# MPI. Основная идея и недостатки

В мире функционального программирования наиболее распространенной технологией создания программ для кластеров с распределенной памятью является MPI. Основным способом взаимодействия параллельных процессов в данном случае выступает передача сообщений от одного узла другому. Стандарт MPI фиксирует интерфейс, который должен соблюдаться как системой программирования на каждой вычислительной платформе, так и пользователем при создании своих программ.

MPI поддерживает работу с языками Fortran и C. MPI-программа — это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений от одного процесса другому. [1].

Несмотря на то, что MPI-программы показывают высокий уровень производительности, сама технология имеет ряд недостатков:

* низкий уровень (программирование на MPI часто сравнивают с программированием на ассемблере), необходимость детального управления распределением массивов и витков циклов между процессами, а также обменом сообщениями между процессами – все это приводит к высокой трудоемкости разработки программ;
* необходимость избыточной спецификации типов данных в передаваемых сообщениях, а так же наличие жестких ограничений на типы передаваемых данных;
* сложность написания программ, способных выполняться при произвольных размерах массивов и произвольном количестве процессов – делает практически невозможным повторное использование имеющихся MPI-программ;
* сложность выражения многоуровневого параллелизма программы;

При использовании MPI распределение вычислительной нагрузки и данных между узлами, обеспечение необходимых пересылок осуществляется в прикладной программе. Данное обстоятельство приводит к тому, что разработка параллельных программ становится сложной задачей и ложится на плечи программиста. Между тем, многие механизмы, связанные с распределением нагрузки и данных между вычислительными узлами, являются достаточно общими, и реализация их заново в каждом параллельном приложении приводит к дублированию кода. Необходимость равнозначного владения на высоком уровне технологией программирования на языке высокого уровня (C или Fortran) и MPI еще дальше отдаляет «конечных» пользователей – специалистов в предметных областях, потребителей суперкомпьютерных технологий, от возможности участия в разработке параллельных программных приложений.

# ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ С ОБЩЕЙ И РАСПРЕДЕЛЕННОЙ ПАМЯТЬЮ

Рассмотрим классификацию (по Джонсону), которая чаще других используется при проектировании высокопроизводительных ВС. Это системы с общей памятью и распределенной памятью, или как еще их называют симметричные мультипроцессорные системы (SMP) и системы с массовым параллелизмом (МРР) соответственно.

## Си­сте­мы с об­щей па­мя­тью

Си­сте­ма­ми с об­щей па­мя­тью на­зы­ва­ют си­сте­мы, в ко­то­рых не­сколь­ко про­цес­со­ров име­ют об­щую опе­ра­тив­ную па­мять (рисунок 1). Ча­ще все­го встре­чаю­щие­ся си­сте­мы это­го ти­па — ком­пью­те­ры с мно­го­ядер­ны­ми про­цес­со­ра­ми.

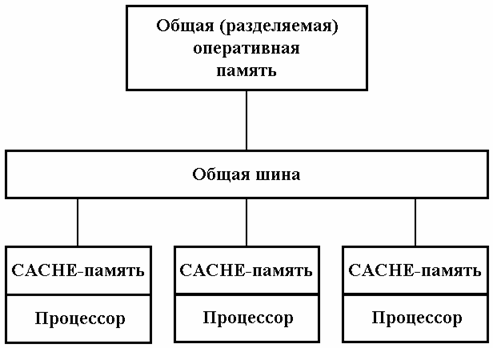


Рис.1 – Архитектура ВС с общей памятью

Пре­иму­ще­ства:

* Не тре­бу­ет­ся об­мен дан­ны­ми: дан­ные, по­ме­щён­ные в па­мять од­ним про­цес­со­ром, ав­то­ма­ти­че­ски ста­но­вят­ся до­ступ­ны­ми дру­гим про­цес­со­рам. Со­от­вет­ствен­но, си­сте­ма не долж­на тра­тить вре­мя на пе­ре­сыл­ку дан­ных.
* Для та­ких си­стем про­сто пи­сать про­грам­мы: мож­но, на­при­мер, со­здать не­сколь­ко вы­чис­ли­тель­ных по­то­ков, или же снаб­дить про­грам­му спе­ци­аль­ны­ми ди­рек­ти­ва­ми (на­при­мер, тех­но­ло­гия OpenMP), ко­то­рые под­ска­жут ком­пи­ля­то­ру, как рас­па­рал­ле­ли­вать про­грам­му. Кро­ме то­го, воз­мож­но пол­но­стью ав­то­ма­ти­че­ское рас­па­рал­ле­ли­ва­ние про­грам­мы ком­пи­ля­то­ром.
* Ком­пакт­ность си­стем: мо­жет быть реа­ли­зо­ва­на в ви­де не­сколь­ких про­цес­со­ров на од­ной ма­те­рин­ской пла­те, и/или в ви­де не­сколь­ких ядер внут­ри про­цес­со­ра.

Не­до­стат­ки:

* Про­бле­ма сов­мест­но­го до­сту­па к па­мя­ти: нуж­но осто­рож­но ра­бо­тать с те­ми участ­ка­ми па­мя­ти, для ко­то­рых воз­мож­но од­но­вре­мен­ное вы­пол­не­ние за­пи­си од­ним про­цес­со­ром и дру­гой опе­ра­ции (за­пи­си или чте­ния) дру­гим про­цес­со­ром.
* Про­бле­ма син­хрон­но­сти кэ­шей: для уско­ре­ния до­сту­па к па­мя­ти про­цес­со­ры снаб­жа­ют­ся кэ­ша­ми. Ес­ли один про­цес­сор из­ме­нил дан­ные в опе­ра­тив­ной па­мя­ти, и эти дан­ные про­кэ­ши­ро­ва­ны дру­ги­ми про­цес­со­ра­ми, то их кэ­ши долж­ны ав­то­ма­ти­че­ски об­но­вить­ся. Дан­ная про­бле­ма от­сут­ству­ет в мно­го­ядер­ных про­цес­со­рах, ис­поль­зую­щих об­щий кэш.
* Про­бле­ма мед­лен­но­го об­ра­ще­ния к опе­ра­тив­ной па­мя­ти и её огра­ни­чен­но­го объ­ё­ма: про­цес­сор ра­бо­та­ет быст­ро, а па­мять — мед­лен­но, по­это­му да­же од­но­му про­цес­со­ру при­хо­дит­ся ждать за­груз­ки дан­ных из опе­ра­тив­ной па­мя­ти. Ес­ли же про­цес­со­ров не­сколь­ко, то им при­хо­дит­ся ждать ещё доль­ше. Ско­рость ра­бо­ты каж­до­го про­цес­со­ра с па­мя­тью ста­но­вит­ся тем мень­ше, чем боль­шее чис­ло про­цес­со­ров име­ет­ся в си­сте­ме. Кро­ме то­го, объ­ём па­мя­ти не мо­жет быть сде­лан сколь угод­но боль­шим, так как для это­го при­дёт­ся уве­ли­чи­вать раз­ряд­ность ши­ны па­мя­ти.
* Про­бле­ма мас­шта­би­ру­е­мо­сти: очень слож­но сде­лать по­доб­ную си­сте­му с большим чис­лом про­цес­со­ров, так как очень силь­но воз­рас­та­ет сто­и­мость и па­да­ет эф­фек­тив­ность ра­бо­ты из-за опи­сан­ных вы­ше про­блем. Прак­ти­че­ски все по­доб­ные си­сте­мы име­ют ≤ 8 про­цес­со­ров.

## Си­сте­мы с рас­пре­де­лён­ной па­мя­тью

Си­сте­ма со­дер­жит не­сколь­ко про­цес­со­ров, каж­дый име­ет свою опе­ра­тив­ную па­мять (рисунок 2). Для обес­пе­че­ния об­ме­на ин­фор­ма­ци­ей про­цес­со­ры со­еди­не­ны ка­на­ла­ми свя­зи. По ха­рак­те­ру свя­зей та­кие си­сте­мы де­лят­ся на си­сте­мы с уни­вер­саль­ной ком­му­та­ци­ей (каж­дый про­цес­сор мо­жет пе­ре­дать ин­фор­ма­цию лю­бо­му дру­го­му про­цес­со­ру) и си­сте­мы с жёст­кой (фик­си­ро­ван­ной) ком­му­та­ци­ей (каж­дый про­цес­сор мо­жет пе­ре­дать ин­фор­ма­цию толь­ко огра­ни­чен­но­му чис­лу дру­гих про­цес­со­ров).

Си­сте­мы с рас­пре­де­лён­ной па­мя­тью, в ко­то­рых каж­дый вы­чис­ли­тель­ный узел пред­став­ля­ет со­бой пол­но­цен­ный ком­пью­тер со сво­ей ко­пи­ей опе­ра­ци­он­ной си­сте­мы, на­зы­ва­ют кла­стер­ны­ми (или «кла­сте­ра­ми»). Кла­сте­ры обыч­но пред­став­ля­ют со­бой шка­фы с ком­пакт­ны­ми си­стем­ны­ми бло­ка­ми, ко­то­рые со­еди­не­ны друг с дру­гом ка­на­ла­ми свя­зи (по­сред­ством спе­ци­аль­ных ком­му­та­то­ров), пе­ре­даю­щи­ми дан­ные со ско­ро­стью 10 ГБит/сек и бо­лее.

Пре­иму­ще­ства:

* Про­сто­та и де­ше­виз­на по­строе­ния: мож­но взять боль­шое ко­ли­че­ство обыч­ных ком­пью­те­ров, со­еди­нить их ка­на­ла­ми свя­зи (на­при­мер, Ethernet), и по­лу­чить кла­стер.
* Эф­фек­тив­ное ре­ше­ние за­дач, тре­бую­щих ма­ло­го об­ме­на дан­ны­ми: каж­дый ком­пью­тер бу­дет ра­бо­тать в пол­ную мощ­ность, не ожи­дая, по­ка осво­бо­дит­ся до­ступ к опе­ра­тив­ной па­мя­ти.
* Воз­мож­ность ре­шать за­да­чи, тре­бую­щие очень боль­ших объ­ё­мов опе­ра­тив­ной па­мя­ти: сум­мар­ный объ­ём па­мя­ти си­сте­мы мож­но сде­лать сколь угод­но боль­шим. Тре­бу­ет­ся лишь, что­бы за­да­ча раз­би­ва­лась на от­но­си­тель­но не­за­ви­си­мые под­за­да­чи.
* Воз­мож­ность мас­шта­би­ро­ва­ния: мож­но со­еди­нить сколь­ко угод­но вы­чис­ли­тель­ных уз­лов вме­сте, при этом сто­и­мость си­сте­мы бу­дет про­пор­цио­наль­на чис­лу уз­лов. В свя­зи с этим боль­шин­ство са­мых мощ­ных вы­чис­ли­тель­ных си­стем в ми­ре яв­ля­ют­ся кла­стер­ны­ми.

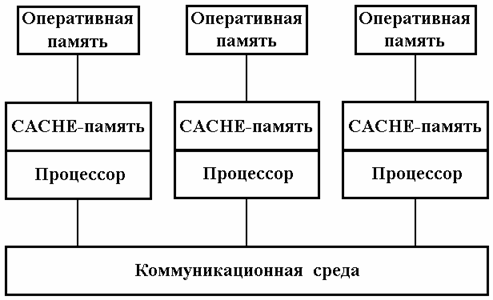


Рис. 2 – Архитектура ВС с распределенной памятью

Не­до­стат­ки:

* Про­бле­ма об­ме­на дан­ны­ми: об­мен дан­ны­ми в та­ких си­сте­мах обыч­но идёт очень мед­лен­но по срав­не­нию со ско­ро­стью вы­чис­ле­ний (и с боль­ши­ми за­держ­ка­ми). По­это­му за­да­чи, тре­бую­щие ин­тен­сив­но­го об­ме­на, не­воз­мож­но ре­шить на та­ких си­сте­мах эф­фек­тив­но.
* Слож­ное про­грам­ми­ро­ва­ние: про­грам­мист дол­жен про­ду­мать об­мен дан­ны­ми, ко­то­рый бу­дет при­сут­ство­вать в си­сте­ме, дол­жен сам за­про­грам­ми­ро­вать этот об­мен (на­при­мер, с по­мо­щью MPI). При не­пра­виль­ном про­грам­ми­ро­ва­нии ве­ли­ка ве­ро­ят­ность вза­им­ных бло­ки­ро­вок: ко­гда, на­при­мер, два про­цес­со­ра ждут дан­ных друг от дру­га. Про­бле­ма бло­ки­ро­вок есть и в си­сте­мах с об­щей па­мя­тью, но здесь она про­яв­ля­ет се­бя го­раз­до ча­ще. Ав­то­ма­ти­че­ская ор­га­ни­за­ция об­ме­на дан­ны­ми воз­мож­на лишь для не­ко­то­рых ча­ст­ных слу­ча­ев.
* Боль­шой раз­мер си­стем и боль­шое энер­го­по­треб­ле­ние: кла­стер­ные си­сте­мы за­ни­ма­ют це­лые ком­на­ты и да­же зда­ния.

# СПОСОБЫ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

## Явный параллелизм и автоматическое распараллеливание

Работы в области моделирования и построения параллельных алгоритмов можно разделить на два больших направления:

* Неявный параллелизм. Это направление изучает методы автоматической генерации параллельных алгоритмов на основе их последовательных прототипов (автоматического распараллеливания последовательных алгоритмов).
* Явный параллелизм. Разработка методов организации вычислений, изначально ориентированных для реализации на ЭВМ с параллельной архитектурой.

Исследования в области автоматического распараллеливания вычислительных алгоритмов необходимы в связи с наличием большого объема ранее разработанных методов, алгоритмов и программ для решения различных задач на последовательных ЭВМ. Их реализация на параллельных ЭВМ требует модификации, связанной с распределением данных и вычислений по узлам параллельной ЭВМ, а также с адаптацией под особенности архитектуры конкретной ЭВМ. Этот процесс важно автоматизировать, чтобы максимально сократить его длительность и избавить исследователей - специалистов в различных областях, которые часто являются авторами и пользователями вычислительных программ, от знакомства со спецификой конкретной ЭВМ, на которой программа будет исполняться.

Автоматическое распараллеливание имеет большое значение и при создании новых вычислительных программ. Последовательные алгоритмы удобны и естественны для человека в силу того, что люди привыкли думать и действовать последовательно. Вместе с тем, любая современная ЭВМ обладает определенной степенью параллелизма.

Работы в области автоматического распараллеливания вычислительных алгоритмов принадлежат таким ученым, как Абрамов С.М.[2, 3], Воеводин В.В., Воеводин Вл.В.[4], Легалов А.И.[5], Тыугу Э.Х.[6, 7] и др.

Наряду с несомненными достоинствами, такими как возможность использования ранее разработанных и хорошо отлаженных последовательных программ, сохранение привычного для человека последовательного стиля разработки вычислительных алгоритмов, обеспечение переносимости программ, автоматическое распараллеливание обладает недостатками. Основным недостатком является ограниченная область применения. К сожалению, не все последовательные алгоритмы допускают эффективное распараллеливание. Иногда сам численный метод, на основе которого построен алгоритм, не допускает распараллеливания.

Для достижения максимальной производительности необходимо уже на этапе разработки алгоритма учитывать параллелизм и явно выделять участки, которые должны выполняться одновременно. Более того, необходимо учитывать архитектуру и особенности конкретной параллельной ЭВМ, на которой будет исполняться программа.

Основные сложности, с которыми сталкиваются исследователи в области построения параллельных алгоритмов с явным параллелизмом, в первую очередь связаны с наглядным представлением алгоритма.

Текстовая нотация, традиционно используемая в математике и программировании, удобна для представления последовательных процессов. Однако последовательная природа самого текста значительно затрудняет восприятие текстового описания параллельных вычислений. На первый план выдвигаются графические способы описания параллелизма.

## Графические модели параллельных процессов

Основой подавляющего большинства графических способов представления параллельных процессов является форма представления в виде графа, то есть совокупности вершин (узлов), соединенных между собой дугами (ребрами). В отличие от текстовой формы записи, в которой объекты (символы и слова) образуют *последовательность*, а каждый объект связан только с левым и правым «соседом», графовая форма позволяет наглядно изображать более сложные *взаимосвязи*, поскольку в ней каждый объект может соединяться с несколькими другими объектами. В этом смысле текстовая форма одномерна, в то время как графовая форма – многомерна. Возможность варьировать геометрические размеры, форму и цвет вершин, внешний вид и толщину дуг, изменять взаимное расположение вершин без изменения топологии графа значительно увеличивают выразительные возможности графовой формы представления.

Графические модели обычно являются ориентированными графами, в которых дуги определяют направление передачи данных или зависимость между вершинами. Вершины и дуги обычно снабжаются текстовыми аннотациями, которые именуют их, перечисляют их содержимое или свойства. Различные графические модели отличаются друг от друга семантикой вершин и дуг. Основные классы графических моделей параллельных процессов:

* сети Петри [8];
* графы переходов, используются в SWITCH-технологии /64, 65/ и графическом языке Statecharts /89/;
* диаграммы потоков данных, используются в системах CODE 2.0 /83/, Paralex /77/;
* диаграммы потоков управления, примерами моделей этого типа являются графический язык системы HeNCE /74/, технология графо-символического программирования /32/

Графическая нотация является более наглядной и компактной, по сравнению с текстовым описанием. За счет использования графических моделей удается не только сократить время разработки параллельных алгоритмов, но и повысить их качество, т.к. графическая нотация допускает формальное математическое описание модели, по которому может быть проведена ее автоматическая верификация и оптимизация.

## Концепция параллельной технологии ГСП

Технология ГСП в работе [9] определена как технология проектирования и кодирования алгоритмов программного обеспечения (ПО), базирующаяся на графическом способе представления программ, преследующую цель полной или частичной автоматизации процессов проектирования, кодирования и тестирования ПО.

Модель представляется четверкой <D, F, P, G>, где D – множество данных некоторой предметной области, F – множество операторов, определенных над данными предметной области, Р – множество предикатов, действующих над структурами данных предметной области, G = {A, Ψ, Φ, R} – ориентированный помеченный граф. A = {A1, A2, …, An} – множество вершин графа. Каждая вершина Ai помечена локальным оператором   
fi(D) ∈F. На графе задано множество дуг управления Ψ = { Ψ1i1, Ψ1i2, …, Ψjm} и множество дуг синхронизации Ф = {Ф1i1, Ф1i2, …, Фj*l*}. R – отношение над множествами вершин и дуг, определяющее способ их связи. Дуга управления, соединяющая любые две вершины Ai и Aj, имеет три метки: предикат pij(D) ∈P, приоритет k(Ψij) ∈N и тип дуги T(Ψij) ∈N. Каждая дуга синхронизации Фij помечена сообщением μij∈N. В классической (последовательной) модели вычислительного процесса, используемой в технологии ГСП, отсутствуют дуги синхронизации, которые вводятся в модель параллельного вычислительного процесса для решения задач синхронизации между его различными участками.

Под предметной областью программирования (ПОП) будем понимать некоторую среду программирования, имеющую общую цель - разработку программного обеспечения автоматизации расчетов в некоторой области практических интересов (авиационные двигатели, бизнес, медицинские приборы и т.д.), общую область данных и общую область знаний.

Развитие вычислительного процесса, описываемого моделью, ассоциируется с переходами из вершины в вершину по дугам управления. При этом переход по дуге управления возможен лишь в случае истинности предиката, которым она помечена. Если несколько предикатов, помечающих исходящие из вершины дуги, одновременно становятся истинными, переход осуществляется по наиболее приоритетной дуге. Функционирование модели начинается с выполнения оператора f0(D), помечающего начальную вершину A0.

Граф-модель в данном случае заменяет текстовую (вербальную) форму описания алгоритма программы, при этом:

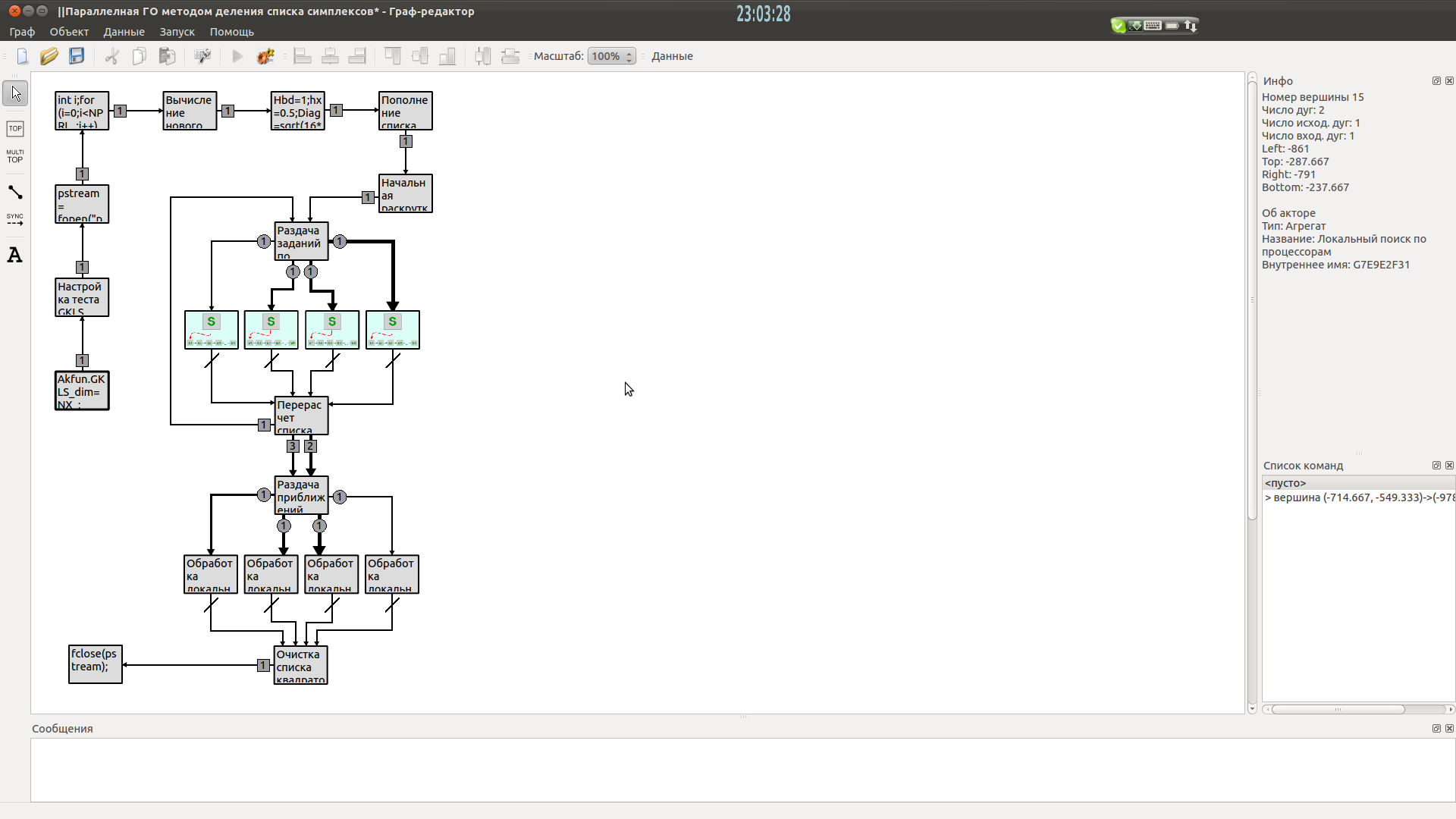


Рис 3. Граф-модель параллельного алгоритма нахождения глобального экстремума

1. Реализуется главная цель - представление алгоритма в визуальной (графосимволической) форме. Изображение программ в виде ориентированного помеченного графа естественно для восприятия человеком (рисунок 3). Направленная дуга служит очевидным изображением перехода из одного состояния вычислительного процесса в другое, вершина - выполняемой вычислительной функции, а в целом ориентированный граф наглядно представляет все пути, по которым может развиваться вычислительный процесс. В этом случае логические особенности разрабатываемого программного модуля проявляются в характерной для него топологии графа. Можно сказать, что графическое представление программ позволяет задействовать непосредственное образное восприятие, расширяя возможности человека при разработке и анализе сложных программ.
2. Происходит декомпозиционное расслоение основных компонент описания алгоритма программного продукта. Так структура алгоритма представляется графом G, элементы управления собраны во множестве предикатов P и, как правило, значимы не только для объекта F, но и для всей предметной области. Спецификация структур данных, а также установка межмодульного информационного интерфейса по данным “пространственно” отделена от описания структуры алгоритма и элементов управления.

## Создание граф-моделей параллельных вычислений

В наиболее общем виде создание модели состоит из этапов, представленных на рисунке 4.

Словарь

данных

Базовые

модули

Объекты

(

(акторы)

Граф-модель

(

(агрегат)

1

2

3

4

Рис. 4 - Этапы создания модели

1. Создание словаря данных ПО. На данном этапе создаются новые типы и структуры данных, а также происходит накопление словаря данных, где хранится информация обо всех переменных программы. Этот этап целесообразно реализовать с помощью удобного, гибкого многооконного интерфейса пользователя с информационным фондом системы.
2. Разработка базовых модулей. Это этап традиционного текстового программирования, на котором программист работает с исходными текстами программ, учитывая, однако, требования стандарта ГСП к оформлению этих текстов.
3. Создание объектов ПО. Этот этап производится автоматически после привязки формальных параметров базовых модулей к фактическим данным предметной области.
4. Конструирование агрегатов. На этапе графического программирования пользователь может создать графовый образ новой программы.

Разработанный и отлаженный агрегат, в свою очередь, может быть использован в качестве исходного материала при конструировании следующих агрегатов. Следовательно, в общем случае агрегат имеет иерархическую структуру.

Эффективность программирования в технологии ГСП возрастает по мере развития пользователем своей среды программирования. Доля текстового программирования с традиционной трудоемкой отладкой постепенно снижается, и программирование перерастает в конструирование агрегатов из надежных программных модулей. Отладка при этом заключается только в корректировке структуры графа. Для корректировки структуры графа и работы с ним, зачастую возникает необходимость визуализировать весь граф целиком. Но при иерархической структуре это сделать проблематично.

## Управление вычислительными процессами. Граф-машина

В технологии ГСП для объектов – агрегатов используется мониторная схема организации вычислений. В основу способа положено централизованное управление процессом вычислений, осуществляемое специальной программой – граф-машиной.

Граф-машина универсальна для любого алгоритма. Исходной информацией для граф-машины служит, описанная выше, модель графа управления вычислительным процессом. Анализируя его графическую модель, представленную виде структур на смежной памяти [ссылка], она выполняет в соответствующем порядке акторы и агрегаты, вычисляет значения предикатов и управляет синхронизацией. Для каждой параллельной ветви запускается по одному экземпляру граф-машины, которая представляет собой отдельный процесс в вычислительной системе.

Работа граф-машины начинается с выполнения актора в корневой вершине. Затем строится список дуг, исходящих из текущей вершины. Этот список просматривается граф-машиной последовательно, начиная с самой приоритетной дуги. Вычисляется значение предиката, помечающего дугу, и в случае его истинности, происходит переход к обработке следующей вершины. В результате обработки параллельной дуги в отдельном процессе запускается другая граф-машина, обрабатывающая порождаемую данной дугой параллельную ветвь. После запуска всех параллельных ветвей происходит переход в вершину, в которой они терминируются. Родительская граф-машина ожидает завершения выполнения всех дочерних граф-машин, если не задано альтернативное условие.

Централизация функций управления в рамках одной программы (граф-машины) на самом деле очень удобное решение, поскольку позволяет:

* контролировать вычислительный процесс в целом. И, в случае нештатных ситуаций, принимать системные решения;
* реализовать сбор статистической информации о характеристиках надежности каждого из модулей; вычислительной сложности модулей; маршрутах развития вычислительного процесса и т.п.

# МЕЖМОДУЛЬНЫЙ ИНТЕРФЕЙС ПАРАЛЛЕЛЬНОГО ОБМЕНА ДАННЫМИ

## Стандарт хранения и использования данных в ГСП

В технологии ГСП используется стандарт при организации межмодульного информационного интерфейса. Стандарт обеспечивается выполнением пяти основных правил:

1. Вводится единое для всей предметной области программирования (ПОП) хранилище данных, актуальных для всей области. Полное описание данных размещено в словаре данных ПОП. Любые переменные, не описанные в словаре данных, считаются локальными данными для тех объектов ГСП, где они используются.
2. В пределах ГСП описание типов данных размещается централизовано в архиве типов данных.
3. Данные, актуальные для формируемого программного приложения, объединяются в единую универсальную структуру - класс TPOData.
4. В базовых модулях в качестве механизма доступа к данным допускается только передача параметров по адресу, ссылающемуся на универсальную структуру данных.
5. Привязка данных объектов ПОП к формальным параметрам базовых модулей реализована в паспортах объектов ПОП.
6. В технологии ГСП не рекомендуется использовать иные способы организации межпрограммных связей по данным.

Данных подход к организации межмодульного информационного интерфейса приводит к тому, что формируемые (автоматически или автоматизировано) программные коды и информационные связи “пространственно” отделены друг от друга. Модификация любого из объектов (актора, предиката или агрегата) не требует переделки кодов других объектов, входящих в ПОП. Физически данные ПОП хранятся в общей области памяти базового компьютера. Параллельный вариант граф модели при использовании технологии MPI предполагает распределенное размещение данных на разных компьютерах некоторого кластера [10]. Для соблюдения вышеназванных условий в параллельной версии технологии ГСП программно реализована модель общей памяти для аппаратной архитектуры с распределенной памятью.

## Способ реализации общей памяти в ГСП

В среде выполнения программы выбирается машина, на которой будет запущен процесс, отвечающий за хранение переменных ПОП. Учитывая аппаратные особенности и топологию ВС, это может быть узел с наибольшим объемом оперативной памятью или центральный узел, имеющий минимальное время доступа от любого из остальных узлов кластера. Преимущество данного подхода в том, что значительно экономится ресурс памяти на вычислительных узлах, т.к. на узлах память выделяется только под те переменные, которые используются.

Предлагаемый способ обмена данными требует введения понятия диспетчера данных – подпрограммы, выполняющей функции хранения, чтения и модификации данных предметной области.

## Менеджер памяти

Диспетчер данных исполняется в отдельном процессе параллельной программы. Он порождает объект, описываемый классом TPOData, который хранит значения данных предметной области. В каждом из процессов, содержащих параллельные ветви граф-модели, также порождается объект класса TPOData. Однако функции доступа к членам-данным у объекта диспетчера данных и у объектов параллельных ветвей различаются. Диспетчер данных хранит все данные в локальной памяти и для обращения к ним использует обычные указатели. На остальных процессах используется ленивая инициализация памяти под переменную при первом доступе.

В параллельных ветвях граф-модели для чтения или записи некоторого данного осуществляется обращение к диспетчеру памяти с помощью совокупности сообщений. В первом сообщении пересылается запрос на чтение или запись конкретного данного. Каждая переменная из ПОП получает уникальный номер, по которому диспетчер памяти может ее идентифицировать. В случае чтения параллельная ветвь переходит к ожиданию ответа от диспетчера данных. При записи во втором сообщении пересылается новое значение данного. Диспетчер данных циклически принимает и обрабатывает запросы параллельных ветвей (рисунок 5).

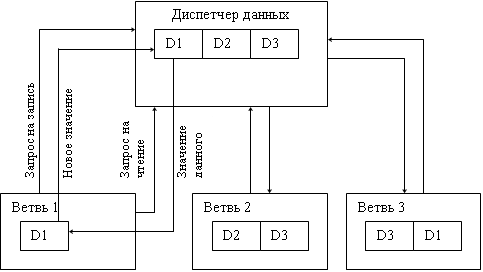


Рис. 5 – Модель общих данных системы PGRAPH

## Обзор класса

Для сохранения очевидных преимуществ, возникающих при использовании модели общей памяти, в системе PGRAPH последняя эмулируется за счет использования возможностей объектно-ориентированной парадигмы программирования: класса TPOData.

Класс TPOData – это ядро механизма хранения и передачи данных в технологии ГСП. Класс. Класс TPOData инкапсулирует все данные ПОП и предоставляет доступ к ним через поля-свойства. Полей ровно столько, сколько переменных в ПОП и их названия совпадают с названиями переменных. Для переменных простых типов в классе TPOData описано по одному методу доступа для чтения и установки. Для массивов определено по два метода чтения и установки: доступ ко всему массиву и к элементу по индексу.

## Реализация свойств

Свойство — способ доступа к внутреннему состоянию объекта, имитирующий переменную некоторого типа. Обращение к свойству объекта выглядит так же, как и обращение к полю объекта, но, в действительности, реализовано через вызов функции. При попытке задать значение данного свойства вызывается один метод (*setter*), а при попытке получить значение данного свойства — другой (*getter*).

Реализацией свойств в технологии ГСП являются шаблонные классы \_\_property\_rw и \_\_indexed\_property\_rw. Классы описан в файле property.h. В качестве параметра шаблону передается тип переменной, которая будет доступна через данное свойство, и ссылка на класс, в котором это свойство будет использоваться, в нашем случае это TPOData. После инициализации объекта свойства ему необходимо установить методы доступа. В качестве методом доступа используются соответствующие методы из класса TPOData. Об указателях на функции-члены классов можно прочесть в RSDN Magazine #6-2004.

Для простых типов и для массивов используются различные классы свойств. Для простых типов данных необходимо указать по одному методу для получения значения и для установки. Методы соответственно должны иметь сигнатуру:

\_T (TPOData:: \*)(void); //для getter

\_T (TPOData:: \*)(\_T const &); // для setter ,

где \_T – тип переменной.

Для массивов необходимо указать 2 метода получения значения и 2 метода установки значения. Одна пара методов используется для получения всего массива, другая пара используется для получения одного элемента по индексу.

\_T\* (TPOData:: \*)(void); // для получения всего массива

\_T\* (TPOData:: \*)(\_T\* const &); // для установки всего массива

\_T (TPOData:: \*)(\_I const &); // для получения одного элемента по индексу

\_T (TPOData:: \*)(\_I const &, \_T const &); // для установки одного элемента по индексу

Это один из вариантов реализации свойств в языке C++ спроектированный с учетом особенностей технологий ГСП и MPI. Об общих подходах к реализации свойств можно прочесть в RSDN

## Компилятор данных.

Компилятор данных собирает из типов и данных ПОП класс TPOData. В основном при создании класса TPOData решается задача определения типа переменной.

Выходные файлы строятся компилятором на основе шаблонов. В шаблоне определены теги, которые заменяются компилятором на соответствующие конструкции. Всего имеется три файла шаблона:

* utypes.h.template – шаблон, в котором описаны типы данных
* tpodata.h.template – заголовочный файл класса TPOData
* tpodata.cpp.template – исходный текст класса TPOData

По количеству шаблонов создается 3 выходных файла. Для каждого определенного типа данных в файле utypes.h имеется запись. Если речь идет о программе на MPI, то в файле utypes.h создается соответствующий тип MPI для каждого пользовательского типа.

Для каждой переменой из ПОП в классе TPOData создается поле-свойство, для доступа к переменной и метод установки и получения значений, работающие с этим свойством. Класс TPOData описываются в файле tpodata.h. а его реализация, соответственно, помещается в файл tpodata.cpp.

## Доступ к D

Класс TPOData и экземпляр класса напрямую недоступен для разработчика граф-программ и программист не обязан знать о его существовании. Когда программист обращается к переменной ПОП, на самом деле он обращается к полям-свойствам объекта D.

## Пример использования свойств

Рассмотрим использование свойств на примере создания inline-актора.

Пусть в ПОП определены следующие переменные и типы:

|  |  |  |
| --- | --- | --- |
| Название | Определение | Описание |
| int |  | Встроенный тип целых чисел |
| double |  | Встроенный тип чисел с двойной точностью |
| array | typedef int array[100] | Массив целых чисел длины 100 |

|  |  |
| --- | --- |
| Название | Тип |
| a | int |
| b | double |
| c | array |

Пример доступа к переменным:

a = 100; //устанавливает значение a в 100

double \_b = b; //читает значение b в локальную переменную \_b

b+=1; //инкрементирует значение b

for (int i = 0; i < c.length; i++)

c[i] = i; //инициализирует каждый элемент массива

В примере показана дополнительная возможность массивов – это получение их длины, как значения поля length.

## Ограничения использования свойств

В целом, с плеч программиста снимается большой груз забот об управлении данными с помощью функций MPI, и обмен данными между процессами становится “похож” на обмен данными между нитями внутри одного процесса. Однако данная реализация свойств накладывает ряд ограничений на использование данных.

1. Переменная ПОП не может использоваться в качестве параметра функции с переменным числом аргументов (например printf(const char \*, …)) и не может передаваться по указателю в функцию (например scanf(const char \*, void \*)). В подобных случаях необходимо создать локальную буферную переменную, через которую читать и писать в переменную ПОП. Это ограничение касается использования свойств только в inline-модулях.
2. Переменные в общей памяти не могу иметь тип “указатель”. Следствием этого является то, что в общей памяти нельзя создавать массивы переменной длины.
3. В качестве переменных предметной области нельзя использовать многомерные массивы, как следствие запрета 2. Необходимо представить многомерный массив в виде одномерного.

# ЭКСПЕРИМЕНТАЛЬНЫЕ ИЗМЕРЕНИЯ НАКЛАДНЫХ РАСХОДОВ НА ПЕРЕДАЧУ ДАННЫХ В ГСП

## Условия эксперимента

Для оценки эффективности предложенной реализации модели общей памяти для распределенной вычислительной системы кластера были проведены сравнительные тесты программы, написанной на PRAPH с использованием общей памяти и программы на “чистом” MPI. Сравнивалось время передачи массива из 1 000 000 элементов типа double как единого массива и передача массива “поэлементно”. Эксперименты проводились на суперкомпьютере СГАУ “Сергей Королев”. В эксперименте было задействовано до 30 процессоров, расположенных на разных узлах кластера. В качестве системной сети для связи узлов использовалась QLogic/Voltaire InfiniBand DDR со скоростью передачи до 20Гбит/с. Использовалась библиотека MVAPICH2 оптимизированная под InfiniBand.

## Результаты эксперимента

### Измерение времени передачи целого массива

В первом эксперименте измерялось время передачи массива из 1 000 000 элементов целиком. Время засекалось от момента начала ожидания массива до его получения. Все процессы предварительно были синхронизированы через MPI\_Barier().

Результаты эксперимента приведены в таблице и на рисунке 6.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Граф** | **Чистый MPI** |  | **Граф** | **Чистый MPI** |  |
| Номер процесса | **Весь массив** | | Накладные расходы | **Поэлементно** | | Накладные расходы |
| 1 | 0,0033541 | 0,0087700 | 0,3824486 | 4,1762000 | 0,3362190 | 12,42 |
| 2 | 0,0054910 | 0,0101080 | 0,5432351 | 4,1843500 | 0,6731550 | 6,22 |
| 3 | 0,0077121 | 0,0253921 | 0,3037216 | 4,2014400 | 1,0249600 | 4,10 |
| 4 | 0,0098951 | 0,0265560 | 0,3726122 | 4,2228200 | 1,3744200 | 3,07 |
| 5 | 0,0120518 | 0,0354850 | 0,3396308 | 8,3733600 | 1,7241700 | 4,86 |
| 6 | 0,0142450 | 0,0419271 | 0,3397564 | 8,4177600 | 2,0622800 | 4,08 |
| 7 | 0,0163870 | 0,0372140 | 0,4403450 | 8,4629200 | 2,4115400 | 3,51 |
| 8 | 0,0184159 | 0,0359800 | 0,5118371 | 8,5056200 | 2,7689600 | 3,07 |
| 9 | 0,0204589 | 0,0429502 | 0,4763400 | 12,7302000 | 3,1016900 | 4,10 |
| 10 | 0,0224710 | 0,0448780 | 0,5007130 | 12,8189000 | 3,4355600 | 3,73 |
| 11 | 0,0244930 | 0,0558629 | 0,4384484 | 12,8835000 | 3,7703100 | 3,42 |
| 12 | 0,0265100 | 0,0525172 | 0,5047870 | 12,9495000 | 4,1292900 | 3,14 |
| 13 | 0,0285220 | 0,0573850 | 0,4970288 | 17,1755000 | 4,4815000 | 3,83 |
| 14 | 0,0305281 | 0,0614181 | 0,4970538 | 17,2822000 | 4,8265400 | 3,58 |
| 15 | 0,0325871 | 0,0649281 | 0,5018952 | 17,3967000 | 5,1809600 | 3,36 |
| 30 | 0,0629568 | 0,1341530 | 0,4692910 | 35,2747000 | 10,4065000 | 3,39 |

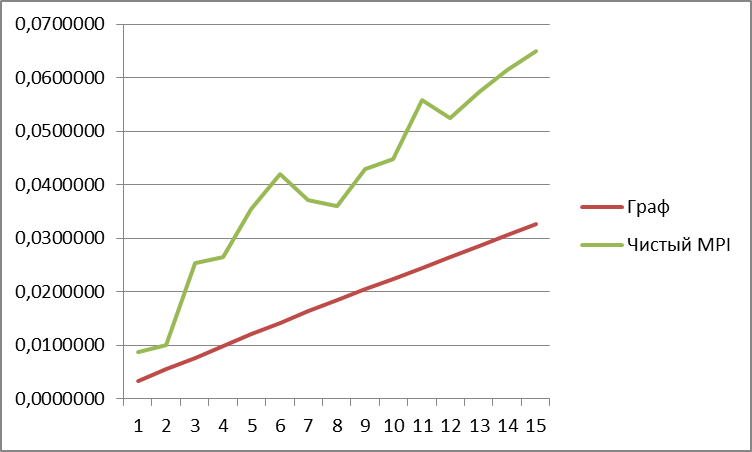


Рис. 6 – Время передачи целого массива.

На таблицы видно, что время получения всеми 15-ю процессами всего массива оставил 0,032 сек, а в программе на MPI все процессы получили массив только через 0,064 сек. Время рассылки данных в программе использующей механизм общей памяти технологии ГСП меньше времени рассылки в программе на “чистом” MPI в 2 раза. Подробнее карта пересылок показана на рисунке 7 и 8.

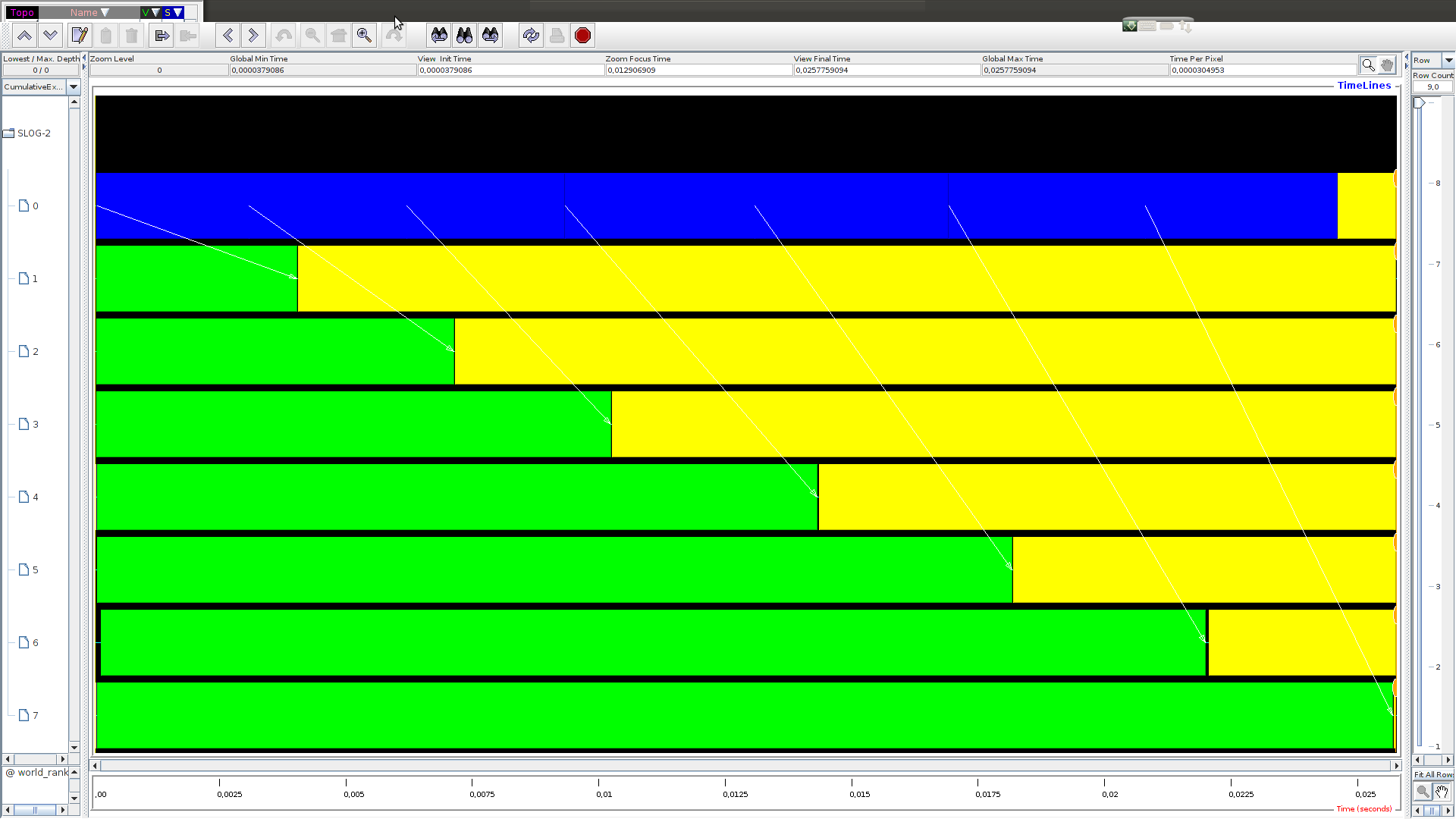


Рис. 7 – Карта процессов для программы на “чистом” MPI

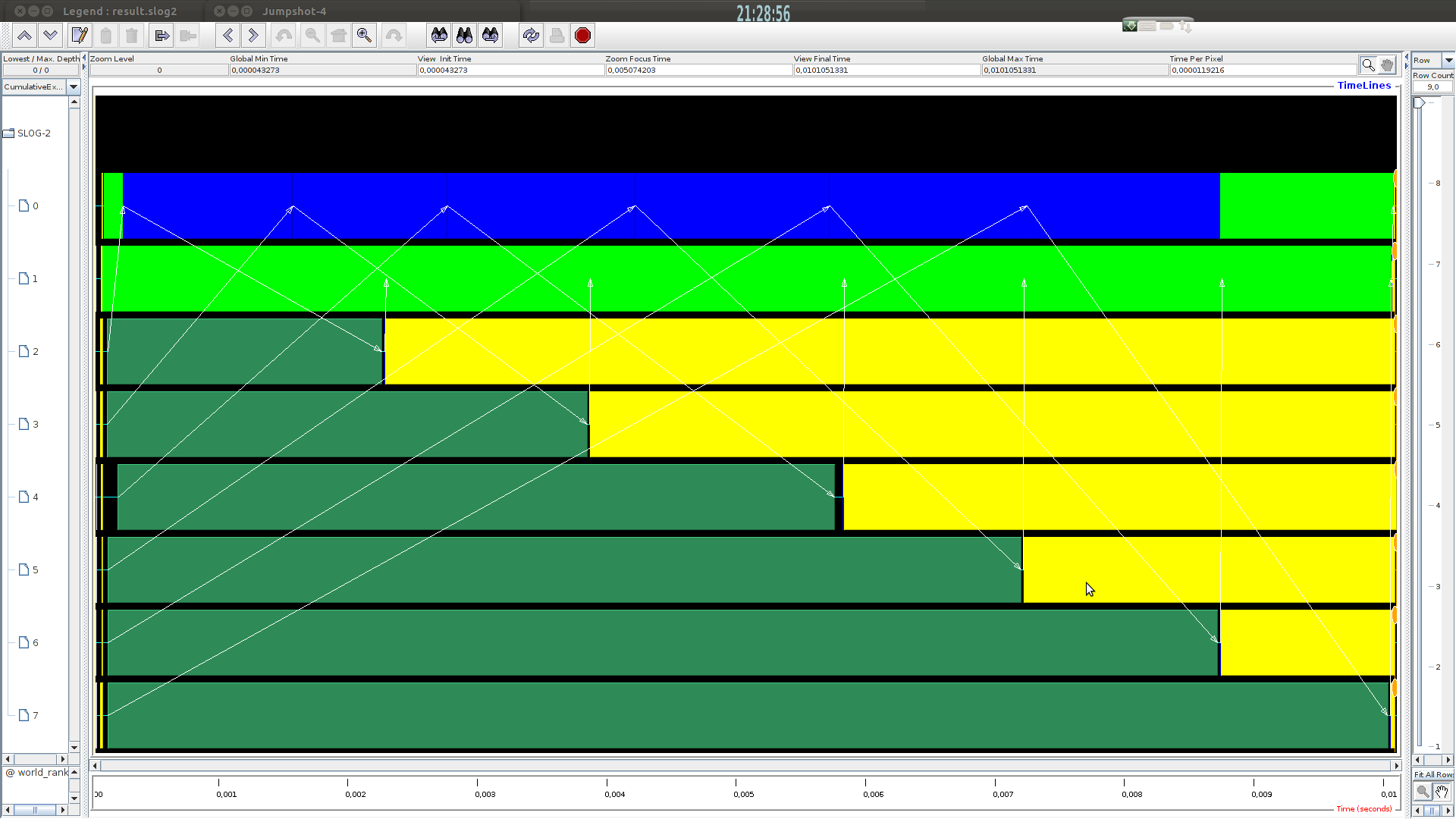


Рис. 8 – Карта процессов для программы на графе.

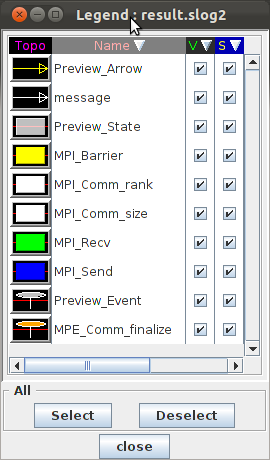


Рис. 9 – Легенда для карты процессов

## Измерение времени передачи массива поэлементно

В этом эксперименте массив из 1 000  000 элементов передавался по одному числу. Результаты эксперимента приведены в таблице и на рисунке 10.

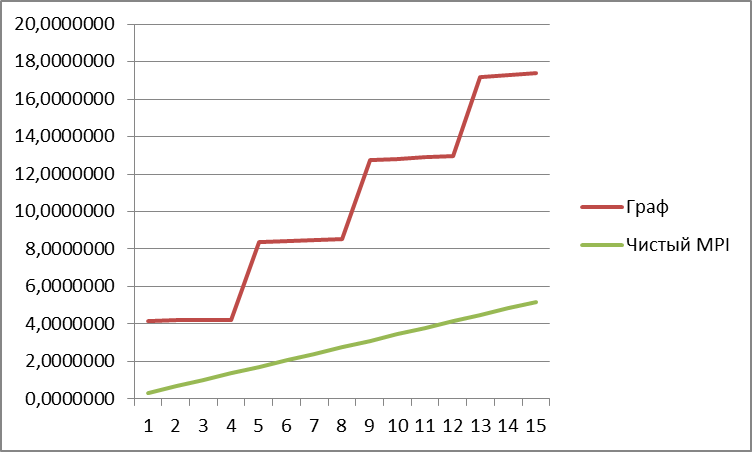


Рис. 10 – Время передачи массива поэлементно

Время передачи массива для программы на графе имеет ступенчатую структуру. Это связано с наполнением буфера. (КАКОГО???)

## Вывод

Как видно из сравнения графика на рисунке 6 и 10 механизм эмуляции общей памяти, основанный на технологии MPI, может оптимизировать работу самой библиотеки, но для многих мелких передач не годится. Стоит отметить, что сделанные выводы годятся лишь для определённой архитектуры кластера и не проверялись для других архитектур.

ССЫЛКИ

1. Антонов, А. С. Параллельное программирование с использованием технологии MPI. М.: Издательство МГУ.
2. Абрамов С.М., Кузнецов А.А., Роганов В.А. Кроссплатформенная версия T-системы с открытой архитектурой // Вычислительные методы и программирование. - 2007. - Том 8, Раздел 2. - С. 18-23.
3. Введение в программирование для параллельных ЭВМ и кластеров: Учебное пособие. / Авторы-составители: Кравчук В.В., Попов С.Б., Привалов А.Ю., Фурсов В.А., Шустов В.А.; под ред. В.А.Фурсова. // Самара: Самарский научный центр РАН, СГАУ, 2000.   
   - 87 с.
4. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления - СПб.: БХВ-Петербург, 2002. - 608 с.
5. Легалов А., Кузьмин Д., Казаков Ф., Привалихин Д. На пути к переносимым параллельным программам. // Открытые системы. - 2003. - № 5.
6. Тыугу Э.Х. Концептуальное программирование. - М.: Наука, 1984. - 256 с.
7. Tyugu, E., Valt, R. Visual programming in NUT. // Journal of visual languages and programming. -1997. - v. 8. - C. 523 - 544.
8. Котов В.Е. Сети Петри. - М.: Наука, 1984. - 160 с.
9. Коварцев А.Н. Автоматизация разработки и тестирования программных средств. - Самар. гос. аэрокосм. ун-т., Самара, 1999. – 150 с.
10. Гергель В.П. Теория и практика параллельных вычислений. – М.: Бином, 2007. – 423 с.