будем работать с одним, с общим числом итераций NN. При каждой итерации этого общего цикла рассчитываются индексы виртуальных циклов при помощи процедуры calcIndexes(), которая выглядит следующим образом:

```
void calcIndexes(unsigned int iii, int *N, int *i) {
for (int j = NUM_OF_PARAMS-1; j >= 0; j--) {
i[j] = iii % N[j];
iii = (iii - i[j]) / N[j];
}
}
```

Статистические значения ВНП будем хранить в массиве Y_, а расчетные значения для каждого набора параметров – в массиве Y. В качестве критериев близости расчетного и статистического временных рядов используем коэффициент близости U(X, Y) = 1 - E(X, Y), где E(X. Y) – индекс Тэйла [6].

Расчет коэффициента близости производится при помощи функции bliz():

```
float bliz(float *x, float *y, int n) {
        float d1 = 0.0, d2 = 0.0, d3 = 0.0;
        for (int p = 0; p < n; p++) {
                d1 += pow(x[p] - y[p], 2);
                d2 += pow(x[p], 2);
                d3 += pow(y[p], 2);
        }
        return 1.0 - sqrt(d1 / (d2 + d3));
}
```

При расчетах используются несколько констант, которые предварительно идентифицированы из статистических данных. Константы, необходимые для расчетов, хранятся в массиве fCONST.

Основной вычислительный цикл выглядит следующим образом:

```
for (unsigned int j = jStart; j < jEnd; j++) {
        calcIndexes(j, N, i);

        for (int p = 0; p < NUM_OF_PARAMS; p++) {
                par[p] = limits1[p] + i[p] * step[p];
        }

        float fkLast, y[YEARS], Y[YEARS];
        for (int t = 0; t < YEARS; t++) {
                float fk;
                if (t == 0) {
                        fk = 1;
                } else {
                        fk = fkLast - par[3] * fkLast + par[4] *
fCONST[3] * y[t-1] / fp(t-1);
                }
                fkLast = fk;
y[t] = pow(par[0] * pow(fl(t), -par[1]) +
```

```
(1 - par[0]) * pow(fk, - par[1]), -par[2] / par[1]);
                    Y[t] = y[t] * Y_[0];
            }

            float F = bliz(Y_, Y, YEARS);
            if (/*F < 0.990 && */F > bestF) {
                  // найдено решение, лучшее, чем предыдущее
                     copyArray(par, bestPar, NUM_OF_PARAMS);
                     copyArray(Y, bestY, YEARS);
                     bestF = F;
                     // как менялось лучшее решение...
                     fprintf(perProcLog, "%d %f: ", j, F);
                     printFloatArray(par, NUM_OF_PARAMS,
                              perProcLog);

            }
      }

      float fp(int t) {
            return fCONST[0] + (1 - fCONST[0]) * (1 + t) *
exp(- fCONST[1] * t);
      }

      float fl(int t) {
            return exp(fCONST[2] * t);
      }
```

В каждом вычислительном процессе переменные jStart и jEnd вычисляются автоматически согласно номеру процесса:

```
int slice = (int) (NN / numproc);
            int jStart = procind * slice;
            int jEnd = (procind + 1) * slice;
```

При завершении расчета каждый процесс, отличный от нулевого, посылает нулевому процессу результаты своих расчетов, свое локальное лучшее решение:

```
MPI_Send(&bestF, 1, MPI_FLOAT, 0, 77, MPI_COMM_WORLD);
MPI_Send(bestPar, NUM_OF_PARAMS, MPI_FLOAT, 0, 77,
        MPI_COMM_WORLD);
MPI_Send(bestY, YEARS, MPI_FLOAT, 0, 77,
      MPI_COMM_WORLD);
```

Нулевой процесс в свою очередь принимает от остальных процессов решения и выбирает из них наилучшее, отображая его на экран.

2.4. Результаты идентификации модели и выводы

В качестве основной задачи, поставленной в исследовании, выделим создание сценариев развития экономической ситуации в Кировской области.

В модели необходимо идентифицировать огромное количество параметров. А именно: ν_i, r_i, q_i, b_i, k_i, ρ_i, n_i, m_i при i = 1..6. Это уже сорок восемь параметров. Необходимо так же идентифицировать параметры, связанные с демографией. А именно: η, γ, ξ_i , λ_i , δ_i^0 $\beta_i(t)$, χ_i , θ_i , δ_i^1 при i = 1..6. Кроме того, из-за невозможности определения численности населения в каждой страте необходимо ввести еще шесть параметров, определяющих долю каждой страты в общей численности населения. Таким образом, общее количество параметров больше сотни.

Очевидно, что при идентификации модели необходимы значительные упрощения для сокращения количества параметров в несколько раз.

Демографические параметры возьмем те, что были получены в [4].

Уровень штрафных санкций предполагаем равным нулю (m = 0). Уровни налогообложения всех страт будем считать одинаковыми ($n_i = n$). Функции ν_i и r_i полагаем константами. Таким образом, остается идентифицировать тридцать семь параметров: ν_i, r_i, q_i, b_i, k_i, ρ_i, n.

Однако вычислительных мощностей персонального компьютера, которые используется для вычислений, недостаточно, необходимо дальнейшее сокращение параметров.

Следует выбрать параметры, которые с высокой мерой точности можно определить, опираясь на расчеты, полученные для России [4].

Параметр k_i чрезвычайно сильно влияет на модель. Его мы и будем определять, перебирая все возможные значения на определенном интервале. Также необходимо добавить к этим шести параметрам, уровень налогообложения n. Таким образом, необходимо перебрать семь параметров.

При этом шаг перебора установим равным 0,05, а затем уменьшим его на 0,02, уменьшив область перебора вокруг получившегося значения.

В качестве меры близости будем использовать коэффициент несовпадения Тэйла для значений ВРП, полученных из расчета и статистики ВРП Кировской области.

$$T = \sqrt{\frac{\sum_{t=2000}^{2009}\left(X_j(t) - Y_j(t)\right)^2}{\sum_{t=2000}^{2009} X_j^2(t) + \sum_{t=2000}^{2009} Y_j^2(t)}},$$

где X(t) – рассчитанный по модели ряд; Y(t) – статистический ряд.

В результате расчетов получены следующие значения: $k_1 = 0{,}62$, $k_2 = 1{,}18$, $k_3 = 1{,}18$, $k_4 = 0{,}98$, $k_5 = 1{,}04$, $k_6 = 0{,}94$. При этом уровень налогообложения n = 0,28.

При данных показателях коэффициент несовпадения Тэйла равен 0,0367, что говорит о высокой степени совпадения рядов.

Коэффициент несовпадения Тэйла для динамики численности населения равен 0,000002, что говорит о чрезвычайно сильной связи между рядами.

Рассчитанный ВРП довольно близок к статистическому на протяжении всех периодов, кроме последнего. В результате кризиса ВРП резко упал, что никак не могло быть спрогнозировано моделью.

Коэффициент несовпадения Тэйла равен 0,036 и средняя процентная ошибка составляет 3%.

По данным расчетов получилось, что доходы элиты после небольшого спада начинают быстро расти. Доходы всех остальных

страт постепенно растут, а доходы в сельском хозяйстве и пенсии постепенно выходят на уровень зарплат бюджетников.

2.5. Сценарии развития экономической ситуации

Сценарий 1. Все параметры останутся те же. Тогда построим прогноз до 2020 г.

В результате проделанных операций получаем, что население региона продолжает снижаться, однако темпы падения значительно уменьшаются. При этом ВРП продолжает расти, но темпы роста также снижаются.

Доходы элиты увеличиваются нарастающими темпами. Доходы других страт также растут, но значительно медленнее. Стоит выделить более высокий рост доходов в пятой страте (сельское хозяйство и пенсионеры). Такой рост говорит о том, что при данных параметрах модель показывает развитие региона при высоком уровне сельского хозяйства.

Можно рассмотреть также другие сценарии развития региона.

Сценарий 2. Предположим, что в дальнейшем (начиная с 2010 г.) резко увеличивается уровень налогообложения до показателя n = 0,4.

Тогда по результатам прогнозов получим, что с ростом налогов доходы чиновников резко растут, однако после нескольких лет устанавливаются на одном уровне в связи с уменьшением поступлений в бюджет от второй и третьей страт.

Сильнее всего рост налогов сказывается на промышленности и торговле, так как дотации из бюджета в эти отрасли малы. Однако на доходы «бюджетников» и «андеграунда» рост налогов влияет положительно. Общий ВРП при этом заметно падает.

Таким образом, можно говорить о том, что рост налогов должен приводить к временному росту доходов отдельных страт, но в целом на экономике сказывается отрицательно.

Сценарий 3. Предположим, что в экономике с 2010 г. происходит широкая поддержка сельского хозяйства государством.

На доходах элиты это почти не сказывается, так как коррупционные поступления от данной страты небольшие. В

результате поддержки сельского хозяйства доходы соответствующей страты значительно растут. И в целом вливание денег в сельское хозяйство приводит к значительному росту ВРП.

3. Алгоритмы параллельных программ

3.1. Параллельное программирование в MPI

Изложение данной главы следует электронному курсу, представленному в Интернет на сайте

http://www.ccas.ru/mmes/educat/lab04k/01/basics.html.

Хорошее введение в MPI содержится также в [8].

3.1.1. Обзор MPI

3.1.1.1. Что такое MPI

MPI – это библиотека передачи сообщений, собрание функций на C/C++ (или подпрограмм в Фортране, которые, зная MPI для C/C++, легко изучить самостоятельно), облегчающих коммуникацию (обмен данными и синхронизацию задач) между процессами параллельной программы с распределенной памятью. Акроним (сокращение по первым буквам) установлен для Message Passing Interface (интерфейс передачи сообщений). MPI является на данный момент фактическим стандартом и самой развитой переносимой библиотекой параллельного программирования с передачей сообщений.

MPI не является формальным стандартом, подобным тем, что выпускают организации стандартизации, такие как Госкомстандарт РФ, ANSI или ISO. Вместо этого он является «стандартом по консенсусу», спроектированным на открытом форуме, который включал крупных поставщиков компьютеров, исследователей, разработчиков библиотек программ и пользователей, представляющих более 40 организаций. Такое широкое участие в его развитии гарантировало быстрое

превращение MPI в широко используемый стандарт для написания параллельных программ передачи сообщений.

«Стандарт» MPI был введен MPI-форумом в мае 1994 г. и обновлен в июне 1995 г. Документ, который определяет его, озаглавлен «MPI: A Message-Passing Standard», опубликован университетом Тэннеси и доступен по World Wide Web в Argonne National Lab. Если вы еще не знакомы с MPI, то, возможно, вы захотите распечатать перевод этого документа на русский язык и использовать его для получения справок о синтаксисе функций MPI, которые данный курс не может охватить полностью за исключением иллюстрации частных случаев.

MPI 2 производит расширения к стандарту MPI передачи сообщений. Эти усилия не изменили MPI, они расширили применение MPI на следующие сферы.

Управление динамическими процессами.

Ассимметричные операции.

Параллельный ввод/вывод (I/O).

Привязка к C++ и ФОРТРАН 90.

Внешние интерфейсы.

Расширенные коллективные коммуникации.

Расширения реального времени.

Другие сферы.

MPI 2 был завершен в июле 1997 г.

MPI предлагает переносимость, стандартизацию, эффективную работу, функциональность и имеет ряд высококачественных реализаций.

Стандартизация

MPI стандартизован на многих уровнях. Например, поскольку синтакс стандартен, вы можете положиться на ваш MPI код при запуске в любой реализации MPI, работающей на вашей архитектуре. Поскольку функциональное поведение вызовов MPI довольно стандартизовано, нет нужды беспокоиться о том, какая реализация MPI установлена сейчас на вашей машине; ваши вызовы MPI должны вести себя одинаково независимо от реализации. Эффективность работы тем не менее слегка меняется в зависимости от реализации.

Переносимость

В быстро изменяющемся окружении высоко производительных компьютеров и технологий коммуникации, переносимость (мобильность) почти всегда важна. Кто захочет развивать программу, которая может выполняться только на одной машине или только с дополнительными затратами труда на других машинах? Все системы массивной параллельной обработки обеспечивают своего рода библиотеку передачи сообщений, которая точно определена аппаратными средствами используемой ЭВМ. Они обеспечивают прекрасную эффективность работы, но прикладной код, написанный для одной платформы не может быть легко перенесен на другую.

MPI позволяет писать портативные программы, которые все еще используют в своих интересах спецификации аппаратных средств ЭВМ и программного обеспечения, предлагаемого поставщиками. К счастью, эти заботы в основном берут на себя запросы MPI, потому что конструкторы настроили эти вызовы на основные аппаратные средства ЭВМ и окружающую среду программного обеспечения.

Эффективность работы

Множество внешних инструментов, включая PVM, Express и P4, пытались обеспечить стандартизованное окружение для параллельных вычислений, тем не менее ни одна из этих попыток не показала такой высокой эффективности работы, как MPI.

Богатство возможностей

MPI имеет намного больше одной качественной реализации. Эти реализации обеспечивают асинхронную коммуникацию, эффективное управление буфером сообщения, эффективные группы, и богатые функциональные возможности. MPI включает большой набор коллективных операций коммуникации, виртуальных топологий и различных способов коммуникации, и, кроме того, MPI поддерживает библиотеки и неоднородные сети.

Имеющиеся в настоящее время реализации включают

– MPICH: реализация Argonne National Lab и Mississippi State University,

- LAM: реализация Ohio Supercomputer Center,
- MPI/Pro: реализация MPI Software Technology,
- IBM MPI: реализация фирмы IBM для кластерных рабочих станций SP и RS/6000,
- CHIMP: реализация Edinburgh Parallel Computing Centre,
- UNIFY: реализация Mississippi State University.

MPI имеется в наличии на многих массивно параллельных системах.

ВЦ РАН и МСЦ используют MPICH-реализацию MPI, последнюю версию которой можно свободно скачать с сайта Argonne National Lab.

3.1.1.2. Как использовать MPI

Если у вас уже есть последовательная версия программы и вы собираетесь ее модифицировать, используя MPI, перед распараллеливанием убедитесь, что ваша последовательная версия безукоризненно отлажена. После этого добавьте вызовы функций MPI в соответствующие места вашей программы.

Если вы пишете программу MPI с чистого листа и написать сначала последовательную программу (без вызовов MPI) не составляет для вас большого труда, сделайте это. Повторим, идентификация и удаление непараллельных ошибок вначале намного облегчит отлаживание параллельной программы. Проектируйте ваш параллельный алгоритм, используя в своих интересах любой параллелизм, свойственный вашему последовательному коду, например большие массивы, которые можно разбить на подзадачи и обрабатывать независимо.

Отлаживая параллельную версию удостоверьтесь сначала, что запуски вашей программы успешны на нескольких узлах. Затем постепенно увеличивайте число узлов, например, от 2 до 4, затем 8, и т.д. Таким путем вы не будете тратить впустую много машинного времени на дополнительные ошибки.

3.1.2. Программы MPI

В этом параграфе дано введение к простой программе с MPI. Это нужно, чтобы дать наглядное представление об относящихся к делу вопросах, к которым можно потом вернуться, если у вас возникнут вопросы по таким вещам, как

- в каком порядке следует делать эти вызовы?
- как выглядит список параметров?

Программа сама по себе представляет широко известную программу **Hello, world**, поэтому мы не будем касаться понимания цели алгоритма, лучше мы сфокусируемся полностью на механизме осуществления параллельной версии этой чрезвычайно простой задачи

3.1.2.1. Функции MPI

Во-первых, рассмотрим форматы фактических вызовов, используемых MPI.

Привязка к языку C

Для C, общий формат имеет вид

```
rc = MPI_Xxxxx(parameter, ... )
```

Заметим, что регистр здесь важен. Например, MPI должно быть заглавным, так же как и первая буква после подчеркивания. Все последующие символы должны быть в нижнем регистре. Переменная **rc** есть некий код возврата, имеющий целый тип. В случае успеха он устанавливается в MPI_SUCCESS.

Программа на C должна включать файл «mpi.h». Он содержит определения для констант и функций MPI.

Привязка к языку Фортран

В случае языка Фортран общая форма выглядит так

```
Call MPI_XXXXX(parameter,..., ierror)
```

Заметим, что здесь регистр не важен. Поэтому, эквивалентной формой будет

```
call mpi_xxxxx(parameter,..., ierror)
```

В отличие от C, в которой функции MPI возвращают код ошибки, Фортран-версия подпрограмм MPI обычно имеет один дополнительный параметр в списке вызова ierror, который равен

коду возврата. В случае успешного вызова ierror устанавливается в значение MPI_SUCCESS.

Программы Фортран должны обычно включать 'mpif.h' (для компилятора Compaq 'mpif90.h') Этот файл определения для констант и функций MPI.

Для обеих привязок и С, и Фортрана исключением к приведенным выше функциям есть функции времени (MPI_Wtime и MPI_Wtick), которые являются функциями как С, так и Фортрана и возвращают действительные числа с двойной точностью.

В дальнейшем изложении мы **ограничимся привязкой только к языку С**. Изучив привязку MPI к С, желающие, как уже говорилось, могут самостоятельно освоить соответствующие подпрограммы Фортрана.

Основная схема программы MPI подчиняется следующим общим шагам:

1. Инициализация для коммуникаций.

2. Коммуникации распределения данных по процессам.

3. Выход «чистым» способом из системы передачи сообщений по завершении коммуникаций.

MPI имеет свыше ста двадцати пяти функций. Тем не менее начинающий программист обычно может иметь дело только с шестью функциями, которые иллюстрируют нашу простейшую программу и указаны ниже.

Три функции инициализации для коммуникаций:

MPI_Init инициализирует окружение MPI,

MPI_Comm_size возвращает число процессов,

MPI_Comm_rank возвращает номер текущего процесса (ранг = номер по-порядку).

Две функции коммуникации распределения данных по процессам:

MPI_Send отправляет сообщение,

MPI_Recv получает сообщение.

Шестая функция для выхода из системы передачи сообщений:

MPI_Finalize.

3.1.2.2. Пример MPI-программы

Покажем эти шесть базовых вызовов функций MPI в следующем коде на языке C.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
main(int argc, char **argv
{
  char message[20];
  int  i, rank, size, type = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0)
  {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 14, MPI_CHAR, i, type,
MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, type,
MPI_COMM_WORLD, &status);
  printf( "Message from process = %d : %.14s\n",
rank,message);
  MPI_Finalize();
}
```

Покажем синтаксис и дадим подробное описание используемых функций.

int MPI_Init(int *argc, char ***argv)

MPI_Init должна быть первой функцией MPI, которую вы вызываете в каждом процессе. Ее можно вызвать только один раз. Она устанавливает окружение, необходимое для запуска MPI. Это окружение может быть приспособлено для любых флагов запуска MPI, обеспечиваемых реализацией MPI.

int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_Comm_size возвращает число процессов внутри коммуникатора. Коммуникатор – это механизм MPI для создания отдельных коммуникационных областей («вселенных»). Наша пробная программа использует предопределенный «мировой коммуникатор» MPI_COMM_WORLD, который включает все ваши процессы. MPI может определить число процессов, поскольку вы определяете это, когда задействуете команду mpirun, используемую для запуска программ MPI.

int MPI_Comm_rank(MPI_Comm comm, int *rank)

Ранг (номер по порядку) используется, чтобы специфицировать отдельный процесс. Ранг является целым в интервале от 0 до size - 1, где size есть число процессов, которое возвращает функция MPI_Comm_size. MPI_Comm_rank, возвращает ранг вызываемого процесса в точно определенном коммуникаторе.

Часто для процесса необходимо знать его собственный ранг. Например, вам может захотеться разбить вычислительную работу в цикле по всем вашим процессам, так чтобы каждый процесс выполнял подмножество в исходном диапазоне цикла. Один способ сделать это – для каждого процесса использовать его ранг для вычисления его диапазона в индексах цикла.

Позднее, когда мы изучим коммуникаторы, вы увидите, что процесс может принадлежать более чем одному коммуникатору и может иметь различный ранг для каждого коммуникатора. Но сейчас будем полагать, что мы имеем дело только с предопределенным коммуникатором MPI_COMM_WORLD, который состоит из всех ваших процессов, как проиллюстрировано в выбранном примере.

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

В MPI существует множество оттенков для отправок и получений. Это одна из причин того, что MPI снабжена более чем 125 функциями. В нашем основном множестве из шести вызовов мы рассмотрим только один тип отправки и один тип получения.

MPI_Send является блокирующей отправкой. Это означает, что вызов не возвращает управление в вашу программу до тех пор, пока все данные не будут скопированы из расположения, которое вы точно определили в листе параметров. Из-за этого вы можете изменить данные после вызова, что не отразится на оригинальном сообщении. Однако следует отметить существование неблокирующих отправлений, в которых это не так.

Параметры:
buf – начало буфера, содержащего данные, которые будут отосланы (для C это адрес),
count – число элементов, которые будут отосланы (не байт),
datatype – тип данных,
dest – ранг процесса, места назначения, для сообщения,
tag – произвольное число, которое можно использовать для отличия от других сообщений,
comm – (определенный) коммуникатор,
(ierror – возвращаемый функцией код ошибки).

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Передача сообщения в MPI-1 требует двухстороннюю коммуникацию (связь). Односторонняя коммуникация является одной из черт, добавленной в MPI-2. Каждый раз, когда один процесс отправляет сообщение, другой процесс должен явно получить сообщение. Итак, для каждого места в вашем приложении, в котором вы вызываете функцию MPI отправки, должно быть соответствующее место, где вы вызываете функцию

MPI получения. Следует проявить заботу, чтобы увериться в том, что параметры отправки и получения сочетаются.

Подобно MPI_Send, MPI_Recv является блокирующей. Это означает, что вызов не возвращает управление в вашу программу до тех пор пока все полученные данные не будут запомнены в переменной(ых), которую(ые) вы точно определили в листе параметров. Из-за этого вы можете использовать эти данные после вызова и быть уверены, что все они здесь. Существуют неблокирующие получения, в которых это не так.

Параметры:
buf – начало буфера, в котором входящие данные должны быть запомнены (для C это адрес),
count – число элементов (не байт) в вашем буфере получателя,
datatype – тип данных,
source – ранг процесса, от которого данные будут приняты (он может быть любым (джокером, дикой картой) при задании параметром MPI_ANY_SOURCE),
tag – произвольное число, которое можно использовать для отличия от других сообщений (оно может быть любым (дикой картой) при задании параметром MPI_ANY_TAG),
comm – (определенный) коммуникатор,
status – массив или структура возвращаемой информации (если вы определяете дикой картой источник source или тег tag, статус скажет вам действительный ранг или тег для полученного сообщения).

При этом ierror – возвращаемый функцией код ошибки.

int MPI_Finalize()

Последним вызовом, который вам надлежит сделать в каждом процессе, является вызов MPI_Finalize. Это помогает увериться в том, что MPI выходит чисто и что ничего не отложено на узлах, когда вы это делаете. Для того чтобы все приложение завершить чисто, все ожидающие (неблокирующие) коммуникации следует завершить до вызова MPI_Finalize.

Отметим, что ваш код может продолжать выполнение после вызова MPI_Finalize, однако он не может больше вызывать функции MPI.

Резюмируя эту программу, можно сказать: это код SPMD (Single Program, Multiple Data stream), так что копии этой программы выполняются на множестве процессоров. Каждый процесс инициализирует себя в MPI (MPI_Init), определяет число процессов (MPI_Comm_size) и узнает его ранг (MPI_Comm_rank). Затем один процесс (с рангом 0) посылает сообщения в цикле (MPI_Send), устанавливает целевой аргумент (предназначения) в индекс цикла, чтобы быть уверенным, что в каждый из оставшихся процессов посылается одно сообщение. Оставшиеся процессы получают одно сообщение (MPI_Recv). Затем все процессы печатают сообщение и выходят из MPI (MPI_FInalize).

Здесь нет заботы о том, что не произойдет в этой программе. Нет функций, которые вызывают дополнительные копии программы на выполнение. Для запуска программы на выполнение на суперкомпьютере используют команду mpirun.

В упражнении лабораторной работы, изложенной в разделе практических заданий, от вас потребуется расширить эту программу некоторыми дополнительными вызовами функций MPI.

3.1.3. Сообщения MPI

Сообщения MPI состоят из двух основных частей: отправляемые/получаемые данные и сопроводительная информация (записи на конверте /оболочке/), которая помогает отправить данные по определенному маршруту. Обычно существуют три вызываемых параметра в вызовах передачи сообщений MPI, которые описывают данные и три других параметра, которые определяют маршрут:

сообщение = данные (3 параметра) + оболочка (3 параметра).

Данные включают в себя параметры старт буфера, число, тип данных, а оболочка включает цель, тег и коммуникатор.

Каждый параметр в данных и оболочке (конверте) будет обсужден более детально ниже, включая информацию о том, когда

эти параметры следует координировать между отправителем и получателем.

3.1.3.1. Данные

Для передачи данных необходимы следующие параметры.

Буфер в вызовах MPI есть место в компьютерной памяти, из которого сообщения должны быть посланы или где они накапливаются. В этом смысле буфер – это просто память, которую компилятор выделил для переменной (часто массива) в вашей программе. Следующие три параметра вызова MPI необходимы, чтобы определить буфер.

- **Старт буфера –** это адрес, где данные начинаются. Например, начало массива в вашей программе.

- **Число** элементов (пунктов) данных в сообщении. Заметим, что это элементы, а не байты. Это делается для переносимости кода, ибо нет необходимости беспокоиться о различных представлениях типов данных на различных компьютерах. Реализация матобеспечения MPI определяет число байт автоматически. Число, определенное при получении должно быть больше чем или равно числу, определенному при отправке. Если посылается больше данных, чем имеется в хранилище принимающего буфера, то произойдет ошибка.

- **Тип данных.** Необходимо знать тип данных, которые будут передаваться, данные с плавающей точкой, например. Этот тип данных должен быть тем же самым для вызовов отправки и получения. Исключением из этого правила является тип данных MPI_PACKED, который является одним из способов обработки сообщений со смешанным типом данных предпочтительным методом является метод с производными типами данных). Проверка типов не нужна в этом случае. Типы данных, уже определенные для вас, называются «основными типами данных» и перечислены в табл. 1.

Основные типы данных MPI для языка C

Таблица 1. Сопоставление типов данных для C и MPI

Типы данных MPI	Типы данных C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Производные типы данных

Могут быть также определены дополнительные типы данных, названные производными типами данных. Вы можете определить тип данных для несмежных данных или для последовательности смешанных основных типов данных. Такое определение часто облегчает программирование и обеспечивает более быстрое выполнение кода. Производные типы данных находятся вне сферы этого введения, но охвачены в модуле "Производные типы данных".

Некоторые производные типы данных:

- Contiguous,
- Vector,
- Hvector,
- Indexed,
- Hindexed,
- Struct.

3.1.3.2. Оболочка (конверт)

Напомним, что сообщение состоит из данных и оболочки (конверта) сообщения. Оболочка дает информацию о том, как связаны отправления с получениями. Три параметра используются для определения оболочки (конверта) сообщения.

Назначение или источник

Этот аргумент устанавливается к рангу в комммуникаторе (см ниже). Ранг меняется от 0 до (size-1), где size – это число процессов в коммуникаторе. Назначение определяется отправкой и используется, чтобы определить маршрут сообщения к соответствующему процессу. Источник определяется получением. Только сообщения, идущие от этого источника, могут быть приняты при вызове получения, но получение может установить источник в MPI_ANY_SOURCE, чтобы указать, что любой источник приемлем.

Тег

Тег (ярлык, метка) – произвольное число, которое помогает различать сообщения. Теги, определяемые отправителем и получателем, должны совпадать, но получатель может определить его как MPI_ANY_TAG, чтобы показать, что любой тег приемлем.

Коммуникатор

Коммуникатор, определенный при отправке должен равняться коммуникатору, определенному при получении. Коммуникаторы будут обсуждаться более глубоко чуть позже в этом же модуле. Сейчас будет достаточно знать, что коммуникатор определяет коммуникационную «вселенную», и то, что процессы могут принадлежать к более чем одному коммуникатору. В этом модуле мы будем иметь дело только с предопределенным коммуникатором MPI_COMM_WORLD, который включает все процессы приложения.

Аналогия

Чтобы легче понять параметры окружения сообщения, рассмотрим аналогию с агентством, выпускающим иски

(платежные требования – квитанции) по нескольким потребностям. Посылая иск, агентство должно указать:

- Лицо, получающее иск (более определенно, его идентификационный номер ИН). Это назначение.
- Какой месяц охватывает этот иск. Так как лицо получит двенадцать исков в год, ему необходимо знать, за какой месяц приходит этот иск. Это тег (ярлык, метка).
- На какую потребность выпускается иск. Лицу надо знать, иск ли это за электричество или за телефон. Это коммуникатор.

3.1.4. Коммуникаторы

3.1.4.1. Зачем нужны коммуникаторы

Раскроем теперь немного больше понятие коммуникатора. Не будем углубляться во все детали – только в те, что обеспечивают некоторое понимание того, как они используются. Коммуникаторы охвачены более глубоко в модуле "Управление группами и коммуникаторами MPI".

Приемлемость сообщения для захвата точно определенным вызовом принятия зависит от его источника, тега и коммуникатора. Тег позволяет программе различать типы сообщений. Источник упрощает программирование. Вместо того, чтобы иметь уникальный тег для каждого сообщения, каждый процесс, посылающий ту же самую информацию, может использовать тот же тег. Но зачем нужен коммуникатор?

Пример

Предположим, вы посылаете сообщения между вашими процессами и, кроме того, вызываете ряд библиотек, полученных откуда-либо, которые также порождают процессы, выполняемые на различных узлах, взаимодействующих друг с другом с помощью MPI. В этом случае вам хотелось бы быть уверенными, что отправленные вами сообщения придут к вашим процессам и не будут смешаны с сообщениями, отправленными между процессами изнутри библиотечных функций.

В этом примере мы имеем три процесса, взаимодействующих друг с другом. Каждый процесс также

вызывает библиотечную функцию, при этом три параллельные части библиотеки взаимодействуют друг с другом. Нам хочется иметь два различных «пространства» сообщений, одно для наших сообщений и одно для библиотечных сообщений. Нам бы не хотелось получить какое-либо перемешивание этих сообщений.

Блоки представляют части параллельных процессов. Время растет сверху вниз на каждой диаграмме. Цифры в круглых скобках НЕ параметры, а номера процессов. Например, send(1) означает отправку сообщения процессу 1. Recv(any) означает получение сообщения от любого процессора. Пользовательский (вызывающий) код находится в белом (незатененном) блоке. Затененный блок (вызываемый) представляет собой пакет библиотеки (параллельной), вызванной пользователем. Наконец, стрелки представляют собой перемещение сообщения от отправителя получателю.

Что бы произошло в этом случае, можно увидеть на рис. 1. В этом случае все работает, как намечено.

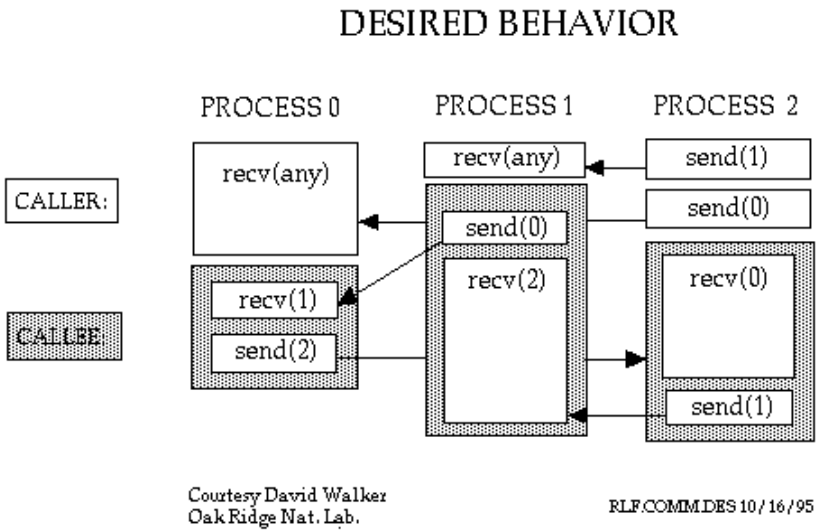

Рис. 1. Верное поведение процесса

Тем не менее нет никакой гарантии, что события произойдут в этом порядке, так как относительное расписание

процессов на различных узлах может меняться от выполнения к выполнению. Предположим, мы изменили третий процесс, добавив некоторые вычисления вначале. Последовательность событий может оказаться такой, как показано на рис. 2.

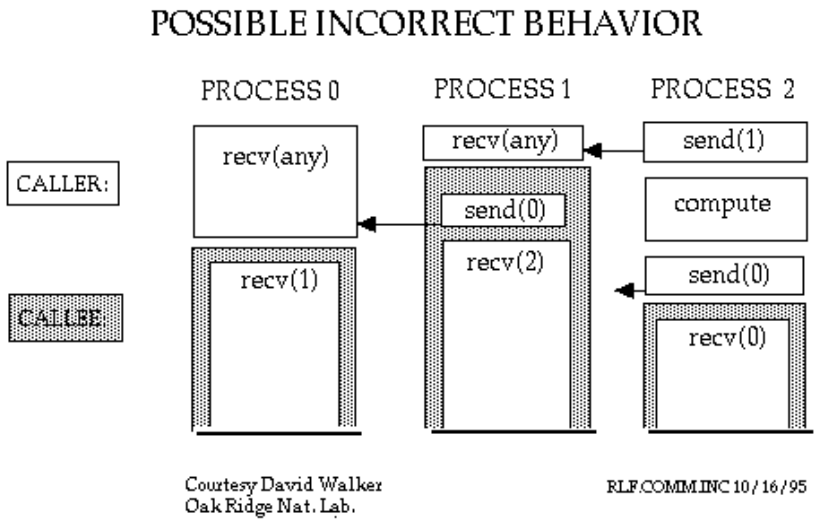

Рис. 2. Неверное поведение процесса

В этом случае коммуникации не происходят, как намечено. Первый «receive» в процессе 0 теперь получает «send» из библиотечной функции в процессе 1, а ненамеченный (и теперь задержанный) «send» – из процесса 2. В результате все три процесса подвисают.

Проблема решается за счет того, что разработчик библиотеки запрашивает новый и уникальный коммуникатор и определяет этот коммуникатор во всех вызовах отослать и получить, которые делаются библиотекой. Это создает библиотечное (вызываемое) пространство сообщений, отдельное от пользовательского (вызывающего) пространства сообщений.

Можно ли использовать теги, чтобы осуществить отдельные пространства сообщений? Проблема с тегами состоит в том, что их значения задаются программистом и он(а) может использовать тот же тег, что и параллельная библиотека,

использующая MPI. С коммуникаторами система, а не программист назначает идентификацию – система назначает коммуникатор пользователю и она назначает отличный коммуникатор библиотеке, так что не возникает возможность перекрытия.

3.1.4.2. Группы коммуникаторов и процессов

В дополнение к разработке параллельных библиотек, коммуникаторы полезны также в организации коммуникаций внутри приложения. Мы описывали коммуникаторы, которые включают все процессы в приложении. Но программист может также определить подмножество процессов, называемое группой процессов, и прикрепить один или больше коммуникаторов к этой группе процессов. Коммуникация определяет, что коммуникатор будет теперь ограничиваться этими процессами.

Заново напомним аналогию с выпуском исков: одно лицо может иметь счет от электрической и телефонной компаний (два коммуникатора), но ни одного от водопроводной компании. Электрический коммуникатор может содержать людей, отличающихся от телефонного коммуникатора. Персональный ИН номер (ранг) может изменяться с потребностью (коммуникатором). Итак, критически важно заметить, что ранг, заданный как источник или назначение сообщения, есть ранг в точно определенном коммуникаторе.

3.1.5. Запуск MPI программ

3.1.5.1. Компиляция

Компиляция и перенос приложений обсуждает различные способы исполнения выполнения на суперкомпьютере. Пожалуйста, обращайтесь к этому модулю для общих наставлений по созданию исполняемых программ. Для создания параллельной исполняемой программы вам следует включить директорию, содержащую MPI и библиотеки MPI, когда вы активизируете компилятор. Это подразумевает, что компилятор и MPICH установлены на машине, где вы компилируете исполняющую программу.

Если вы компилируете программу с помощью MS Visual C/C++, то вам необходимо добавить библиотеку MPICH.lib. Если же компиляция происходит из командной строки, то нужно вписать строку:

mpicc prog.c (после активизации setup_visualc).

3.1.5.2. Запуск программ, использующих MPI

Запуск на исполнение MPI-программы производится с помощью команды:

mpirun [параметры_mpirun...] <имя_программы> [параметры_программы...] [-host]

Параметры команды mpirun следующие:

-h

интерактивная подсказка по параметрам команды mpirun.

-maxtime <максимальное_время>

Максимальное время счета. От этого времени зависит положение задачи в очереди. После истечения этого времени задача принудительно заканчивается.

-np <число_процессоров>

Число процессоров, требуемое программе.

-quantum <значение_кванта_времени>

Этот параметр указывает, что задача является фоновой, и задает размер кванта для фоновой задачи.

-restart

Указание этого ключа приведет к тому, что после своего завершения задача будет вновь поставлена в очередь. Для удаления из очереди такой задачи пользуйтесь стандартной командой mqdel, а для ее завершения – командами mkill или mterm.

-stdiodir <имя_директории>

Этот параметр задает имя каталога стандартного ввода/вывода, в который будут записываться протокол запуска задачи, файл стандартного вывода и имена модулей, на которых запускалась задача.

-stdin <имя_файла>

Этот параметр задает имя файла, на который будет перенаправлен стандартный ввод задачи.

-stdout <имя_файла>

Этот параметр задает имя файла, на который будет перенаправлен стандартный вывод задачи.

-stderr <имя_файла>

Этот параметр задает имя файла, на который будет перенаправлен стандартный вывод сообщений об ошибках задачи.

-interactive

Задание этого ключа делает задачу интерактивной (см. п. 13), а также отменяет действия ключей stdin, stdout, stderr.

-ldmname <имя_ресурса_ЛДП>

Этот параметр указывает на то, что задача будет использовать ресурс ЛДП с указанным именем. Если на момент запуска задачи ресурс ЛДП с указанным именем не существует, он будет создан временным, о чем пользователь извещается специальным сообщением, например:

Warning! LDM resource ldm does not exists, it will be created temporary.

Для временного ресурса пользователь обязан задать размер требуемой дисковой памяти на каждом модуле (параметр ldmspace).

Для любых типов ресурсов (как разовых, создаваемых на время счета задачи, так и постоянных, т.е. уже выделенных на момент запуска задачи) требуется указание каталога монтирования (параметр mountdir).

Если в очереди или в счете у пользователя уже есть задача, использующая (или требующая) временный ресурс с таким же именем, то система выдаст сообщение об ошибке, и данная задача не будет поставлена в очередь. Подробнее о ресурсах ЛДП см. п. 5.

-ldmspace <размер_ресурса_ЛДП_на_одном_модуле>

Этот параметр имеет действие только, если указано имя ресурса ЛДП (параметр ldmname) и ресурс ЛДП с заданным именем не существует (т.е. для задачи требуется разовый или временный ресурс ЛДП). Параметр ldmspace задает размер локальной дисковой памяти на одном модуле, требуемой под временный ресурс ЛДП. Локальная дисковая память заданного размера будет выделена на каждом модуле. Размер ресурса ЛДП задается в килобайтах. Подробнее о ресурсах ЛДП см. п. 5.

-mountdir <имя_каталога_монтирования>

Этот параметр имеет действие только, если указано имя ресурса ЛДП (параметр ldmname). Параметр mountdir задает имя каталога монтирования для ресурса ЛДП. Монтирование ресурса ЛДП к указанному пользователем каталогу будет производиться перед стартом задачи на каждом вычислительном модуле. ВНИМАНИЕ! Монтирования ресурса ЛДП на сервер доступа или управляющую ЭВМ при старте задачи не производится! Подробнее о ресурсах ЛДП см. п. 5.

-termtime <дополнительное_время>

Этот параметр задает дополнительное время для завершения задачи. Время задается в минутах. Подробнее о дополнительном времени см. п. 4.

-termsignal <сигнал_для_завершения>

Этот параметр имеет действие только, если задано дополнительное время для завершения задачи (параметр termtime). Параметр termsignal задает сигнал, который будет разослан всем процессам задачи в качестве предупреждения о предстоящем завершении. Формат задания сигнала должен соответствовать команде kill (pkill). Пользователь должен самостоятельно определить обработчик сигнала в своей программе. Подробнее о дополнительном времени см. п. 4. Подробнее о сигналах см. описание системных вызовов kill() и signal().

-transform <имя_командного_файла>

При запуске задачи происходит преобразование списка выделенных задаче вычислительных модулей в формат среды программирования MPICH. По умолчанию системой используется командный файл /common/runmvs/bin/p4togm.sh. Параметр имя_командного_файла задает командный файл, который выполнит указанное преобразование вместо стандартного файла p4togm.sh. Необходимо учесть, что при вызове данный командный файл получит два параметра: файл со списком узлов, выделенных задаче, и полное имя файла запускаемой программы. Подробнее форматы этих файлов и описание ключа transform можно найти в разделе «Интерфейс между средой для разработки параллельных программ и системой управления прохождением задач» Руководства администратора.

-width

Использование альтернативного способа нумерации процессоров. По умолчанию процессы задачи распределяются по процессорам выделенных модулей в следующем порядке: 1-й процесс – на 1-й процессор 1-го модуля, 2-й процесс – на 1-й процессор 2-го модуля, 3-й процесс – на 1-й процессор 3-го модуля и т.д. После занятия всех 1-х процессоров всех выделенных модулей занимаются 2-е процессоры в том же порядке.

При указании ключа width используется другой способ нумерации: 1-й процесс – на 1-й процессор 1-го модуля, 2-й процесс – на 2-й процессор 1-го модуля, 3-й процесс – на 1-й процессор 2-го модуля и т.д.

Ключ -host описан в п. 14.

Удачно запущенная задача получает определенный номер, который добавляется к имени задачи. Это позволяет пользователю запускать одновременно несколько экземпляров задачи с одним и тем же именем – система присвоит каждому экземпляру задачи уникальный номер. Для каждого экземпляра будет создан отдельный каталог стандартного ввода/вывода. По завершении задачи ее номер «освобождается» и будет использован повторно. При удачном старте система выдаст пользователю следующую информацию:

– имена свободных узлов в системе на момент запуска задачи;

– имена выделенных под задачу узлов;

– сведения о принятых системой установках по умолчанию.

Завершает выдачу сообщение об удачном старте задачи, причем в сообщении указывается присвоенный задаче номер, например:

Task «test.1» started successfully

Может случиться так, что задача не будет запущена сразу, а поставлена в очередь. В этом случае реакция системы будет следующей:

Task «test.1» was queued

В процессе работы команды mpirun образуется файл паспорта задачи <имя_программы>.img, формат которого описан ниже. Данный файл может быть использован в команде mrunf.

3.1.5.3. Запуск в пакете

Процедура запуска параллельной работы в пакете немного сложнее, чем запуска последовательной работы.

- Когда переключаетесь между узлами, помните, что развиваемые узлы содержат по 2 процессора каждый, а некоторые ускоренные машины имеют и по 4 процессора на каждом узле.
- Для выполнения исполняемой программы следует использовать mpirun.
- Файлы необходимо скопировать на все узлы работы и из всех узлов работы.

3.1.6. Резюме

Хотя MPI обеспечивает расширенное множество вызовов, функциональная программа на MPI может быть записана с помощью всего шести базовых вызовов:

- MPI_Init,
- MPI_Comm_rank,
- MPI_Comm_size,
- MPI_Send,
- MPI_Recv,
- MPI_Finalize.

Для удобства программирования и оптимизации кода вам следует рассмотреть использование других вызовов, таких, как описано в модулях по попарной и коллективной коммуникациям.

Сообщения MPI

Сообщения MPI состоят из двух частей:

- данные (старт буфера, число, тип данных),
- оболочка (назначение/источник, тег, коммуникатор).

Данные определяют информацию, которая будет отослана или получена. Оболочка (конверт) используется в маршрутизации сообщения к получателю и связывает вызовы отправки с вызовами получения.

Коммуникаторы

Коммуникаторы гарантируют уникальные пространства сообщений. В соединении с группами процессов их можно

использовать, чтобы ограничить коммуникацию к подмножеству процессов.

3.1.7. Вопросы для самопроверки по основам программирования в MPI

1. Что вы всегда можете ожидать от MPI?
   - верная программа MPI должна выполняться на любой машине, которая поддерживает MPI;
   - верная программа MPI должна давать сопоставимое представление на любой машине, которая поддерживает MPI.
2. Отметьте каждый верный пункт о MPI:
   - MPI – библиотека передачи сообщений;
   - MPI – официальный стандарт;
   - программы MPI переносимы;
   - функции, начинающиеся с «MPE» являются частью MPI.
3. Что неправильно в следующем вызове функции MPI на Fortran (вы должны отвечать без вызова man-страницы руководства)?

```
count = 12
dest = 0
tag = 100
call MPI_Send (buffer, count, MPI_CHARACTER, dest, tag, MPI_COMM_WORLD)
```

   - в этом случае неверно имя подпрограммы MPI;
   - пропущен последний аргумент, который должен быть кодом ошибки.
4. Какие из следующих предложений о ранге **не** верны?
   - ранг есть целое число между 0 и nproc - 1, где nproc равно числу процессов в приложении;
   - каждый ранг уникален внутри коммуникатора;
   - ранг возвращается посредством обращения к MPI_Comm_rank.
5. «Буфер» в MPI является:
   - временным размещением выхода;
   - коммуникационным путем;
   - пространством в памяти.

6. Параметр «число» в вызовах отправки и получения в MPI измеряется в единицах:

- в световых единицах;
- в байтах данных;
- элементов данных;
- пакетов данных.

7. Правда или нет: уникальный тег должен быть точно определен при каждом вызове получения:

- правда;
- ложь.

8. Как много байт в данных сообщения передается в данном вызове из C?

MPI_Send(buffer, 1024, MPI_INT, dest, tag, MPI_COMM_WORLD)

- 1024 байт;
- 1024 * число байт, используемых для записи целого числа.

9. Проверьте все, что подходит: что должно быть верно для того, чтобы сообщение было «маршрутизировано» к точно определенному вызову получения?

- Коммуникатор определенный при отправке должен совпадать с коммуникатором, определенном при получении.
- Тег сообщения, определенный при отправке, должен равняться тегу сообщения, определенного при получении.
- Тег сообщения, определенный при отправке, должен *соответствовать* тегу сообщения, определенного при получении.

10. Основная цель коммуникатора состоит в том, чтобы

- определить размер группы процессов,
- помочь в маршрутизации сообщений,
- определить число задач в функции параллельной библиотеки.

11. С каким числом функций MPI обычно может иметь дело программа для начинающих?

- 2,
- 4,
- 6,
- 8.

12. Для каждого из следующих вызовов MPI из C программы проверьте, когда этот вызов *НЕ* работает. Предполагается, что все другие необходимые команды представлены верно.

- MPI_Send(msg, 12, MPI_CHARACTER, i, tag, MPI_COMM_WORLD)
- MPI_Comm_size(MPI_COMM_WORLD, &size)
- rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank, &ierror)

3.1.8. Упражнения по основам программирования в MPI

Предварительные требования

Практические занятия первой части курса выполняются на виртуальной параллельной машине, устанавливаемой в Windows XP/7 на вашем компьютере.

Во-первых, перед началом работы вам следует установить MPICH для соответствующего Windows на ваш компьютер.

Затем вам необходимо изучить инструкцию пользователя MPICH на английском языке или руководство пользователя российского суперкомпьютера МВС 1000М, с тем чтобы вы знали ответы на следующие вопросы.

- Компиляция при использовании библиотек MPI.
- Использование команды mpirun.
- Создание файла.
- Понимание механизма выполнения работы MPI на двух узлах.

Обзор

В первых двух упражнениях этой лабораторной работы требуется модифицировать очень простую программу MPI, используя только шесть базовых вызовов MPI.

В третьем упражнении дана простая последовательная программа, названная karp, которая вычисляет PI, используя цикл for для вычисления аппроксимации интеграла. Требуется преобразовать ее в параллельную программу, используя представление SPMD (одна программа множество данных).

Для относительных новичков в передаче сообщений, эти упражнения возможно потребуют следующее время для выполнения.

- Упражнение 1: 20–30 минут.
- Упражнение 2: 20–30 минут.
- Упражнение 3: 45–60 минут.

Чтобы посмотреть на синтаксис и описание вызовов MPI, справляйтесь с руководством программиста российского суперкомпьютера МВС 1000М.

Упражнение 1. Добавить отправку сообщений от рабочих мастеру

Создадим в блокноте исходный С файл: hello.c.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, type = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0)
  {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 14, MPI_CHAR, i, type,
MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, type,
MPI_COMM_WORLD, &status);
```

```
    printf( "Message from process = %d : %.14s\n",
rank,message);
    MPI_Finalize();
    return 0;
}
```

Файл решения на C записать в: helloex1.c

Hello является программой SPMD (Single Program Multiple Data = одна программа множество данных), то есть одна и та же программа выполняется и как процесс «мастер», и как процессы «рабочие». Программа определяет, является ли она мастером (ранг 0) или рабочим (ранг 1 или выше) посредством предложения if и затем разветвляется по соответствующим сегментам программы.

Мастер посылает сообщение («Hello world!») всем рабочим и затем распечатывает сообщение на стандартный вывод stdout. Каждый рабочий получает его сообщение, затем распечатывает его на stdout.

Ваша миссия заключается в том, чтобы модифицировать программу hello так, чтобы каждый рабочий вместо распечатки подтверждения посылал сообщение обратно мастеру, добавив в него рабочий ранг. Мастер должен получить эти сообщения и распечатать сообщение и соответствующие ранги на stdout.

Запустите эту программу на 4 процессорах.

Упражнение 2. Согласовать сообщения, используя теги

Исходный C файл: helloex1.c, файл решения на C записать в: helloex2.c

Приложение может использовать параметр тег в вызовах отправки и получения, чтобы отличать сообщения. Модифицируйте программу, полученную в упражнении 1 так, чтобы мастер посылал сообщения каждому рабочему, используя два различных тега. Используя теги, заставьте рабочих получать сообщения в обратном порядке и затем ответьте мастеру, как в упражнении 1. И опять заставьте мастера получить и распечатать каждое сообщение и соответствующий ранг на stdout.

Запустите эту программу на 8 процессорах. Создайте командный файл для запуска ex2.bat и очистки ex2cu,bat.

Упражнение 3. Преобразовать последовательный код в параллельный

Исходный С файл: karp.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define f(x) ((double)(4.0/(1.0+x*x)))
#define pi ((double)(4.0*atan(1.0)))
void startup (void);
int solicit (void);
void collect (double sum);
int main()
  {
  double sum, w;
  int i, N;
  /*
   * The startup routine will create parallel tasks
   */
  /* startup(); */
  /*
   * The solicit routine will get and propagate the value of N
   */
  N = solicit();

  while (N > 0) {
    w = 1.0/(double)N;
    sum = 0.0;
    for (i = 1; i <= N; i++)
      sum = sum + f(((double)i-0.5)*w);
    sum = sum * w;
    collect (sum);
    N = solicit ();
  }
  return (0);
}
```

```
void startup (void)
{
}

int solicit (void)
{
  int N;
  printf ("Enter number of approximation intervals:(0 to exit)\n");
  scanf("%d",&N);
  return (N);
}
void collect(double sum)
{
  double err;
  err = sum - pi;
  printf("sum, err = %7.5f, %10e\n", sum, err);
}
```

Файл решения на C записать в: karpsoln.c.
Входной файл данных: values.

```
10
100
0
```

Программа karp вычисляет PI, используя интегральную аппроксимацию. Вам предоставлена последовательная версия программы karp, и от вас требуется модифицировать ее в параллельную версию в форме SPMD.

а) Разберитесь с тем, как работает последовательная версия.

Скопируйте и запустите последовательную программу, затем найдите ответ на следующие вопросы:

- Как программа считает PI?
- Как точность вычисления зависит от числа шагов аппроксимации N? (Совет: отредактируйте values для различных входных значений от 10 до 10000).

– Как вы думаете, что будет с точностью, с которой мы вычисляем PI, когда мы разобьем работу по узлам?

б) Добавьте вызовы MPI чтобы создать некую SPMD karp программу.

В этом разделе вы разбиваете работу по параллельным процессам.

Цель состоит в том, чтобы получить реальную, работающую, SPMD программу, осуществленную на MPI.

Ваши действия заключаются в редактировании программы karp. Чтобы разбить работу по процессам, используйте только шесть базовых вызова MPI.

Совет: мастеру необходимо дать знать всем рабочим полное число итераций, и затем каждый рабочий вычисляет его индексы цикла, так чтобы он проделал его часть работы. Когда это сделано, каждый рабочий посылает его частную сумму назад мастеру, который получает их и вычисляет окончательную сумму.

Рассчитать задачу на одном из вводимых узлов.

Выполнить задачу с четырьмя процессорами и с восемью процессорами.

Вопросы

– Считается ли программа?
– Дает ли она правильный ответ?
– Для данного N будет ли вычисленное значение PI всегда тем же?
– Как может операция «распространения» (broadcast), в которой одна задача посылает одно и то же сообщение всем другим задачам, помочь вам? Операция MPI «распространения» (broadcast) будет изучаться позднее.

4. Попарный обмен сообщениями в MPI, I

Изложение следует http://www.ccas.ru/mmes/educat/lab04k/02/p2pCommI.html.

Хорошее изложение можно найти также в [9].

4.1. Обзор

Попарная коммуникация (от точки к точке) включает передачу сообщения между одной парой процессов, что противостоит коллективной коммуникации, включающей группу задач. MPI отличается более широким кругом попарных коммуникаций, чем большинство других библиотек передачи сообщений.

Во многих библиотеках передачи сообщений, таких как PVM или MPI, метод, по которому система обрабатывает сообщения, выбран разработчиком библиотеки. Выбранный метод дает приемлемую надежность и эффективность работы для всех возможных сценариев коммуникации. Но это может скрыть возможные проблемы программирования либо не дать наилучшую из возможных эффективность в точно определенных условиях. Для MPI это эквивалентно стандартному способу коммуникации, который будет приведен в этом модуле.

В MPI большая часть контроля того, как система управляет сообщением, отдано программистам, которые выбирают способ коммуникации в момент выбора функции отправки. В дополнение к стандартному способу MPI обеспечивает так же синхронный, по готовности и буферный способы. В данном разделе курса (модуле) посмотрим на поведение системы для каждого способа и обсудим их преимущества и недостатки.

В дополнение к выбору способа коммуникации программист должен решить, будут ли вызовы отправки и получения блокирующими или неблокирующими. Блокирующая или неблокирующая отправка может быть спарена с блокирующим или неблокирующим получением.

Блокирующий подвешивает исполнение, до тех пор пока буфер сообщения безопасно используется. В обоих способах отправления и получения буфер, используемый для содержания сообщения, может оказаться неиспользуемым ресурсом, и данные могут быть искажены, когда он используется, до того как проходящий перевод завершен; блокирующие коммуникации гарантируют, что этого никогда не случится. Когда контроль возвращается от блокирующего вызова, буфер может быть безопасно модифицирован без какой-либо опасности испортить некоторую часть процесса.

Неблокирующий отделяет коммуникацию от вычисления. Неблокирующий может эффективно гарантировать, что прерывание будет сгенерировано, когда сообщение (перевод, транзакция) готово к выполнению, тем самым позволяя оригинальной нити вернуться назад к вычислительно-ориентированной обработке.

4.2. Блокирующее поведение

Прежде чем приступить к изложению способов коммуникации давайте повторим синтаксис блокирующих отправки и получения:

MPI_Send является блокирующей отправкой (передача сообщения с блокировкой). Это означает, что вызов не возвращает управление к вашей программе до тех пор, пока данные не будут скопированы в место, определенное вами в листе параметров. Из-за этого вы можете изменить данные после вызова, не воздействуя на оригинальное сообщение. Заметим, что существуют неблокирующие отправки, в которых это не имеет места.

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Параметры:

buf – начало буфера, содержащего данные для отправки (для C это адрес),
count – число элементов для отправки (не байт),
datatype – тип данных,
dest – ранг процесса, который является назначением сообщения,
tag – произвольное число, которое можно использовать для отличия данного сообщения от других,
comm – коммуникатор.

Подобно MPI_Send, MPI_Recv является блокирующей. Это означает, что вызов не возвращает управление в вашу программу до тех пор, пока все получаемые данные не будут запомнены в переменных, которые вы определили в листе параметров. Из-за этого вы можете использовать данные после вызова и быть

уверены, что все они здесь. Существуют неблокирующие получения, в которых это не имеет места.

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Параметры:

buf – начало буфера для хранения приходящих данных (для C – адрес),
count – число элементов (не байт) в вашем буфере получения,
datatype – тип данных,
source– источник, то есть ранг процесса, из которого данные будут приняты,
tag – тег, то есть произвольное число, которое можно использовать для отличия данного сообщения от других,
comm – коммуникатор,
status – статус, который является массивом или структурой возвращаемой информации.

Источник отправления может быть назначен любым (джокером – wildcard), если определить параметр MPI_ANY_SOURCE.

Точно так же и тег принимаемого сообщения может быть любым (джокером – wildcard), если определить параметр MPI_ANY_TAG.

Если вы определяете источник или тег любым, status покажет вам действительный ранг или тег для полученного сообщения.

4.2.1. Способы коммуникации

Способ коммуникации выбирается функцией отправки. Существуют четыре функции блокирующей отправки и четыре функции неблокирующей отправки, соответствующие четырем способам коммуникации. Функция получения не определяет способ коммуникации – она является просто блокирующей или неблокирующей.

Вызовы отправки и получения, которые будут описаны в данном разделе, представлены в табл. 2.

Таблица 2. Способы коммуникаций и виды функций

Способ коммуникации	**Блокирующие функции**	**Неблокирующие функции**
Синхронный	MPI_Ssend	MPI_Issend
По готовности	MPI_Rsend	MPI_Irsend
Буферизованный	MPI_Bsend	MPI_Ibsend
Стандартный	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv
	MPI_Sendrecv	
	MPI_Sendrecv_replace	

Начнем изучение поведения блокирующей коммуникации для четырех способов, начав с синхронного способа. Для компактности изложения придержим изучение неблокирующего поведения до следующего раздела.

4.2.1.1. Блокирующая синхронная отправка

На рис. 3, приведенном ниже, время течет слева направо. Жирная горизонтальная линия, помеченная S, представляет время исполнения для задачи отправки (на одном узле), а жирная пунктирная линия, помеченная R, представляет время исполнения для задачи получения (на втором узле). Разрывы в этих линиях представляют прерывания, обусловленные событием передачи сообщения.

Когда блокирующая синхронная отправка MPI_Ssend исполняется, задача отправки отсылает задаче получения сообщение «готова к отправке». Когда задача получателя исполняет вызов получения, она посылает сообщение «готова к получению». Затем данные передаются.

Существуют два источника накладок (overhead) в передаче сообщений. **Системная накладка** вызывается копированием данных сообщения из буфера сообщения отправителя и копированием данных из сети в буфер сообщения получателя.

Синхронизационная накладка есть время, потраченное на событие ожидания другой задачи. На рис. 3 отправитель должен ждать для исполнения события получения перед тем, как

сообщение может быть передано. Получатель также подвержен некоторой синхронизационной накладке при ожидании завершения встречи. Неудивительно, что синхронизационная накладка может быть значительной в синхронном способе. Как мы увидим, другие способы используют разные стратегии для сокращения этой накладки.

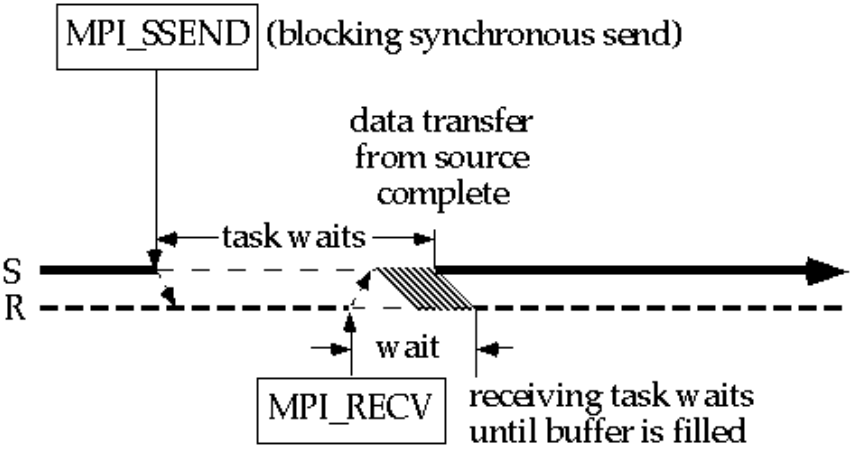

Рис. 3. Блокирующая синхронная отправка

Только один относительный выбор времени для вызовов. (MPI_Ssend и MPI_Recv показан, но они могут прийти в любом порядке. Если вызов получения предшествует отправке, большую часть синхронизационной накладки будет нести получатель.

Кто-то может надеяться, что если работа соответствующим образом сбалансирована по загрузке, синхронизационная накладка будет минимальна для обеих задач: и отправления, и получения. Это не всегда реально. Если ничего больше не вызывает отсутствия синхронизации, то системные службы, которые выполняются в непредсказуемое время на различных узлах, вызовут несинхронизированные задержки. Кто-то может ответить на это, сказав, что будет проще просто часто вызывать MPI_Barrier, чтобы сохранить задачи синхронными, но этот вызов сам по себе испытывает синхронизационную накладку и не гарантирует, что задачи будут синхронными несколькими секундами позже. Таким образом, вызов барьера почти всегда

является потерей времени. MPI_Barrier блокирует вызовы, пока все члены группы не вызовут его.

4.2.1.2. Блокирующая отправка по готовности

Отправка способом по готовности MPI_Rsend просто отсылает сообщение вовне над сетью. Она требует, чтобы прибыло уведомление «готова к получению», указывая, что задача получения послана на получение. Если сообщение «готова к получению» не прибыло, то отправка способом по готовности выдаст ошибку. По умолчанию код будет существовать. Программист может связать различное управление по ошибке с коммуникатором, чтобы избежать этого поведения по умолчанию. Диаграмма показывает последнюю посылку MPI_Recv, которая не вызвала бы ошибку.

Способ по готовности имеет целью минимизировать системную накладку и синхронизационную накладку, вызванную задачей отправления. В блокирующем случае единственным ожиданием на отправляющем узле является ожидание того момента, пока все данные не будут перемещены вовне из буфера сообщения задачи. Получение может все еще нести существенную синхронизационную накладку, зависящую от того, насколько ранее оно исполняется, чем соответствующая отправка.

Этот способ не следует использовать кроме случая, когда пользователь уверен, что соответствующее уведомление о готовности получения послано.

4.2.1.3. Блокирующая буферизованная отправка

Блокирующая буферизованная отправка MPI_Bsend (S) копирует данные из буфера сообщения в буфер, предложенный пользователем, и затем возвращает. Отправляющая задача может поэтому выполнять вычисления, которые модифицируют оригинальный буфер сообщения, зная, что эти модификации не будут отражены в данных, действительно отправленных. Данные будут скопированы из предложенного пользователем буфера на сеть, когда будет получено уведомление «готова получить».

Буферизованный способ испытывает дополнительную системную накладку из-за дополнительного копирования буфера сообщения в предложенный пользователем буфер.

Синхронизационная накладка исключается на задаче отправки – время получения теперь не имеет отношение к отправителю. Всякий раз как получение исполняется до отправки, оно должно ожидать прибытия сообщения до того, как оно может вернуться.

Другой выгодой для пользователя является возможность обеспечить то количество буферного пространства для исходящих сообщений, в котором программа нуждается. С обратной стороны, пользователь ответственен за управление и прикрепление этого буферного пространства. Буферизованный способ отправки, который требует больше буферного пространства, чем имеется в наличии, сгенерирует ошибку, и (по умолчанию) программа завершит работу.

Буферное управление

Для буферизованного способа отправки пользователь должен обеспечить буфер: он может быть статически распределенным массивом или память для буфера может быть динамически распределена функцией malloc. Количество памяти, распределенное для предложенного пользователем буфера, должно превышать сумму данных сообщения, так как заголовки сообщения должны также быть запомнены.

Это пространство должно быть идентифицировано, как предложенный пользователем буфер, посредством вызова MPI_Buffer_attach. Когда в этом больше нет нужды, это отделяют MPI_Buffer_detach. В каждый момент времени может быть активным только один предложенный пользователем буфер сообщения. Он запомнит многочисленные сообщения. Когда сообщения, в конце концов, покидают буфер, система сохраняет след и переиспользует буферное пространство. Безопасность программы не должна зависеть от этого обстоятельства.

4.2.1.4. Блокирующая стандартная отправка

Для стандартного способа библиотечный исполнитель определяет системное поведение, которое будет работать наилучшим образом для большинства пользователей на заданной системе (см. для MPICH).

Размер сообщения меньше, чем порог. В этом случае блокирующая стандартная отправка MPI_Send копирует сообщение на сети в системный буфер на узле получателя.

Стандартная отправка затем возвращается, и отправляющая задача может продолжать вычисление. Системный буфер прикрепляется, когда программа стартует – пользователю нет нужды управлять этим каким-либо способом. Существует один системный буфер на задачу, который будет содержать множественные сообщения. Сообщение будет скопировано из системного буфера в задачу получения, когда вызов получения исполняется.

Как и в буферизованном способе, использование буфера уменьшает вероятность синхронизационной накладки на отправляющей задаче за счет увеличения системной накладки, получающейся из дополнительного копирования в этот буфер. Как всегда, синхронизационная накладка может быть навлечена получающей задачей, если получение послано первым.

В отличие от буферизованного способа задача отправки не влечет за собой ошибки, если буферное пространство превышается. Вместо этого задача отправки заблокируется до тех пор, пока задача получения вызовет получение, которое выгрузит данные из системного буфера. Таким образом, отправляющая задача все еще может влезть в синхронизационную накладку.

Когда размер сообщения больше, чем порог, поведение блокирующей стандартной отправки MPI_Send, по существу, то же самое, как для синхронного способа.

Почему поведение стандартного способа отличается по размеру сообщения? Маленькие сообщения выигрывают от уменьшающегося шанса синхронизационной накладки, получающейся от использования системного буфера. Тем не менее по мере увеличения размера сообщения стоимость копирования в буфер возрастает и в конце концов становится невозможным обеспечить достаточное пространство системного буфера. Таким образом, стандартный способ пытается обеспечить наилучший компромисс.

4.2.1.5. Блокирующие отправка и получение

Операции отправки и получения могут быть скомбинированы в один вызов. MPI_Sendrecv осуществляет блокирующие отправку и получение, в которых буферы для отправки и получения должны быть разъединены. MPI_Sendrecv_replace также осуществляет блокирующие отправку

и получение, но заметим, что в этом случае существует только один буфер вместо двух, поскольку получаемое сообщение переписывает отправляемое.

- Процесс отправки и получения можно объединить.
- Sendrecv может отправить на регулярный Recv.
- Sendrecv может получить из регулярного Send.
- Sendrecv может быть прозондирован операцией зондирования (апробирования).

4.2.2. Выводы: способы отправки

Синхронизированный способ является «безопаснейшим», и потому также наиболее переносимым. «Безопасность» означает, что если код выполняют при одном и том же наборе условий, то есть размерах сообщений или архитектуре, то он выполнится при всех условиях. Синхронный код безопасен, так как он не зависит от порядка, в котором отправка и получение исполняются (в отличие от способа по готовности) или количества буферного пространства (в отличие от буферизованного способа или стандартного способа). Синхронизационный способ может вызвать существенную синхронизационную накладку.

Способ по готовности имеет наименьшую суммарную накладку. Он действительно не требует встречи (рукопожатия) отправителя и получателя (подобно синхронизированному способу) или дополнительного копирования в буфер (подобно буферизованному или стандартному способу). Тем не менее получение должно предшествовать отправке. Этот способ не подходит для всех сообщений.

Буферизованный способ разъединяет отправителя и получателя. Это исключает синхронизационную накладку на отправляющей задаче и гарантирует, что порядок исполнения отправки и получения не имеет значения (в отличие от способа по готовности). Дополнительным преимуществом является то, что программист может контролировать размер сообщений, которые будут буферизованы и полное количество буферного пространства. Существует дополнительная системная накладка, вызванная копированием в буфер.

Поведение стандартного способа определяется реализацией. Разработчик библиотеки выбирает системное

поведение, что обеспечивает хорошую эффективность работы и приемлемую безопасность.

4.3. Неблокирующее поведение

Блокирующие вызовы отправки и получения подвешивают (приостанавливают) исполнение программы до момента, когда буфер сообщения безопасно использовать для отправления/получения. В случае блокирующей отправки это означает, что данные, которые будут отправлены, должны быть скопированы из буфера отправки, но они не обязаны быть получены получающей задачей. Содержимое буфера отправки может быть модифицировано без воздействия на отправленное сообщение. Завершение блокирующего получения подразумевает, что данные в буфере получения правильные.

Неблокирующие вызовы возвращаются немедленно после инициации коммуникации. Программист не знает, являются ли данные, которые надо отправить, скопированными из буфера отправки или являются ли данные, которые надо получить, прибывшими. Таким образом, перед использованием буфера сообщения программист должен проверить его статус. Описание статуса будет охвачено в следующем модуле MPI попарный обмен сообщениями II.

Программист может выбрать блокировку, пока буфер сообщения безопасно использовать, посредством вызова MPI_Wait и его вариантов или только возвращая текущий статус коммуникации посредством MPI_Test и его вариантов.

Различные варианты вызовов Wait и Test позволяют вам проверить статус точно определенного сообщения либо проверить все, любой или некоторые из списка сообщений.

Интуитивно ясно, почему вам надо проверить статус неблокирующего получения: вам действительно не хочется считывать сообщение, пока вы не убедились, что это сообщение прибыло. Менее очевидно, почему вам нужно проверить статус неблокирующего отправления. Это чрезвычайно необходимо, когда вы имеете цикл, который повторно заполняет буфер сообщения и отправляет сообщение. Вы не можете записать что-нибудь новое в буфер, пока вы не уверитесь наверняка, что предыдущее сообщение успешно скопировано из буфера. Если

даже буфер отправки не используется повторно, выгодно завершить коммуникацию, так как это освобождает системные ресурсы.

4.3.1. Синтаксис неблокирующих вызовов

Неблокирующие вызовы имеют тот же синтаксис, как и блокирующие, с двумя исключениями:

1). Каждый вызов имеет «I», сразу следующий за «_».

2). Последний аргумент есть способ доступа (рукоятка) к скрытому объекту запроса, который содержит информацию о сообщении, т.е. его статус.

Например, стандартная неблокирующая отправка и соответствующий вызов Wait выглядят для языка С подобно этому:

MPI_Isend (buf,count,dtype,dest,tag,comm,request)

MPI_Wait (request, status).

Аналогично неблокирующий вызов получения есть MPI_Irecv.

Вызовы Wait и Test берут один или несколько способов доступа к запросу как вход и возвращают один или более статусов. В дополнение Test указывает, завершилась ли любая из коммуникаций, к которым запрос обращается. Wait, Test и статус обсуждаются в деталях в модуле о попарном обмене сообщениями II.

4.3.2. Пример: неблокирующая стандартная отправка

Ранее было рассмотрено блокирующее поведение для каждого способа коммуникации. Теперь можно обсудить неблокирующее поведение для стандартного способа. Поведение других способов может быть выведено отсюда.

Как и раньше, отправка стандартным способом выполнится по-разному в зависимости от размера сообщения. Отправляющая задача посылает неблокирующую стандартную отправку, когда содержимое буфера сообщения готово к тому, чтобы быть перемещено. Ее возврат происходит немедленно без ожидания завершения копирования в удаленный системный буфер. MPI_Wait вызывается как раз перед тем, как отправляющая задача нуждается в перезаписи буфера сообщения.

Получающая задача вызывает неблокирующее получение, как только буфер сообщения будет пригоден вместить сообщение. Возврат из неблокирующего получения происходит без ожидания прибытия сообщения. Задача получения вызывает MPI_Wait, когда ей нужно использовать данные входящего сообщения (то есть когда необходимо быть уверенным, что они прибыли).

Системная накладка не отличается существенно от накладки при блокирующих вызовах отправки и получения за исключением того, что перевод данных и вычисления могут происходить одновременно. Соответственно, если ЦПУ может понадобиться одновременно выполнить как передачу данных, так и вычисления, в этом случае вычисления будут прерваны на обоих (отправляющем и получающем) узлах, чтобы переправить сообщение. Если прерывание все-таки случается, то оно не должно как-то особо влиять на выполняемую программу. Даже для архитектур, которые перекрывают вычисления и коммуникацию, факт, что этот случай применяется только к малым сообщениям, означает, что не стоит ожидать большой разницы в эффективности работы.

Преимущество в использовании неблокирующей отправки происходит, когда отдаленный системный буфер полон. В этом случае блокирующая отправка должна была бы ожидать, пока принимающая задача выгрузит некоторые данные сообщения из буфера. Если используется неблокирующий вызов, то вычисление может быть сделано во время этого интервала.

Преимущество неблокирующего получения перед блокирующим может быть значительным, если получение послано перед отправкой. Вычисление задачи может продолжаться, пока не будет послана функция Wait. Это уменьшает итог синхронизационной накладки.

Неблокирующие вызовы могут гарантировать, что не получится, что оба процессора ожидают друг друга. Этот Wait должен быть послан после того, как вызовы понадобятся, чтобы завершить коммуникацию.

4.3.3. Пример: неблокирующая стандартная отправка, большое сообщение

Случай неблокирующей стандартной отправки MPI_Isend для сообщения, которое превышает порог, более интересен.

Для блокирующей отправки синхронизационной накладкой будет период между блокирующим вызовом и копированием в сеть. Для неблокирующего вызова синхронизационная накладка сокращается на количество времени между неблокирующим вызовом и MPI_Wait, во время которого полезные вычисления выполняются.

И опять неблокирующее получение MPI_Irecv сократит синхронизационную накладку на принимающей задаче для случая, в котором получение послано первым. Существует также выгода в использовании неблокирующего получения, когда отправка послана первой. Рассмотрим, как картина изменилась бы, если бы блокирующее получение было послано. В типичной ситуации блокирующие получения посылаются непосредственно перед тем, как данные сообщения должны быть использованы (чтобы дать максимальное время для завершения коммуникации). Таким образом, блокирующее получение было бы послано на место MPI_Wait. Это задержало бы синхронизацию с вызовом отправки до этой более поздней точки в программе и таким образом увеличило бы синхронизационную накладку на отправляющей задаче.

4.3.4. Выводы: неблокирующие вызовы

Избегайте тупиков

Избегайте получения процессов, ожидающих один и тот же ресурс или друг друга

Уменьшайте синхронизационную накладку

Неблокирующие вызовы имеют то преимущество, что вычисления могут продолжаться почти немедленно, даже если сообщение не может быть отослано. Это может исключить тупик и сократить синхронизационную накладку.

Некоторые системы: сократи системную накладку

На некоторых машинах системная накладка может быть сокращена, если транспортировка сообщения может быть проведена в фоновом режиме без получения какого-либо воздействия на прогресс в вычислениях на обоих процессах - и отправителе и получателе.

Получив некоторый опыт с основами MPI, проконсультируйтесь со стандартом или имеющимися учебниками для получения информации по постоянным запросам. Блокирующий вызов передает его с намного меньшими системными издержками, чем неблокирующий вызов. При этом если оба сообщения должны быть отправлены немедленно, а передача не является немедленно удовлетворяемой, то неблокирующий вызов выигрывает, потому что «полезные вычисления» могут быть завершены прежде их вынужденного окончания.

Лучше всего послать неблокирующие отправки и получения так рано, как это возможно, и сделать их ожидание настолько поздним, насколько возможно.

Некоторое дополнительное программирование требуется для неблокирующих вызовов, чтобы протестировать завершение коммуникации. Лучше всего послать отправки и получения настолько рано, насколько это возможно, и ждать завершения настолько позже, насколько это возможно. «Рано, насколько это возможно» выше означает, что данные в буфере, которые будут посланы, должны быть правильными и также буфер, который будет получен, должен иметься в наличии.

Следует избегать записывания в буфер отсылки между MPI_Isend и MPI_Wait и следует избегать считывания и записывания в буфер получения между MPI_Irecv и MPI_Wait.

Безопасно считывать буфер отправки должно быть возможным даже после того, как отправка послана, но ничего не должно быть записано в этот буфер до тех пор, пока статус не будет проверен, чтобы дать гарантию, что оригинальное

сообщение отослано. С другой стороны, содержимое оригинального сообщения может быть затерто. Никакого пользовательского чтения или записывания в буфер получения не должно иметь места между посылкой неблокирующего получения и определения того, что сообщение получено. Чтение может давать либо старые данные, либоё новые (приходящие) данные сообщения. Запись может переписать текущее прибывшее сообщение.

4.4. Рекомендации по программированию

В общем случае разумно начинать программирование с неблокирующих вызовов и стандартного способа. Неблокирующие вызовы могут исключить возможность тупика и сократить синхронизационную накладку. Стандартный способ дает обычно хорошую эффективность работы.

Блокирующие вызовы могут потребоваться, если программист хочет синхронизовать задачи. Также если программа требует, чтобы за неблокирующим вызовом немедленно следовал Wait, то намного эффективнее использовать блокирующий вызов. Если используются блокирующие вызовы, то может быть выгодным начать в синхронном способе и затем переключиться на стандартный способ. Тестирование в синхронном способе гарантирует, что программа не зависит от присутствия достаточного системного буферного пространства.

Следующим шагом является анализ кода и оценка его эффективности работы. Если неблокирующие получения посланы рано, значительно ранее соответствующих отправок, будет выгодно использовать способ по готовности. В этом случае получающая задача должна, возможно, извещать отправителя после того, как получения посланы. После получения уведомления отправитель может выполнить отправки.

Если существует слишком большая синхронизационная накладка на отправляющей задаче, особенно для больших писем, буферизованный способ может оказаться более эффективным.

Вопросы для самопроверки: MPI попарные коммуникации I

1. Какие из следующих вызовов не могут использоваться, чтобы получить сообщение, отправленное MPI_Isend?

- A. MPI_Irecv
- B. MPI_Recv
- C. MPI_Sendrecv
- D. оба: B и C

2. Вы думаете, что ваш код является тупиковым. Каков наибыстрейший и наигарантирующий способ определить, где тупик случается?

- Перекомпилировать с ключом -g и прикрепить отладчик в следующий раз, когда тупик случится.
- Внести операторы записи в вероятных местах и перекомпилировать.
- Перекомпилировать с ключом –p.
- Внимательно просмотреть код.

3. В зависимости от того, «кто пришел первым» для различных запусков с тем же самым исполняемым кодом и теми же самыми данными, получаются различные результаты. Какие из следующих действий не генерируют состояние, в котором важно «кто пришел первым»?

- A. чтение из буфера после MPI_Isend.
- B. чтение из буфера после MPI_Irecv.
- C. запись в буфер после MPI_Isend.
- D. запись в буфер после MPI_Irecv.

Предполагаем в каждом случае, что ни MPI_Wait, ни MPI_Test не вызываются для соответствующего запроса.

4. Какое утверждение справедливо для всех блокирующих отправок?

- На возврат из блокирующей отправки сообщение было получено удаленной задачей.
- На возврат из блокирующей отправки сообщение покинуло локальный узел.
- На возврат из блокирующей отправки буфер сообщения безопасно переписать.

5. Отметьте все, что подходит: в ситуации, когда блокирующая стандартная отправка исполняется вперед соответствующему ей получению и размер сообщения больше, чем порог, какие из следующих действий *могут* уменьшить синхронизационную накладку на отправке?

- Переключить на синхронизирующий способ.
- Переключить на способ по готовности.
- Переключить на буферизованный способ.
- Увеличить значение порога для стандартной отправки (если ваша реализация MPI позволяет это).
- Переключить на неблокирующую отправку.

Упражнения: MPI попарные коммуникации I

Предварительные требования

Упражнение следует делать после изучения "MPI основы попарного обмена сообщениями I". Вам следует вначале полностью завершить упражнения лабораторной работы по основам MPI, прежде чем приступать к этой работе.

Чтобы изучать или найти ссылки по синтаксису вызовов MPI, получите доступ к стандарту передачи сообщений из http://www-unix.mcs.anl.gov/mpi/ или из руководства программиста Межведомственного суперкомпьютерного центра.

Настоящее упражнение, во-первых, познакомит вас с использованием блокирующих и неблокирующих вызовов. Затем вы будете работать на простом коде, чтобы улучшить образцы декомпозиции и коммуникации данных.

Упражнение 1

Рабочий файл на C: deadlock.c

```
#include <stdio.h>
#include "mpi.h"
#define MSGLEN 4072          /* length of message in elements
*/
#define TAG_A 100
#define TAG_B 200
main( argc, argv )
    int argc;
    char **argv;
{
 float message1 [MSGLEN],          /* message buffers
*/
```

```
    message2 [MSGLEN];
int rank,                /* rank of task in communicator        */
   dest, source,         /* rank in communicator of destination  */
                         /* and source tasks                     */
   send_tag, recv_tag,   /* message tags                         */
   i;
MPI_Status status;       /* status of communication              */
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
printf ( " Task %d initialized\n", rank );
/* initialize message buffers */
for ( i=0; i<MSGLEN; i++ )  {
  message1[i] = 100;
  message2[i] = -100;
}

/* ------------------------------------------------------------
 * each task sets its message tags for the send and receive, plus
 * the destination for the send, and the source for the receive
 * ------------------------------------------------------------ */
if ( rank == 0 )  {
  dest = 1;
  source = 1;
  send_tag = TAG_A;
  recv_tag = TAG_B;
}
else if ( rank == 1)  {
  dest = 0;
  source = 0;
  send_tag = TAG_B;
  recv_tag = TAG_A;
}

/* ------------------------------------------------------------
 * send and receive messages
 * ------------------------------------------------------------ */
printf ( " Task %d has sent the message\n", rank );
```

```
    MPI_Send ( message1, MSGLEN, MPI_FLOAT, dest,
send_tag, MPI_COMM_WORLD );
    MPI_Recv ( message2, MSGLEN, MPI_FLOAT, source,
recv_tag, MPI_COMM_WORLD,
            &status );
    printf ( " Task %d has received the message\n", rank );

    MPI_Finalize();
    return 0;
}
```

Файл решения на C: fixed.c

Это упражнение демонстрирует, что неблокирующие функции безопаснее блокирующих функций.

Скомпилируйте эту тупиковую программу (deadlock). Например, для mpich в Unix, следует использовать команду:

mpicc -o deadlock deadlock.c.

Назначив для использования два узла и максимальное время исполнения 5 минут, запустите программу:

mpirun -np 2 deadlock -maxtime 5.

Программа выдаст несколько строк выходных данных и затем встанет. Чтобы не ждать пять минут до выброса по предельному времени счета, вам следует завершить запущенную задачу командой вида

mkill deadlock.1,

где deadlock – имя задачи, а 1 – ее номер.

Прочитайте всю тупиковую программу (deadlock). Понимаете ли вы, почему она попадает в тупик?

Откорректируйте тупиковую программу так, чтобы она могла работать до нормального завершения, с помощью замены блокирующего вызова на неблокирующий.

Упражнение 2

Рабочие файлы для C: least-squares-pt2pt.c, xydata

```
/* --------------------------------------------------------------------
 * FILE: least-squares-pt2pt.c
 *
 * PROBLEM DESCRIPTION:
 *  The method of least squares is a standard technique used to find
 *  the equation of a straight line from a set of data. Equation for a
 *  straight line is given by
 *         y = mx + b
 *  where m is the slope of the line and b is the y-intercept.
 *
 *  Given a set of n points {(x1,y1), x2,y2),...,xn,yn)}, let
 *     SUMx = x1 + x2 + ... + xn
 *     SUMy = y1 + y2 + ... + yn
 *     SUMxy = x1*y1 + x2*y2 + ... + xn*yn
 *     SUMxx = x1*x1 + x2*x2 + ... + xn*xn
 *
 *  The slope and y-intercept for the least-squares line can then be
 *  calculated using the following equations:
 *         slope (m) = ( SUMx*SUMy - n*SUMxy ) / ( SUMx*SUMx -
n*SUMxx )
 *  y-intercept (b) = ( SUMy - slope*SUMx ) / n
 *
 * PROGRAM DESCRIPTION:
 *  o This program computes a linear model for a set of given data.
 *  o Data is read from a file; the first line is the number of data
 *    points (n), followed by the coordinates of x and y.
 *  o Data points are divided amongst processes such that each process
 *    has naverage = n/numprocs points; remaining data points are
 *    added to the last process.
 *  o Each process calculates the partial sums
 *   (mySUMx,mySUMy,mySUMxy,
 *    mySUMxx) independently, using its data subset. In the final step,
 *    the global sums (SUMx,SUMy,SUMxy,SUMxx) are calculated to
 *    find the least-squares line.
 *  o For the purpose of this exercise, communication is done strictly
 *    by using the MPI point-to-point operations,
 *    MPI_SEND and MPI_RECV.
 *
```

```
 * USAGE: Tested to run using 1,2,...,10 processes.
 *
 * AUTHOR: Dora Abdullah (MPI version, 11/96)
 * LAST REVISED: RYL converted to C (12/11)
 * ---------------------------------------------------------------------- */
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv) {
  double *x, *y;
  double mySUMx, mySUMy, mySUMxy, mySUMxx, SUMx, SUMy,
SUMxy,
        SUMxx, SUMres, res, slope, y_intercept, y_estimate;
  int i,j,n,myid,numprocs,naverage,nremain,mypoints,ishift;
  /*int new_sleep (int seconds);*/
  MPI_Status istatus;
  FILE *infile;
  infile = fopen("xydata", "r");
  if (infile == NULL) printf("error opening file\n");

  MPI_Init(&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myid);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  /* ----------------------------------------------------------
   * Step 1: Process 0 reads data and sends the value of n
   * ---------------------------------------------------------- */
  if (myid == 0) {
    printf ("Number of processes used: %d\n", numprocs);
    printf ("------------------------------------\n");
    printf ("The x coordinates on worker processes:\n");
    /* this call is used to achieve a consistent output format */
    /* new_sleep (3);*/
    fscanf (infile, "%d", &n);
    x = (double *) malloc (n*sizeof(double));
    y = (double *) malloc (n*sizeof(double));
    for (i=0; i<n; i++)
      fscanf (infile, "%lf %lf", &x[i], &y[i]);
```

```
   for (i=1; i<numprocs; i++)
     MPI_Send (&n, 1, MPI_INT, i, 10, MPI_COMM_WORLD);
  }
  else {
    MPI_Recv (&n, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
&istatus);
    x = (double *) malloc (n*sizeof(double));
    y = (double *) malloc (n*sizeof(double));
  }
  /* ------------------------------------------------------- */
   naverage = n/numprocs;
  nremain = n % numprocs;
  /* ------------------------------------------------------
   * Step 2: Process 0 sends subsets of x and y
!  * ------------------------------------------------------- */
  if (myid == 0) {
    for (i=1; i<numprocs; i++) {
      ishift = i*naverage;
      mypoints = (i < numprocs -1) ? naverage : naverage + nremain;
      MPI_Send (&ishift, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
      MPI_Send (&mypoints, 1, MPI_INT, i, 2, MPI_COMM_WORLD);
      MPI_Send (&x[ishift], mypoints, MPI_DOUBLE, i, 3,
MPI_COMM_WORLD);
      MPI_Send (&y[ishift], mypoints, MPI_DOUBLE, i, 4,
MPI_COMM_WORLD);
    }
  }
  else {
    /* ---------------the other processes receive---------------- */
    MPI_Recv (&ishift, 1, MPI_INT, 0, 1, MPI_COMM_WORLD,
&istatus);
    MPI_Recv (&mypoints, 1, MPI_INT, 0, 2, MPI_COMM_WORLD,
&istatus);
    MPI_Recv (&x[ishift], mypoints, MPI_DOUBLE, 0, 3,
MPI_COMM_WORLD,
                &istatus);
    MPI_Recv (&y[ishift], mypoints, MPI_DOUBLE, 0, 4,
MPI_COMM_WORLD, &istatus);
```

```
  printf ("id %d: ", myid);
  for (i=0; i<n; i++) printf("%4.2lf ", x[i]);
  printf ("\n");
  /* ------------------------------------------------------- */
 }

 /* ------------------------------------------------------
  * Step 3: Each process calculates its partial sum
  * ------------------------------------------------------ */
 mySUMx = 0; mySUMy = 0; mySUMxy = 0; mySUMxx = 0;
 if (myid == 0) {
  ishift = 0;
  mypoints = naverage;
 }
 for (j=0; j<mypoints; j++) {
  mySUMx = mySUMx + x[ishift+j];
  mySUMy = mySUMy + y[ishift+j];
  mySUMxy = mySUMxy + x[ishift+j]*y[ishift+j];
  mySUMxx = mySUMxx + x[ishift+j]*x[ishift+j];
 }

 /* ------------------------------------------------------
  * Step 4: Process 0 receives partial sums from the others
  * ------------------------------------------------------ */
 if (myid != 0) {
  MPI_Send (&mySUMx, 1, MPI_DOUBLE, 0, 5,
MPI_COMM_WORLD);
  MPI_Send (&mySUMy, 1, MPI_DOUBLE, 0, 6,
MPI_COMM_WORLD);
  MPI_Send (&mySUMxy,1, MPI_DOUBLE, 0, 7,
MPI_COMM_WORLD);
  MPI_Send (&mySUMxx,1, MPI_DOUBLE, 0, 8,
MPI_COMM_WORLD);
        }
 else {
  SUMx = mySUMx; SUMy = mySUMy;
  SUMxy = mySUMxy; SUMxx = mySUMxx;
  for (i=1; i<numprocs; i++) {
```

```
    MPI_Recv (&mySUMx, 1, MPI_DOUBLE, i, 5,
MPI_COMM_WORLD, &istatus);
    MPI_Recv (&mySUMy, 1, MPI_DOUBLE, i, 6,
MPI_COMM_WORLD, &istatus);
    MPI_Recv (&mySUMxy,1, MPI_DOUBLE, i, 7,
MPI_COMM_WORLD, &istatus);
    MPI_Recv (&mySUMxx,1, MPI_DOUBLE, i, 8,
MPI_COMM_WORLD, &istatus);
     SUMx = SUMx + mySUMx;
     SUMy = SUMy + mySUMy;
     SUMxy = SUMxy + mySUMxy;
     SUMxx = SUMxx + mySUMxx;
   }
  }

  /* ------------------------------------------------------------
   * Step 5: Process 0 does the final steps
   * ------------------------------------------------------------ */
  if (myid == 0) {
   slope = ( SUMx*SUMy - n*SUMxy ) / ( SUMx*SUMx - n*SUMxx
);
   y_intercept = ( SUMy - slope*SUMx ) / n;
   /* this call is used to achieve a consistent output format */
   /*new_sleep (3);*/
   printf ("\n");
   printf ("The linear equation that best fits the given data:\n");
   printf ("       y = %6.2lfx + %6.2lf\n", slope, y_intercept);
   printf ("------------------------------------------------\n");
   printf ("   Original (x,y)     Estimated y     Residual\n");
   printf ("------------------------------------------------\n");

   SUMres = 0;
   for (i=0; i<n; i++) {
    y_estimate = slope*x[i] + y_intercept;
    res = y[i] - y_estimate;
    SUMres = SUMres + res*res;
    printf ("   (%6.2lf %6.2lf)      %6.2lf       %6.2lf\n",
    x[i], y[i], y_estimate, res);
```

```
    }
    printf("-------------------------------------------------\n");
    printf("Residual sum = %6.2lf\n", SUMres);
  }
  /* ------------------------------------------------------      */
  MPI_Finalize();
}
15
0.1  6.5
0.3  6.0
0.5  5.6
0.55 5.5
0.67 5.08
0.93 4.25
1.15 3.52
1.2  3.4
1.38 3.15
1.4 3.0
1.5 2.53
1.7  2.2
1.8  2.0
1.98 1.46
2.0 1.04
```

Это упражнение было разработано, чтобы представить вам множество возможностей по параллельному программированию, особенно в областях попарной коммуникации и декомпозиции данных. Вы можете начать с программы least-squares-pt2pt.c и проделать каждую из нижеследующих задач независимо. Альтернативно вы можете начать с программы least-squares-pt2pt.c и постепенно, шаг за шагом, создать программу, которая содержит решение всех задач.

Прочитайте внимательно всю программу и попытайтесь хорошо разобраться в ее алгоритме. Посмотрите на использование блокирующих вызовов для отправок и получений и на то, как данные декомпозируются.

Есть в наличии также последовательная версия этой программы least-squares.c.

```
/* ------------------------------------------------------------------
 * FILE: least-squares.c
 * This program computes a linear model for a set of given
data.
 *
 * PROBLEM DESCRIPTION:
 *  The method of least squares is a standard technique used to
find
 *  the equation of a straight line from a set of data. Equation
for a
 *  straight line is given by
 *         y = mx + b
 *  where m is the slope of the line and b is the y-intercept.
 *
 *  Given a set of n points {(x1,y1), x2,y2),...,xn,yn)}, let
 *     SUMx = x1 + x2 + ... + xn
 *     SUMy = y1 + y2 + ... + yn
 *     SUMxy = x1*y1 + x2*y2 + ... + xn*yn
 *     SUMxx = x1*x1 + x2*x2 + ... + xn*xn
 *
 *  The slope and y-intercept for the least-squares line can be
 *  calculated using the following equations:
 *              slope (m) = ( SUMx*SUMy - n*SUMxy ) / (
SUMx*SUMx - n*SUMxx )
 *  y-intercept (b) = ( SUMy - slope*SUMx ) / n
 *
 * AUTHOR: Dora Abdullah (Fortran version, 11/96)
 * REVISED: RYL (converted to C, 12/11/96)
 * ------------------------------------------------------------------ */
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char **argv) {

  double *x, *y;
  double SUMx, SUMy, SUMxy, SUMxx, SUMres, res, slope,
```

```
        y_intercept, y_estimate ;
    int i,n;
    FILE *infile;

    infile = fopen("xydata", "r");
    if (infile == NULL) printf("error opening file\n");
    fscanf (infile, "%d", &n);
    x = (double *) malloc (n*sizeof(double));
    y = (double *) malloc (n*sizeof(double));

    SUMx = 0; SUMy = 0; SUMxy = 0; SUMxx = 0;
    for (i=0; i<n; i++) {
      fscanf (infile, "%lf %lf", &x[i], &y[i]);
      SUMx = SUMx + x[i];
      SUMy = SUMy + y[i];
      SUMxy = SUMxy + x[i]*y[i];
      SUMxx = SUMxx + x[i]*x[i];
    }
    slope = ( SUMx*SUMy - n*SUMxy ) / ( SUMx*SUMx -
n*SUMxx );
    y_intercept = ( SUMy - slope*SUMx ) / n;

    printf ("\n");
    printf ("The linear equation that best fits the given data:\n");
    printf ("       y = %6.2lfx + %6.2lf\n", slope, y_intercept);
    printf ("--------------------------------------------------\n");
    printf ("   Original (x,y)     Estimated y     Residual\n");
    printf ("--------------------------------------------------\n");

    SUMres = 0;
    for (i=0; i<n; i++) {
      y_estimate = slope*x[i] + y_intercept;
      res = y[i] - y_estimate;
      SUMres = SUMres + res*res;
      printf ("   (%6.2lf %6.2lf)      %6.2lf       %6.2lf\n",
                x[i], y[i], y_estimate, res);
    }
    printf("--------------------------------------------------\n");
```

```
  printf("Residual sum = %6.2lf\n", SUMres);
}
```

Из-за способа, каким параллельное окружение управляет сигналом, стандартную функцию sleep() нельзя использовать. Поэтому вызывается функция new_sleep, содержащаяся в файле new_sleep.c.

```
#if defined(_AIX)
#include <errno.h>
#include <sys/time.h>
int new_sleep (int *amount)
{
    struct timestruc_t Requested, Remaining;
    double famount = *amount;
    int rc;
    while (famount > 0.0) {
        Requested.tv_sec = (int) famount;
        Requested.tv_nsec =
            (int) ((famount - Requested.tv_sec)*1000000000.);
        rc = nsleep ( &Requested, &Remaining );
        if ((rc == -1) && (errno == EINTR)) {
            /* Sleep interrupted.  Resume it */
            famount = Remaining.tv_sec + Remaining.tv_nsec /
                                        1000000000.;
            continue;
        }
        else /* Completed sleep.  Set return to zero */
        {
                return (0);
        }
    }   /* end of while */

    /* famount = 0.; exit */
    return (0);
}
```

```
#elif defined(__INTEL_COMPILER) || (__PGI)
#include <unistd.h>
int new_sleep (int *amount)
{
  unsigned int secs = *amount, remain;
  remain=sleep(secs);
  while(remain) {
    remain=sleep(remain);
  }

  return 0;
}
int new_sleep_(int *amount)
{
  return(new_sleep(amount));
}
/* A simulated version of IBMs irtc() */
#include <sys/time.h>
long int irtc() {
  struct timeval tv;
    gettimeofday ( &tv, (struct timezone*)0 );
    return 1000*( tv.tv_sec * 1000000 + tv.tv_usec);
}
long int irtc_() {
  return irtc();
}
#else
#error No new_sleep function defined
#endif
```

Откомпилируйте программу.

```
cc -c new_sleep.c
mpicc least-squares-pt2pt.c new_sleep.o -o least-squares-pt2pt.exe
```

Запустите программу least-squares-pt2pt.exe с различным числом процессов (от 2 до 10). Обратите внимание, как экземпляры данных распределяются по процессам.

Все нижеследующие программы должны быть откомпилированы тем же способом, как программа «least-squares-pt2pt.exe», тем не менее объектный код «new_sleep.o» следует создавать только однажды.

Чтобы попрактиковаться в использовании неблокирующих отправки и получения, замените блокирующие вызовы отправки и получения в least-squares-pt2pt.c на неблокирующие вызовы.

Файл решения на C запишите в: pt2pt-nblk-comm.c

Программа нуждается в некотором улучшении при работе с балансированием загрузки. Это так, потому что последнему процессу дается наибольшее число экземпляров данных, когда их не распределяют поровну по числу процессов.

Перепишите шаги 2 и 3 программы least-squares-pt2pt.c, чтобы обеспечить наиболее равномерное распределение экземпляров данных по процессам. Когда существуют избыточные данные, некоторые процессы должны иметь n/numprocs, а некоторые иметь (n/numprocs) + 1 экземпляров данных. Заметим: предполагается целое деление для n/numprocs.

Файл решения на C запишите в: pt2pt-data-decomp.c.

На шаге 4 программы f least-squares-pt2pt.c процесс 0 получает все частные суммы от других процессов. Перепишите шаг 4, используя двоичное дерево, как описано ниже.

Используйте число процессов, которое равняется двум в некоторой степени, например, 2, 4, или 8.

Разделите процессы на две группы. Тогда каждый процесс из второй группы посылает его частную сумму некоторому процессу в первой группе. Первая группа затем разделяется на две, и этот шаг повторяется, пока процесс 0 не получит все частные суммы (предполагая, что процесс 0 находится в первой группе). Нажмите здесь (http://www.ccas.ru/mmes/educat/lab04k/gifs/bi-tree.gif), чтобы получить иллюстрацию этого метода при использовании 8 процессов.

Заметим, что это может быть сделано с библиотекой коллективных коммуникаций.

Файл решения на C запишите в: pt2pt-bi-reduc.c.

Как возможное (но необязательное) упражнение скомбинируйте все зафиксированное в одну программу.

Файл решения на C запишите в: pt2pt-combo.c.

5. Попарный обмен сообщениями в MPI, II

Изложение данной главы следует электронному курсу, представленному в Интернет на сайте

http://www.ccas.ru/olenev/educat/lab04/03/p2pCommII.html.

Для ознакомления со стандартом MPI можно ознакомиться с соответствующим форумом [10].

5.1. Обзор

Начнем с обзора альтернатив программирования, представленных в этом модуле.

Тупик

Ситуация тупика (дэдлока) часто встречается в параллельной обработке. Это возникает тогда, когда два или более процесса соревнуются за один и тот же набор ресурсов. В типичном сценарии в коммуникацию вовлечены два процесса, желающих обменяться сообщениями, причем оба пытаются отдать их противоположному, хотя ни один еще не готов принять сообщение. Здесь описан ряд стратегий, помогающих избежать такого рода проблемы, встречающиеся в приложениях.

Проверка и исполнение по «состоянию» коммуникаций

При вовлечении в неблокирующие транзакции (пересылки) вызов функций wait (подождать), test (проверить) и probe (прозондировать) дает приложению возможность запросить статус (status) отдельных сообщений или проверить любое сообщение, обладающее определенным набором характеристик, без извлечения любой завершенной транзакции из очереди.

Проверка информации (статуса), возвращаемой из транзакции, позволяет приложению произвести, например, корректирующие действия, когда случилась ошибка.

Использование специальных параметров для специальных случаев

Джокеры – MPI_ANY_SOURCE и MPI_ANY_TAG

Использование вызовов с джокерами позволяет получающему процессу конкретизировать получаемое сообщение, используя джокер для указания либо отправителя, либо типа сообщения.

Пустые процессы и запросы

Специальные пустые параметры могут упростить кодирование приложений, делая структуры данных регулярными.

5.2. Основной тупик

Явление тупика (дэдлока) наиболее распространено при блокирующих коммуникациях. Тупик случается, когда все задачи ожидают событий, которые еще не были инициализированы.

Можно рассмотреть две задачи типа SPMD (единственная программа – множество данных). Возникает, например, ситуация, когда обе задачи вызывают блокирующие стандартные отправки в одной и той же точке программы. Спаренное с отправкой получение у каждой задачи происходит позже в другой программе этой задачи.

Простейший пример тупика: каждая отправка из двух ждет соответствующего ей получения, чтобы завершиться, но эти получения исполняются после отправок, так что если отправки не завершены и не вернулись, то получения никогда не смогут исполниться и оба набора коммуникаций подвиснут на неопределенное время.

Более сложный пример тупика может произойти, если размер сообщения больше порога; тупик произойдет из-за того, что ни одна задача не может согласоваться во времени (синхронизироваться) с соответствующим ей получением. Тупик все же может произойти и в случае, когда размер сообщения не превышает порога, если нет достаточного места в системном буфере. Обе задачи будут ожидать получения, чтобы переписать данные сообщения из системного буфера, но эти получения не могут исполниться, так как обе задачи заблокированы в отправке.

Решения для исключения тупиков

Есть четыре способа избежать тупика.

1. Различный порядок вызовов у задач

Организуйте порядок, в котором одна задача объявляет первой свое получение, а вторая объявляет первой свою отправку. Это прояснение даст порядок, в котором сообщение в одном из направлений проходит раньше, чем в другом.

2. Неблокирующие вызовы

Заставляют каждую задачу объявить неблокирующее получение до того, как она произведет какую-нибудь другую коммуникацию. Это дает возможность получить каждое сообщение независимо от того, над чем задача трудится, когда сообщение прибывает, и независимо от порядка, в котором отправки объявлены.

3. MPI_Sendrecv

MPI_Sendrecv_replace

Использование MPI_Sendrecv – это элегантное решение, использующее возможности самой библиотеки MPI для исключения тупика. В этой версии используется два буфера: один для отправляемого сообщения, а другой для получаемого. В версии _replace система выделяет некоторое единственное буферное пространство (не зависящее от порогового предела) для обработки обмена сообщениями. Отправленное сообщение в этом буфере замещается полученным.

Заметим здесь, что обмен с процессом, который имеет пустое значение MPI_PROC_NULL, не дает результата, а отправка на пустой процесс MPI_PROC_NULL всегда успешна.

4. Буферизованный способ

Используйте буферизованную отправку, чтобы можно было осуществлять вычисления после копирования отправляемого сообщения в пользовательский буфер. Это дает возможность исполнения получений.

5.3. Определение информации о сообщении

В приложениях можно использовать специальные вызовы для контроля положения дел у незавершенных транзакций или транзакций, состояние которых неизвестно, без необходимости

завершения этих операций (это аналогично ожиданию информации о том, пришло ли тетино письмо, не заботясь о его содержании). Приложение можно запрограммировать так, чтобы знать состояние его коммуникаций, и поэтому продуманно действовать в различных ситуациях.

5.3.1. Wait, Test и Probe

Имея успешно расцепленные коммуникации по вашей основной вычислительной нити, вам тем не менее нужно быть способным как к получению информации о статусе коммуникационной транзакции, так и к систематизации, чтобы осуществить определенные действия, относящиеся к найденным вами условиям. Три вызова, описанные здесь, принадлежат к тем, что обычно наиболее полезны для разрешения этих типов ситуаций.

MPI_Wait

Полезен для обоих участников неблокирующей коммуникации: отправителя и получателя.

Блокирующие коммуникации включают автоматический вызов wait, поэтому невозможно увидеть нетривиальный вызов при использовании таких операций. В обоих случаях: и в случае отправки, и в случае получения (для неблокирующих операций), – вызывающий процесс откладывает функционирование до тех пор, пока работа, заданная ссылкой wait, не завершится, и только после этого исполнение возобновится в вызывающем процессе.

Под управлением программиста получающий процесс заблокирован, пока сообщение принимается.

На стороне получателя процесс уже объявил неблокирующее получение, которое будет завершено независимо от того, что вызывающий процесс делает. Программист поэтому способен определить, совершено или нет какое-либо полезное вычисление, перед тем, как запрашивается информация в еще не полученном сообщении. Если есть в наличии полезная работа, приложение определенно приобретет эффективность, совершая эту работу, пока сообщение все еще в пути. В некоторой точке, конечно, сообщение понадобится, и wait будет использован.

Отправляющий процесс заблокирован до тех пор, пока операция отправки завершается, к этому времени буфер сообщения способен к новому использованию.

На стороне отправки процесс также освобождается от транзакции, за исключением того факта, что он принужден не записывать что-либо в этот особый буфер до тех пор, пока его текущее использование не завершится. Если отправитель пытается положить некоторое другое сообщение в этот буфер до того, как предшествующая пересылка завершится, результаты не будут определены и будут основаны исключительно на том, какая часть процесса была в обработке, когда произошла перезапись. Осуществление wait на сообщение гарантирует, что повторное использование буфера не будет деструктивным.

MPI_Test

Используется обоими участниками неблокирующей коммуникации: отправителем и получателем.

Там, где wait задерживает исполнение до тех пор, пока операция завершится, test возвращается немедленно с информацией о ее текущем состоянии.

Получатель проверяет, чтобы удостовериться, что определенный отправитель отослал сообщение, которое ожидает свою передачу, при этом сообщения от всех других отправителей игнорируются.

Вызов test возвращает значение true только в том случае, когда отправитель задал в объекте проверку на то, отослано ли некое сообщение, которое теперь находится в очереди на передачу, при этом поток от всех прочих источников игнорируется.

Отправителю надо выяснить, может ли буфер сообщения быть заново использован, но при этом он должен подождать, пока операция завершится.

На стороне отправителя test является неблокирующим аналогом wait, предоставляя приложению знание текущего состояния без требования его блокировки до завершения, что позволяет приложению делать другую работу, если какая-нибудь есть.

MPI_Probe

Получатель извещается, когда (т.е. это некий блокирующий вызов) сообщения от потенциально любого отправителя прибывают и готовы быть выполнены.

Все предыдущие вызовы наметили конкретные сообщения и отправителей; вызов probe может быть предназначен к возврату «распределяемой» информации, касающейся как любого отправителя, так и конкретного.

5.3.2. Статус (Status)

Статус возвращает источник, тег и ошибку (в стандартном случае).

Статус – это объект, на который смотрят, чтобы определить информацию об источнике и теге сообщения и любой ошибке, которой может быть подвергнут коммуникационный вызов. В C они возвращаются как status.MPI_SOURCE, status.MPI_TAG и status.MPI_ERROR. В случае отправок информация о статусе, скорее всего, не отличается от той, что была в вызове, так что ее редко используют.

MPI_Get_count возвращает число элементов

Это может быть полезно, если вы распределили буфер получателя, который может быть больше, чем входящее сообщение, или если вы хотите узнать длину сообщения, идентифицированного MPI_Probe.

Условия, в которых имеет смысл проверять статус

Эта информация, которую представляет статус, становится очень удобной, когда имеют дело со следующей ситуацией, предлагаемой без обсуждения, как изменяющиеся подробности, уже охваченные, где бы то ни было (за исключением MPI_ANY_TAG и MPI_ANY_SOURCE, которые будут кратко описаны):

Блокирующее получение или ожидание, когда MPI_ANY_TAG или MPI_ANY_SOURCE должны быть использованы:

- MPI_ANY_TAG, принимает сообщение с любым значение тега;

- MPI_ANY_SOURCE, принимает сообщение от любого источника;
- MPI_Probe или MPI_Iprobe для получения информации по приходящим сообщениям:
- MPI_Probe – блокирующий тест для некоторого сообщения;
- MPI_Iprobe – неблокирующий тест для некоторого сообщения;

MPI_Test служит для того, чтобы узнать завершена ли коммуникация.

5.4. Специальные параметры

Джокеры (заменители) – MPI_ANY_SOURCE и MPI_ANY_TAG.

Родовой термин, имеющий значение «что-либо, отвечающее очень общему множеству характеристик». MPI_ANY_SOURCE позволяет получателю получить сообщения от любого отправителя, и MPI_ANY_TAG позволяет получателю получить любой тип сообщения от отправителя.

Пустые (фиктивные) процессы и запросы

Приложения часто имеют дело с регулярными структурами данных, такими как прямоугольные гипермассивы, и выполняют один и тот же тип коммуникаций везде внутри них за исключением краев, для которых приходится писать специальный код, чтобы не осуществлять коммуникации там, где нет действительных «соседей» для отправки или получения; специальные пустые параметры перемещают логику этого из пользовательского кода в систему, упрощая создание приложения.

В реализации MPICH стандартный, синхронный и буферизованный способы работают так, как предписано стандартом.

5.5. Рекомендации по программированию

Избегайте тупика посредством продуманного размещения вызовов отправок/получений или ранним объявлением неблокирующих получений.

Если вы выбрали использование блокирующих транзакций, пытайтесь обеспечить исключение тупика посредством

тщательной разработки вашей коммуникационной стратегии, в которой отправки и получения надлежащим образом спарены в необходимом порядке; в противном случае, объявите неблокирующие получения настолько рано, насколько это возможно, так что отправки будут простаивать в системе так мало времени, как это необходимо.

Используйте уместный вызов «состояния операции» («wait», «test» или «probe»), чтобы держать под контролем функционирование вызовов неблокирующих коммуникаций.

Корректное знание состояния коммуникационных транзакций позволяет приложению продуманно управляться с работой, улучшая эффективность использования имеющихся циклов. В конце концов подвешенный обмен должен быть принят, но такое действие может долго быть в пути и многие транзакции, возможно, могут быть выполнены, хотя и не завершены. Вызовы wait, test и probe позволяют приложению соединить соответствующую деятельность с индивидуальной ситуацией.

Проверьте значения полей «статуса (status)» для сообщения проблемы.

Не предполагайте просто, что все пройдет гладко – сделайте общим правилом проверку на успех любой транзакции и то, чтобы фатальные ошибки немедленно сообщались, и так много относящейся к делу информации, насколько возможно, развивается и делается доступным для отладки.

Продуманное использование джокеров может значительно упростить логику и кодирование.

Используя общие параметры, получения способны обработать более чем один тип потока сообщений (в терминах либо отправителя, либо типа сообщения, либо обоих), что может значительно упростить структуру вашего приложения и потенциально может дать экономию на системных ресурсах (если у вас есть привычка использовать уникальный буфер сообщения для каждой транзакции).

Пустые (фиктивные) процессы и запросы перемещают тесты вовне пользовательского кода.

Использование MPI_PROC_NULL и MPI_REQUEST_NULL не дает освобождения от граничных

тестов, а просто позволяет программисту использовать вызов, который будет проигнорирован системой.

5.6. Вопросы: MPI попарные коммуникации II

1. Отметьте все, что подходит. Какие из следующих утверждений о тупике (дэдлоке) справедливы?

- Тупик не может никогда случиться для сообщения, которое буферизуется.
- Тупик не может никогда случиться для неблокирующей отправки, укомплектованной функцией ожидания (wait).
- Тупик не может никогда случиться на отправке по готовности.

2. Некое единственное сообщение отправляется между двумя процессами. После того, как получатель вызывает функцию MPI_Irecv, какой вызов из приведенных не подходит?

- A. MPI_Cancel.
- B. MPI_Probe.
- C. MPI_Test.
- D. MPI_Wait.

3. Два вызова MPI_Irecv осуществляются при точно заданных отличающихся буферах и тегах, но при одинаковом местоположении отправителя и запроса. Как можно определить, что буфер, заданный в первом вызове, имеет верные данные?

- A. Вызвав MPI_Probe.
- B. Вызвав MPI_Testany с тем же самым запросом, перечисленным дважды.
- C. Вызвав MPI_Wait дважды с тем же запросом.
- D. Посмотрев на данные в буфере и пытаясь определить, являются ли они отличающимися от тех, что были там до того, как MPI_Irecv был запущен.

4. Размер объекта MPI_Status равен:

- A. 1 байту.
- B. 8 байтам.
- C. числу, не зависящему от реализации.
- D. числу, заданному константой MPI_STATUS_SIZE.

5. После того, как сообщение было получено функцией MPI_Recv, можно определить, как много элементов было в действительности получено:

- A. посредством вызова MPI_Probe.
- B. из аргумента «число (count)» функции MPI_Recv.
- C. из декларированного размера буфера получателя.
- D. посредством вызова MPI_Get_count.

6. Один из ваших процессов является «мастером» и раздает работу другим процессам. Он посылает MPI_Irecv с MPI_ANY_TASK для завершения сообщений и начинает делать некоторую работу сам. Какой вызов вам следует делать периодически, чтобы выяснить закончили ли какие-либо процессы работу?

- A. MPI_Iprobe.
- B. MPI_Probe.
- C. MPI_Test.
- D. MPI_Wait.

7. Какие из следующих функций являются частью библиотеки MPI?

- A. MPI_Iprobe.
- B. MPI_Itest.
- C. MPI_Iwait.
- D. MPI_Uwait.

Упражнения

Предварительные требования

Работу следует делать после изучения MPI основы попарного обмена сообщениями II. Прежде чем приступить к этой работе вы должны завершить модуль и лабораторную работу по основам программирования в MPI.

Цели

Настоящая лабораторная работа познакомит вас с четырьмя способами коммуникации, имеющимися в MPI для попарного обмена сообщениями (синхронного, по готовности, буферизованного и стандартного).

Упражнение 1
Файл на С для работы:

blocksends.c

```
/* ---------------------------------------------------------------------
* Code:   blocksends.c
* Lab:    MPI Point-to-Point Communication
Time message sends that use the four blocking communication
modes:
synchronous, ready, buffered, standard
Usage:  blocksends <message_length_in_number_of_floats>
Run on two nodes.
Author: Roslyn Leibensperger  Last revised: 8/29/95 RYL
---------------------------------------------------------------------*/

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#define NMSG 4
#define SYNC 0
#define READY 1
#define BUF 2
#define STANDARD 3
#define BUF_ALLOC 4
#define GO 10
void print_time ( int event, double tbegin, double tend );

main( argc, argv )
int argc;
char **argv;
{
float  *message  [NMSG];              /* pointers to messages
*/
float *buffer;                  /* buffer required by buffered send
*/
int rank,               /* rank of task in communicator        */
count,                  /* number of elements in message       */
dest, source,           /* rank in communicator of destination  */
```

```
        /* and source tasks                    */
        tag[NMSG], etag,          /* message tags                    */
        mlen,                     /* message length specified by user     */
        bsize,                    /* buffer size for buffered send, bytes */
        i, j;
        double tbegin [NMSG+1],   /* used to measure elapsed time */
        tend [NMSG+1];
        MPI_Comm comm;            /* communicator                    */
        MPI_Datatype datatype;            /* type of data in message
*/
        MPI_Request request[NMSG]; /* handle for pending
        communication     */
        MPI_Status status[NMSG];     /* status of communication   */

        if (argc != 2) {
        printf ( " Usage: blocksends
        <message_length_in_number_of_floats>\n" );
        return -1;
        }

        /* -----------------------------------------------------------------
        do initial housekeeping: allocate memory for messages,
        initialize program with MPI, define message tags
        ----------------------------------------------------------------- */

        mlen = atoi (argv[1]);
        for (i=0; i<NMSG; i++)
        /* message[i] = (float *)MPIF_Malloc ( mlen * sizeof(float) );
*/
        message[i] = (float *)malloc ( mlen * sizeof(float) );

        MPI_Init( &argc, &argv );
        MPI_Comm_rank( MPI_COMM_WORLD, &rank );
        printf ( " Message size = %5d floats\n", mlen );
        printf ( " Task %d initialized\n", rank );

        tag[SYNC] = SYNC;
        tag[READY] = READY;
```

```
tag[BUF] = BUF;
tag[STANDARD] = STANDARD;

/*  ------------------------------------------------------------------
task 0 will receive a ready message, and then send 4 messages
using the different communication modes
------------------------------------------------------------------ */
if ( rank == 0 )  {
/* initialize all message contents */
for ( i=0; i<NMSG; i++ )
for ( j=0; j<mlen; j++ )
message[i][j] = (float)i;

/* receive empty message, indicating all receives have been
posted */
count = 0;
datatype = MPI_INT;
source = 1;
etag = GO;
comm = MPI_COMM_WORLD;
MPI_Recv ( (int*)0, count, datatype, source, etag, comm,
&status[1] );
printf( " Ready to send messages\n" );

/* these message parameters apply to all succeeding sends by
task 0 */
count = mlen;
datatype = MPI_FLOAT;
dest = 1;
comm = MPI_COMM_WORLD;

/* time a synchronous send */

tbegin[SYNC]= MPI_Wtime();
MPI_Ssend ( message[SYNC], count, datatype, dest,
tag[SYNC], comm );
tend[SYNC]= MPI_Wtime();
print_time ( SYNC, tbegin[SYNC], tend[SYNC] );
```

```
/* time a ready send */

tbegin[READY]= MPI_Wtime();
MPI_Rsend ( message[READY], count, datatype, dest,
tag[READY], comm );
tend[READY]= MPI_Wtime();
print_time ( READY, tbegin[READY], tend[READY] );

/* time overhead for buffer allocation for buffered send */

tbegin[BUF_ALLOC]= MPI_Wtime();
bsize = count * sizeof (float) + 100;
/*    buffer = (float *)MPIF_Malloc ( bsize ); */
buffer = (float *)malloc ( bsize );
MPI_Buffer_attach ( buffer, bsize );
tend[BUF_ALLOC]= MPI_Wtime();
print_time ( BUF_ALLOC, tbegin[BUF_ALLOC],
tend[BUF_ALLOC] );

/* time a buffered send */

tbegin[BUF]= MPI_Wtime();
MPI_Bsend ( message[BUF], count, datatype, dest, tag[BUF],
comm );
tend[BUF]= MPI_Wtime();
print_time ( BUF, tbegin[BUF], tend[BUF] );
MPI_Buffer_detach ( buffer, &bsize );

/* time a standard send */
tbegin[STANDARD]= MPI_Wtime();
MPI_Send ( message[STANDARD], count, datatype, dest,
tag[STANDARD],
comm );
tend[STANDARD]= MPI_Wtime();
print_time ( STANDARD, tbegin[STANDARD],
tend[STANDARD] );
}
```

```
/*  ------------------------------------------------------------------
task 1 will post 4 receives, and then send a "ready" message
------------------------------------------------------------------ */

else if ( rank == 1 )  {
count = mlen;
datatype = MPI_FLOAT;
source = 0;
comm = MPI_COMM_WORLD;

/* post non-blocking receives */
for (i = 0; i < NMSG; i++)
MPI_Irecv ( message[i], count, datatype, source, tag[i],
comm, &request[i] );

/* send ready message indicating all receives have been posted
*/
count = 0;
datatype = MPI_INT;
dest = 0;
etag = GO;

MPI_Send ( (int *)0, count, datatype, dest, etag, comm );

/* wait for all receives to complete */
MPI_Waitall ( NMSG, request, status );
}
MPI_Finalize();
return 0;
}

/* ------------------------------------------------------------------
calculate elapsed time and print results
------------------------------------------------------------------ */
void print_time ( int event, double tbegin, double tend )
{
int dt;
```

```
        dt = (int)1000000.*(tend - tbegin);

        switch (event)  {
        case SYNC:
        printf( " Elapsed time for synchronous send = %8d uSec\n", dt
);
        break;
        case READY:
        printf( " Elapsed time for ready send = %8d uSec\n", dt );
        break;
        case BUF:
        printf( " Elapsed time for buffered send = %8d uSec\n", dt );
        break;
        case STANDARD:
        printf( " Elapsed time for standard send = %8d uSec\n", dt );
        break;
        case BUF_ALLOC:
        printf( " Elapsed time for buffer allocation = %8d uSec\n", dt );
        break;
        default:
        break;
        }
        }
```

cbrcv.mak - C Makefile;

```
        #
        CC = mpicc.exe
        #
        all: blocksends.exe brecv.exe nbrecv.exe

        blocksends.exe:  blocksends.c new_sleep.obj
                $(CC)  blocksends.c  new_sleep.obj

        brecv.exe:  brecv.c new_sleep.obj
                $(CC)  brecv.c  new_sleep.obj

        nbrecv.exe:  nbrecv.c new_sleep.obj
                $(CC)  nbrecv.c  new_sleep.obj
```

```
new_sleep.obj:  new_sleep.c
        $(CC)  -c new_sleep.c

clean:
        del /Q *.obj *.exe *.mod
```

Это упражнение касается одного кусочка информации, которая относится к относительной эффективности четырех способов коммуникации (синхронного, по готовности, буферизованного и стандартного): минимального времени, потраченного на вызов блокирующей отправки. Заметим, что это не является мерой времени для завершения коммуникации или даже мерой полной системной накладки.

Программа blocksends сообщает истекшее (wallclock) время, потраченное на вызовы блокирующих отправок для четырех способов коммуникации. Все получатели вызваны (объявлены) до того, как какие-либо сообщения отосланы. Отличающиеся относительные времена могут случиться, если получатели не объявлены первыми.

1. Прочитайте текст программы и отметьте аналогию синтаксиса между вызовами отправки, дополнительные шаги нужны для осуществления буферизованной отправки и использования неблокирующих получений.

2. Откомпилируйте программу, используя Makefile:

для C: nmake /f cbrcv.mak blocksends.exe

3. Запустите программу на 2 процессорах, используя различные длины сообщений. Команды таковы:

mpirun -np 2 blocksends.exe

message_length_in_number_of_floats(reals)

Отметим время, потраченное на блокирующую отправку для различных способов коммуникации. Вам может понадобиться несколько запусков, чтобы получить представительные (репрезентативные) длительности.

Упражнение 2

Файлы для работы на C:

brecv.c - исходный файл.

```
#include <stdio.h>
#include "mpi.h"
#define TAG 100
void print_time ( double tbegin, double tend );
main( argc, argv )
    int argc;
    char **argv;
{
  float *message;    /* message buffer                  */
  int rank,                /* rank of task in communicator */
      i;
  int mlen;  /* dimmension of the mesage */
  MPI_Status status;         /* status of communication      */
  double tbegin, tend;       /* used to measure elapsed time */

  if (argc != 2) {
    printf ( " Usage: blocksends
<message_length_in_number_of_floats>\n" );
    return -1;
  }
  /* ----------------------------------------------------------------
   * do initial housekeeping: allocate memory for messages,
   * initialize program with MPI, define message tags
   *---------------------------------------------------------------- */
  mlen = atoi (argv[1]);
  message = (float *)malloc ( mlen * sizeof(float) );
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  printf ( " Process %d initialized\n", rank );
  printf ( " Message size = %6d floats\n", mlen );
  printf ( " Total size = %6d bytes\n", (mlen* sizeof(float))  );
  /* ----------------------------------------------------------------
   * task 0 will report the elapsed time for a blocking send
   * ---------------------------------------------------------------- */
  if ( rank == 0 )  {
    for (i=0; i<mlen; i++) message[i] = 100;
    printf ( " Task %d sending message\n", rank );
```

```
   tbegin=MPI_Wtime();
   MPI_Send ( message, mlen, MPI_FLOAT, 1, TAG,
MPI_COMM_WORLD );
   tend=MPI_Wtime();
   print_time ( tbegin, tend );
  }
  /* ---------------------------------------------------------------
   * task 1 sleeps for 10 seconds, and then calls a blocking
   * receive.the sleep is intended to simulate time spent in useful
   * computation
   * --------------------------------------------------------------- */
  else if ( rank == 1)  {
   for (i=0; i<mlen; i++) message[i] = -100;
/*    MPI_Sleep(10); */
   new_sleep(10);
   MPI_Recv ( message, mlen, MPI_FLOAT, 0, TAG,
MPI_COMM_WORLD,
              &status );
   printf ( " Task %d received message\n", rank );
  }
  MPI_Finalize ();
  return 0;
}
/* ---------------------------------------------------------------
 * calculate elapsed time and print results
 * --------------------------------------------------------------- */
void print_time ( double tbegin, double tend )
{
  int dt;
  dt = (int)((tend - tbegin)*1000000.0);
  printf ( " Elapsed time for send = %8d uSec\n", dt );
}
```

new_sleep.c - простая программа на C;

```
#include <time.h>
int SLEEP(clock_t wait)
{
   clock_t goal;
   wait *= 1000;
```

```
    goal = wait + clock();
    while (goal > clock() );
    return (0);
}
int new_sleep (int amount)
{
        SLEEP(amount);
        return (0);
}
```

nbrecv.c – название файла для записи решения;
cbrcv.mak – C Makefile;

```
CC = mpicc.exe
all: blocksends.exe brecv.exe nbrecv.exe
blocksends.exe:  blocksends.c new_sleep.obj
        $(CC)  blocksends.c  new_sleep.obj
brecv.exe:  brecv.c new_sleep.obj
        $(CC)  brecv.c  new_sleep.obj
nbrecv.exe:  nbrecv.c new_sleep.obj
        $(CC)  nbrecv.c  new_sleep.obj
new_sleep.obj:  new_sleep.c
        $(CC)  -c new_sleep.c
clean:
        del /Q *.obj *.exe *.mod
```

Это упражнение демонстрирует, что замена блокирующего получения на неблокирующее может уменьшить синхронизационную накладку для соответствующей стандартной блокирующей отправки для сообщений, размер которых превышает размер буфера по умолчанию для коммуникаций на кластере.

Программа **brecv.exe** является весьма искусственной: в ней задача 0 осуществляет блокирующую отправку, а задача 1 бездействует (спит) в течение десяти секунд перед выполнением блокирующего получения. Вызов бездействия (сна) предназначен для имитации затрат времени на полезные вычисления.

Программа на С вызывает функцию **new_sleep()**, чтобы имитировать некоторые вычисления.

Компиляция программы **brecv.exe** для языка программирования С:

```
nmake /f cbrcv.mak brecv.exe
```

Определив, что использоваться будут два узла ровно, запустим программу. Она предоставит время, потраченное задачей 0 на блокирующую отправку.

Отредактируйте **brecv.c**, чтобы заменить блокирующее получение на вызов MPI_Wait. Пошлите неблокирующее получение перед сном. Откомпилируйте и запустите код.

Откомпилируйте новую программу (**nbrecv.exe**):

для языка программирования С

```
nmake /f cbrcv.mak nbrecv.exe
```

Определив, что использоваться будут два узла ровно, запустим программу. Она предоставит время, потраченное задачей 0 на неблокирующую отправку.

Проанализируйте значительную разницу между **brecv.exe** и **nbrecv.exe**, когда вы точно определяете значение внутри и, напротив, выше известного в данной версии MPICH буферного предела. (2048*4 байт – буфер по умолчанию для MPICH for Lynix):

```
mpirun -np 2 brecv.exe 2048
mpirun -np 2 brecv.exe 2049
mpirun -np 2 nbrecv.exe 2048
mpirun -np 2 nbrecv.exe 2049
```

Заключение

Данная работа посвящена построению алгоритмов и программ косвенной идентификации внешних параметров в моделях экономики России и в региональных моделях экономики с социальной идентификацией. При такой идентификации здесь

нужно применять параллельные вычисления на суперкомпьютерах, поскольку модели экономики с социальной стратификацией содержат огромное число внешних параметров, данные для которых нельзя получить на основе прямых оценок из статистики.

К счастью, новые модели экономики с социальной стратификацией содержат в себе естественную параллельность по стратам. Эта естественная параллельность используется при алгоритмическом и программном обеспечении косвенной идентификации параметров для нового типа моделей с социальной стратификацией.

На первом этапе параметры каждой страты идентифицировались при упрощающем предположении об отсутствии социальных лифтов, то есть в условиях, когда параметры потоков населения между стратами обнулялись. В каждой страте число параметров, которые нельзя определить напрямую из статистики, очень велико, для их определения используется косвенная идентификация – верификация экономических и демографических макропоказателей страты по историческим данным.

Алгоритмы и программы второго этапа идентификации параметров учитывают параметры всех страт модели, включая параметры, определяющие заданные переходы населения между стратами. Часть параметров, определенных на первом этапе при этом уточняется.

В алгоритмах и программах косвенной идентификации моделей содержатся те или иные меры близости для временных рядов, в данном случае для рядов макропоказателей, рассчитанных по модели и построенных на основе статистики. В работе используются критерии ошибок идентификации, основанные на двух мерах близости. Одна из них исходит от классической евклидовой меры в математике или среднеквадратичного отклонения в физике, которые в экономике для применения к экспоненциально растущим траекториям преобразуются в критерий неравенства Тэйла. Другая мера близости построена нами на основе вейвлет коэффициентов для оценки похожести рядов, что очень важно в социально-экономических явлениях, где данные статистики и расчета по модели могут отличаться из-за

разных методик учета, например теневого оборота. Оба этих критерия в процессе счета были модифицированы для ускорения расчета, что важно в параллельных вычислениях, где многократно повторяются однотипные операции.

Построены возможные сценарии развития экономики, в которых заданы, в частности, параметры переходов населения между стратами. В результате опыта идентификации моделей страны и региона уравнения моделей были модифицированы для учета возможных переходов и ограничений.

Получен опыт использования параллельных алгоритмов и программ глобальной оптимизации, разработанных в Нижегородском университете, для идентификации моделей. Разработаны алгоритмы и программы расчета по идентифицированным моделям страны и региона, учитывающим социальную стратификацию.

Литература

1. *Оленев Н.Н.* Основы параллельного программирования в системе MPI. М.: ВЦ РАН, 2005. 80 с.
2. LAM/MPI Parallel Computing. URL: http://www.lam-mpi.org (дата обращения: 18.12.2014).]
3. *Кантор М.* Бунт сытых // Новая газета. 2009. № 127 (16 ноября). URL: http://www.novayagazeta.ru/society/42632.html (дата обращения 18.12.2014).
4. *Оленев Н.Н.* Параллельные вычисления в моделировании российской экономики с учетом социальной стратификации // Параллельные вычислительные технологии (ПаВТ’2010): труды международной научной конференции (Уфа, 29 марта – 2 апр. 2010г.). Челябинск: Изд. ЮУрГУ, 2010. С. 276–286. URL: http://omega.sp.susu.ac.ru/books/conference/PaVT2010/full/021.pdf (дата обращения 18.12.2014).
5. *Оленев Н.Н.* Параллельные вычисления в идентификации динамических моделей экономики //Параллельные вычислительные технологии (ПаВТ'2008): труды международной научной конференции. Челябинск: Изд. ЮУрГУ, 2008. С. 207–214. URL: http://www.ccas.ru/olenev/022.pdf (дата обращения 18.12.2014).

6. *Тейл Г.* Экономические прогнозы и принятие решений. М.: Статистика, 1971. 488 с.
7. *Заславская Т.И.* Социальная структура современного российского общества // Общественные науки и современность, 1997. № 2. С. 5–23.
8. *Богачев К.Ю.* Основы параллельного программирования. М.: БИНОМ : Лаборатория знаний, 2003. 342 с.
9. *Немнюгин С.А., Стесик О.Л.* Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002. 400 с.
10. MPI: A Message Passing Interface Standard. June 12, 1995 [Electronic resource]. URL: http://www.cluster.bsu.by/MPI_ALL.htm (дата обращения 18.12.2014).
11. *Фетинина А. И.* Модель региональной экономики на основе социальной стратификации // Научно-технический вестник СПбГУ ИТМО, 2011. № 5 (75). С. 120–124.
12. *Долматова А.И., Оленев Н.Н.* Параллельные вычисления в моделировании региональной экономики: учебное пособие. Киров: ПРИП ФГБОУ ВПО «ВятГУ», 2012. 125 с.

Содержание

Долматова Анна Игоревна, Оленёв Николай Николаевич

Моделирование динамики социальной стратификации: параллельные алгоритмы и программы

Подписано в печать 23.12.2014
Формат бумаги 60x84 1/16
Уч.-изд.л. 5. Усл.-печ.л. 7
Тираж 120 экз. Заказ 29

Отпечатано на ротапринтах
в Федеральном государственном бюджетном учреждении науки
Вычислительный центр им. А.А. Дородницына Российской академии наук
119333, Москва, ул. Вавилова, 40