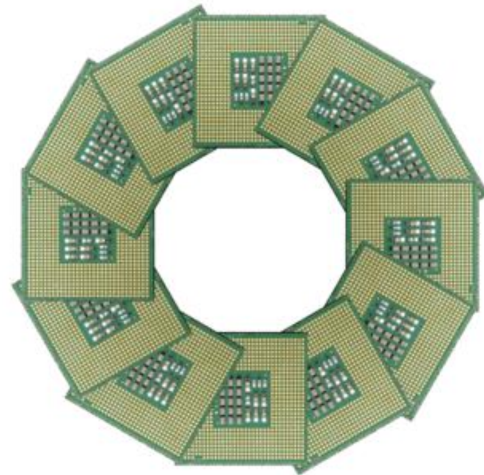
Программирование для высокопроизводительных вычислений



План изложения:

- Две парадигмы программирования
- Методология проектирования параллельных алгоритмов
- Декомпозиция для выделения параллелизма
- Концепция передачи сообщений
- Передача сообщений

Две парадигмы программирования. Проблемы:

С появлением параллельных систем возникли новые проблемы:

- как обеспечить эффективное решение задач на той или иной параллельной системе,
- какими критериями эффективности следует пользоваться.

Проблемы описания классов:

- задач, которые естественно решать на данной параллельной системе,
- задач, не поддающихся эффективному распараллеливанию.

Проблемы:

Метод распараллеливания алгоритма?

Желательно обеспечить переносимость полученной программы на систему с другой архитектурой.

Требуется сохранить работоспособность программы и улучшить ее характеристики при модификации данной системы:

как обеспечить работоспособность программы при увеличении количества параллельных модулей.

Причины возникающих трудностей:

- идеология распараллеливания трудно воспринимается после приобретения навыков последовательного программирования,
- методы распараллеливания задач недостаточно разработаны.

Два основных подхода к распараллеливанию вычислений

Параллелизм данных и параллелизм задач.

В англоязычной литературе соответствующие термины – data parallel и message passing.

В основе обоих подходов лежит распределение вычислительной работы по доступным пользователю процессорам параллельного компьютера.

Какие проблемы приходится решать?

Достаточно равномерная загрузка процессоров, так как если основная вычислительная работа будет ложиться на один из процессоров, мы приходим к случаю обычных последовательных вычислений, и в этом случае никакого выигрыша за счет распараллеливания задачи не будет.

Скорость обмена информацией между процессорами. Если загрузка процессоров достаточно равномерная, но скорость обмена данными низкая, основная часть времени будет тратиться впустую на ожидание информации, необходимой для дальнейшей работы данного процессора.

Модель параллелизма данных

Основывается на параллелизме, который заключается в регулярном манипулировании элементами больших монолитных структур данных – в применении одной и той же операции к множеству элементов структур данных таких, как массивы.

Желательно различные фрагменты такого массива обрабатывать на векторно-конвейерном процессоре или на разных процессорах параллельной машины.

Распределением данных между процессорами также занимается программа.

Векторизация или распараллеливание в этом случае чаще всего выполняется уже на этапе компиляции – переводе исходного текста программы в машинные команды.

Роль программиста в этом случае обычно сводится к заданию опций векторной или параллельной оптимизации компилятору, директив параллельной компиляции, использованию специализированных языков для параллельных вычислений.

Модель параллельных данных (2)

- Не обладает свойствами, привязывающими ее к какой-то конкретной параллельной архитектуре.
- Основным архитектурным признаком является то, что компьютер, реализующий эту модель, работает одновременно с набором слов памяти, а не с каждым словом отдельно, как это происходит в последовательном компьютере.
- В языках программирования с параллелизмом данных набор элементов составляет "параллельный" объект, в котором каждый элемент можно интерпретировать как содержимое памяти одного из процессоров машины с параллелизмом данных.
- Языками для параллельных вычислений являются Высокопроизводительный ФОРТРАН (High Performance FORTRAN – HPF) и параллельные версии языка C (например, C*).

Параллелизм задач

- Стиль программирования подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач, и каждый процессор загружается своей собственной подзадачей.
- Компьютер при этом представляет собой MIMD-машину.
- MIMD - компьютер, выполняющий одновременно множество различных операций над множеством, вообще говоря, различных и разнотипных данных.
- Для каждой подзадачи пишется своя собственная программа на обычном языке программирования,
- Чем больше подзадач, тем большее число процессоров можно использовать, тем большей эффективности можно добиться.

Параллелизм задач (2)

- Все программы должны обмениваться результатами своей работы, практически такой обмен осуществляется вызовом процедур специализированной библиотеки.
- Программист контролирует распределение данных между процессорами и подзадачами и обмен данными.
- Требуются усилия для того, чтобы обеспечить эффективное совместное выполнение различных программ.
- По сравнению с предыдущим подходом, данный подход более трудоемкий.

- Привлекательными особенностями являются: большая гибкость и большая свобода, в разработке программы, эффективно использующей ресурсы параллельного компьютера и, как следствие, возможность достижения максимального быстродействия.

- Примерами специализированных библиотек являются библиотеки MPI (Message Passing Interface) и PVM (Parallel Virtual Machines). Эти библиотеки являются свободно распространяемыми и существуют в исходных кодах.

Методология проектирования параллельных алгоритмов

Самый простой вариант попробовать ускорить имеющуюся программу – это воспользоваться встроенными в транслятор (обычно с ФОРТРАНа или Си) средствами векторизации или распараллеливания.

При этом никаких изменений в программу вносить не придется. Однако вероятность существенного ускорения в разы или десятки раз невелика.

Трансляторы с ФОРТРАНа и Си векторизуют и распараллеливают программы очень аккуратно, и при любых сомнениях в независимости обрабатываемых данных оптимизация не проводится.

Второй этап работы

Анализ затрачиваемого времени разными частями программы и определение наиболее ресурсопотребляющих частей.

Последующие усилия должны быть направлены именно на оптимизацию этих частей.

В программах наиболее затратными являются циклы, и усилия компилятора направлены, прежде всего, на векторизацию и распараллеливание циклов.

Диагностика компилятора поможет установить причины, мешающие векторизовать и распараллелить циклы.

Возможно, что простыми действиями удастся устранить эти причины. Это может быть простое исправление стиля программы, перестановка местами операторов (цикла и условных), разделение одного цикла на несколько, удаление из критических частей программы лишних операторов типа операторов отладочной печати.

Третий этап

Замена алгоритма вычислений в наиболее критичных частях программы.

Способы написания оптимальных с точки зрения быстродействия программ существенно отличаются в двух парадигмах программирования – в последовательной и в параллельной (векторной).

Программа, оптимальная для скалярного процессора, с большой вероятностью не может быть векторизована или распараллелена.

Специальным образом написанная программа для векторных или параллельных компьютеров будет исполняться на скалярных машинах довольно медленно.

Замена алгоритма в наиболее критических частях программы может привести к ускорению программы при небольших потраченных усилиях.

Дополнительные возможности предоставляют специальные векторные и параллельные библиотеки подпрограмм.

Используя библиотечные функции, которые оптимизированы для конкретной ЭВМ, можно упростить себе задачу по написанию и отладке программы. Но программа может стать непереносимой на другие машины (даже того же класса), если на них не окажется аналогичной библиотеки.

Написание программы "с нуля"

Одинаково сложно (или одинаково просто) для машин любых типов.

Идеальный способ для разработки эффективных векторных или параллельных программ.

Изучение специфики программирования для векторных и параллельных ЭВМ.

Изучение алгоритмов, которые наиболее эффективно реализуются на ЭВМ данных типов.

Определить возможность применения векторизуемых и распараллеливаемых алгоритмов для решения конкретной задачи.

Возможна переформулировка части задачи, для применения векторных или параллельных алгоритмов.

Программа, специально написанная для векторных или параллельных ЭВМ, даст наибольшее ускорение при ее векторизации и распараллеливании.

Этапы проектирования

Описываемая методология проектирования предлагает подход к распараллеливанию, который рассматривает машинно-независимые аспекты реализации алгоритма, такие как параллелизм, на первой стадии, а особенности проектирования, связанные с конкретным параллельным компьютером, – на второй.

Подход разделяет процесс проектирования на 4 отдельных этапа:

- декомпозиция (partitioning),
- коммуникации (communications),
- кластеризация (agglomeration) и
- распределение (mapping).

Первые два этапа призваны выделить в исходной задаче параллелизм и масштабируемость, остальные этапы связаны с различными аспектами производительности алгоритма.

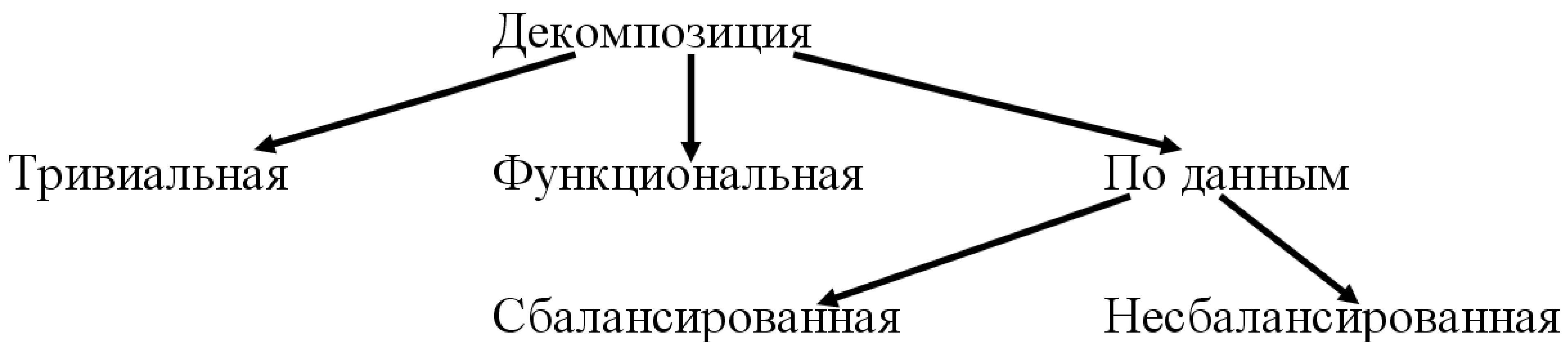
Декомпозиция для выделения параллелизма

На этапе декомпозиции, общая задача вычислений и обработки данных делится на подзадачи меньшего размера.

Игнорируются проблемы практической реализации, например, число процессоров используемого в будущем компьютера.

Все внимание сосредотачивается на возможном параллелизме исходной задачи.

Методы декомпозиции



Тривиальная декомпозиция

Предполагает, что имеется последовательная программа, которая может исполняться независимо от большого числа различных исходных данных.

Можно ввести параллелизм путем запуска некоторого числа этих программ параллельно.

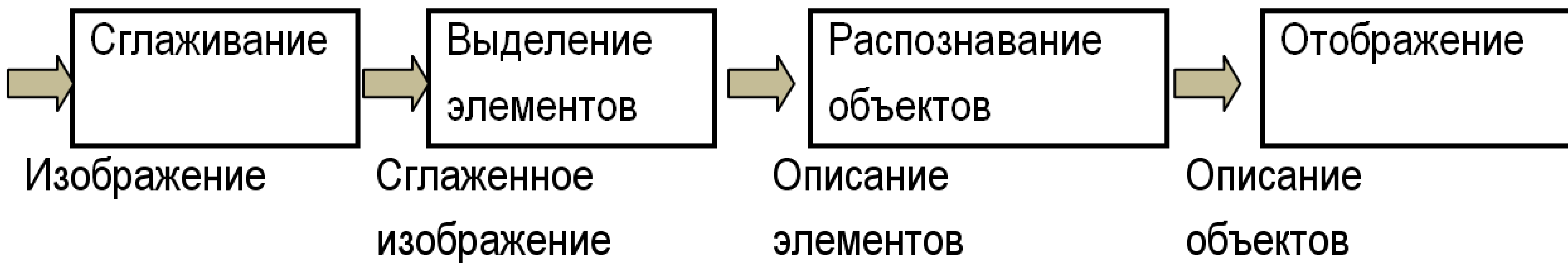
Так как нет зависимости между разными запусками программы, то число процессоров, которые могут быть использованы, ограничено только числом исполняемых запусков.

В этом случае время исполнения набора запусков будет временем исполнения самого длинного по времени запуска в наборе.

Функциональная декомпозиция

Настоящий метод декомпозиции, разбивающий программу на подпрограммы.

Простая форма функциональной декомпозиции называется "конвейер", которая очень схожа с конвейерной обработкой команд в процессоре.



Пример конвейера для обработки изображения

В примере параллелизм появляется, когда несколько входных порций данных перемещаются по конвейеру совместно.

Конвейер первоначально пуст.

Первая порция данных поступает в первую стадию конвейера (сглаживание).

Как только эта порция сглажена, она передается на следующую стадию конвейера (выделение элементов).

В то время как первая порция проходит стадию выделения элементов, вторая порция может поступать на стадию сглаживания.

Параллелизм появляется, когда вторая порция данных, проходя через конвейер, находится там одновременно с первым элементом.

Этот процесс заполнения продолжается, пока все части конвейера продвигают вперед данные.

Когда набор данных исчерпывается, появляется период просачивания, во время которого число занятых стадий в конвейере падает до нуля.

Ферма задач

Другой вид функциональной декомпозиции – это «фермы задач».

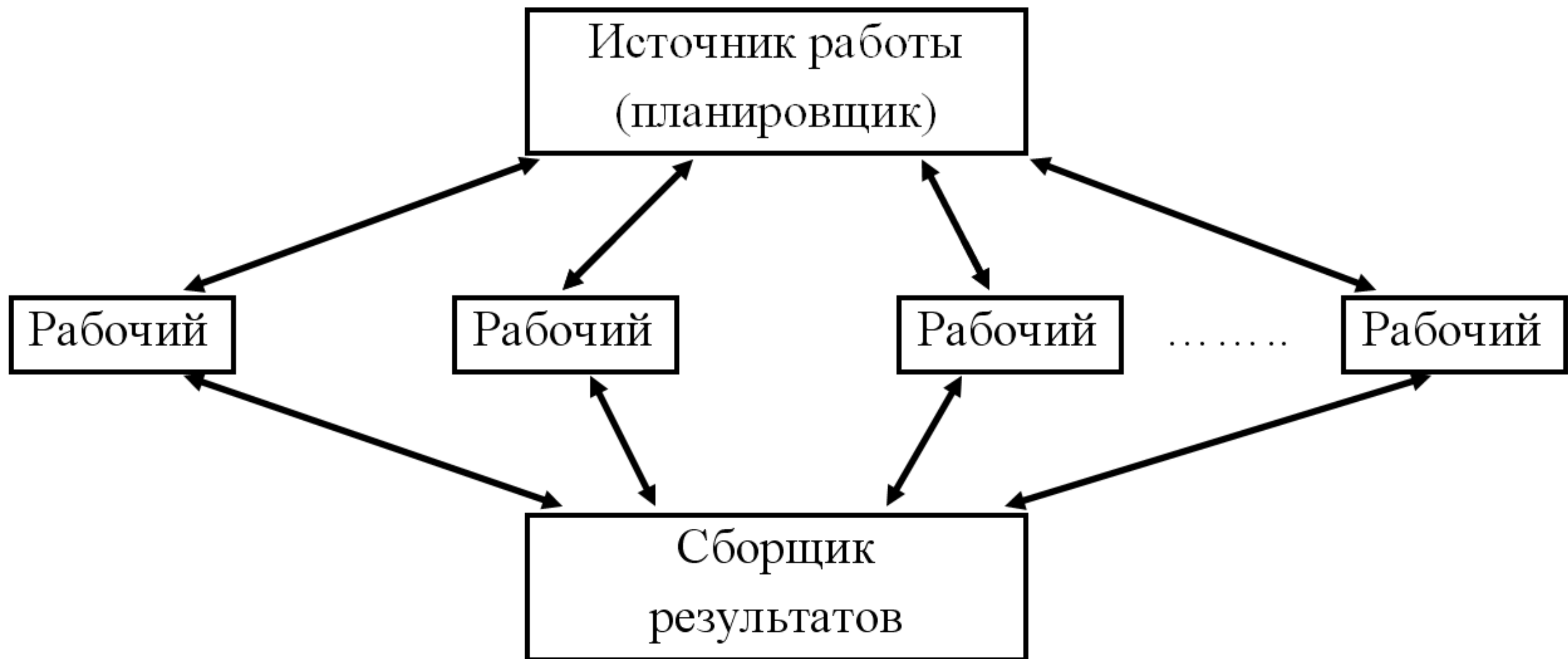
Термин "*farm*" ("ферма«) обозначает способа организации многомашинной системы, при котором одна из машин действует как планировщик, а другие – как рабочие.

Отдельная задача в функциональной декомпозиции может быть распределена между некоторым числом процессоров. Эта технология обычно так и называется – ферма задач (*task farming)*.

Для обработки выбираются небольшие порции данных и один процессор, *источник работы*, или планировщик, который управляет набором необработанных порций данных.

Некоторое число *рабочих* процессоров периодически запрашивают порцию данных из источника, обрабатывают ее и затем отправляют результаты к *сборщику* результатов, который может совпадать или нет с источником работы.

Простейшая ферма задач



Преимущество технологии фермы задач

Не требует предварительных предположений о наборе данных.

Если порции данных достаточно малы, равномерная загрузка будет обеспечена за счет того, что рабочие процессоры, которые получают порции данных, не требующих большого объема работы над ними, будут просто запрашивать больше порций данных.

Работа фермы задач требует постоянного потока запросов и ответов для частиц между рабочими машинами и управляющей.

Стоимости, связанные с поддержанием динамического баланса загрузки, являются существенными в случае, когда выполняется много итераций над набором данных.

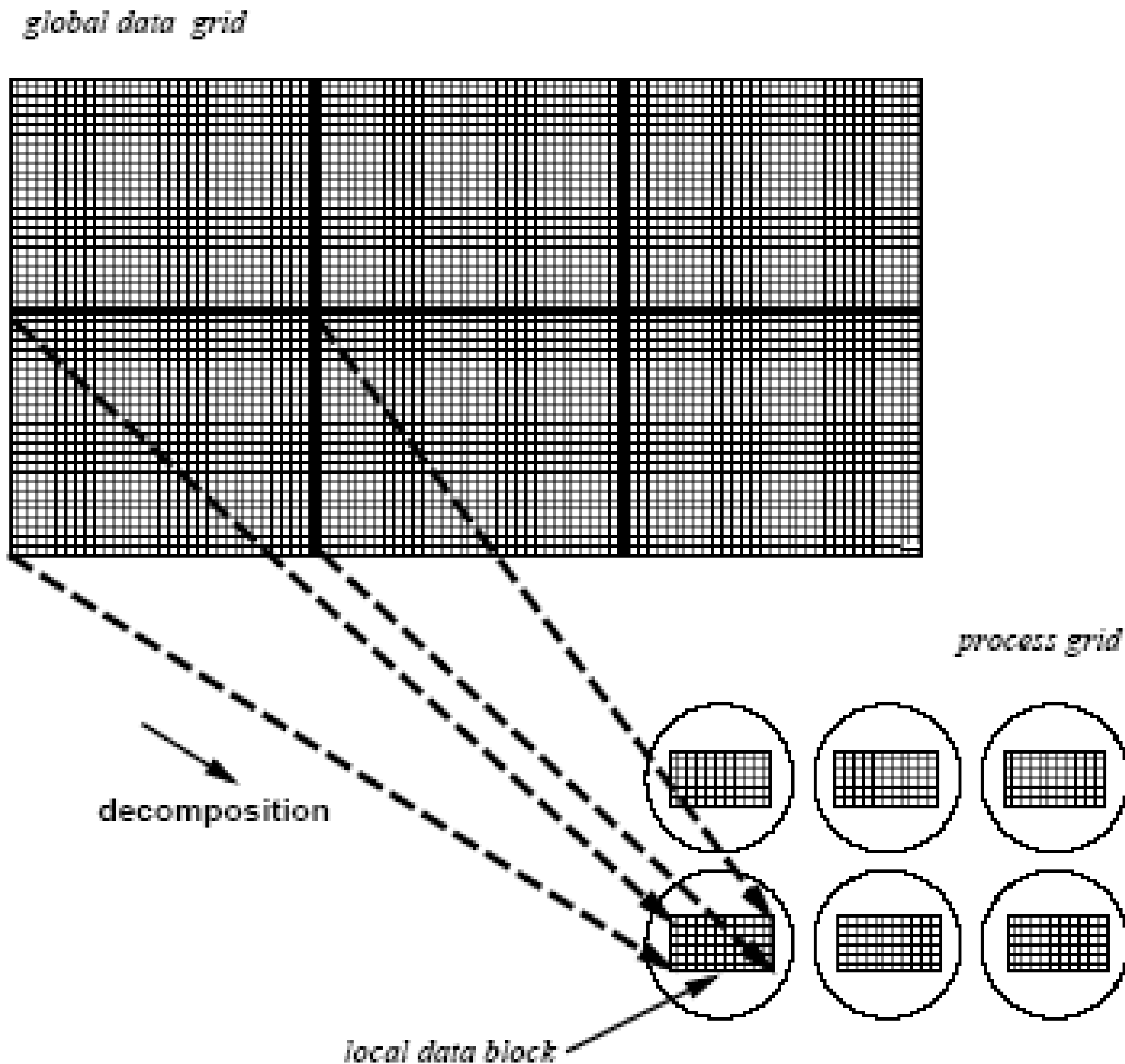
Декомпозицию по данным

Многие задачи связаны с обработкой очень больших наборов данных, распределенных в регулярной сеточной структуре, при этом применяются некоторые операции трансформации над элементами данных.

Когда данные могут быть разделены на регулярные подсетки и распределены между несколькими процессами, тогда преобразования могут быть применены параллельно, что позволяет решить задачу в меньшие сроки, чем это было бы возможно в обычных условиях.

В методе декомпозиции регулярной области берется большая сетка элементов данных, разбивается на регулярные подсетки (блоки данных), и эти блоки распределяются по отдельным процессам, где они обрабатываются.

Декомпозиция регулярной области



Область перекрытия

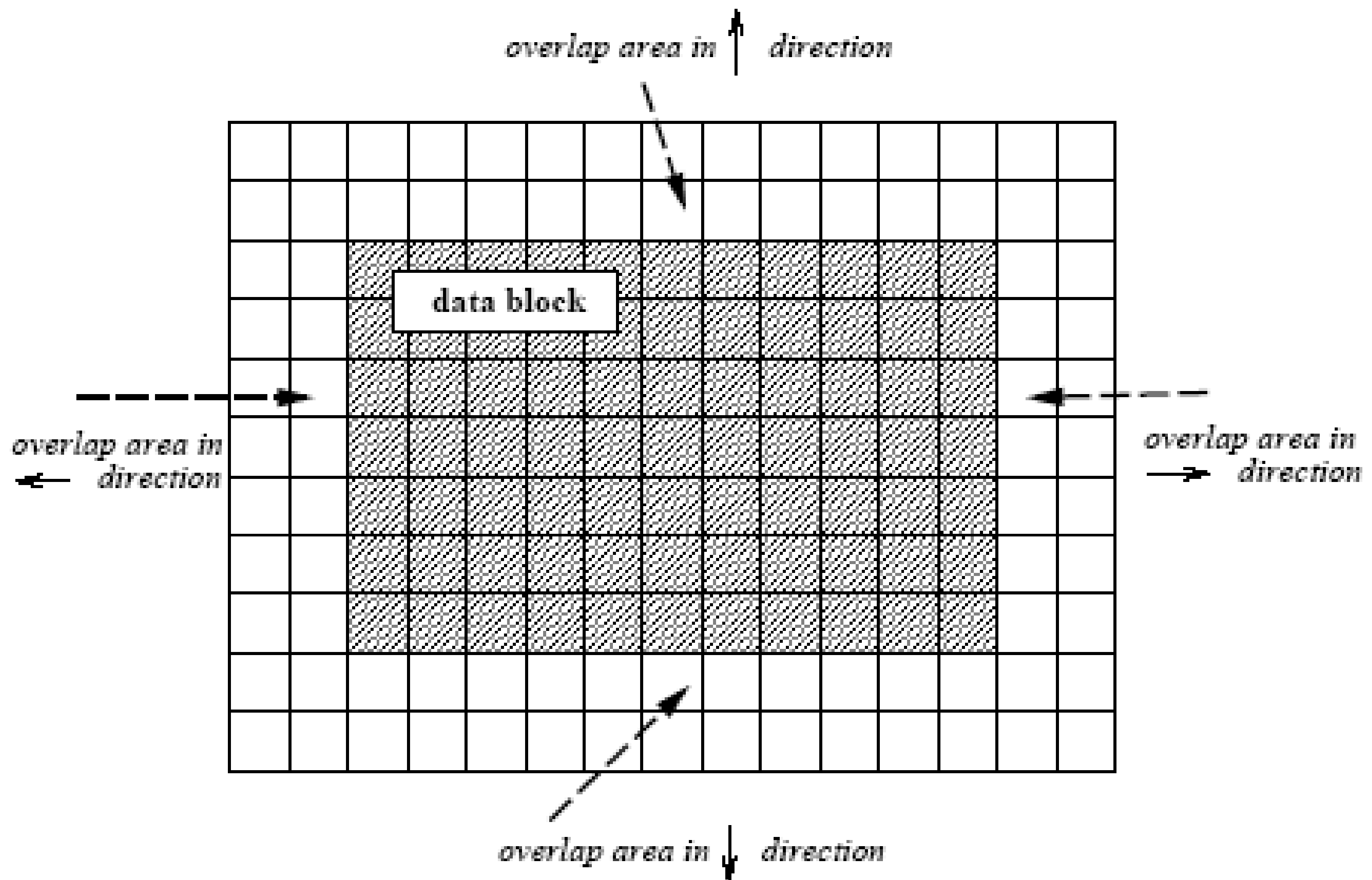
Совсем не требуется, чтобы соседние элементы были обязательно расположены в локальном блоке данных одного процессора.

Для доступа к элементам из других блоков, каждый процесс владеет некоторым "ореолом" элементов данных, покрывающих области локальных блоков данных соседних процессов.

Эта область называется областью *перекрытия*.

Блок данных, окруженный областью перекрытия, называется массивом данных,

Область перекрытия

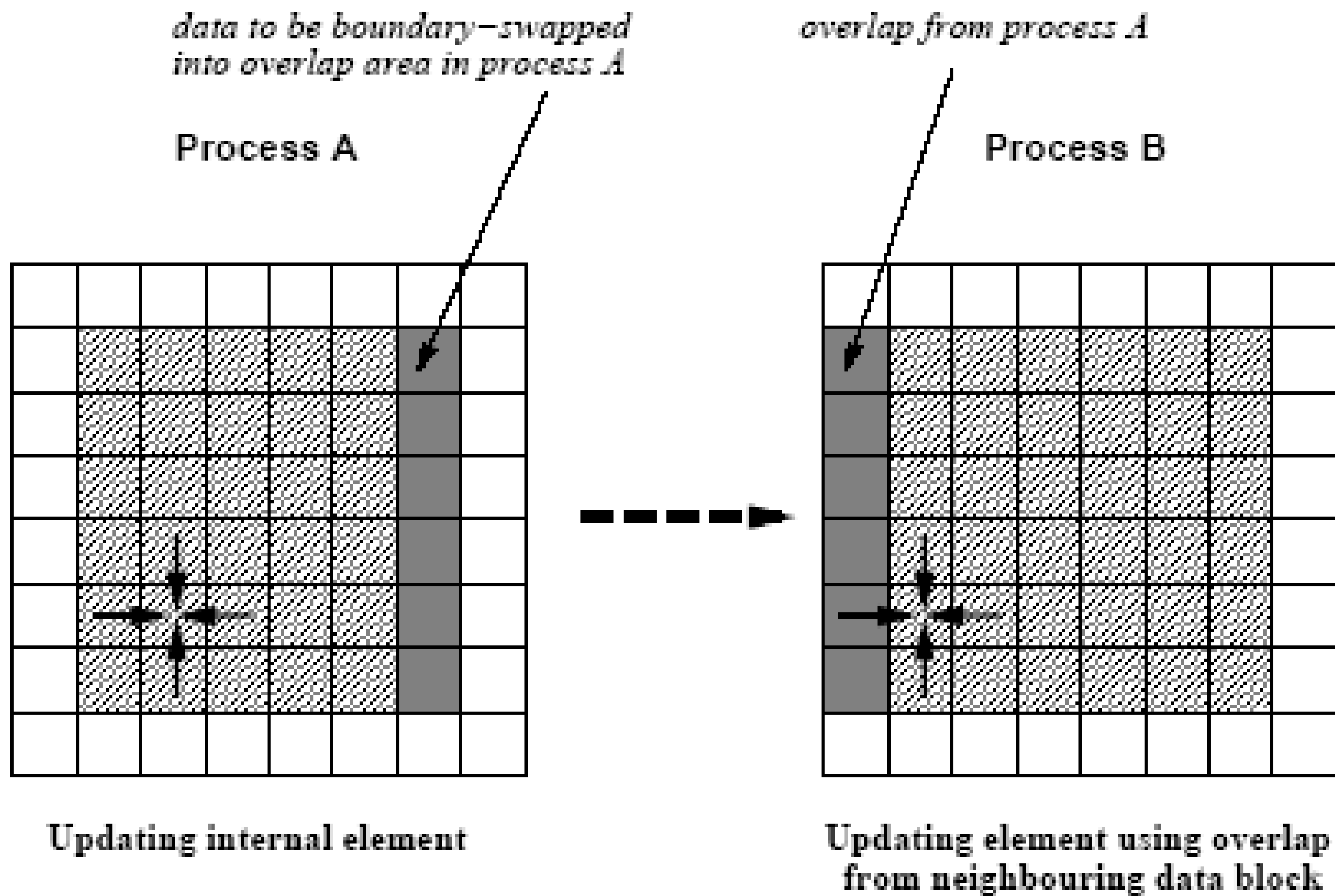


Граничная подкачка

Элементы около ребра блока данных каждого процесса требуют области элементов из блока данных соседнего процесса, расположенного в области перекрытия ореола, то каждый процесс должен быть готовым послать копию элементов ребра своего собственного блока данных соседнему процессу, позволяя ему сохранять область перекрытия в актуальном состоянии. Это называется граничной подкачкой (swapping).

На следующем слайде показаны два процесса, A и B. Процесс A выполняет обновление внутреннего элемента и поэтому он не обращается ни к каким данным, содержащимся в области перекрытия. Процесс B выполняет обновление элемента, которое требует данных из области перекрытия с процессом A. Процесс A должен запланировать исполнение граничной подкачки с процессом B перед тем как процессу B будет позволен доступ к этим данным.

Граничная подкачка (2)



Граничные условия

Имеется два основных типа граничных условий – *периодические* и *статические*.

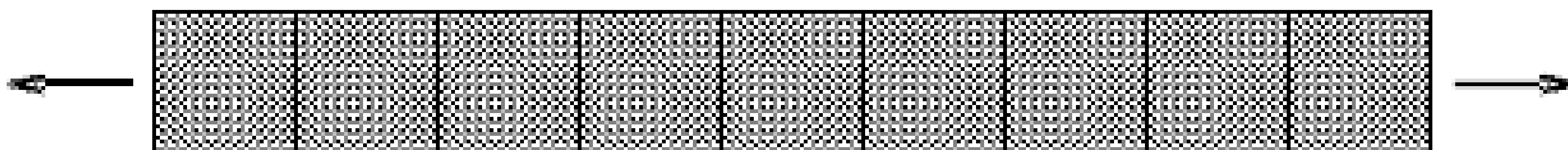
Периодические граничные условия - это такие условия, в которых сеточные данные на границе конца области перекрываются с данными на границе начала области.

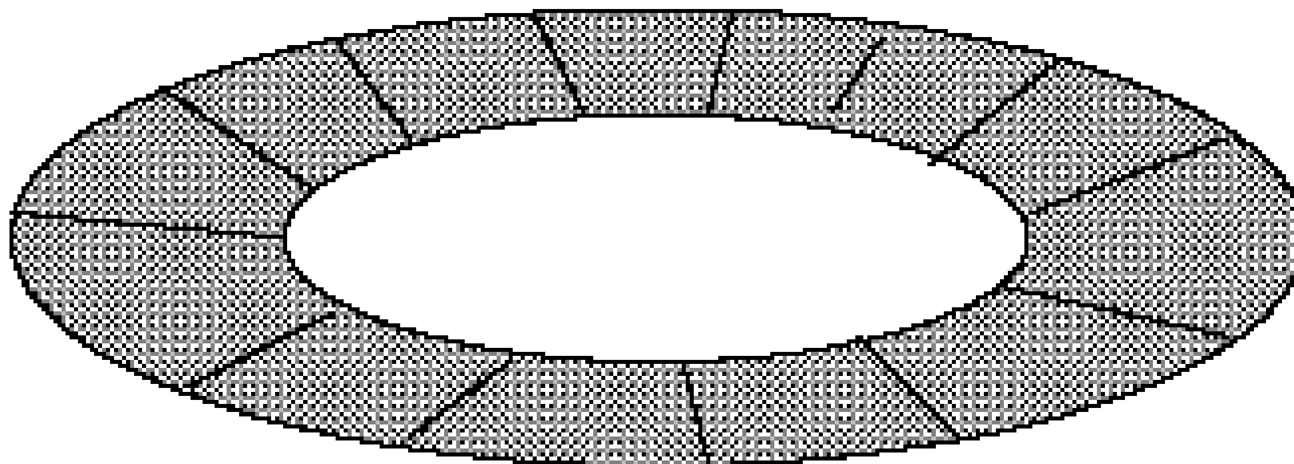
Одномерная сетка (или "вектор") с периодическими граничными условиями позволяет рассматривать сетку как "кольцо".

Двумерная сетка становится тором, и она имеет периодические границы в обоих измерениях.

Периодические граничные условия

global 1−dimensional data grid





1−dimensional grid with periodic boundary viewed as 'wraparound'

Статические граничные условия

Условия, в которых граница не завертывается.

Области перекрытия на границе глобальной сетки данных не соответствуют элементам данных вовсе.

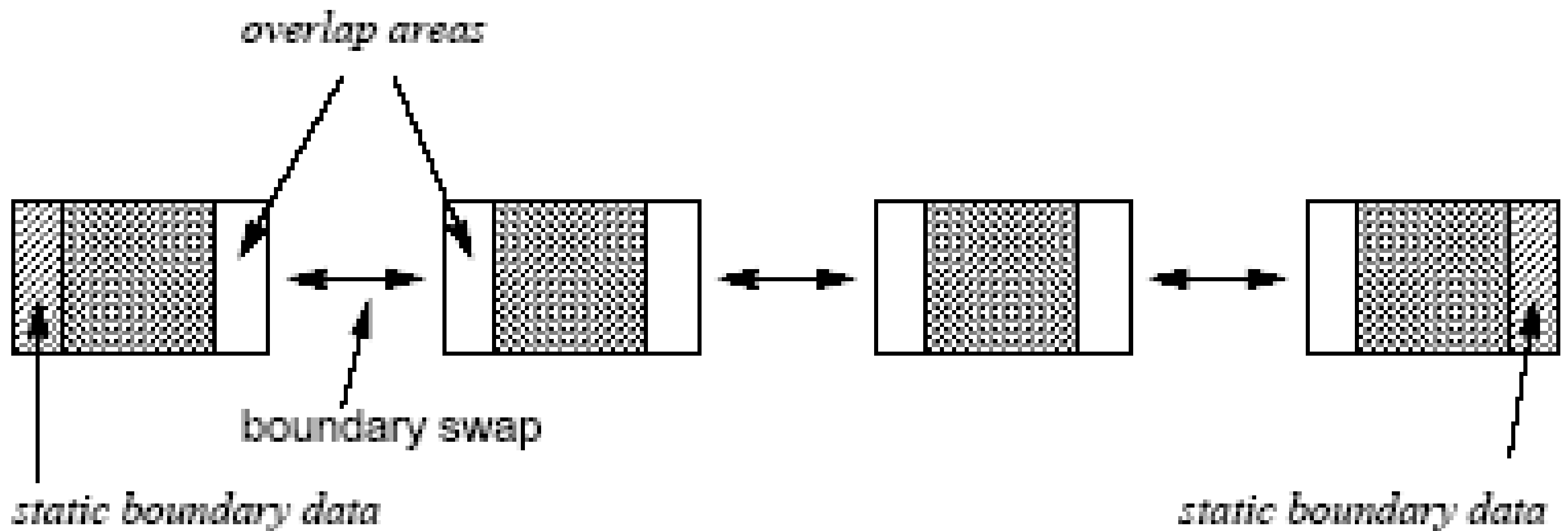
Когда приложение использует статические граничные условия, статические граничные данные должны быть инициализированы в этих областях перекрытия перед выполнением любой итерации обновления.

Следующий слайд показывает одномерную сетку со статическими граничными условиями.

Сетка разделяется между 4 процессами.

Перекрывающиеся области между процессами заполняются путем граничной подкачки, в то время как "реберные" области перекрытия содержать статические граничные данные.

Статические граничные условия (2)



Концепция передачи сообщений

В модели программирования на основе передачи сообщений, каждый процесс имеет локальную память, и никакие другие процессы не могут непосредственно иметь доступ на чтение/запись к этой локальной памяти.

Модель программирования с распределенной памятью предполагает, что нет глобально адресуемой памяти как в модели с параллельной памятью.

Общая ситуация для параллельных компьютеров – поддерживать несколько моделей программирования.

Ответственность за решение, какая из этих моделей подходит лучше для приложения, перекладывается на программиста.

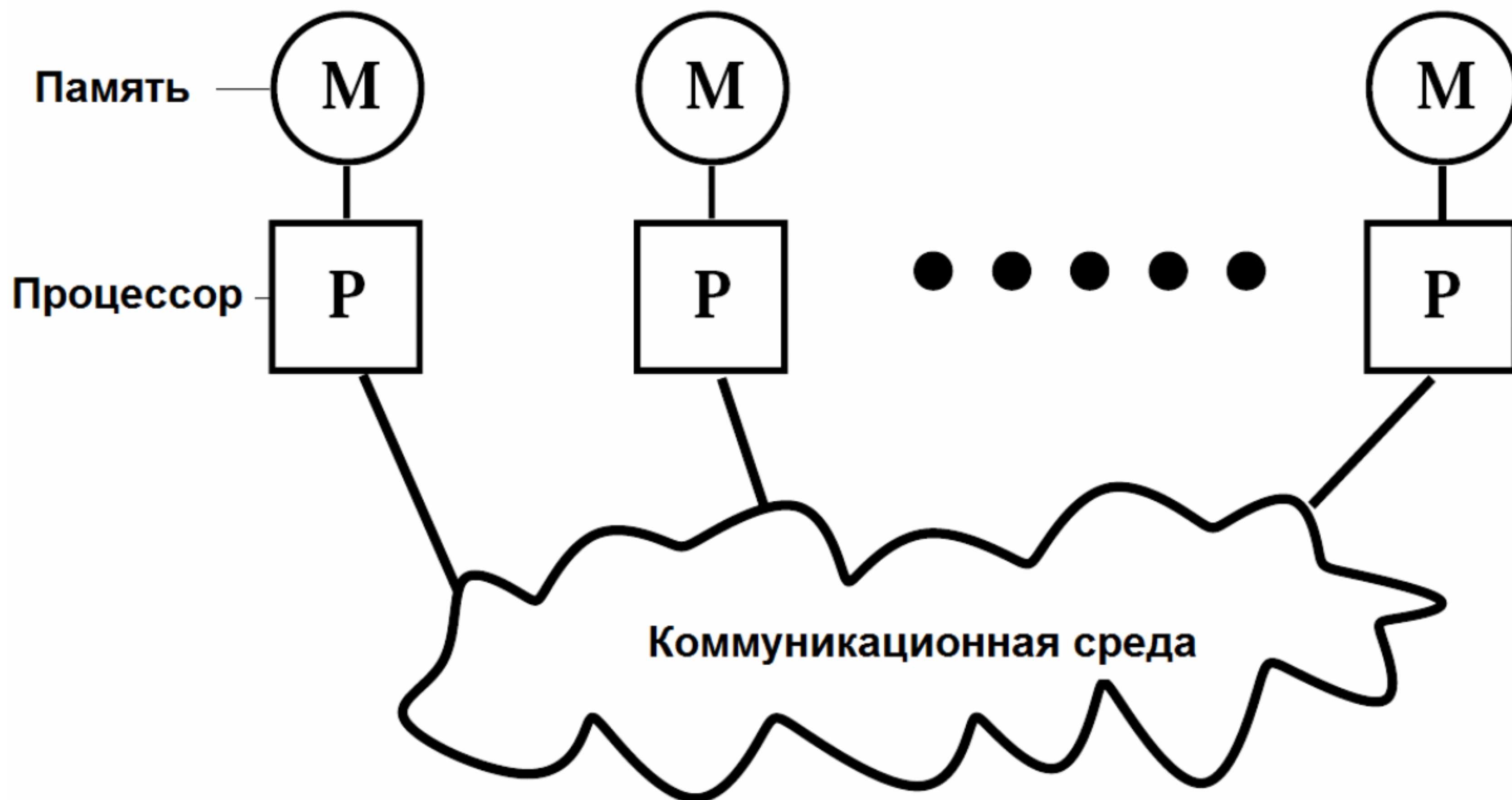
Процесс

Для прояснения целей, мы будем обращаться к отдельному процессу, координированному в параллельных вычислениях, как образующему параллельную программу или просто программу.

Термин *процесс* будет использоваться только для обозначения единичной вычислительной активности на отдельном процессоре.

Программы с передачей сообщений пишутся на тех же самых языках, что и обычные последовательные программы, однако в любой точке программист определяет действия процесса, выполняющего часть вычисления, а не описывает действия всей программы.

Архитектура параллельных компьютеров, поддерживающих модель передачи сообщений



Модель параллельных вычислений *SPMD*

Встречается ситуация, в которой программист пишет единственный исходный код программы, который компилируется и соединяется на клиентском компьютере.

Получающийся в результате объектный код копируется в локальную память каждого процессора, принимая участие в вычислениях.

Параллельная программа выполняется путем принуждения всех процессоров исполнять тот же самый объектный код.

Эта модель параллельных вычислений называется *Одна-Программа-Много-Данных (Single-Program*-Multiple-Data– *SPMD)*.

Если все переменные с одним и тем же именем имели одно и то же значение, и если все эти процессы имели доступ к тем же самым данным, они не должны быть параллельными вовсе, каждый процесс должен действовать точно также как все остальные.

Модель SPMD (2)

В контексте передачи сообщений коммуникационный интерфейс обычно включает функции или процедуры, которые возвращают значение, идентифицирующее процесс, которое называет эту функцию или процедуру.

То же самое значение будет всегда возвращаться к тому же самому процессу.

В этом случае будем называть это значение идентификатором процесса.

Последовательность инструкций, выполняемая процессом, будет определяться входными данными и идентификатором этого процесса.

Отличительной особенностью модели SPMD является то, что все процессы являются процессами того же самого типа.

Необходимо надлежащим образом запускать процессы различного типа, записывая условные операторы, чьи ветви вызывают соответствующий код для каждого процесса, в зависимости от его соответствующего идентификатора.

Модель параллельных вычислений– SPMD

На практике имеются случаи, когда SPMD модель не подходит.

Например, согласно SPMD модели выполнение всех типов процессов должно сопровождаться их загрузкой на каждый процессор.

В зависимости от приложения это может занимать слишком большое количество памяти каждого процессора.

Альтернативная модель параллельных вычислений называется *Много-Программ-Много-Данных* (Multiple-Program-Multiple-Data – SPMD), где различные загрузочные модули могут быть исполнены на различных процессорах.

Модель MPMD (2)

Предполагает возможность загружать объектный код на процессор во время параллельных вычислений и является очень естественной на сети рабочих станций.

Некоторые поставщики мультикомпьютеров упрощают своё системное программное обеспечение путем обеспечения только параллельной среды, которая поддерживает SPMD программы, обеспечение MPMD модели на мультикомпьютерах является активной исследовательской задачей.

Сообщения

Это центральное понятие для модели передачи сообщений – они передаются между процессорами.

Когда два процесса обмениваются сообщением, данные копируются из локальной памяти одного процесса в локальную память другого.

Данные пересылаются в сообщении, состоящем из двух частей: содержание и оболочка.

Содержание сообщения

Это данные пользователя, которые не интерпретируются ни коммуникационным интерфейсом, ни коммуникационной системой, стоящей за этим интерфейсом.

Оболочка сообщения

Данные в оболочке используются коммуникационной системой для копирования содержания сообщения между локальными памятями, в частности, должна быть определена следующая информация:

- Какой процессор посылает сообщение?
- Где находятся данные на посылающем процессоре?
- Какой тип данных посылается. Сколько этих данных?
- Какой процессор принимает сообщение?
- Где данные должны быть помещены на принимающем процессоре?
- Сколько данных принимающий процессор готовится принять?

Передача сообщений

В общем случае процессы посылки и приема будут кооперироваться в обеспечении передаваемой информации.

Некоторая из этой информации, которая обеспечивается посылающим процессом, будет присоединяться к сообщению при его путешествии через систему.

Другая информация может быть обеспечена принимающим процессом.

Связь точка-точка

Простейшая форма сообщения, в котором сообщение посылается от посылающего процесса к принимающему.

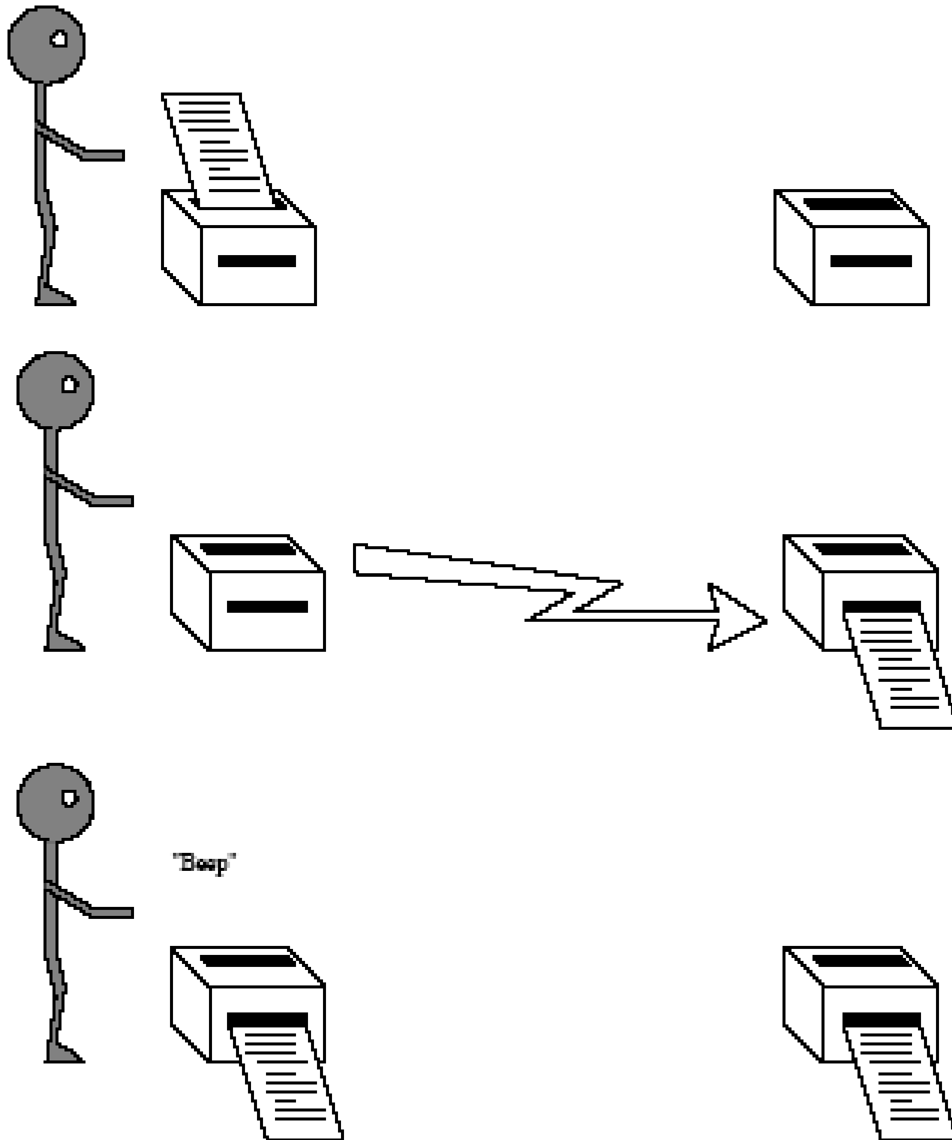
Только эти два процесса должны знать об этом сообщении.

Сообщение само по себе состоит из двух операций: отсылка и получение.

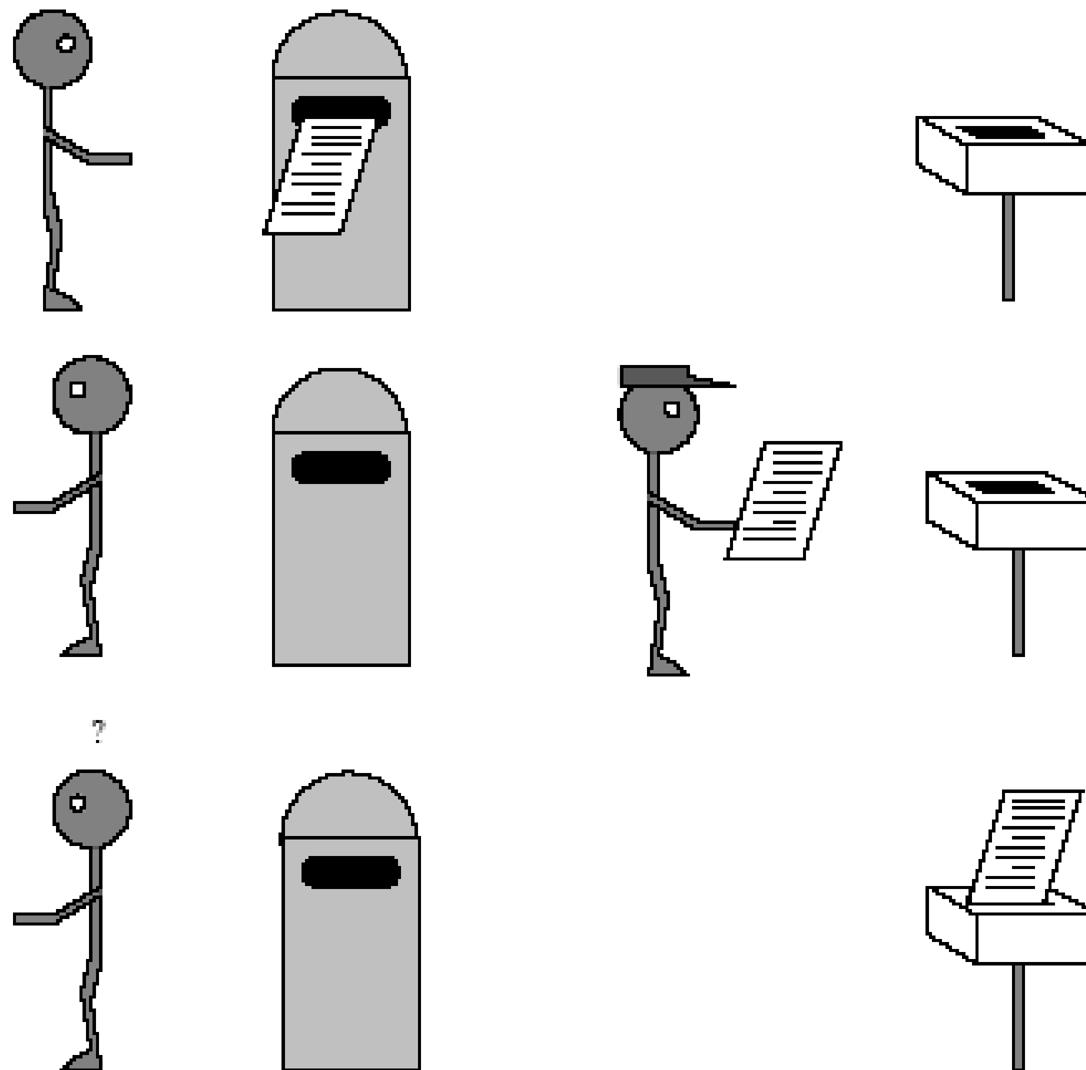
Синхронная отсылка завершается только тогда, когда о соответствующем сообщении заботится некоторый принимающий процесс.

Асинхронная отсылка завершается, как только соответствующее сообщение доставлено коммуникационной системой.

Синхронная отсылка



Асинхронная отсылка



Примеры синхронных и асинхронных операциями

Факсовое сообщение или заказное письмо являются синхронными операциями. Отправитель всегда знает, было ли его сообщение доставлено.

Почтовая карточка является асинхронным сообщением. Отправитель только знает, что положил его в почтовый ящик, но не имеет информации относительно того, было ли оно доставлено, если только получатель не пошлет ответ.

Блокирующие и неблокирующие процедуры

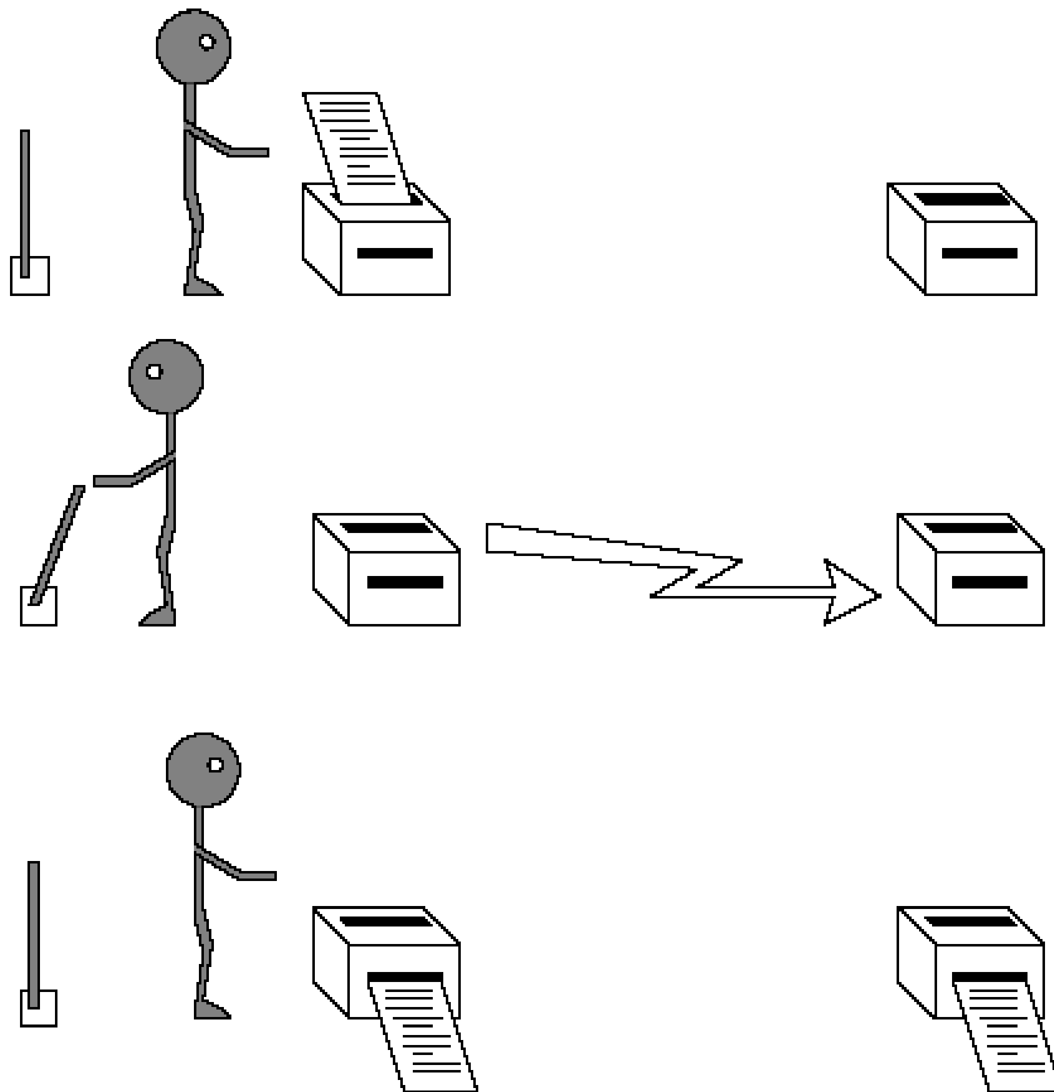
Имеется две основные формы коммуникационных процедур: блокирующие и неблокирующие.

Это разделение основано на времени завершении операции коммуникации и процедуре, которая инициирует коммуникационную операцию.

Блокирующие процедуры отдают управление только тогда, когда соответствующая коммуникация завершается.

Неблокирующие процедуры отдают управление сразу же и позволяют процессу продолжать другие действия. Некоторое время спустя процесс может протестировать завершение соответствующей коммуникации.

Неблокирующие коммуникации позволяют выполнить полезные действия во время ожидания завершения коммуникации



Примеры блокирующих и неблокирующих процедур

Обыкновенная факс-машина обеспечивает блокирующую коммуникацию. Факс остается занятым до тех пор, пока сообщение не будет отослано. Это позволяет вам загружать документы в память, и если удаленный номер занят, машина может продолжать пытаться справиться с отсылкой, пока вы ходите и делаете что-либо более важное.

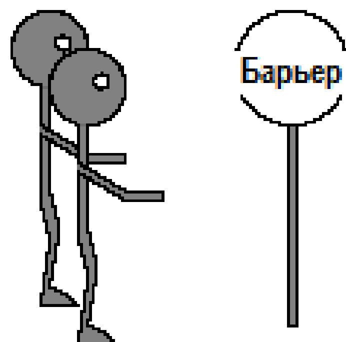
Получение сообщения может быть неблокирующей операцией. Например, если Вы включили факс-машину и оставили её включенной, так что сообщение может прибыть. Вы затем периодически тестируете её путем захода в комнату с факсом и просмотра – пришло ли сообщение.

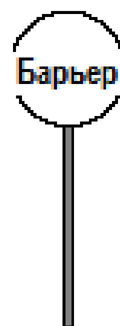
Коллективные коммуникации

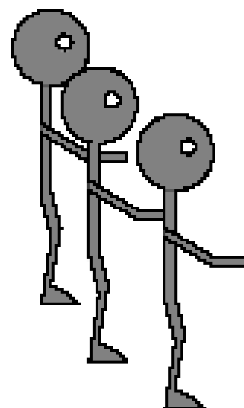
Рассмотрим 3 основных типа коллективных коммуникаций.

- Барьерная операция синхронизирует процессоры. Не происходит обмена данными, но барьер блокирует до тех пор пока все участвующие процессы не вызовут процедуру барьера
- Вещание – это коммуникация «один-ко-многим». Один процесс посылает сообщение нескольким адресатам за одну операцию.
- Операции сокращения берут несколько единиц данных от нескольких процессов и сокращают их до одной единицы данных, которая может стать, а может и не стать доступной для всех участвующих процессов.

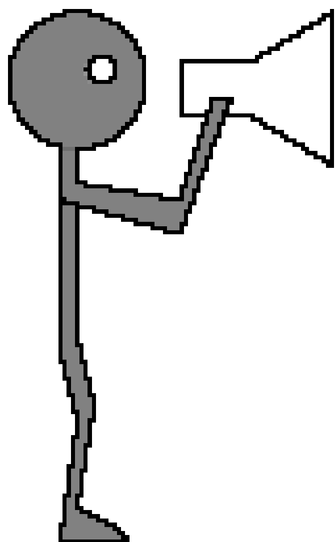
Барьер

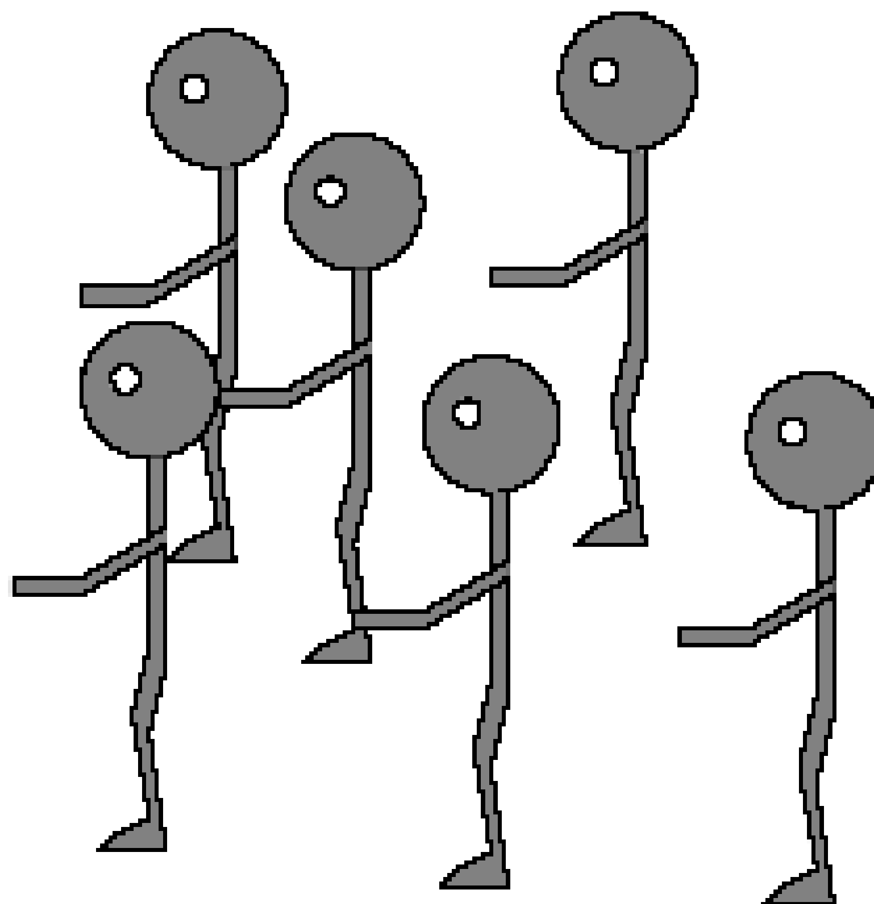




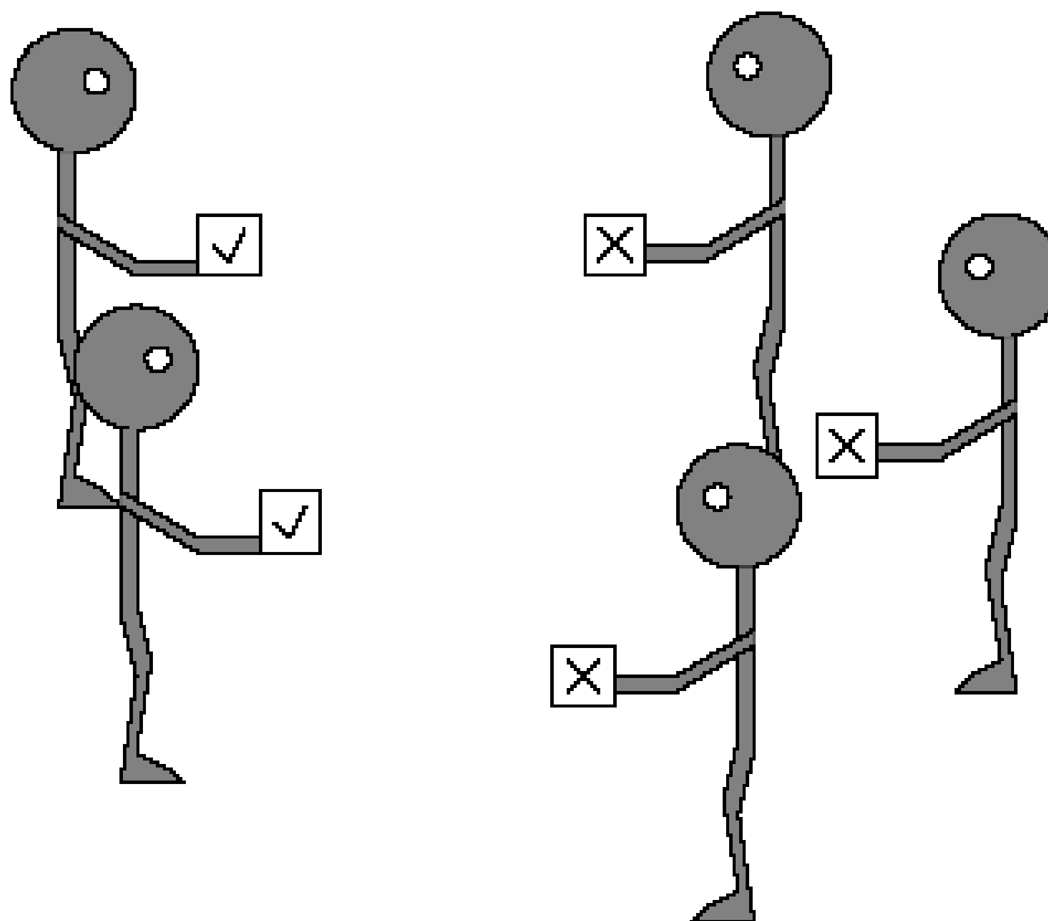


Вещание





Сокращение



Примеры

Один пример операции сокращения – это голосование нажатием, где тысячи голосов сокращаются до одного решения.

Другой пример операции сокращения – это суммирование всех значений, сохраненных во всех переменных, с именем x, в каждом процессе, принимающем участие в операции.

Перспективы парадигмы передачи сообщений

Причина популярности парадигмы передачи сообщений в том, что широкий круг платформ поддерживает эту модель.

Программы, написанные в стиле передачи сообщений, могут выполняться на распределенных процессорах или с разделяемой памятью, на сети рабочих станций, или даже на однопроцессорных системах.

Аргумент переносимости в пользу парадигмы передачи сообщений может ослабевать с годами, по мере того как другие модели программирования становятся более широко поддерживаемыми поставщиками.

Две характеристики парадигмы передачи сообщений кажутся важными в настоящее время: возможность писать коммуникационные ядра, которые полностью используют топологию коммуникационной сети и возможность менять по время выполнения программы способы распределения данных по процессорам.

Интерфейс передачи сообщений, такой, как MPI или PVM, делает возможным написание высококачественного программного обеспечения, которое является переносимым на различные платформы без жертвования производительностью.

Передача сообщений популярна не потому, что она чрезвычайно проста, а т.к. она так обща.

Перспективы программирования для высокопроизводительных вычислений

Ранее одним из недостатков параллельных вычислений была необходимость переписывать код при переносе с одной платформы на другую и техническая сложность осуществления этого.

Появление стандартов параллельных вычислений, таких, как High Performance Fortran (HPF) для программной модели параллельных данных и библиотеки Message Passing Interface (MPI) для программной модели передачи сообщений, позволили значительно продвинуться в решении этой проблемы.

Это также привлекло к параллельным вычислениям пользователей, которые не хотели тратить время и усилия на изучение низкоуровневых языков программирования, но хотели бы использовать их преимущества, в частности соотношения цены и производительности параллельных систем.

Перспективы (2)

Осведомленность в этих прошлых проблемах побудила производителей учитывать потребности расширяющегося числа пользователей.

Существенный прогресс был достигнут в области обеспечения дружественной и стабильной среды программирования, ее технического развития и поддержки.

Пользователям были предоставлены средства разработки, которые для обычных ПК уже считаются неотъемлемыми.

Параллельные вычисления, бывшие ранее заповедной зоной академических и индустриальных исследований, теперь открываются все большей аудитории, которая осознает преимущества и потенциал параллелизма.

Такие традиционно консервативные области, как финансы и коммерция, принимают параллельные вычисления как способ достижения конкурентоспособности при эффективных вложениях средств.

Возможно, именно эти области станут определяющими для будущего параллельных вычислений.