

**Федеральное агентство по образованию
Государственное образовательное учреждение
высшего профессионального образования
САМАРСКИЙ ГОСУДАРСТВЕННЫЙ
АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ
имени академика С.П. КОРОЛЕВА
(Национальный исследовательский университет)**

**«МЕТОДЫ И СРЕДСТВА ВИЗУАЛЬНОГО ПАРАЛЛЕЛЬНОГО
ПРОГРАММИРОВАНИЯ. АВТОМАТИЗАЦИЯ ПРОГРАММИРОВАНИЯ»**

*Электронные методические указания
к лабораторным работам*

Самара 2010

Составители: Коварцев Александр Николаевич,
Жидченко Виктор Викторович

В методических указаниях рассматриваются различные аспекты визуального параллельного программирования: методология визуальной разработки программных продуктов на примере системы графосимволического программирования PGRAPH, отладка параллельных программ в среде Microsoft Visual Studio 2005, выполнение заданий под управлением Microsoft Compute Cluster Server 2003, использование автоматизированных методов поиска и устранения ошибок в параллельных программах.

Методические указания предназначены для магистров направления 010400.68 «Прикладная математика и информатика», изучающих курс «Методы и средства визуального параллельного программирования. Автоматизация программирования», и предназначены для более глубокого освоения программы курса.

Лабораторная работа №1
**РАЗРАБОТКА ПОСЛЕДОВАТЕЛЬНЫХ АЛГОРИТМОВ
В СИСТЕМЕ ГРАФОСИМВОЛИЧЕСКОГО
ПРОГРАММИРОВАНИЯ GRAPH**

1. ТЕХНОЛОГИЯ ГРАФОСИМВОЛИЧЕСКОГО ПРОГРАММИРОВАНИЯ (ГСП)

1.1 Концептуальная модель ГСП

Технология ГСП - это технология проектирования и кодирования алгоритмов программного обеспечения (ПО), базирующаяся на графическом способе представления программ. Технология преследует цель полной или частичной автоматизации процессов проектирования, кодирования и тестирования ПО.

Данная технология программирования исповедует два основополагающих принципа:

- *визуальную, графическую форму представления алгоритмов программ и других компонент их спецификации;*
- *принцип структурированного процедурного программирования.*

В качестве методологической основы для представления алгоритмов в ГСП используется модель объекта с дискретными состояниями [27]. Основу такой модели составляет предположение, что для любого объекта программирования тем или иным способом можно выделить конечное число состояний, в которых он может пребывать в каждый момент времени. Тогда развитие вычислительного процесса можно ассоциировать с переходами объекта из одного состояния в другое.

Под состояниями в технологии ГСП понимают ансамбли конкретизаций структур данных (входных или вычисляемых), используемых в алгоритме. На каждом шаге алгоритма реализуется переработка текущего состояния структур данных D в новое состояние D^* с помощью некоторой локальной функции $D^* = A_k(D)$. Процесс переработки $D^0 = D$ в $D^1 = A_{k_1}(D^0)$, D^1 в $D^2 = A_{k_2}(D^1)$ и т.д. продолжается до тех пор, пока не появится сигнал о получении решения.

При этом алгоритм интерпретируется как некоторая вычислимая функция $A : in(D) \rightarrow out(D)$,

где $in(D)$ - множество входных данных программного модуля A , $out(D)$ - множество выходных (вычисляемых) данных программного модуля A .

Определим граф состояний G как ориентированный помеченный граф, вершины которого - суть состояния, а дугами отмечаются переходы системы из состояния в состояние.

Каждая вершина графа помечается соответствующей локальной вычислимой функцией f_k . Одна из вершин графа, соответствующая начальному

состоянию, объявляется начальной вершиной и, таким образом, граф оказывается *инициальным*.

Дуги графа проще всего интерпретировать как *события*. С позиций данной работы *событие* - это изменение *состояния* объекта **O**, которое влияет на развитие вычислительного процесса.

На каждом конкретном шаге работы алгоритма в случае возникновения коллизии, когда из одной вершины исходят несколько дуг, соответствующее *событие* определяет дальнейший ход развития вычислительного процесса алгоритма. Активизация того или иного события так или иначе зависит от состояния объекта, которое в свою очередь определяется достигнутой конкретизацией структур данных **D** объекта **O**.

Для реализации *событийного управления* на графе состояний **G** введем множество предикативных функций $P = \{P_1, P_2, \dots, P_l\}$. Под *предикатом* будем понимать логическую функцию $P_i(D)$, которая в зависимости от значений данных **D** принимает значение, равное 0 или 1.

Дугам графа **G** поставим в соответствие предикативные функции. Событие, реализующее переход $S_i \rightarrow S_j$ на графе состояний **G**, инициируется, если модель объекта **O** на текущем шаге работы алгоритма находится в состоянии S_i и соответствующий предикат $P_{ij}(D)$ (помечающий данный переход) истинен.

Если несколько предикатов, помечающих дуги, исходящие из одной вершины, принимают истинное значение, то переход осуществляется по наиболее приоритетной дуге. Для этого все дуги, исходящие из одной вершины, помечаются различными натуральными числами, определяющими их приоритеты.

Определим универсальную алгоритмическую модель технологии графосимволического программирования четверкой $\langle D, \mathfrak{F}, P, G \rangle$, где **D** - множество данных (ансамбль структур данных) некоторой предметной области программирования; \mathfrak{F} - множество вычислимых функций некоторой предметной области $f_k(D) \in \mathfrak{F}$; **P** - множество предикатов, действующих над структурами данных предметной области **D**; **G** - граф состояний объекта **O**.

Представленная алгоритмическая модель является универсальной, поскольку допускает описание любых алгоритмов.

Граф в данном случае заменяет текстовую (вербальную) форму описания алгоритма программы.

Существенно, что изображение программ в виде ориентированного помеченного графа естественно для восприятия человеком. Направленная дуга служит очевидным изображением перехода из одного состояния вычислительного процесса в другое, вершина - изображением 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.

Предложенная алгоритмическая модель $\langle D, \mathfrak{J}, P, G \rangle$, в конечном счете, описывает некоторую вычислимую функцию $f_G(D)$ и в этом смысле может служить “исходным материалом” для построения алгоритмических моделей других программ. Последнее означает, что технология ГСП допускает построение иерархических алгоритмических моделей. Уровень вложенности граф-моделей в ГСП не ограничен.

1.2 Объекты технологии ГСП

При программировании в технологии ГСП в качестве программных единиц рассматриваются *объекты*. По способу порождения и функциональному назначению различают три типа объектов: *акторы, агрегаты и предикаты*. Все они имеют конкретный содержательный смысл и действуют в рамках предметной области программирования (ПОП).

В предметной области, как правило, заранее уже определен терминологический словарь данных (параметров, переменных или констант). Так, например, в теории газотурбинных двигателей перечень параметров, их обозначение и содержание регламентировано государственными стандартами, которые во многом унифицированы с международными стандартами. Аналогичное положение дел имеет место в самолетостроении, ракетостроении, радиоэлектронике и т.д.

Под *предметной областью программирования* в дальнейшем понимается *среда программирования, состоящая из общего набора данных (словарь данных) и набора программных модулей (словарь и библиотека программных модулей)*, имеющая *общую цель* - разработку программного обеспечения автоматизации расчетов в некоторой области практических интересов (авиационные двигатели, бизнес, медицинские приборы и т.д.).

Поэтому программирование в рамках технологии ГСП начинается с утверждения и формирования, так называемого, *словаря данных* ПОП, который служит целям каталогизации данных ПОП, спецификации их семантики и областей значений.

Словарь данных представляет собой таблицу, в которой каждому данному присвоено уникальное имя, задан тип, начальное значение данного и краткий комментарий его назначения в ПОП.

Кроме словаря данных и каталога типов данных информационную среду определяют объекты ГСП. Под объектом понимается специальным образом построенный в рамках технологии ГСП программный модуль, выполняющий определенные действия над данными ПОП.

1.2.1. Акторы

Одним из объектов технологии ГСП является *актор*. Актор формируется из *базового модуля* путем привязки абстрактных типов данных базового модуля к данным предметной области.

Базовые модули - это локальные вычислимые функции, записанные на любом из существующих языков программирования, например, на языке C++.

Базовый модуль осуществляет отображение типов данных из области определения на область их значений $B: T_{i_1}, T_{i_2}, \dots, T_{i_n} \rightarrow T_{j_1}, T_{j_2}, \dots, T_{j_m}$, которое реализуется в соответствии с принятыми в B операциями над типами данных.

Актор производит те же действия, что и породивший его базовый модуль, но над конкретными данными ПОП. В отличие от базовых модулей, каждый актор является содержательным программным модулем, который выполняет понятные функции в рамках заданной предметной области. Акторы в технологии ГСП реализуют отображение над множеством данных предметной области:

$$A_k: D_k^{in} \rightarrow D_k^{out},$$

где $D_k^{in} = \{d_1^{in}, d_2^{in}, \dots, d_n^{in}\}$ - множество входных данных актора A_k ,
 $D_k^{out} = \{d_1^{out}, d_2^{out}, \dots, d_n^{out}\}$ - множество выходных данных актора A_k .
Множества D^{in} и D^{out} образуют в совокупности полное множество данных некоторой предметной области (словарь данных): $D = D^{in} \cup D^{out}$.

Один базовый модуль может породить множество акторов. В данном случае проявляется свойство параметрического полиморфизма базовых модулей технологии ГСП.

Между базовым модулем и актором осуществляется односторонняя связь типа "один ко многим". Каждый актор имеет свой прототип в виде базового модуля, а на основании каждого базового модуля можно построить один или несколько акторов. Это свойство полиморфизма объектов позволяет избежать избыточности при порождении новых акторов, которые различаются между собой только привязкой к данным. Другими словами, на основе одного отлаженного и оттестированного базового модуля за счет механизма автоматизированной привязки по данным можно построить несколько корректных акторов, что позволяет значительно повысить уровень надежности порождаемых программных кодов.

Порождение актора производится путем формирования, так называемого, *паспорта* объекта. Процедура паспортизации базового модуля заключается в установке соответствия между списком типов данных базового модуля и данными предметной области, таким образом, что каждому формальному параметру (типу данных) ставится в соответствие конкретное данное ПОП.

Соответствие между базовым модулем B_i и актором A_j порождает соответствие между подмножеством типов T_i данных и подмножеством самих данных D_j предметной области:

$$\begin{cases} B_i(T_i^{in}, T_i^{out}) \rightarrow A_j(D_j^{in}, D_j^{out}) \\ T_i = (T_i^{in}, T_i^{out}) \rightarrow D_j = (D_j^{in}, D_j^{out}) \end{cases}$$

В этом случае абстрактные операции над типами данных базового модуля превращаются в конкретные функциональные преобразования данных ПОП, т.е. формируется локальная вычислимая функция предметной области, например, термогазодинамический расчет компрессора ГТД.

Сформированное отношение (*паспорт* актора) оформляется как таблица БД (обозначим его $P(t,d)$) информационного фонда ГСП, содержащая перечень имен формальных параметров и соответствующих им имен данных ПО с указанием способа получения ими своих значений. По способу получения своих значений данные в паспорте делятся на три группы:

- 1) иницилируемые (импортируемые) данные (I), которые должны принять значения до их использования объектом;
- 2) вычисляемые (экспортируемые) данные (V), которые впервые получают свои значения в процессе выполнения объекта;
- 3) модифицируемые (изменяемые) данные (M), которые образуются путем пересечения множеств иницилируемых и вычисляемых данных.

1.2.2. Предикаты

В процессе реализации алгоритма в рамках технологии ГСП передача управления между объектами осуществляется с помощью управляющих объектов-предикатов. Формально предикат представляет собой отображение из множества данных предметной области на множество логических значений “истина” или “ложь”:

$$P_k : (d_1, d_2, \dots, d_m) \rightarrow \{ 0, 1 \}.$$

Отличие предиката от актора заключается в том, что предикат не может производить преобразование над данными, то есть все его данные являются входными: $(d_1, d_2, \dots, d_m) \in D^{in}$.

Технологически порождение предикатов ничем не отличается от процедуры порождения акторов. Первоначально строится базис абстрактных логических функций, действующих над типами данных. Множество предикатов ПОП

строится из абстрактных логических функций в результате аппликации их типов данных к данным ПОП, т.е. в результате паспортизации типов данных логических функций. Рассмотрение *акторов* и *предикатов*, как различных категорий *объектов* ГСП, обусловлено не только особенностями реализуемых ими типов отображений, но и, что особенно важно, их ролевыми назначениями.

Акторы, являясь локальными вычислимыми функциями, реализуют преобразование данных ПОП, в то время как предикаты, являясь функциями управления вычислениями, образуют базу знаний для всей предметной области, т.е. они общезначимы для любых программ, разрабатываемых в рамках ПОП.

1.2.3. Агрегаты

Объекты (акторы или предикаты) являются исходным материалом для визуального программирования. Результатом визуального программирования являются агрегаты. Агрегат создается в форме графа, в котором объекты ПОП играют роль вершин и дуг. Дуги - предикаты, а вершины - акторы или агрегаты. Дуги графа определяют передачу управления от одной вершины к другой.

Формально агрегат представляет собой помеченный ориентированный граф с входной (корневой) и несколькими выходными (концевыми) вершинами:

$$G = \{ F, P \},$$

где $F = \{ A_1, A_2, \dots, A_n \}$ - множество акторов, которые являются вершинами графа, $P = \{ P_1, P_2, \dots, P_m \}$ - множество предикатов, которые представляют дуги графа.

Корневой вершиной графа A_0 является такая вершина, из которой есть маршрут по графу в любую другую вершину, и которая помечена как корневая. Из этой вершины начинается выполнение алгоритма, реализованного агрегатом. Аналогично определяется конечная вершина A_n - это вершина, в которую есть маршрут из любой другой вершины, и которая не имеет исходящих дуг-предикатов. Концевых вершин на графе может быть несколько, если все они удовлетворяют поставленным условиям.

Развитие вычислительного процесса в агрегате происходит путем передачи управления из одной вершины в другую, начиная с корневой. Этот процесс может быть завершён по двум причинам: либо достигнута конечная вершина графа, из которой нет исходящих дуг, либо из текущей вершины отсутствуют разрешенные другими предикатами переходы в другие вершины. Если в процессе передачи управления сложилась ситуация, когда истинными одновременно являются несколько предикатов, то управление будет передано по предикату, имеющему наибольший приоритет.

При таком подходе, между агрегатом в технологии ГСП и блок-схемой алгоритма существуют аналогии. Отличие заключается в том, что агрегат не имеет специальных управляющих блоков (условие и выбор), и передача управления всегда осуществляется посредством проверки предиката, который в частном случае может быть тождественно истинным. Это упрощает визуальный

анализ алгоритма, за счет чего можно сократить число структурных ошибок. Например, ошибок, связанных с переусложненной структурой (неправильно вложенные циклы, неверная передача управления и т.п.), либо ошибок вызванных противоречиями в самом графе (непредусмотренные циклы).

В отличие от акторов и предикатов, которые полностью определяются своими паспортами, при порождении агрегата с помощью специального компилятора формируется текст нового объекта ПОП, который после трансляции заносится в библиотеку объектных модулей ПОП.

2. РАЗРАБОТКА ПРОГРАММ В СИСТЕМЕ ГСП GRAPH

2.1 Знакомство с системой

Система ГСП GRAPH – автоматизированная система проектирования и разработки программ с использованием технологии графосимволического программирования. Система предоставляет визуальную среду для создания графического образа агрегатов и спецификации других объектов технологии ГСП, средства автоматизированного анализа проектируемых программ с целью повышения их надежности, а также средства автоматического синтеза исходных текстов программ на основе объектов технологии ГСП. С помощью внешнего компилятора для языка программирования, на котором написаны базовые модули, система позволяет компилировать и запускать на выполнение созданные в ней программы. В текущей версии системы поддерживается язык программирования C++.

Главное окно системы изображено на рисунке 1.

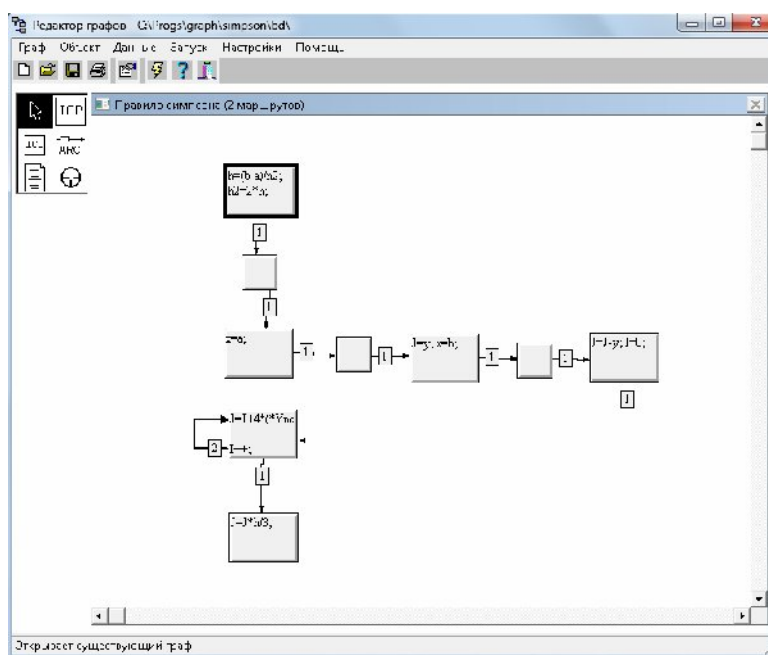


Рисунок 1 – Главное окно системы ГСП GRAPH

2.2 Создание словаря данных ПОП

В соответствии с методологией ГСП, разработка программы начинается с определения используемых типов данных и словаря данных ПОП. Эти действия выполняются в диалоговых окнах «Список типов» и «Словарь данных», вызываемых с помощью одноименных пунктов меню «Данные».

Для вновь создаваемого типа вводится его имя и определение в соответствии с синтаксисом языка C++. Базовые типы языка C++ уже содержатся в информационном фонде системы.

При создании данного (переменной) предметной области пользователь системы определяет:

- имя данного;
- тип данного (выбирается из списка существующих в предметной области типов);
- начальное значение;
- комментарий, описывающий назначение данного.

2.3 Создание базовых модулей

Базовые модули служат основой для построения объектов технологии ГСП (акторов и предикатов).

Исходный текст базового модуля создается во внешнем текстовом редакторе и сохраняется в виде файла с расширением ".C" в рабочий каталог системы PGRAPH.

Для использования базового модуля его необходимо зарегистрировать в системе. Регистрация производится путем выбора меню "Объект" -> "Зарегистрировать модуль" (рисунок 2).

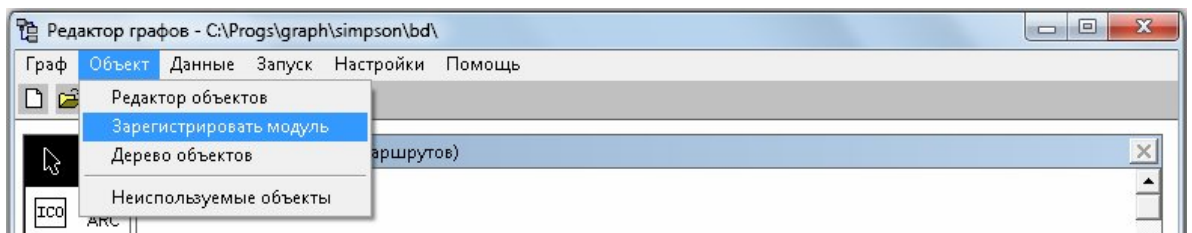


Рисунок 2 – Меню регистрации базового модуля

Диалоговое окно регистрации базового модуля изображено на рисунке 3. Процесс регистрации состоит из двух шагов. На первом шаге нужно выбрать файл с исходным текстом базового модуля. В поле «Комментарий» вводится назначение и краткое описание создаваемого базового модуля.

На втором шаге для каждого параметра базового модуля необходимо определить его класс (Исходный, Вычисляемый или Модифицируемый).

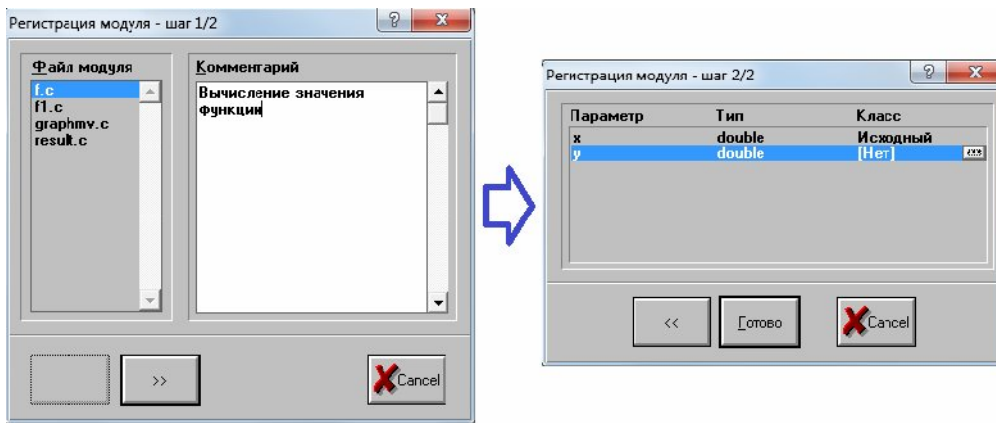


Рисунок 3 - Диалоговое окно регистрации базового модуля

После нажатия на кнопку «Готово» базовый модуль регистрируется в системе.

2.4 Создание объектов технологии ГСП

Граф-программа строится из объектов технологии ГСП: акторов и предикатов.

Для создания объектов ГСП служит «Редактор объектов», вызываемый выбором одноименного пункта в меню «Объект» (рисунок 4).

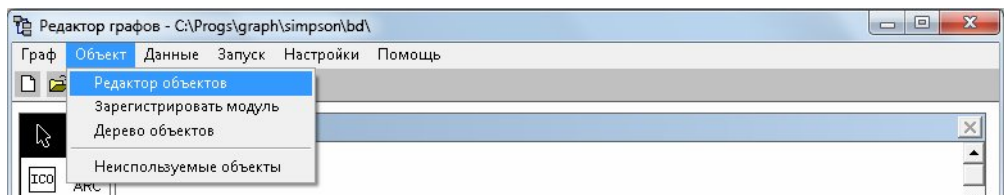


Рисунок 4 – Вызов редактора объектов

Окно редактора объектов приведено на рисунке 5.

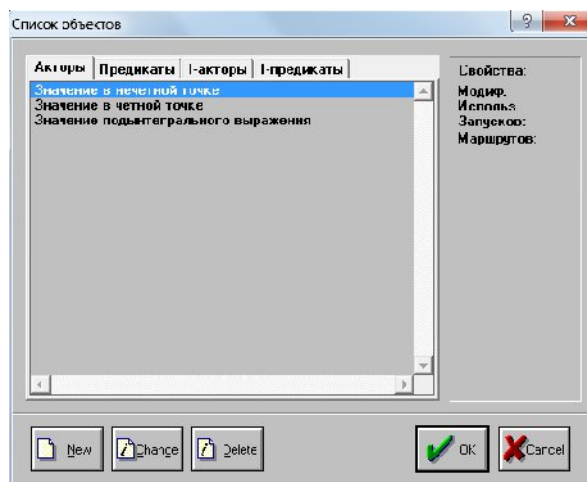


Рисунок 5 – Окно редактора объектов

Объекты ГСП условно делятся на две группы: Inline-объекты (I-акторы, I-предикаты) и обычные объекты (Актеры и Предикаты).

Обычные объекты создаются на основе базовых модулей. Inline-объекты отличаются от обычных тем, что их базовые модули создаются непосредственно при создании самого объекта. Как правило, inline-объекты выполняют простые действия. Например, I-актор может выполнять инкремент счетчика ($J = J + 1$);

В этом случае для него нецелесообразно создавать отдельный файл с базовым модулем. Вместо этого текст базового модуля вводится непосредственно при создании I-актора. Аналогично, несложные предикаты, осуществляющие например, сравнение двух переменных, оформляются в виде I-предикатов.

Для создания актора, предиката, I-актора или I-предиката необходимо выбрать соответствующую вкладку в окне Редактора объектов и нажать кнопку «Создать».

2.4.1 Создание inline-акторов

Создание I-актора состоит из двух шагов, приведенных на рисунках 6, 7.

На первом шаге (рисунок 6) вводится исходный текст I-актора и выбирается картинка (иконка), которая может отображаться на вершинах с этим I-актором вместо текста.



Рисунок 6 – Окно редактирования исходного текста inline-актора

Переход ко второму шагу происходит по нажатию на кнопку ">>".

На втором шаге для каждого данного ПОП, используемого в I-акторе, назначается его класс (исходное, вычисляемое или модифицируемое) и вводится краткий комментарий с описанием этого данного (рисунок 7).

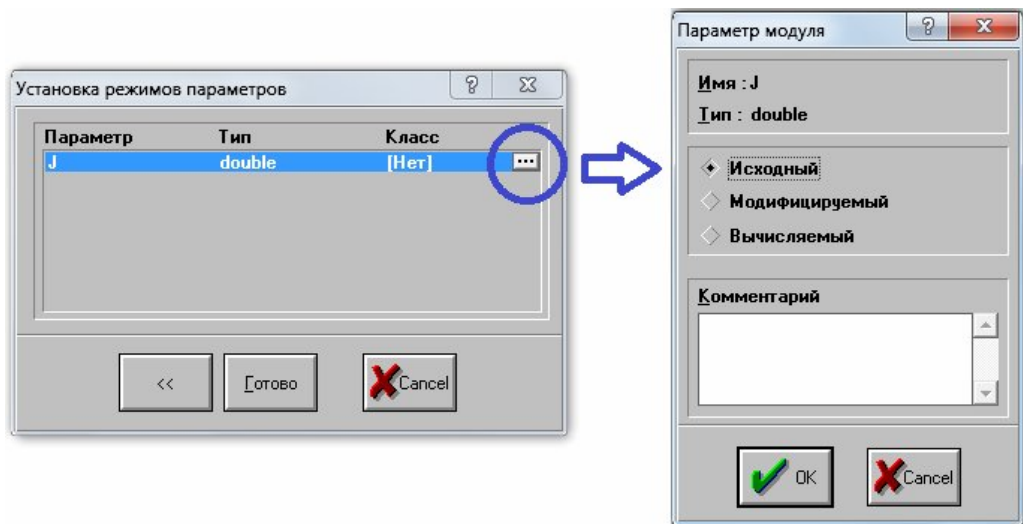


Рисунок 7 – Назначение класса использования для данных, используемых актором

2.4.2 Создание *inline-предикатов*

Создать I-предикат очень просто. Достаточно ввести его исходный текст (т.е. логическое условие над данными ПОП) и нажать кнопку «ОК» (рисунок 8).

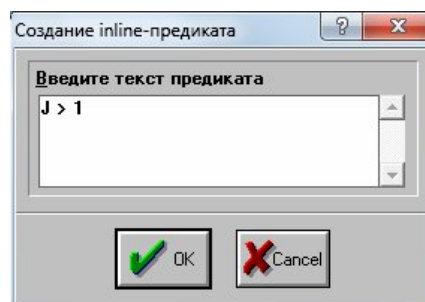


Рисунок 8 – Создание *inline-предиката*

2.4.3 Создание *акторов*

Акторы, в отличие от I-акторов, создаются на основе базовых модулей. Процесс создания актора состоит из трех шагов.

На первом шаге вводится имя для вновь создаваемого актора и выбирается картинка (иконка) для отображения на вершинах в этом актором (рисунок 9).

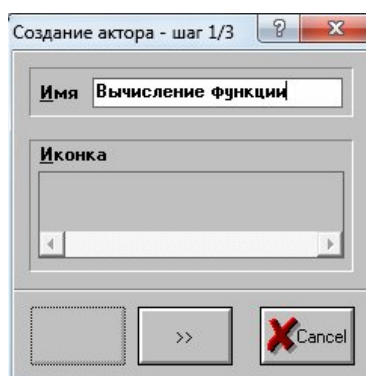


Рисунок 9 – Первый шаг создания актора

На втором шаге выбирается базовый модуль, на основе которого строится актор (рисунок 10).

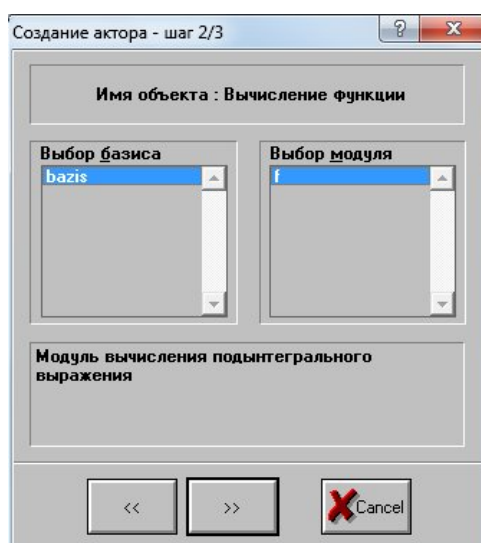


Рисунок 10 – Второй шаг создания актора

На третьем шаге производится паспортизация базового модуля: каждому его параметру ставится в соответствие данное ПОП такого же типа (рисунок 11).

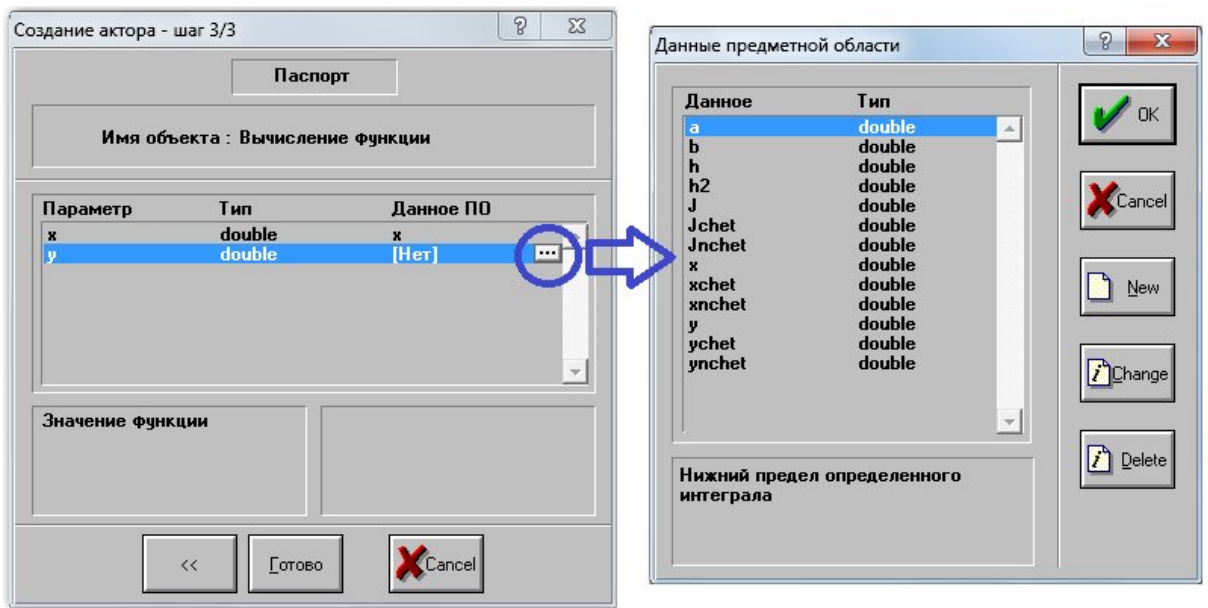


Рисунок 11 – Третий шаг создания актора (паспортизация)

2.4.4 Создание предикатов

Процесс создания предиката аналогичен процессу создания актора. Он также состоит из трех шагов: ввод имени предиката, выбор базового модуля, паспортизация.

2.5 Разработка граф-модели программы

Программа в технологии ГСП представляется в виде граф-модели – ориентированного помеченного графа (агрегата), описывающего развитие вычислительного процесса. Агрегат состоит из вершин и дуг.

Для рисования агрегата используются готовые визуальные элементы, находящиеся на панели редактирования. Внешний вид панель редактирования и назначение ее элементов приведены на рисунке 12.

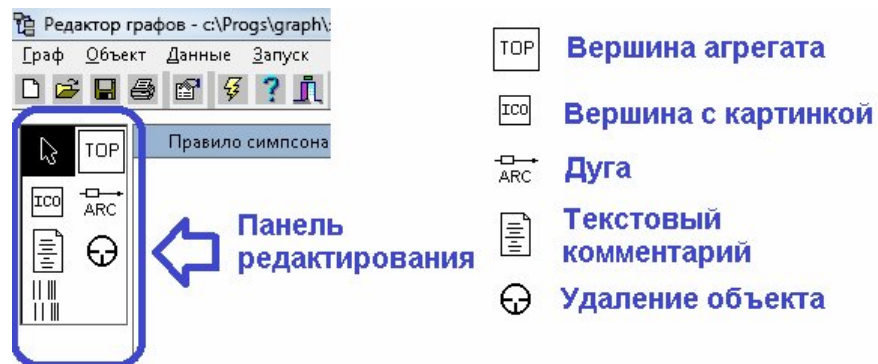


Рисунок 12 – Панель редактирования агрегата

Щелчок на изображении объекта в панели редактирования переводит в режим добавления к агрегату соответствующих объектов (вершин и дуг). Добавление осуществляется щелчком в поле редактирования агрегата.

Рисование агрегата начинается с добавления вершин. Затем вершины соединяются дугами. Пример созданного в системе агрегата приведен на рисунке 13.

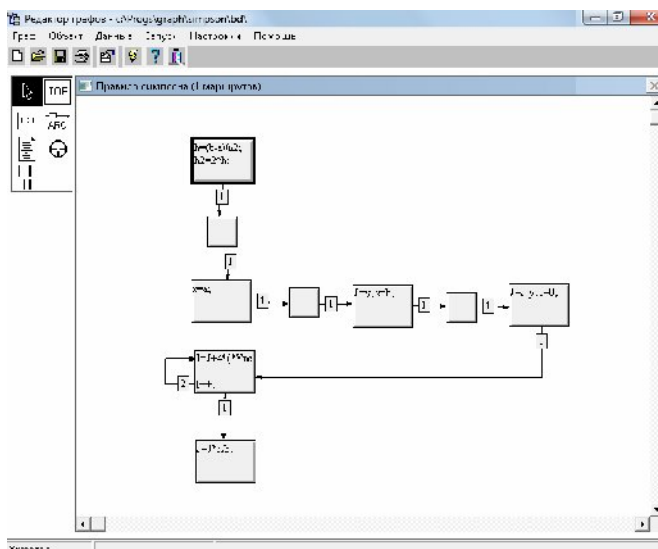


Рисунок 13 – Пример агрегата

Редактирование агрегата завершается назначением актора каждой вершине и назначением предиката каждой дуге. Для назначения актора необходимо дважды щелкнуть на соответствующей вершине и выбрать актор в открывшемся окне «Свойства вершины» (рисунок 14).

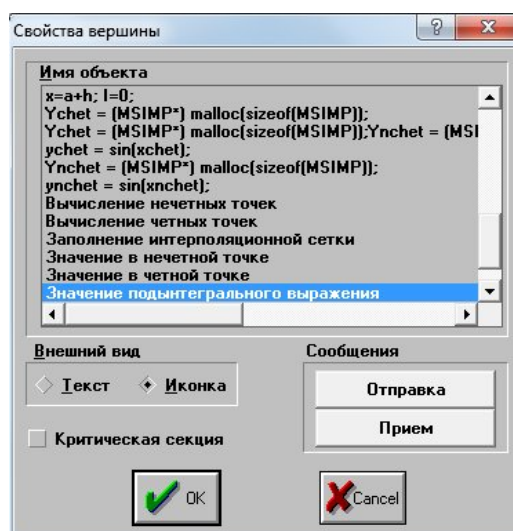


Рисунок 14 – Назначение актора для вершины агрегата

Для назначения предиката некоторой дуге агрегата необходимо дважды щелкнуть на прямоугольнике в начале дуги. После этого откроется окно «Свойства дуги», в котором можно выбрать предикат, а также изменить приоритет дуги (рисунок 15).

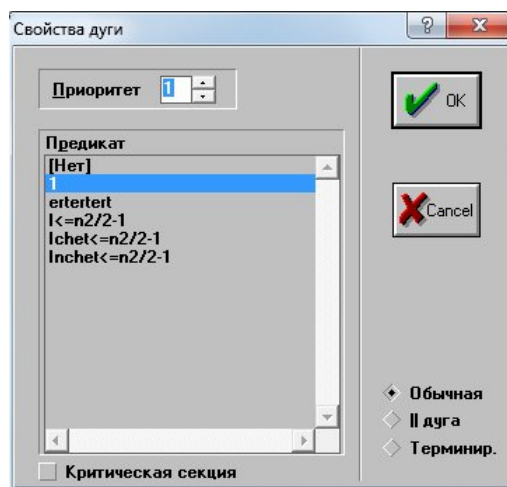


Рисунок 15 – Выбор предиката для дуги агрегата

2.6 Компиляция и запуск программы

Генерация исполняемого файла на основе созданной граф-программы осуществляется выбором пункта «Построение и запуск» в меню «Запуск».

При выборе этого пункта система строит по графу программы ее исходный текст на целевом языке программирования, вызывает внешний компилятор и запускает на выполнение созданный компилятором исполняемый файл.

3. Лабораторная работа №1

Цель работы: Разработка последовательного алгоритма в системе графосимволического программирования GRAPH.

3.1 Задание на самостоятельную работу

1. Получить задание у преподавателя.
2. Сформировать предметную область программирования для решения поставленной задачи.
3. В соответствии с заданием, используя библиотеку базовых модулей, создать необходимый набор акторов и предикатов. Для выполнения тех действий, для которых отсутствуют библиотечные базовые модули, создать inline-акторы и – inline-предикаты.

4. Используя редактор системы GRAPH, построить граф-модель программы из созданных на предыдущем шаге акторов и предикатов. Скомпилировать и произвести тестовый запуск программы..
5. Провести вычислительные эксперименты. Целью экспериментов является измерение времени работы созданной Вами программы при различных значениях размерности задачи.
6. Составить отчет по результатам работы.

3.2 Содержание отчета

1. Постановка задачи.
2. Описание предметной области программирования.
3. Описание используемых базовых модулей.
4. Описание созданных inline-объектов, а также акторов и предикатов, построенных на основе базовых модулей.
5. Описание граф-модели программы.
6. Результаты вычислительных экспериментов.
7. Выводы по работе.

3.3 Контрольные вопросы

1. Опишите концептуальные основы технологии графосимволического программирования.
2. Что такое предметная область программирования? Как она строится в системе ГСП GRAPH?
3. Опишите назначение и основные возможности системы ГСП GRAPH.
4. Что такое базовый модуль? Чем он отличается от актора? Как осуществляется работа с базовыми модулями в системе ГСП GRAPH?
5. Как порождаются акторы и предикаты в системе ГСП GRAPH?
6. Что такое inline-акторы и inline-предикаты?
7. Опишите последовательность действий при визуальной разработке программы в системе ГСП GRAPH.

СПИСОК ЛИТЕРАТУРЫ

1. Коварцев А.Н. Автоматизация разработки и тестирования программных средств. - Самар. гос. аэрокосм. ун-т., Самара, 1999. - 150 с.
2. Коварцев А.Н., Жидченко В.В. Методы и средства визуального параллельного программирования. Автоматизация программирования. Электронный учебник. - Самара, СГАУ, 2010.
3. Коварцев А.Н. Численные методы: курс лекций для студентов заоч. формы обучения. - Самара: СГАУ, 2000. – 177 с.

Лабораторная работа №2
**РАЗРАБОТКА И ОТЛАДКА
ПАРАЛЛЕЛЬНЫХ MPI ПРОГРАММ
В СРЕДЕ MICROSOFT VISUAL STUDIO**

Составитель:
профессор, д.т.н. Гергель В.П.
кафедра математического обеспечения ЭВМ
Нижегородского государственного университета им. Н.И. Лобачевского

Цель лабораторной работы	2
Обзор методов отладки параллельных программ в среде Microsoft Visual Studio 2005	2
Упражнение 1 – Тестовый локальный запуск параллельных программ.....	3
Задание 1 – Локальный запуск параллельной программы вычисления числа Пи на рабочей станции	3
Задание 2 – Настройка Microsoft Visual Studio 2005 для локального запуска параллельной MPI программы в режиме отладки	5
Задание 3 – Запуск параллельной MPI программы в режиме отладки из Microsoft Visual Studio 2005.....	7
Упражнение 2 – Отладка параллельной программы в Microsoft Visual Studio 2005.....	8
Задание 1 – Знакомство с основными методами отладки	8
Задание 2 – Использование точек останова.....	15
Задание 3 – Особенности отладки параллельных MPI программ.....	19
Задание 4 – Обзор типичных ошибок при написании параллельных MPI программ	20
Контрольные вопросы	21

Одним из важнейших этапов разработки программ является процесс поиска и устранения ошибок, неизбежно возникающих при написании сложных программных систем. Для того, чтобы упростить и во многом автоматизировать этот процесс, необходимо использовать специальные программы, называемые *отладчиками*. Несмотря на то, что сложность написания параллельных программ часто существенно превосходит сложность написания последовательных аналогов, длительное время наблюдалась нехватка качественных отладчиков для параллельных программ, разрабатываемых с использованием технологии MPI. Новейшая интегрированная среда разработки Microsoft Visual Studio 2005 имеет в своем составе отладчик, предоставляющий широкие возможности для поиска и устранения ошибок, в том числе и для отладки MPI приложений.

Цель лабораторной работы

Цель данной лабораторной работы – получение практических навыков отладки параллельных MPI программ в среде **Microsoft Visual Studio 2005**. При этом будет предполагаться, что на рабочей станции установлен Compute Cluster Server SDK, и, следовательно, используется реализация MPI от компании Microsoft (библиотека MS MPI):

- Упражнение 1 – Тестовый локальный запуск параллельных программ,
- Упражнение 2 – Отладка параллельной программы в Microsoft Visual Studio 2005.

Примерное время выполнения лабораторной работы: **90 минут**.

Обзор методов отладки параллельных программ в среде Microsoft Visual Studio 2005

Отладка (поиск и устранение ошибок) – один из важнейших этапов в написании программных систем, часто занимающий у разработчика больше времени, чем написание отлаживаемого кода. Для того, чтобы максимально снизить время, затрачиваемое на отладку, необходимо придерживаться рекомендаций крупнейших производителей программного обеспечения и ведущих исследователей в области компьютерных наук еще на этапах проектирования и программирования. Например, подобные рекомендации обычно требуют подготавливать автоматические тесты для всех участков разрабатываемой системы и контролировать корректность значений внутренних переменных с использованием макроса **ASSERT**. Но и сама процедура отладки должна проводиться эффективно, в соответствии с рекомендациями ведущих экспертов в этой области. Огромное значение здесь имеет правильный выбор **отладчика** – специальной программы, существенно упрощающий процесс поиска ошибок, позволяющий выполнять программу в пошаговом режиме, отслеживать значения переменных, устанавливать точки останова и т.д. Важнейшими критериями выбора отладчика являются богатство предоставляемого программисту инструментария и удобство использования. В ходе выполнения данной лабораторной работы мы будем использовать стандартный отладчик среды Microsoft Visual Studio 2005, удачно сочетающий в себе интуитивно понятный пользовательский интерфейс и широкий спектр предоставляемых возможностей.

Наиболее частым приемом отладки является приостановка работы программы в заданный момент времени и анализ значений ее переменных. Момент, в который программа будет приостановлена, определяется выбором, так называемых, точек останова, то есть указанием строк кода исходной программы, по достижении которых выполнение останавливается до получения соответствующей команды пользователя. Помимо простого указания строк кода, где произойдет приостановка, возможно также указание условий, которые должны при этом выполняться. Например, можно дать указание приостановить выполнение программы на заданной строке только в том случае, если значение некоторой переменной программы превысило заданную константу. Правильный выбор момента остановки имеет решающее значение для успеха отладки. Так, зачастую анализ значений переменных непосредственно перед моментом падения программы дает достаточно информации для определения причин некорректной работы.

Другим исключительно полезным инструментом является возможность выполнять программу в пошаговом режиме – по одной строке исходного кода отлаживаемой программы при каждом нажатии пользователем кнопки **F10**. При этом пользователь также имеет возможность заходить внутрь функции, вызов которой происходит на данной строке, или просто переходить к следующей строке. Таким образом, пользователь всегда находится на том уровне детализации, который ему необходим.

Для того, чтобы в полной мере оценить преимущества отладчика, необходимо предварительно скомпилировать программу в специальной отладочной конфигурации (чаще всего такая конфигурация называется **“Debug”**), особенностью которой является добавление в генерируемые бинарные файлы специальной отладочной информации, которая позволяет видеть при отладке исходный код выполняемый программы на языке высокого уровня.

К числу других часто используемых инструментов отладки Microsoft Visual Studio 2005 относятся:

- Окно “**Call Stack**” - окно показывает текущий стек вызова функций и позволяет переключать контекст на каждую из функций стека (при переключении контекста программист получает доступ к значениям локальных переменных функции),
- Окно “**Autos**” - окно показывает значения переменных, используемых на текущей и на предыдущих строках кода. Кроме того, это окно может показывать значения, возвращаемые вызываемыми функциями. Список отображаемых значений определяется средой автоматически,
- Окно “**Watch**” - окно позволяет отслеживать значения тех переменных, которых нет в окне “**Autos**”. Список отслеживаемых переменных определяется пользователем,
- Окно “**Threads**” - окно позволяет переключаться между различными потоками команд процесса,
- Окно “**Processes**” - окно позволяет переключаться между различными отлаживаемыми процессами (например, в случае отладки MPI – программы).

Отладка параллельных MPI программ имеет ряд особенностей, обусловленных природой программирования для кластерных систем. Напомним, что в параллельной программе над решением задачи работают одновременно несколько процессов, каждый из которых, в свою очередь, может иметь несколько потоков команд. Это обстоятельство существенно усложняет отладку, так как помимо ошибок, типичных для последовательного программирования, появляются ошибки, совершаемые только при разработке параллельных программ. К числу таких ошибок можно отнести, например, сложно контролируемую ситуацию *гонки процессов*, когда процессы параллельной программы взаимодействуют между собой без выполнения каких-либо синхронизирующих действий. В этом случае, в зависимости от состояния вычислительной системы, последовательность выполняемых действий может различаться от запуска к запуску параллельной программы. Как результат, при наличии гонки процессов сложным становится применение одного из основных принципов отладки – проверка работоспособности при помощи тестов (однократное выполнение теста для параллельной программы может не выявить ситуации гонки процессов).

Однако инструменты и приемы, используемые в Microsoft Visual Studio 2005 для отладки как последовательных, так и параллельных программ схожи, поэтому, если Вы имеете хороший опыт отладки последовательных программ, то и отладка параллельных программ не покажется Вам слишком сложной.

Упражнение 1 – Тестовый локальный запуск параллельных программ

Перед тем, как запускать скомпилированную программу на многопроцессорной вычислительной системе (кластере), желательно провести несколько **локальных экспериментов** на обычном персональном компьютере (в случае использования однопроцессорного компьютера все процессы параллельной программы запустятся в режиме разделения времени одного имеющегося процессора). Такой способ выполнения параллельных программ является, как правило, более оперативным (не требуется удаленный доступ к кластеру, и отсутствует время ожидания запуска программы в очереди заданий). Кроме того, такие эксперименты выполняются обычно для более простых постановок решаемых задач при небольших размерах исходных данных. Все это позволяет достаточно быстро устранить большое количество имеющихся в параллельной программе ошибок.

Для выполнения локальных экспериментов при установке клиентской части Microsoft Compute Cluster Pack на рабочей станции необходимо установить также Microsoft Compute Cluster Server SDK.

В данном упражнении мы познакомимся с заданием необходимых параметров среды Microsoft Visual Studio 2005 для отладки параллельных MPI программ и запустим программу в режиме отладки.

Задание 1 – Локальный запуск параллельной программы вычисления числа Π на рабочей станции

- Скомпилируйте проект параллельного вычисления числа Π (**parallempi**) в соответствии с инструкциями лабораторной работы “Выполнение заданий под управлением Microsoft Compute Cluster Server 2003” в конфигурации **Release**,
- Откройте командный интерпретатор (“**Start->Run**”, введите команду “**cmd**” и нажмите клавишу “**Ввод**”),
- Перейдите в папку, содержащую скомпилированную программу (например, для перехода в папку “**D:\Projects\senin\mpi_test\parallempi\Release**” введите команду смены диска “**d:**”, а затем перейдите в нужную папку, выполнив команду “**cd D:\Projects\senin\mpi_test\parallempi\Release**”),

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\senin>d:
D:\>cd D:\Projects\senin\mpi_test\parallempi\Release
D:\Projects\senin\mpi_test\parallempi\Release>_
```

- Для запуска программы в последовательном режиме (1 процесс) достаточно просто ввести имя программы и аргументы командной строки. Например, “**parallempi.exe 1500**”,

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\senin>d:
D:\>cd D:\Projects\senin\mpi_test\parallempi\Release
D:\Projects\senin\mpi_test\parallempi\Release>parallempi 1500
Process 0 on ws-2k-114-05.cluster.cmc.unn.net
NumIntervals = 1500
PI is approximately 3.1415926906268359, Error is 0.00000000370370428
D:\Projects\senin\mpi_test\parallempi\Release>_
```

- Для тестового запуска программы в параллельном режиме на локальном компьютере введите “**mpirun.exe -n <число процессов> parallempi.exe <параметры командной строки>**”. В нашем случае параметром командной строки программы вычисления числа Пи является число интервалов разбиения при вычислении определенного интеграла.


```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\senin>d:

D:\>cd D:\Projects\senin\mpi_test\parallempi\Release

D:\Projects\senin\mpi_test\parallempi\Release>parallempi 1500
Process 0 on ws-2k-114-05.cluster.cmc.unn.net
NumIntervals = 1500
PI is approximately 3.1415926906268359, Error is 0.0000000370370428

D:\Projects\senin\mpi_test\parallempi\Release>mpiexec.exe -n 2 parallempi.exe 15
00
Process 0 on ws-2k-114-05.cluster.cmc.unn.net
Process 1 on ws-2k-114-05.cluster.cmc.unn.net
NumIntervals = 1500
PI is approximately 3.1415926906268279, Error is 0.0000000370370348

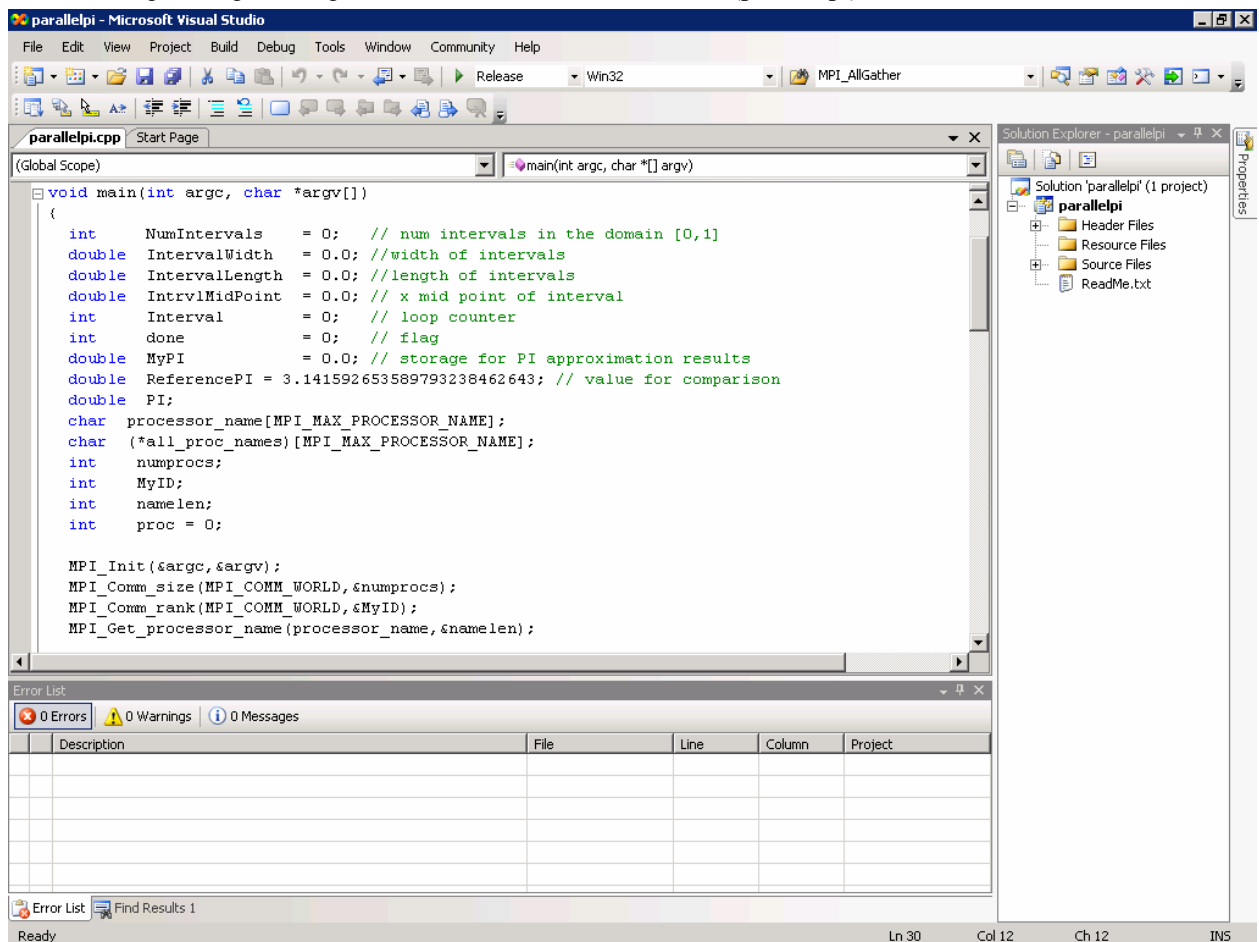
D:\Projects\senin\mpi_test\parallempi\Release>_

```

Задание 2 – Настройка Microsoft Visual Studio 2005 для локального запуска параллельной MPI программы в режиме отладки

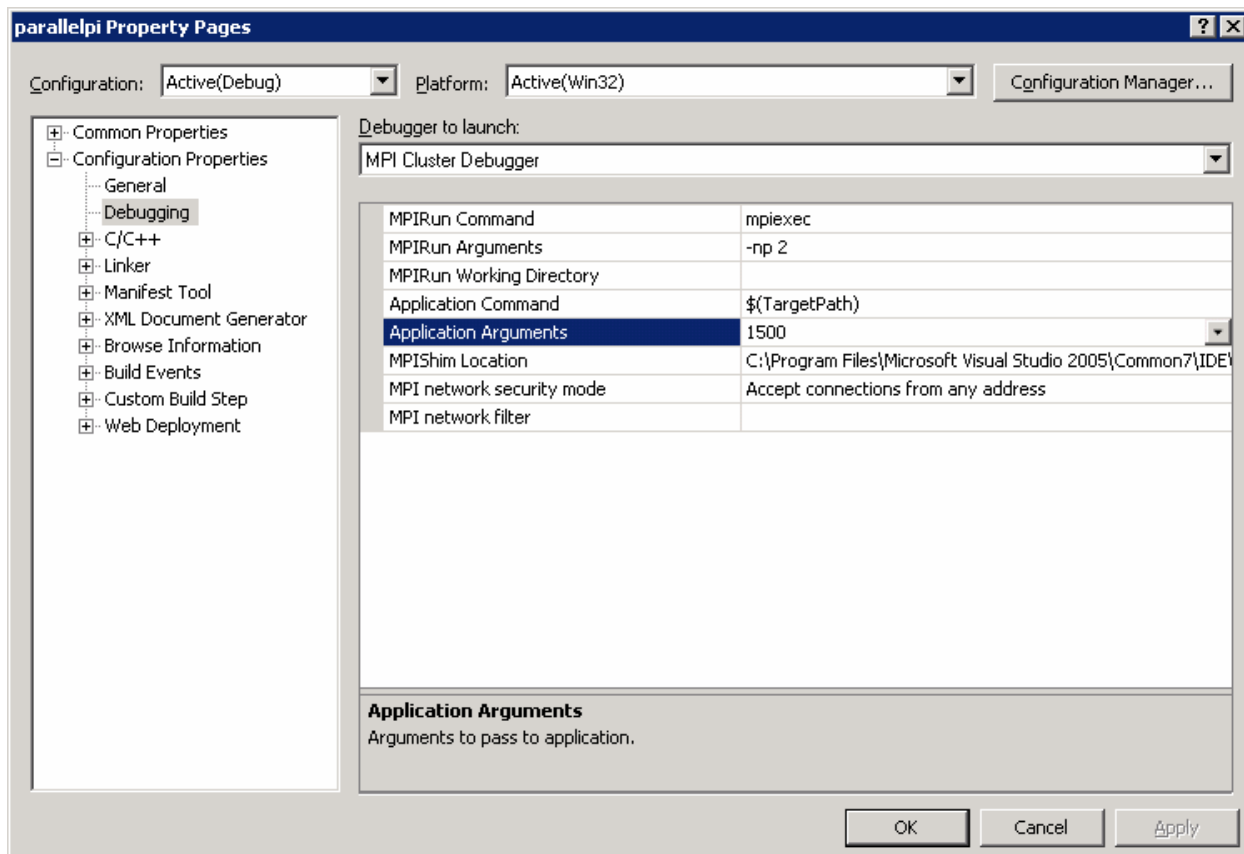
Для того чтобы получить возможность отлаживать параллельные MPI программы, необходимо соответствующим образом настроить Microsoft Visual Studio 2005:

- Откройте проект параллельного вычисления числа Пи (**parallempi**) в Microsoft Visual Studio 2005,

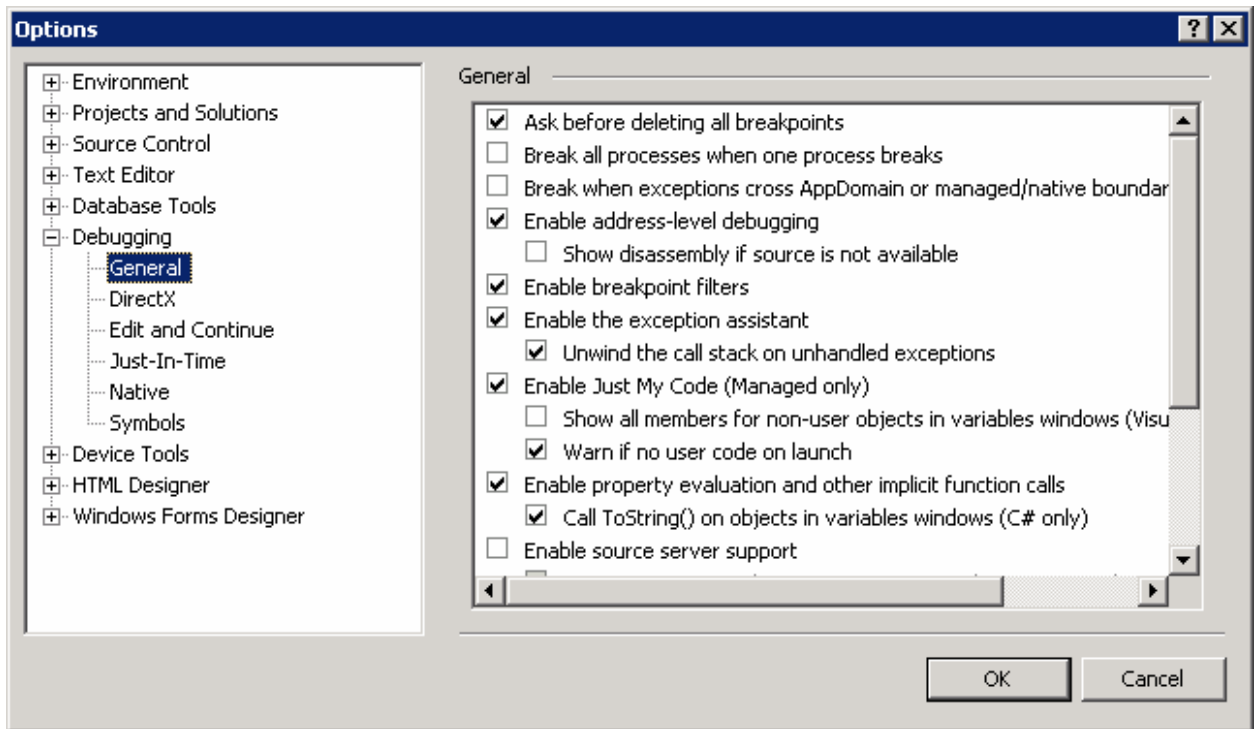


- Выберите пункт меню “**Project->parallempi Properties...**”. В открывшемся окне настроек проекта выберите пункт “**Configuration Properties->Debugging**”. Введите следующие настройки:
 - В поле “**Debugger to launch**” выберите “**MPI Cluster Debugger**”,

- В поле “**MPIRun Command**” введите “**mpixec.exe**” – имя программы, используемой для запуска параллельной MPI программы,
- В поле “**MPIRun Argument**” введите аргументы программы “**mpixec.exe**”. Например, введите “**-np <число процессов>**” для указания числа процессов, которые будут открыты,
- В поле “**Application Command**” введите путь до исполняемого файла программы,
- В поле “**Application Argument**” введите аргументы командной строки запускаемой программы,
- В поле “**MPIShim Location**” введите путь до “**mpishim.exe**” – специальной программы, поставляемой вместе с Microsoft Visual Studio 2005, используемой для отладки удаленных программ,
- Нажмите “**OK**” для сохранения внесенных изменений,

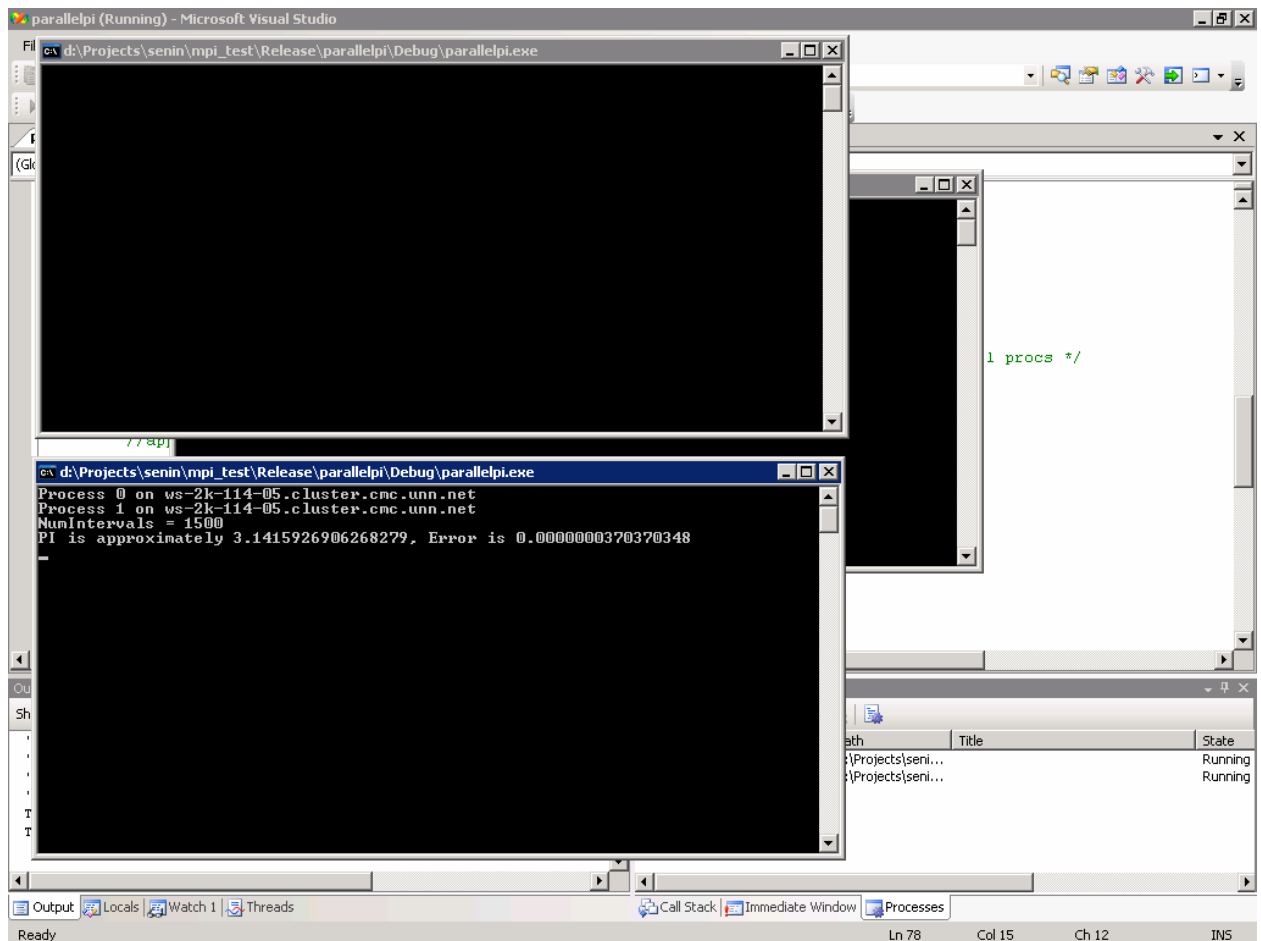


- Выберите пункт меню “**Tools->Options**”. В открывшемся окне настроек проекта выберите пункт “**Debugging->General**”. Поставьте флажок около пункта “**Break all processes when one process breaks**” для остановки всех процессов параллельной программы в случае, если один из процессов будет остановлен. Если Вы хотите, чтобы при остановке одного процесса остальные продолжали свою работу, снимите флажок (рекомендуется).



Задание 3 – Запуск параллельной MPI программы в режиме отладки из Microsoft Visual Studio 2005

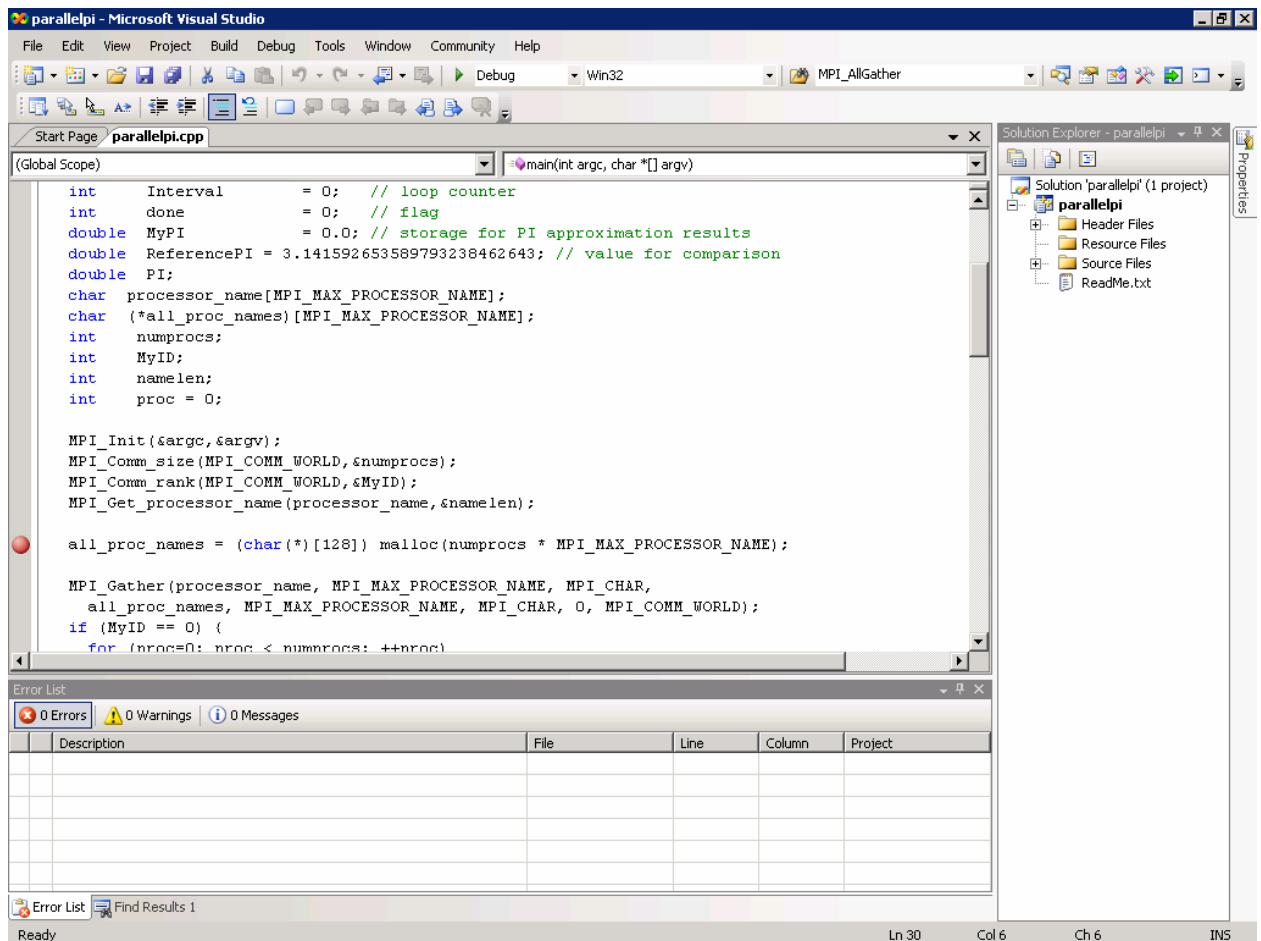
- После настроек, указанных в предыдущем задании, Вы получите возможность отлаживать параллельные MPI программы. Так, запустив программу из меню Microsoft Visual Studio 2005 (команда “**Debug->Start Debugging**”), будут запущены сразу несколько процессов (их число соответствует настройкам предыдущего задания). Каждый из запущенных процессов имеет свое консольное окно. Вы можете приостанавливать любой из процессов средствами Microsoft Visual Studio 2005 и проводить отладку. Подробнее об отладке будет рассказано в следующем упражнении.




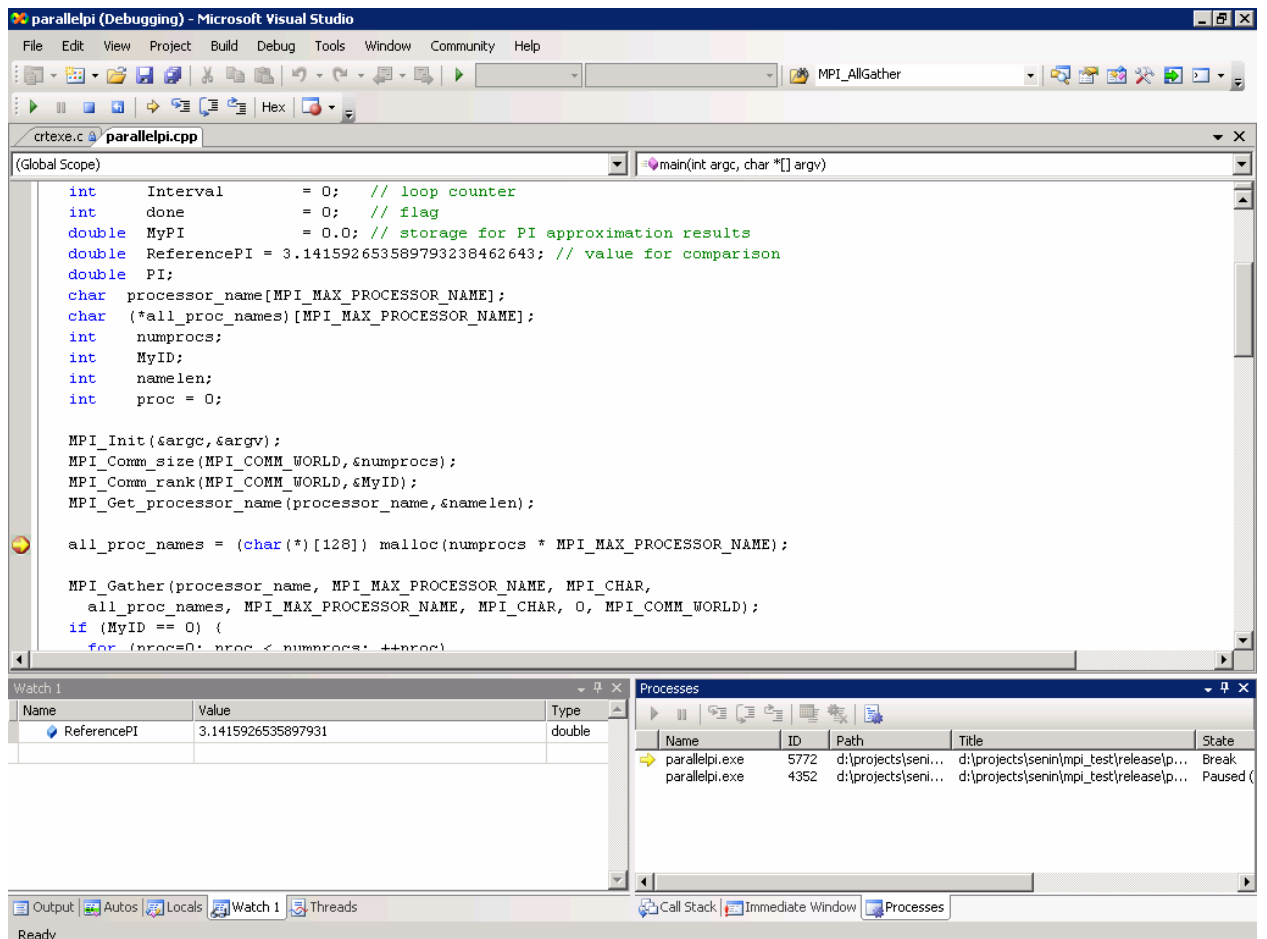
Упражнение 2 – Отладка параллельной программы в Microsoft Visual Studio 2005

Задание 1 – Знакомство с основными методами отладки

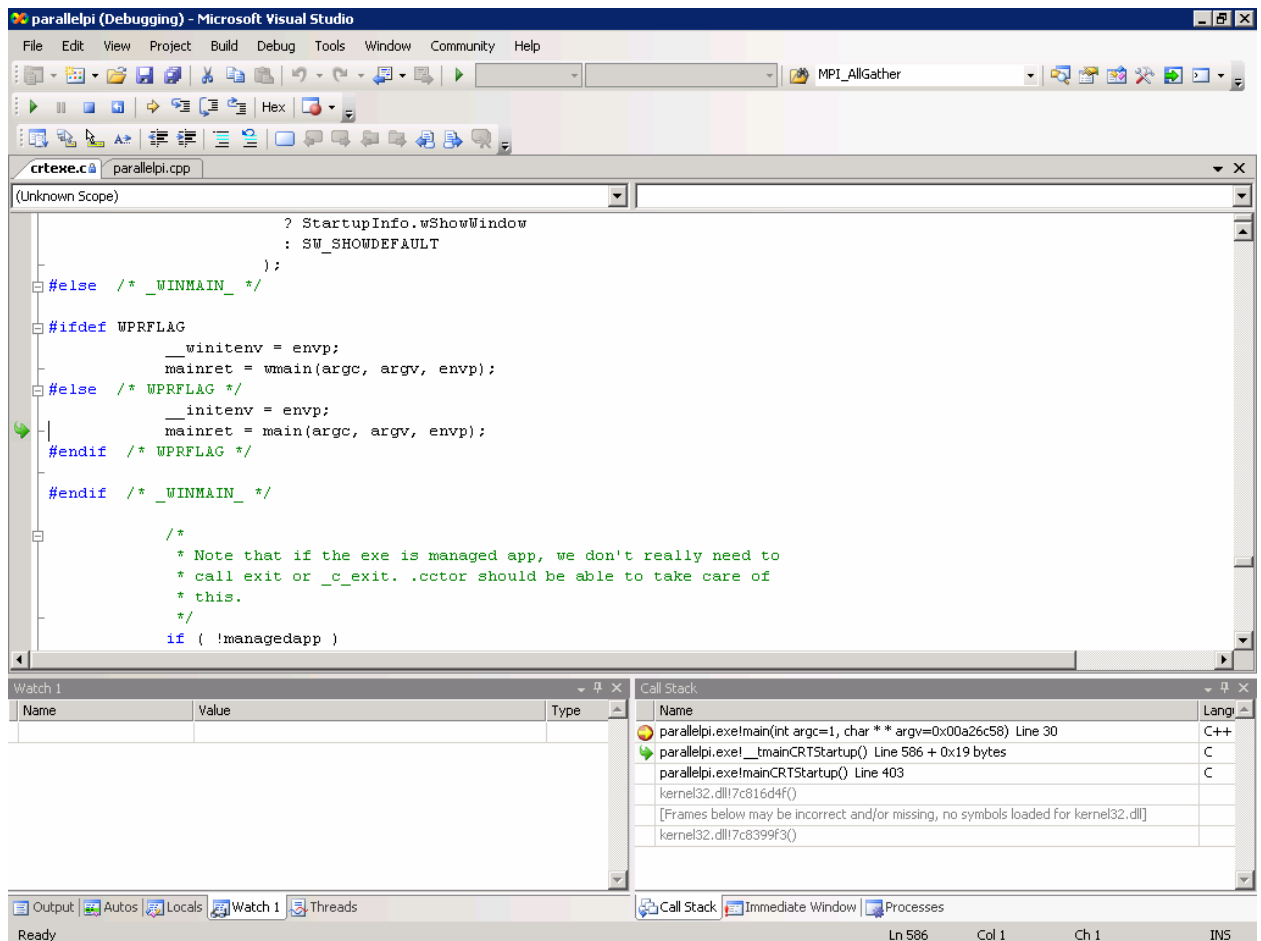
- Откройте проект параллельного вычисления числа Пи (**parallelpi**) и выполните настройки, необходимые для проведения отладки MPI программ (если это еще не было сделано), указав число запускаемых процессов равное 2. Откройте файл "**parallelpi.cpp**" (дважды щелкните на файле в окне "**Solution Explorer**"),
- Поставьте точку останова на строке выделения памяти с использованием функции **malloc**: установите мигающий указатель вводимого текста на ту строку, на которой Вы хотите установить точку останова и нажмите кнопку "**F9**",



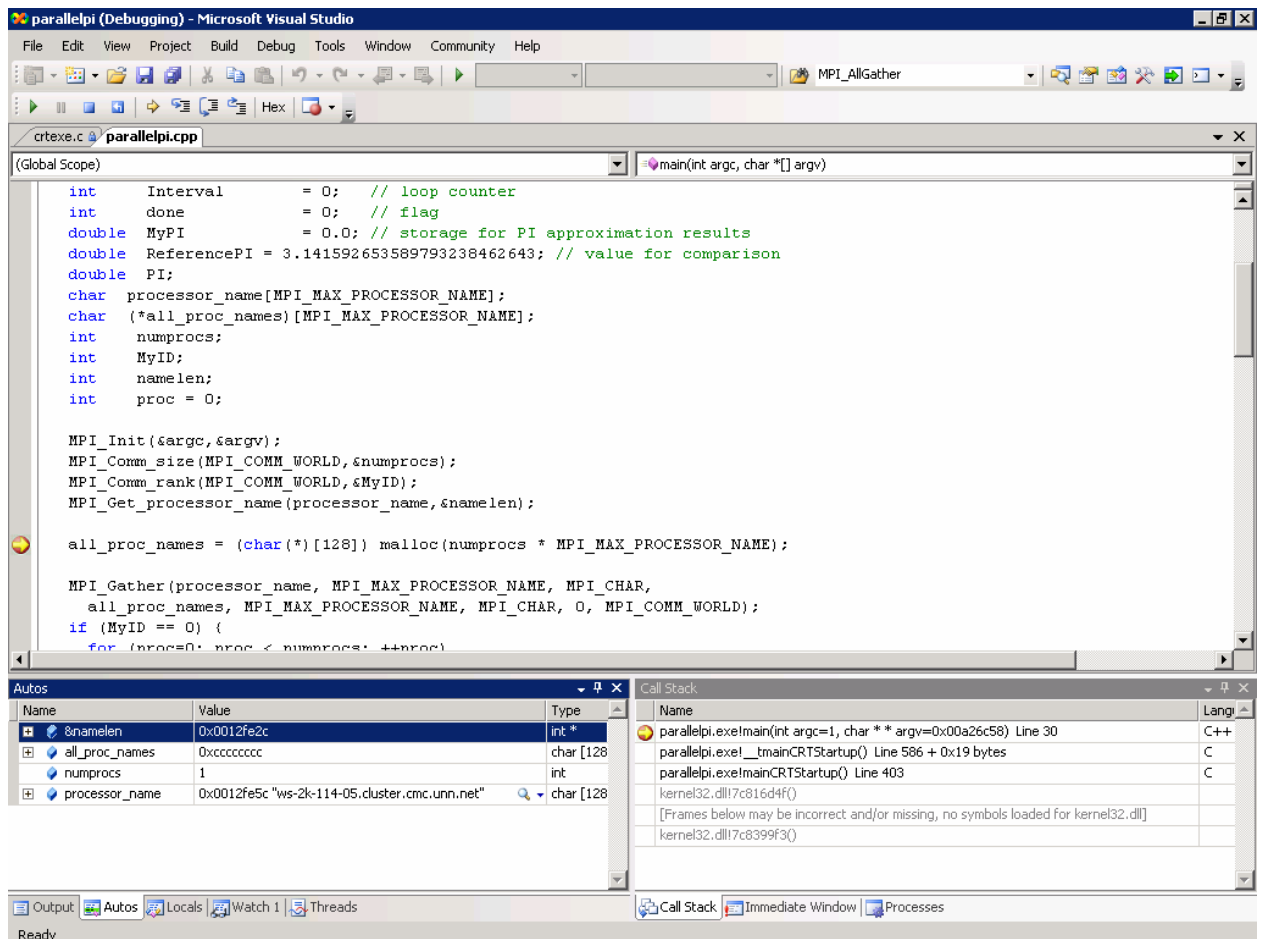
- Выберите отладочную конфигурацию (“**Debug**”) в выпадающем списке конфигураций проекта на панели инструментов. Запустите программу в режиме отладки: выполните команду меню “**Debug->Start Debugging**” или нажмите на кнопке с зеленым треугольником () в панели инструментов Microsoft Visual Studio 2005. Программа запустится, откроется 3 консольных окна: окно процесса “**mpiexec.exe**” и 2 консольных окна параллельной программы вычисления числа Пи. По достижении точки остановки выполнение процессов приостановится, так как оба процесса достигнут указанной точки,



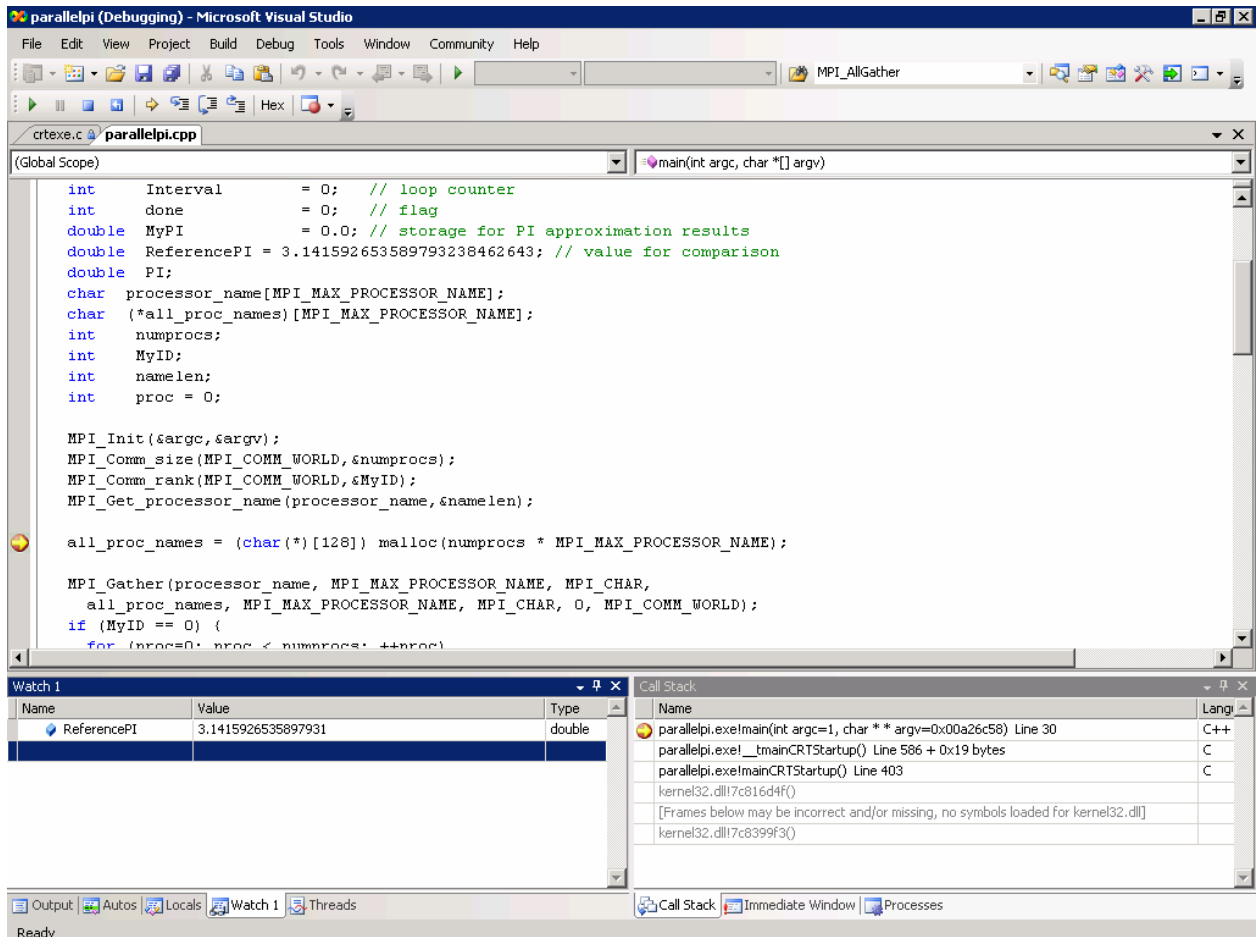
- Откройте окно “Call Stack”: выполните команду меню “Debug->Windows->Call Stack”. Окно показывает текущий стек вызова функций и позволяет переключать контекст на каждую из функций в стеке. При отладке это помогает понять, как именно текущая функция была вызвана, а так же просмотреть значения локальных переменных функций в стеке. Щелкните 2 раза по функции, находящейся ниже текущей функции в стеке. Текущий контекст изменится, и Вы увидите библиотечную функцию, вызвавшую функцию “main”,




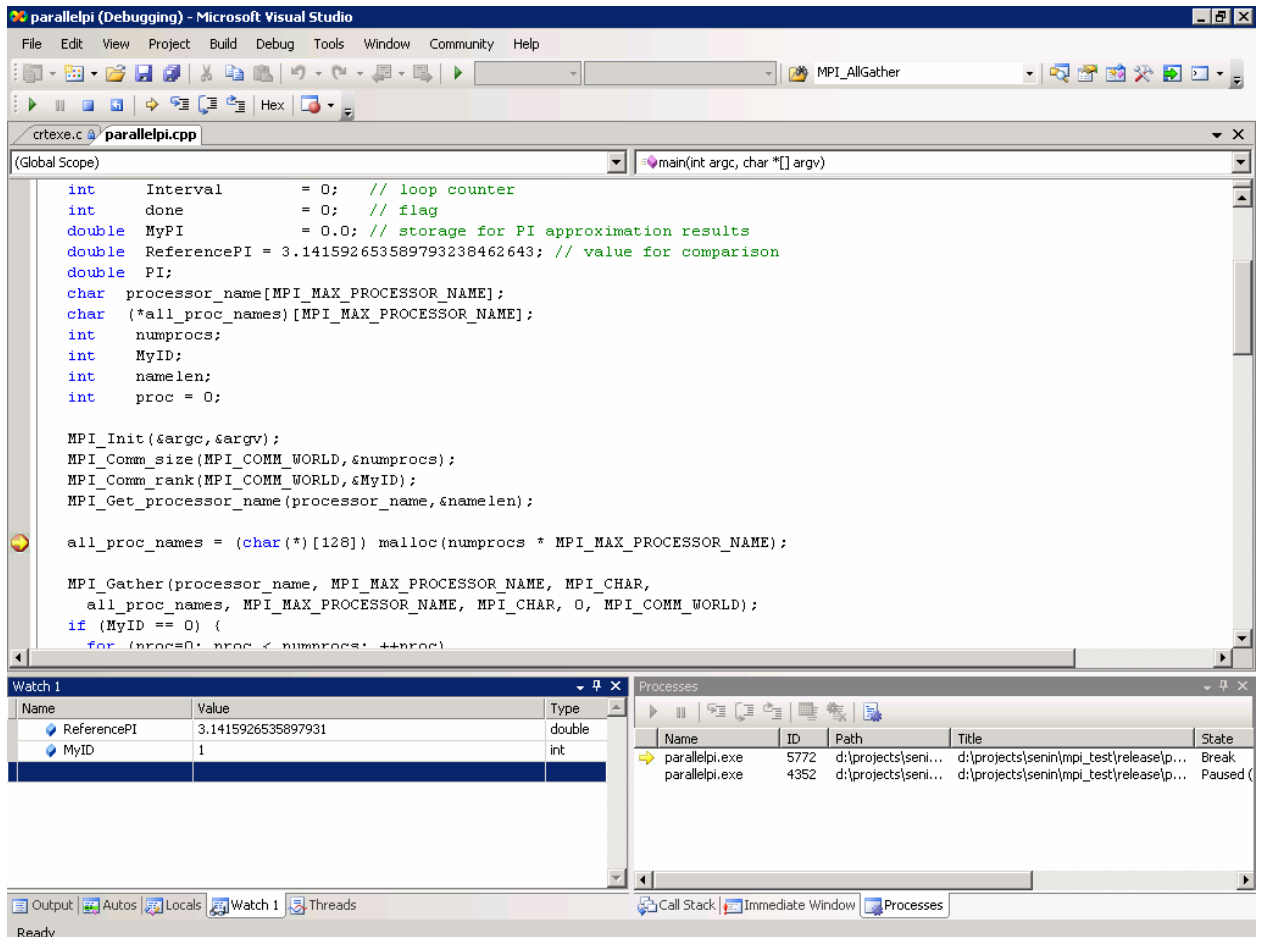
- Щелкните 2 раза на функции “main” в окне “Call Stack”. Откройте окно “Autos” (выполните команду меню “Debug->Windows->Autos”). Окно “Autos” показывает значения переменных, используемых на текущей и на предыдущих строках кода. Вы не можете изменить состав переменных, так как он определяется средой Microsoft Visual Studio 2005 автоматически,



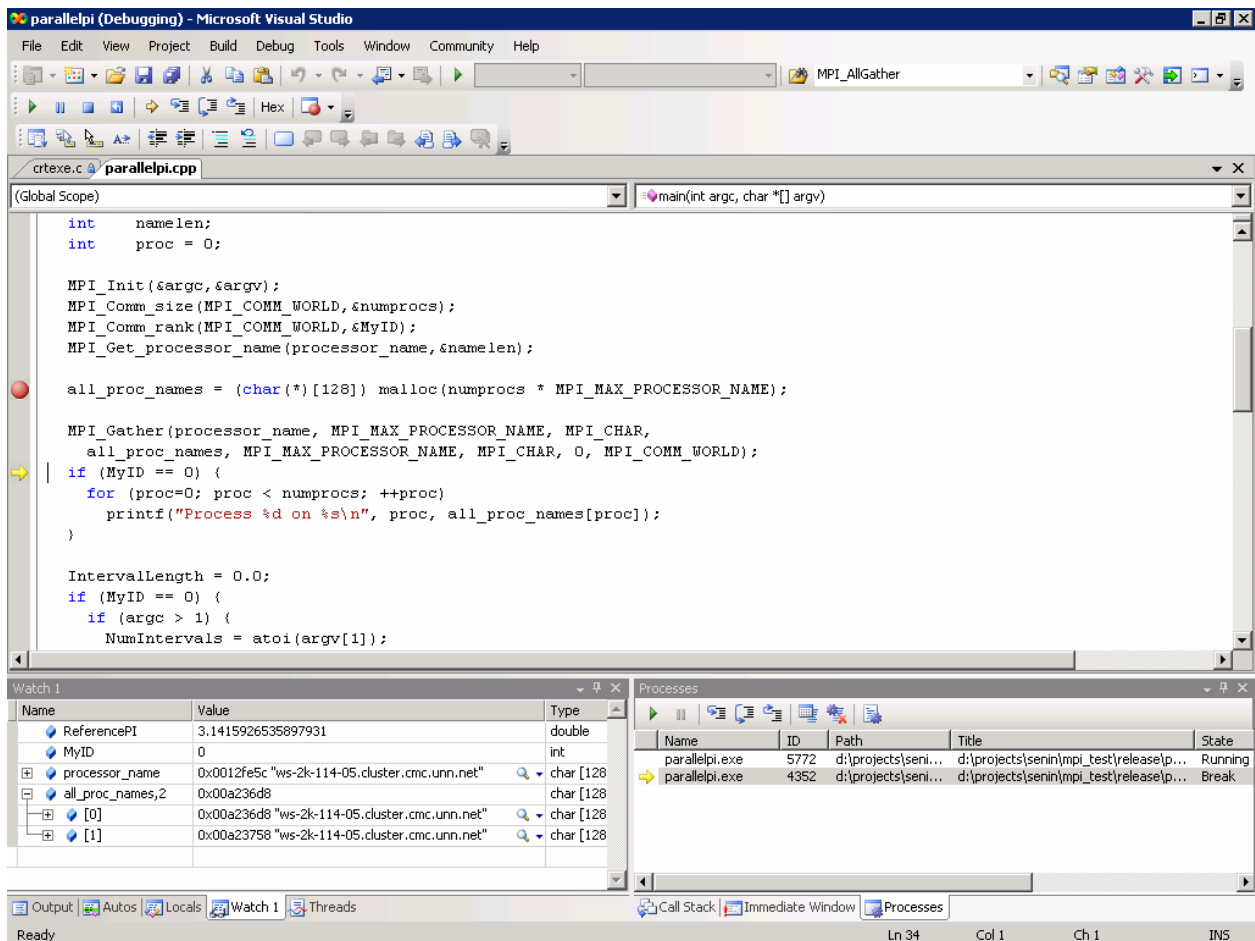
- Откройте окно **“Watch”** (выполните команду меню **“Debug->Windows->Watch->Watch 1”**). В этом окне Вы можете просматривать значения переменных, которых нет в окне **“Autos”**. Например, введите в колонке **“Name”** окна **“Watch”** имя переменной **“ReferencePI”**, нажмите клавишу **“Ввод”**. В колонке **“Value”** появится текущее значение переменной, в колонке **“Type”** – ее тип,



- Откройте окно **“Processes”** (выполните команду меню **“Debug->Windows->Processes”**). Появится окно со списком процессов параллельной MPI программы. В нашем случае их 2. Введите в окне **“Watch”** переменную **“MyID”** – идентификатор текущего процесса. Запомните значение переменной. Затем измените текущий процесс, дважды щелкнув по другому процессу в окне **“Processes”**. Посмотрите значение переменной **“MyID”** – оно должно отличаться от предыдущего, так как у каждого процесса в одной MPI задаче идентификаторы различны. При использовании окна **“Processes”** необходимо учитывать следующие 2 момента:
 - Вы можете сделать активным только приостановленный процесс (для приостановки процесса можно поставить точку остановки или выделить работающий процесс в окне **“Processes”** и выполнить команду **“Break Process”** – кнопка в виде двух вертикальных линий в окне **“Processes”** )
 - В окне **“Processes”** есть колонка **“ID”** – идентификатор процесса в операционной системе Windows. Важно помнить, что указанный идентификатор не имеет отношения к рангу (идентификатору) процесса в MPI задаче,



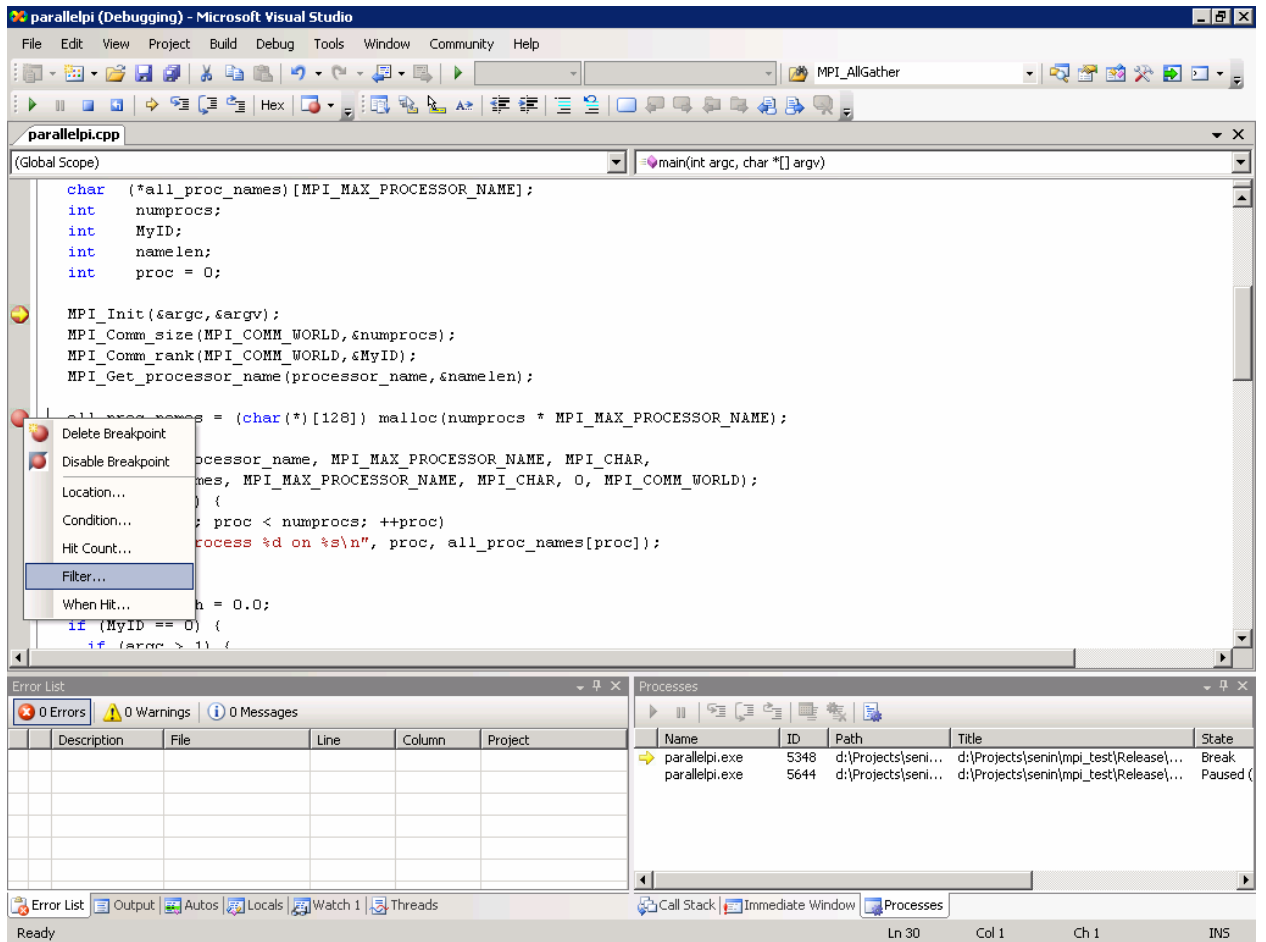
- Перейдите на процесс с нулевым идентификатором “**MyID**”. Введите в окне “**Watch**” переменные “**processor_name**” и “**all_proc_names**”. “**all_proc_names**” – это массив названий узлов, на которых запущены параллельные процессы. Массив используется только на процессе с индексом 0. Для отображения значений, например, двух элементов массива введите “**all_proc_names,2**”. После этого Вы сможете просмотреть значения двух первых элементов массива, нажав на “+” слева от имени переменной. Нажимайте кнопку “**F10**” для пошагового выполнения программы. После каждого шага Вы можете видеть, как изменяются значения внутренних переменных. Так, после вызова функции **MPI_Gather** процесс с индексом 0 будет содержать названия всех узлов, на которых запущены параллельные процессы. Для входа “внутри” функции Вы можете нажимать кнопку “**F11**” (в рассматриваемом примере все вызываемые функции являются системными, поэтому Вы не сможете увидеть их исходный код).



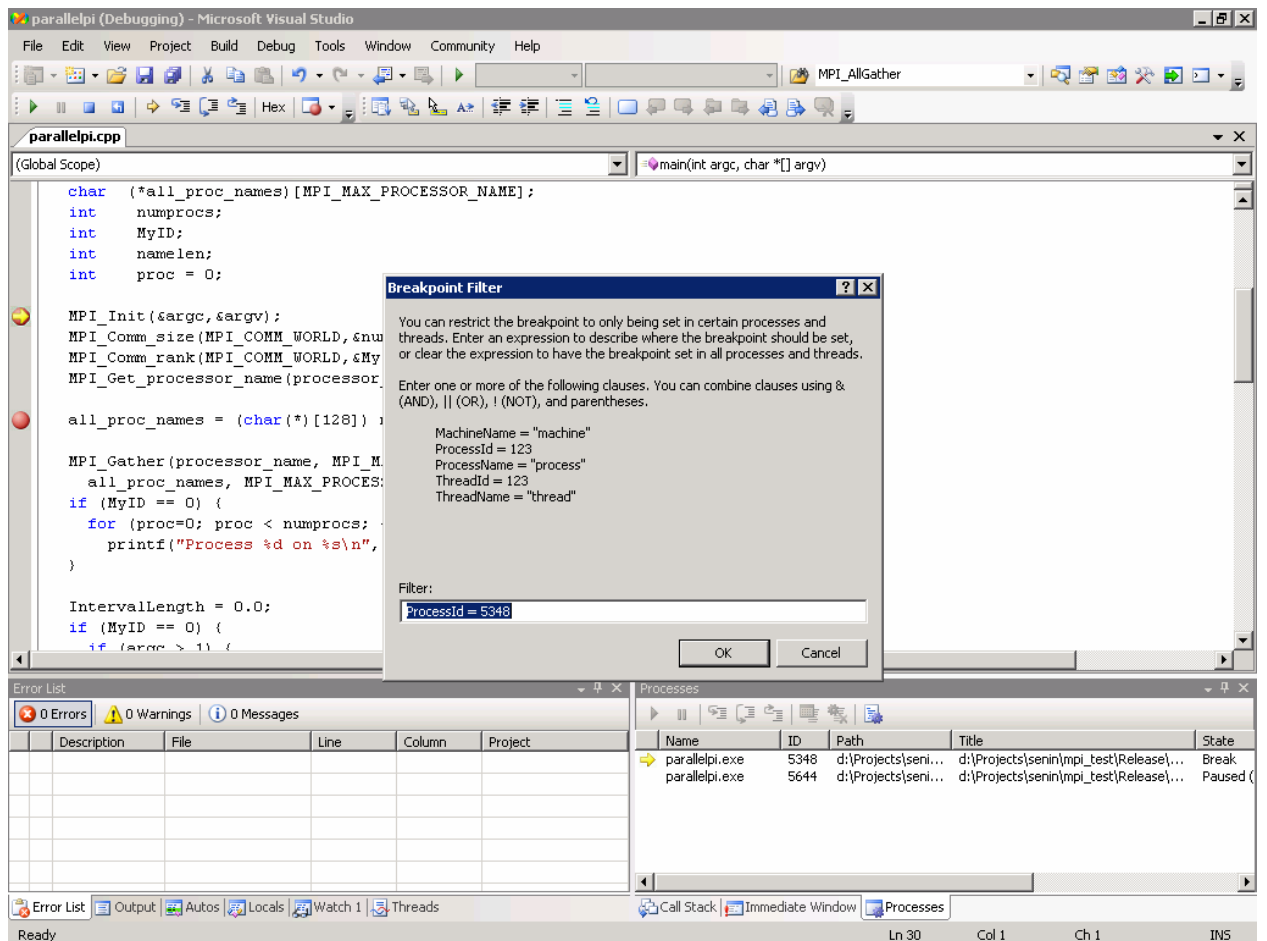
Задание 2 – Использование точек остановки

Мы уже использовали точки остановки в предыдущих пунктах. В данном задании мы рассмотрим вопросы использования **условных точек остановки**, то есть точек остановки, позволяющих указать условия, которые должны выполняться для приостановки работы программы. Использование точек остановки с условиями в некоторых случаях существенно более эффективно, так как позволяет программисту более точно указать необходимые моменты остановки процесса выполнения параллельной программы.

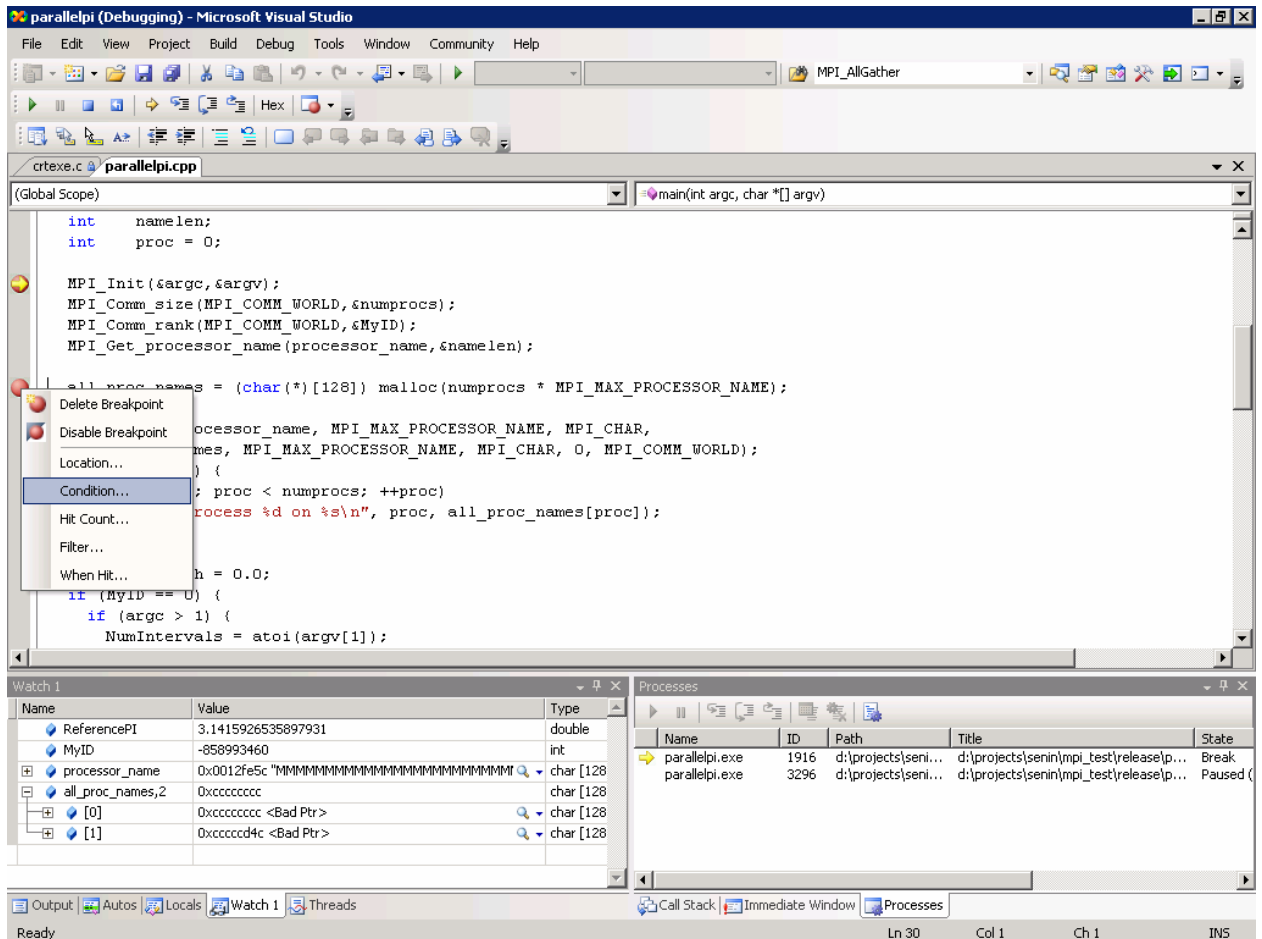
- Откройте проект параллельного вычисления числа Пи (**parallempi**) и выполните настройки, необходимые для проведения отладки MPI программ (если это еще не было сделано), указав число запускаемых процессов равное 2. Откройте файл "**parallempi.cpp**" (дважды щелкните на файле в окне "**Solution Explorer**"),
- Поставьте точку остановки на любой строке текста программы (например, на строке с "**MPI_Init**") и запустите отладку. По достижении точки остановки выполнение процессов приостановится,



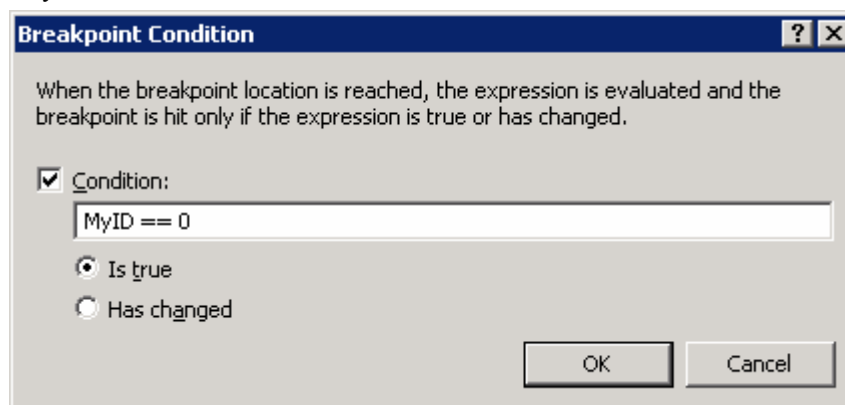
- В открывшемся окне можно указать те процессы и потоки, выполнение которых должно быть прервано по достижении указанной точки остановки. Кроме того, можно указать имя вычислительного узла, на котором точка остановки будет приостанавливать процессы (параметр используется в случае, если часть процессов отлаживаемой задачи запущена на удаленных узлах). Список названий параметров указан в подсказке на окне. При написании логических выражений можно использовать логические функции "и" ("AND", "&"), "или" ("OR", "||"), "не" ("NOT", "!"). Например, для указания того, что приостановить работу должен только один процесс, используется выражение "**ProcessId = <идентификатор процесса>**", где идентификатор процесса можно узнать в окне "Processes". Нажмите "OK" для сохранения условия,



- Кроме того, Вы можете указать условия на значения переменных, которые должны выполняться для приостановки работы программы. Щелкните правой кнопкой мыши на точке остановки и выберите пункт “Conditions”.



- В открывшемся окне Вы можете указать условия 2 типов:
 - Условие первого типа – приостановить работу программы, если выполнены условия. Для использования этого типа условий установите флаг “**Is true**” и введите условное выражение в поле “**Conditions**”. При вводе выражения используется синтаксис, принятый в C++. Так, для остановки только процесса с индексом 0 следует ввести условие “**MyID == 0**”,
 - Условие второго типа – приостановить работу программы, если значение введенного выражения (необязательно логического) изменилось с момента предыдущего выполнения строки, на которой находится точка остановки. Для использования этого типа условий установите флаг “**Has changed**”,
- Поставьте флаг “**Is true**” и введите условие остановки “**MyID == 0**”. Нажмите “**OK**” для сохранения введенного условия.



Задание 3 – Особенности отладки параллельных MPI программ

Отладчик параллельных MPI программ в Microsoft Visual Studio появился впервые в версии Visual Studio 2005. Однако компиляцию и отладку MPI программ можно производить и в более ранних версиях среды разработки. Обычными приемами в случае отсутствия истинной поддержки MPI со стороны отладчика являются следующие:

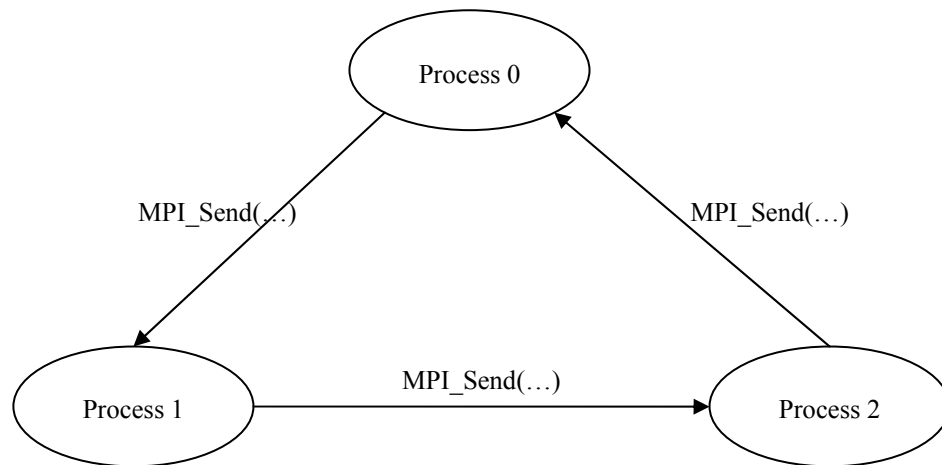
- **Использование текстовых сообщений**, выводимых в файл или на экран, со значениями интересующих переменных и/или информацией о том, какой именно участок программы выполняется. Такой подход часто называют “**printf отладкой**” (“**printf debugging**”), так как чаще всего для вывода сообщений на консольный экран используется функция “**printf**”. Данный подход хорош тем, что он не требует от программиста специальных навыков работы с каким-либо отладчиком, и при последовательном применении позволяет найти ошибку. Однако для того, чтобы получить значение очередной переменной, приходится каждый раз писать новый код для вывода сообщений, перекомпилировать программу и производить новый запуск. Это занимает много времени. Кроме того, добавление в текст программы нового кода может приводить к временному исчезновению проявлений некоторых ошибок,
- **Использование последовательного отладчика**. Так как MPI задача состоит из нескольких взаимодействующих процессов, то для отладки можно запустить несколько копий последовательного отладчика, присоединив каждую из них к определенному процессу MPI задания. Данный подход позволяет в некоторых случаях более эффективно производить отладку, чем при использовании текстовых сообщений. Главным недостатком подхода является необходимость ручного выполнения многих однотипных действий. Представьте, например, необходимость приостановки 32 процессов MPI задания: в случае использования последовательного отладчика придется переключиться между 32 процессами отладчика и вручную дать команду приостановки.

Параллельный отладчик Microsoft Visual Studio 2005 лишен указанных недостатков и позволяет существенно экономить время на отладку. Это достигается за счет того, что среда рассматривает все процессы одной MPI задачи как единую параллельно выполняемую программу, максимально приближая отладку к отладке последовательных программ. К числу особенностей параллельной отладки относятся уже рассмотренное окно “**Processes**”, позволяющего переключаться между параллельными процессами, а также дополнительные настройки среды (см. задание 2 упражнения 1).

Задание 4 – Обзор типичных ошибок при написании параллельных MPI программ

Все ошибки, которые встречаются при последовательном программировании, характерны также и для параллельного программирования. Однако кроме них есть ряд специфических типов ошибок, обусловленных наличием в MPI задаче нескольких взаимодействующих процессов. Дополнительные сложности в разработке параллельных программ обуславливают повышенные требования к квалификации программистов, реализующих параллельные версии алгоритмов. И хотя перечислить все типы ошибок, которые могут возникнуть при программировании с использованием технологии MPI, крайне затруднительно, дадим краткую характеристику ряду наиболее характерных ошибок, преследующих начинающих разработчиков. К числу наиболее типичных ошибок при параллельном программировании с использованием технологии MPI следует отнести:

- **Взаимная блокировка при пересылке сообщений**. Предположим, что n процессов передают информацию друг другу по цепочке так, что процесс с индексом i передает информацию процессу с индексом $i+1$ (для индексов $i=0, \dots, n-2$), а процесс с индексом $n-1$ передает информацию процессу с индексом 0 . Передача осуществляется с использованием функции отправки сообщений **MPI_Send** и функции приема сообщений **MPI_Recv**, при этом каждый процесс сначала вызывает **MPI_Send**, а затем **MPI_Recv**. Функции являются блокирующими, то есть **MPI_Send** возвратит управление только тогда, когда передача будет завершена, или когда сообщение будет скопировано во внутренний буфер. Таким образом, стандарт допускает возможность, что функции отправки сообщения вернут управление только после того, как передача будет завершена. Но передача не может завершиться, пока принимающий процесс не вызовет функцию **MPI_Recv**. А функция **MPI_Recv** будет вызвана, в свою очередь, только после того, как процесс завершит отправку своего сообщения. Так как цепочка передачи сообщений замкнута, отправка сообщений может не завершиться никогда, потому что каждый процесс будет ожидать завершения отправки своего сообщения, и никто не вызовет функцию приема сообщения. Избежать подобной блокировки можно, например, вызывая на процессах с четными индексами сначала **MPI_Send**, а затем **MPI_Recv**, а на процессах с нечетными индексами – в обратной порядке. Несмотря на очевидность ошибки, ее часто допускают, так как при небольших сообщениях высока вероятность, что сообщения уберутся во внутренние структуры библиотеки, функция отправки сообщения вернет управление, и ошибка не даст о себе знать,



- **Освобождение или изменение буферов, используемых при неблокирующей пересылке.** При вызове неблокирующих функций возврат из них происходит немедленно, а адреса массивов с сообщениями запоминаются во внутренних структурах библиотеки MPI. При непосредственной передаче сообщений по сети, которая будет выполнена в отдельном потоке, происходит обращение к исходному массиву с сообщением, поэтому важно, чтобы этот участок памяти не был удален или изменен до окончания передачи (установить, что передача завершена можно вызовом специальных функций). Однако часто об этом правиле забывают, что приводит к непредсказуемому поведению программы. Такого рода ошибки являются одними из самых сложных для отладки,
- **Несоответствие вызовов функций передачи и приема сообщений.** Важно следить, чтобы каждому вызову функции отправки сообщения соответствовал вызов функции приема и наоборот. Однако в том случае, когда число передач заранее не известно, а определяется в ходе вычислений, бывает легко ошибиться и не обеспечить нужных гарантий выполнения данного условия. К этому же типу ошибок можно отнести вызов функций передачи и приема сообщений с несоответствующими тэгами идентификации сообщений.

Контрольные вопросы

- Как локально протестировать работоспособность параллельной программы, разработанной для использования с библиотекой MS MPI до запуска ее на кластере? Какое программное обеспечение должно быть установлено для этого на Вашей рабочей станции?
- Перечислите и дайте краткое описание основным окнам среды Microsoft Visual Studio 2005, используемым при отладке?
- Что такое и для чего используются точки остановки? Что такое условные точки остановки?
- В чем принципиальная особенность отладки параллельных MPI программ в среде Microsoft Visual Studio 2005?
- Какие типичные ошибки при программировании с использованием технологии MPI Вы знаете?

Лабораторная работа №3
**РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ
В СИСТЕМЕ ГРАФОСИМВОЛИЧЕСКОГО
ПРОГРАММИРОВАНИЯ PGRAPH**

1. РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ В ТЕХНОЛОГИИ ГСП

1.1 Концептуальная модель организации параллельных вычислений в технологии ГСП

В параллельной программе, в отличие от последовательной, одновременно существует несколько вычислительных процессов. Вычисления в каждом из процессов выполняются последовательно, поэтому процесс может быть описан обычной «последовательной» граф-моделью. Для того, чтобы в рамках одной модели изобразить несколько таких процессов, ее необходимо расширить. Прежде всего, необходимо выделить различные процессы, обозначить их начало и конец, определить условия их запуска и завершения. Это достигается расширением понятия «дуга» за счет введения дуг различного типа.

Определим формально тип дуги Ψ_{ij} как функцию $T(\Psi_{ij}) \in \{1,2,3\}$, значения которой имеют следующую семантику:

$T(\Psi_{ij}) = 1$ – последовательная дуга (описывает передачу управления на последовательных участках вычислительного процесса);

$T(\Psi_{ij}) = 2$ – параллельная дуга (обозначает порождение нового параллельного вычислительного процесса)

$T(\Psi_{ij}) = 3$ – завершающая дуга (завершает параллельный вычислительный процесс).

Для описания отдельного вычислительного процесса вводится понятие параллельной ветви β - подграфа графа G , начинающегося параллельной дугой (тип этой дуги $T(\Psi_{ij}) = 2$) и заканчивающегося завершающей дугой (тип этой дуги $T(\Psi_{ij}) = 3$). $\beta = \langle A_\beta, \Psi_\beta, R_\beta \rangle$, где A_β – множество вершин ветви, Ψ_β – множество дуг управления ветви, R_β – отношение над множествами вершин и дуг ветви, определяющее способ их связи.

Дуги, исходящие из вершин параллельной ветви β , принадлежат также ветви β . При кодировании алгоритма, описанного с помощью предлагаемой модели, каждая параллельная ветвь порождает отдельный процесс – совокупность подпрограмм, исполняемых последовательно на одном из процессоров параллельной вычислительной системы.

Графическая модель обычно содержит несколько параллельных ветвей, каждая из которых образует отдельный процесс. В этом смысле модель параллельных вычислений можно представить как объединение нескольких параллельных ветвей:

$$G = \cup \beta_k,$$

где β_k – параллельные ветви графа G .

Таким образом, распараллеливание вычислений возможно только на уровне граф-модели. Вычисления в пределах любого актора выполняются последовательно.

Число параллельных ветвей в модели фиксируется при ее построении, при этом максимальное количество ветвей не ограничивается. Каждая ветвь имеет ровно один вход и один выход, для обозначения которых в граф-модели используются два типа дуг: параллельная дуга и терминирующая дуга (рисунок 1):

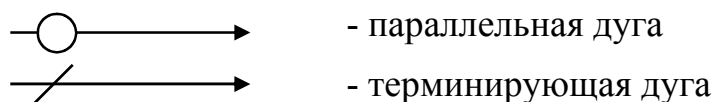


Рисунок 1 – Обозначение параллельных и терминирующих дуг в графической модели параллельных вычислений

Для описания правил построения граф-модели введем следующую систему обозначений:

- To(A_i) – список дуг, входящих в вершину A_i
- Fr(A_i) – список дуг, исходящих из вершины A_i
- H(β_j) – начальная вершина ветви β_j
- |L| – число элементов в списке L

Переход по параллельной дуге начинает работу параллельной ветви, переход по терминирующей дуге – заканчивает ее работу. Параллельная дуга не содержит предиката, т. е. переход по ней происходит безусловно.

$$\forall \Psi_{ij} \in \Psi, T(\Psi_{ij}) = 2 : P_{ij} \equiv 1$$

Из вершины может исходить не менее двух параллельных дуг. Если из вершины исходит параллельная дуга, то дуги других типов (обычная или терминирующая) не могут исходить из данной вершины.

$$T(\Psi_{ij_0}) = 2 \rightarrow \forall j \neq j_0 : T(\Psi_{ij}) = 2, |Fr(A_i)| \geq 2$$

Входящие в вершину дуги не могут быть одновременно параллельными и терминирующими:

$$T(\Psi_{i_0j_0}) = 2 \rightarrow \forall i : T(\Psi_{ij_0}) \neq 3$$

Функционирование модели начинается с запуска единственной ветви, называемой мастер-ветвью (мастер-процессом). Обозначим мастер ветвь β_0 . В вершинах мастер-ветви, имеющих исходящие параллельные дуги, порождаются новые параллельные ветви. Вершины этих ветвей также могут иметь параллельные дуги, таким образом, допускается вложенность параллельных

ветвей. Ветви, породившиеся в одной вершине некоторой ветви, должны терминироваться также в одной вершине этой же ветви:

$$\forall k: H(\beta_k) = A_{k0}, \Psi_{ik0} \in Fr(A_i), A_i \in \beta_1 \rightarrow$$

$$\rightarrow \Psi_{kN} \in T_0(A_j), T(\Psi_{kN}) = 3, A_j \in \beta_1$$

В ветви β_1 управление из вершины A_i после запуска ветвей $\beta_k, k = k_1, k_2, \dots, K$, передается вершине A_j . Вершина A_j запускается на выполнение после завершения работы ветвей $\beta_{k_1} \dots \beta_K$. Таким образом, в каждый момент времени в любой ветви выполняется ровно одна вершина.

Переход между двумя вершинами, принадлежащими различным параллельным ветвям, возможен только по параллельным и терминирующим дугам, то есть, запрещены условные переходы между вершинами различных параллельных ветвей:

$$\forall i, j, A_i \in \beta_1, A_j \in \beta_2 \rightarrow T(\Psi_{ij}) \neq 1$$

Если некоторая ветвь породила новые параллельные ветви, то вычисления в ней приостанавливаются до завершения работы порожденных ветвей. Таким образом, вложенные параллельные ветви выполняются последовательно относительно друг друга.

2. РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ В СИСТЕМЕ ГСП PGRAPH

2.1 Знакомство с системой

Система PGRAPH является развитием системы ГСП GRAPH и предназначена для визуальной разработки параллельных программ с использованием технологии графосимволического программирования.

Интерфейс и общие принципы работы системы в целом аналогичны системе ГСП GRAPH и дополнены средствами описания параллелизма.

Начальные этапы разработки, такие как построение предметной области программирования, создание и регистрация базовых модулей, акторов и предикатов, в обеих системах выполняются одинаково.

В технологии ГСП параллелизм реализуется на уровне агрегатов, поэтому процесс создания параллельного агрегата несколько отличается.

2.2 Создание параллельных агрегатов в системе PGRAPH

Процесс создания параллельного агрегата, как и последовательного, начинается с его рисования при помощи инструментов панели редактирования (рисунок 2).

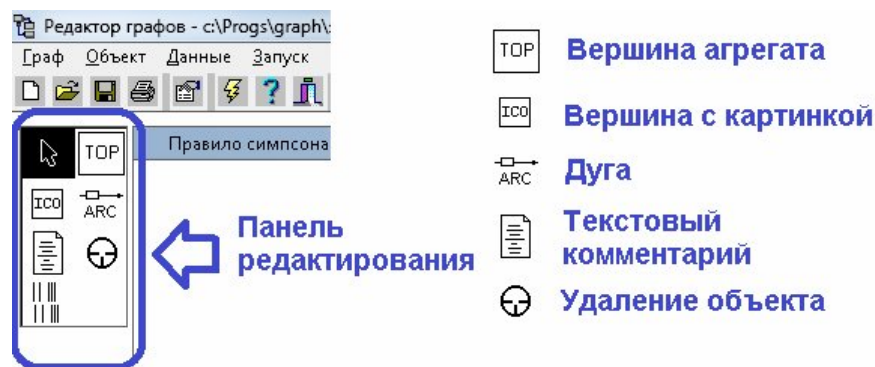


Рисунок 2 – Панель редактирования агрегата

Щелчок на изображении объекта в панели редактирования переводит в режим добавления к агрегату соответствующих объектов (вершин и дуг). Добавление осуществляется щелчком в поле редактирования агрегата.

После создания вершин и дуг необходимо назначить им объекты технологии ГСП – акторы и предикаты. Это выполняется также, как в системе GRAPH: двойной щелчок на вершине или на прямоугольнике в начале дуги открывает окно свойств, в котором выбирается соответственно актор или предикат.

Отличие параллельного агрегата от последовательного заключается в наличии в нем фрагментов, которые могут исполняться одновременно - параллельных ветвей. Любая параллельная ветвь начинается с дуги параллельного типа и заканчивается дугой терминирующего типа.

В системе PGRAPH любая дуга изначально создается как последовательная. После создания дуги можно изменить ее тип в окне «Свойства дуги», которое открывается по двойному щелчку на прямоугольнике в начале дуги (рисунок 3).

При изменении типа дуги изменяется также ее внешний вид. Для параллельной дуги прямоугольник в начале дуги заменяется на кружок, а для терминирующей дуги – на наклонную линию.

Если в граф-модели отсутствует синхронизация, то после рисования агрегата, выбора используемых акторов и предикатов и выделения параллельных ветвей, процесс создания параллельной программы заканчивается, и можно приступить к ее компиляции и запуску.

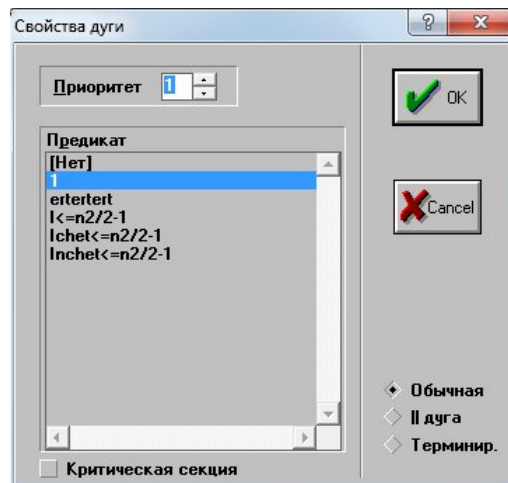


Рисунок 3 – Окно свойств дуги

2.3 Компиляция и запуск параллельной программы

Генерация исполняемого файла на основе созданной граф-программы осуществляется выбором пункта «Построение и запуск» в меню «Запуск».

При выборе этого пункта система сначала проверяет корректность параллельного агрегата, то есть его соответствие правилам построения параллельных граф-моделей в технологии ГСП. Затем синтезируется исходный текст на целевом языке программирования и вызывается внешний компилятор.

Для создания параллельных программ текущая версия системы PGRAPH использует библиотеку и среду исполнения MPI. По умолчанию созданная в системе программа запускается на локальном компьютере, поэтому на нем должна быть установлена одна из реализаций MPI (например, MPICH).

Для проведения вычислительных экспериментов на кластере необходимо отметить галочкой пункт «Запуск на кластере» меню «Настройки». В этом случае система сгенерирует отдельную версию исходных текстов параллельной программы, предназначенную для компиляции на кластере. Каталог, в который система сохраняет сгенерированные файлы, указывается в диалоговом окне «Размещение», в поле «Выходные тексты на С». Указанное окно открывается при выборе пункта «Размещение» меню «Настройки».

Сгенерированные системой исходные тексты необходимо скопировать на кластер, откомпилировать там и запустить полученную программу в соответствии с инструкциями по работе на конкретном кластере.

Инструкции по работе на кластерах суперкомпьютерного центра СГАУ можно найти на сайте hpc.ssau.ru в разделе «Инструкции».

3. Лабораторная работа №3

Цель работы: Разработка параллельного алгоритма в системе графосимволического программирования PGRAPH.

3.1 Задание на самостоятельную работу

1. Используя созданную в лабораторной работе №1 предметную область программирования, разработать параллельный алгоритм решения задачи в соответствии с вариантом, полученным в лабораторной работе №1. Для сокращения времени разработки следует использовать ранее созданный набор акторов и предикатов, при необходимости дополнив его новыми. Параллельный алгоритм должен допускать реализацию на двух и четырех процессорах (в наилучшем варианте алгоритм должен легко масштабироваться на произвольное число процессоров).

При разработке алгоритма необходимо обеспечить корректную работу параллельных ветвей с данными предметной области. Поскольку в настоящей работе не используются средства синхронизации параллельных ветвей, в алгоритме не должно быть критических данных. Это требование означает, что если параллельная ветвь вычисляет значение некоторого данного, то она должна работать с отдельной копией этого данного, не используемой в других параллельных ветвях.

Для выполнения указанного требования, возможно, потребуется создать несколько экземпляров переменных, описывающих одно и то же данное предметной области: по одному для каждой параллельной ветви. Эти экземпляры должны иметь различные имена. Кроме того, необходимо обеспечить корректную инициализацию экземпляра для каждой параллельной ветви, а также сбор результатов вычислений каждой из ветвей для формирования окончательного решения задачи.

Обратите внимание, что если параллельная ветвь использует некоторое данное как исходное и в процессе вычислений не изменяет его, то значение этого данного также не должно изменяться и другими параллельными ветвями.

Для исключения ошибок используйте средство поиска критических данных системы ГСП PGRAPH (Меню «Данные», пункт «Поиск критических данных»). В граф-программе не должно быть критических данных.

2. Используя редактор системы PGRAPH, построить граф-модель алгоритма для реализации на двух процессорах.

3. Скомпилировать и произвести тестовый запуск программы. Убедиться в идентичности результатов работы последовательной и параллельной программы.

4. Используя редактор системы PGRAPH, построить граф-модель алгоритма для реализации на четырех процессорах.

5. Скомпилировать и произвести тестовый запуск программы. Убедиться в идентичности результатов работы последовательной и параллельной программы.
6. Провести вычислительные эксперименты. Целью экспериментов является измерение времени работы созданной Вами программы при различных значениях размерности задачи и различном количестве используемых процессоров (два или четыре).
7. Составить отчет по результатам работы.

3.2 Содержание отчета

1. Постановка задачи.
2. Описание предметной области программирования.
3. Описание используемых базовых модулей.
4. Описание созданных inline-объектов, а также акторов и предикатов, построенных на основе базовых модулей.
5. Описание граф-модели программы.
6. Результаты вычислительных экспериментов с объяснением полученных зависимостей.
7. Выводы по работе.

3.3 Контрольные вопросы

1. Опишите концептуальную модель организации параллельных вычислений в технологии ГСП.
2. Что такое параллельная ветвь? Опишите правила построения параллельных ветвей в модели технологии ГСП.
3. Опишите модель синхронизации параллельных процессов в технологии ГСП.
4. Опишите назначение и основные возможности системы ГСП PGRAPH.
5. Опишите последовательность действий при визуальной разработке параллельной программы в системе ГСП GRAPH.

СПИСОК ЛИТЕРАТУРЫ

1. Коварцев А.Н. Автоматизация разработки и тестирования программных средств. - Самар. гос. аэрокосм. ун-т., Самара, 1999. - 150 с.
2. Коварцев А.Н., Жидченко В.В. Методы и средства визуального параллельного программирования. Автоматизация программирования. Электронный учебник. - Самара, СГАУ, 2010.

3. Коварцев А.Н. Численные методы: курс лекций для студентов заоч. формы обучения. - Самара: СГАУ, 2000. – 177 с.

4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. - М.: Вильямс, 2003. - 512 с.

Лабораторная работа №4
**РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ С
ИСПОЛЬЗОВАНИЕМ ОБЩИХ ДАННЫХ
В СИСТЕМЕ ГРАФОСИМВОЛИЧЕСКОГО
ПРОГРАММИРОВАНИЯ PGRAPH**

1. МОДЕЛЬ СИНХРОНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ В ТЕХНОЛОГИИ ГСП

При создании параллельных программ ключевой проблемой является синхронизация вычислений, т.е. организация согласованного выполнения параллельных фрагментов программы. Существуют различные способы синхронизации, такие как критические интервалы, семафоры, обмен сообщениями и почтовые ящики, мониторы /4/. В предлагаемой модели применяется комбинированный способ, использующий одновременно механизмы передачи сообщений и принципы мониторной синхронизации.

Определим почтовый ящик L_{post} как список сообщений, с помощью которых вершины модели информируют друг друга о своем состоянии:

$$L_{post} = [\mu_{i0,j0}, \dots, \mu_{im,jn}],$$

где $\mu_{i,j}$ – сообщение, посылаемое от вершины A_i вершине A_j . Возможность передачи сообщения от одной вершины граф-модели другой вершине изображается графически дугой синхронизации Φ_{ij} , проведенной из вершины-источника сообщения в вершину-получатель сообщения.

Вершина A_k посылает сообщения другим вершинам, записывая сообщения $L_{AkCom} = [\mu_{k,j0}, \dots, \mu_{k,jn}]$ в список L_{post} . Наличие сообщения $\mu_{i,j}$ в списке L_{post} говорит о том, что оператор вершины A_i завершил работу и хочет сообщить об этом вершине A_j .

Каждой вершине ставится в соответствие семафорный предикат

$$R_{Aj} = r(b_{i0,j}, \dots, b_{im,j}),$$

представляющий собой правильно построенную формулу относительно булевых переменных $b_{ik,j}$, связанных логическими связками типа $\&$, \vee . Булевы переменные определяются следующим образом:

$$b_{kj} = \begin{cases} 1, & \text{если } \mu_{kj} \in L_{post}, \\ 0, & \text{если } \mu_{kj} \notin L_{post}. \end{cases}$$

Семафорный предикат разрешает или запрещает запуск соответствующей вершины. Если семафорный предикат $R_{Aj} = 1$, то запуск оператора вершины A_j разрешается. В противном случае вычисления приостанавливаются до того момента, когда R_{Aj} станет равным 1. Семафорный предикат определяется для всех вершин, в которые входят дуги синхронизации. Для остальных вершин значение семафорного предиката принимается тождественно истинным.

Обмен сообщениями между вершинами, принадлежащими одной параллельной ветви, запрещен:

$$A_i \in \beta_k, A_j \in \beta_k \rightarrow b_{ij} \equiv 0$$

Вершины в пределах одной параллельной ветви исполняются строго последовательно, поэтому их синхронизировать не требуется.

Замечание. Операцию "v" в семафорных предикатах следует использовать только над операндами, определенными над взаимоисключающими вершинами, то есть такими вершинами, факт запуска одной из которых исключает получение управления остальными вершинами. На рисунке 1, а) вершины A_4 и A_5 являются взаимоисключающими.

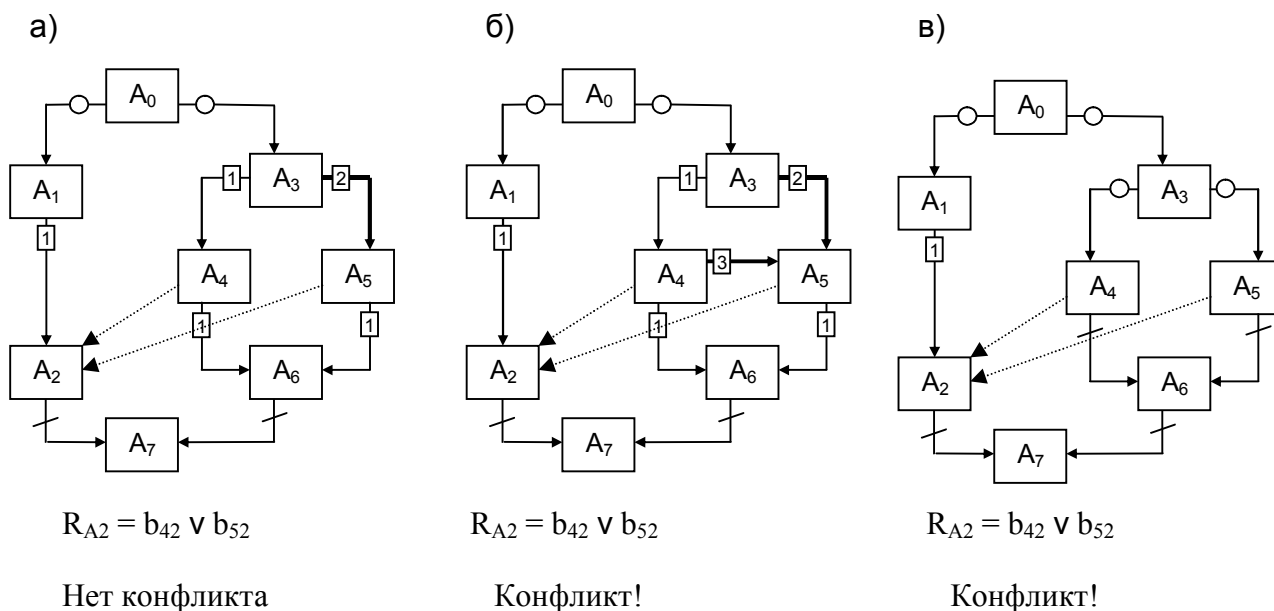


Рисунок 1 – Пример использования операции "v" в семафорных предикатах

Если вершины A_2 , A_4 и A_5 используют общее данное, а операторы в вершинах A_2 , A_3 и A_6 имеют относительно большую длительность исполнения, то имеет смысл ввести две дуги синхронизации $\Phi_{4,2}$ и $\Phi_{5,2}$ и определить семафорный предикат для вершины A_2 с использованием операции "v": $R_{A_2} = b_{42} \vee b_{52}$. В случае передачи управления от вершины A_3 к одной из вершин A_4 или A_5 (например, вершине A_4), другая вершина (A_5) уже не запустится, а значит, вершину A_2 достаточно синхронизировать только с вершиной, получившей управление (A_4). В этом случае для запуска на исполнение операнда в вершине A_2 достаточно истинности одного из операндов ее семафорного предиката, и можно использовать операцию "v".

Если вершины не являются взаимоисключающими, то использование операции "v" над их сообщениями не разрешает конфликта совместного использования данных, так как приход сообщения от одной из таких вершин не исключает запуска другой на исполнение. Примеры подобных ситуаций приведены на рисунке 1, б), в).

На рисунке 1, б) граф-модель дополнена дугой $\Psi_{4,5}$. Если управление передается по этой дуге от вершины A_4 к вершине A_5 , то оператор последней может исполняться одновременно с оператором вершины A_2 , что приведет к конфликту совместного использования данных.

На рисунке 1, в) вершины A_4 и A_5 принадлежат различным параллельным ветвям. В этом случае использование операции "v" в семафорном предикате вершины A_2 также не исключает конфликта, поскольку операторы вершин A_4 и A_5 могут выполняться одновременно, и приход сообщения от одной из них не гарантирует, что оператор другой вершины также завершил исполнение.

Приведенные рассуждения и примеры показывают, что операцию "v" в семафорных предикатах следует применять с осторожностью, и только в тех случаях, когда легко видеть, что вершины, от которых исходят дуги синхронизации, являются взаимоисключающими.

2. ОПИСАНИЕ СИНХРОНИЗАЦИИ В СИСТЕМЕ ГСП PGRAPH

Для описания синхронизации между вершинами граф-модели используется специальный режим графического редактора – режим синхронизации, в котором вершины соединяются дугами синхронизации.

Переход в этот режим осуществляется нажатием на кнопку «Синхронизация» панели редактирования (рисунок 2).

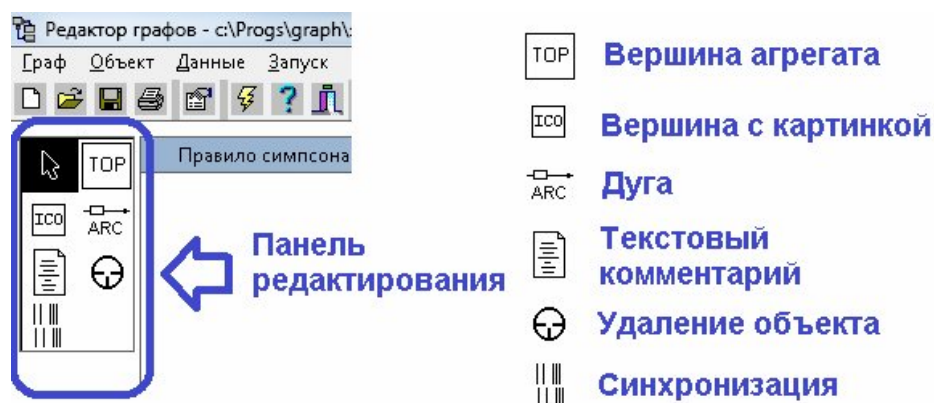


Рисунок 2 – Панель редактирования

Дуги синхронизации определяют направление передачи сообщений между вершинами. Поскольку возможна передача сообщений между вершинами различных агрегатов, в редакторе предусмотрено последовательное продвижение по иерархической структуре граф-модели. Чтобы открыть агрегат, которым помечена некоторая вершина граф-модели, нужно дважды щелкнуть по данной вершине, держа нажатой кнопку «Ctrl». Откроется окно с этим агрегатом в режиме чтения (рисунок 2).

Чтобы создать дугу синхронизации, необходимо определить ее начальную и конечную вершины. Выбор вершины, из которой исходит дуга синхронизации, выполняется в режиме синхронизации одинарным щелчком на соответствующей вершине. Выбор вершины, в которую входит дуга синхронизации, происходит по щелчку на соответствующей вершине, выполняемому при нажатой кнопке «Alt».

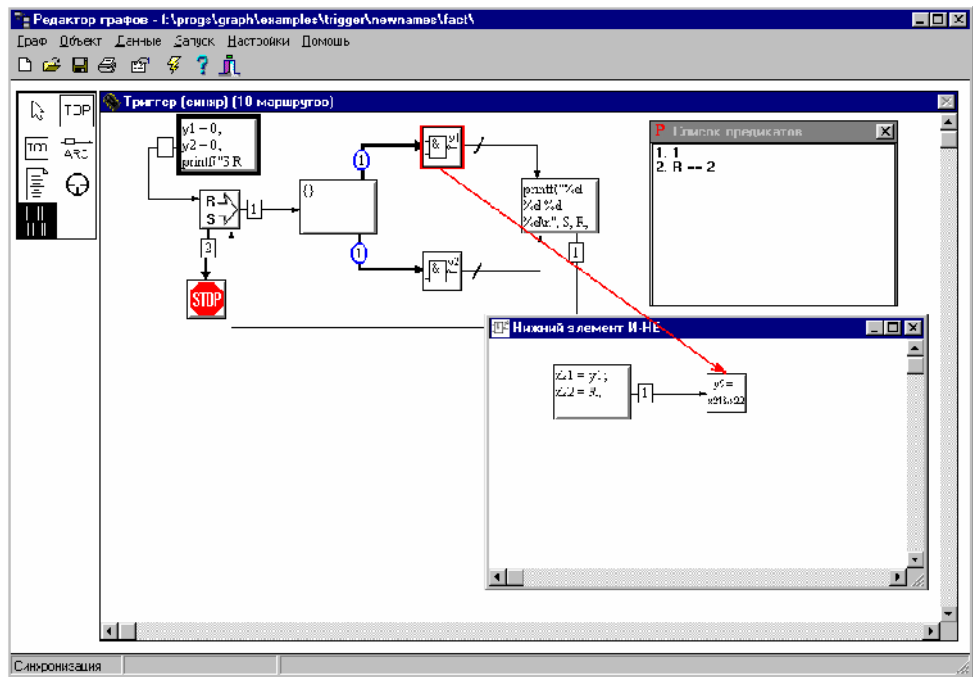


Рисунок 2 – Рисование дуг синхронизации между вершинами различных агрегатов

Каждому сообщению присваивается уникальное имя, которое используется в функциях отправки сообщений и вычисления семафорного предиката. Это имя генерируется системой автоматически, но его можно изменить, нажав на кнопку «Отправка» в окне свойств вершины (рисунок 3).

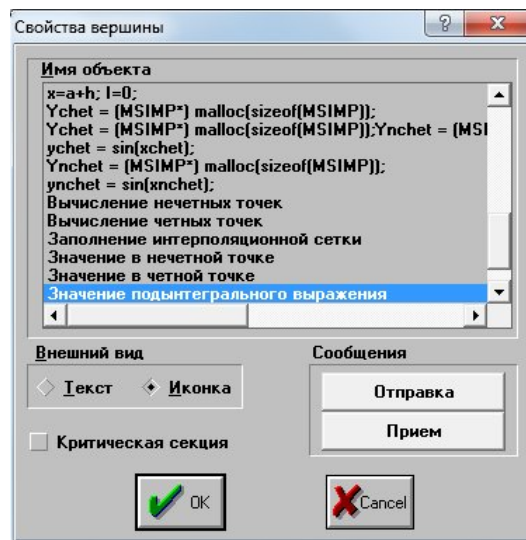


Рисунок 3 – Окно свойств вершины

Функция вычисления семафорного предиката вводится после нажатия на кнопку «Прием» в окне свойств вершины (рисунок 3). Если дуга синхронизации ведет в вершину A_j , имеющую другие входящие дуги синхронизации, то

пользователю предлагается отредактировать логическое выражение семафорного предиката R_{Aj} .

При рисовании дуг синхронизации система производит автоматическую проверку их корректности, запрещая соединять подобными дугами вершины одной параллельной ветви.

3. Лабораторная работа №4

Цель работы: Разработка параллельного алгоритма с использованием общих данных в системе графосимволического программирования PGRAPH.

3.1 Задание на самостоятельную работу

1. Используя механизм синхронизации параллельных ветвей, измените созданный в лабораторной работе №3 параллельный алгоритм так, чтобы в нем были дуги синхронизации. Постарайтесь оптимизировать алгоритм, чтобы после введения в него дуг синхронизации время работы параллельной программы уменьшилось.
2. Скомпилируйте и произведите тестовый запуск программы. Убедитесь в том, что программа с синхронизацией параллельных ветвей выдает те же результаты работы, что и программа без синхронизации.
3. Проведите вычислительные эксперименты для тех же значений размерности задачи и количества используемых процессоров, что и в лабораторной работе №3.
4. Составьте отчет по результатам работы.

3.2 Содержание отчета

1. Постановка задачи.
2. Описание способа распараллеливания и общих принципов работы алгоритма с синхронизацией.
3. Описание предметной области программирования.
4. Описание используемых базовых модулей.
5. Описание созданных inline-объектов, а также акторов и предикатов, построенных на основе базовых модулей.
6. Описание граф-модели программы.
7. Результаты вычислительных экспериментов с объяснением полученных зависимостей.
8. Выводы по работе.

3.3 Контрольные вопросы

1. Опишите концептуальную модель организации параллельных вычислений в технологии ГСП.
2. Что такое параллельная ветвь? Опишите правила построения параллельных ветвей в модели технологии ГСП.
3. Опишите модель синхронизации параллельных процессов в технологии ГСП.
4. Что такое семафорный предикат?
5. Опишите назначение и основные возможности системы ГСП PGRAPH.
6. Опишите последовательность действий при визуальной разработке параллельной программы в системе ГСП PGRAPH.

СПИСОК ЛИТЕРАТУРЫ

1. Коварцев А.Н. Автоматизация разработки и тестирования программных средств. - Самар. гос. аэрокосм. ун-т., Самара, 1999. - 150 с.
2. Коварцев А.Н., Жидченко В.В. Методы и средства визуального параллельного программирования. Автоматизация программирования. Электронный учебник. - Самара, СГАУ, 2010.
3. Коварцев А.Н. Численные методы: курс лекций для студентов заоч. формы обучения. - Самара: СГАУ, 2000. – 177 с.
4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. - М.: Вильямс, 2003. - 512 с.

Лабораторная работа №5
**ВЫПОЛНЕНИЕ ЗАДАНИЙ ПОД УПРАВЛЕНИЕМ
MICROSOFT COMPUTE CLUSTER SERVER 2003**

Составитель:
профессор, д.т.н. Гергель В.П.
кафедра математического обеспечения ЭВМ
Нижегородского государственного университета им. Н.И. Лобачевского

Цель лабораторной работы	1
Общая схема выполнения заданий под управлением Microsoft Compute Cluster Server 2003	1
Упражнение 1 – Компиляция программы для запуска в CCS 2003.....	3
Задание 1 - Установка Microsoft Compute Cluster Pack SDK	4
Задание 2 – Настройка интегрированной среды разработки Microsoft Visual Studio 2005.....	7
Задание 3 – Компиляция параллельной программы в Microsoft Visual Studio 2005	10
Упражнение 2 – Запуск последовательной задачи	15
Запуск программы через графический пользовательский интерфейс	15
Запуск программы с использованием шаблона	25
Запуск программы из командной строки	29
Упражнение 3 – Запуск параллельного задания	31
Упражнение 4 – Запуск множества задач.....	38
Упражнение 5 – Запуск потока задач	45
Дополнительное упражнение. Задача определения характеристик сети передачи данных	54
Описание характеристик, определяющих производительность сети	54
Общая характеристика алгоритма.....	55
Компиляция программы.....	55
Выполнение программы	55
Контрольные вопросы	60

Для эффективной эксплуатации высокопроизводительных кластерных установок необходимо использовать сложный комплекс программных систем. Долгое время пользователям Windows кластеров приходилось одновременно использовать программное обеспечение нескольких производителей, что могло быть причиной проблем с совместимостью различных программ друг с другом. С выходом Compute Cluster Server 2003 (CCS) можно говорить о том, что компания Microsoft предоставляет полный спектр программного обеспечения, необходимый для эффективной эксплуатации кластера и разработки программ, в полной мере использующих имеющиеся вычислительные мощности.

Цель лабораторной работы

Цель данной лабораторной работы – научиться компилировать и запускать программы под управлением **Microsoft Compute Cluster Server 2003**.

- Упражнение 1 – Компиляция программы для запуска в CCS 2003
- Упражнение 2 – Запуск последовательного задания
- Упражнение 3 – Запуск параллельного задания
- Упражнение 4 – Запуск множества заданий
- Упражнение 5 – Запуск потока задач

Примерное время выполнения лабораторной работы: **90 минут**.

Общая схема выполнения заданий под управлением Microsoft Compute Cluster Server 2003

Для эффективного использования вычислительного ресурса кластера необходимо обеспечить не только непосредственный механизм запуска заданий на выполнение, но и предоставить среду управления ходом выполнения заданий, решающую, в том числе, задачу эффективного распределения ресурсов. Эти задачи эффективно решаются с использованием встроенных в CCS 2003 средств.

Дадим определение важнейшим понятиям, используемым в CCS 2003:

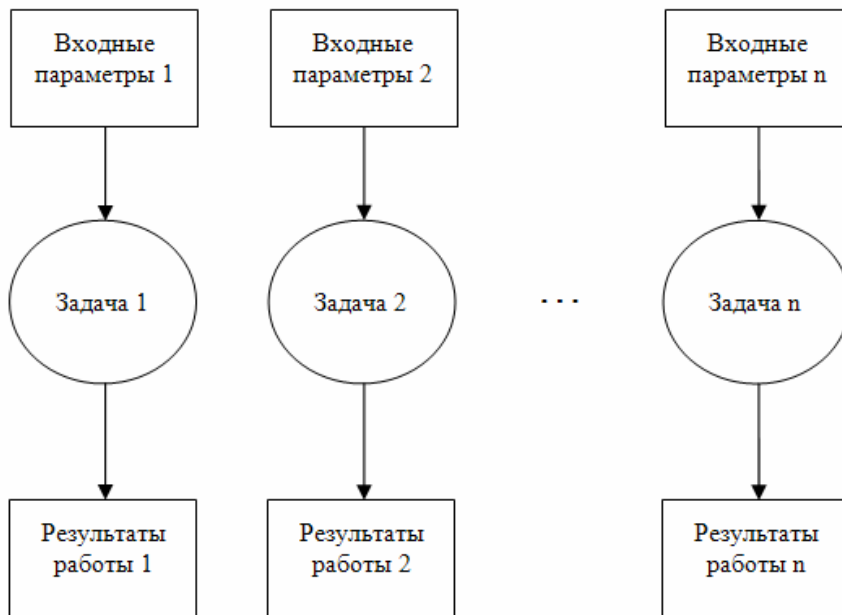
- **Задание (job)** – запрос на выделение вычислительного ресурса кластера для выполнения задач. Каждое задание может содержать одну или несколько задач,
- **Задача (task)** – команда или программа (в том числе, параллельная), которая должна быть выполнена на кластере. Задача не может существовать вне некоторого задания, при этом задание может содержать как несколько задач, так и одну,
- **Планировщик заданий (job scheduler)** – сервис, отвечающий за поддержание очереди заданий, выделение системных ресурсов, постановку задач на выполнение, отслеживание состояния запущенных задач,
- **Узел (node)** – вычислительный компьютер, включенный в кластер под управлением CCS 2003,
- **Процессор (processor)** – один из, возможно, нескольких вычислительных устройств узла,
- **Очередь (queue)** – список заданий, отправленных планировщику для выполнения на кластере. Порядок выполнения заданий определяется принятой на кластере политикой планирования,
- **Список задач (task list)** – эквивалент очереди заданий для задач каждого конкретного задания. Порядок запуска задач определяется правилом FCFS (первыми будут выполнены задачи, добавленные в список первыми), если пользователь специально не задаст иной порядок.

Планировщик заданий CCS 2003 работает как с последовательными, так и с параллельными задачами. Последовательной называется задача, которая использует ресурсы только 1 процессора. Параллельной же называется задача, состоящая из нескольких процессов (или потоков), взаимодействующих друг с другом для решения одной задачи. Как правило, параллельным задачам для эффективной работы требуется сразу несколько процессоров. При этом, в случае использования MPI в качестве интерфейса передачи сообщений, процессы параллельной программы могут выполняться на различных узлах кластера. CCS 2003 включает собственную реализацию стандарта MPI2: библиотеку Microsoft MPI (MS MPI). В случае использования MS MPI в качестве интерфейса передачи сообщений необходимо запускать параллельные задачи с использованием специальной утилиты **mpiexec.exe**, осуществляющей одновременный запуск нескольких экземпляров параллельной программы на выбранных узлах кластера. Важно отметить, что непосредственным запуском задач занимается планировщик, а пользователь может лишь добавить задачу в очередь, так как время ее запуска выбирается системой автоматически в зависимости от того, какие вычислительные ресурсы свободны и какие задания ожидают в очереди выделения им ресурсов. Таким образом, для исполнения программы в CCS 2003 необходимо выполнить следующие действия:

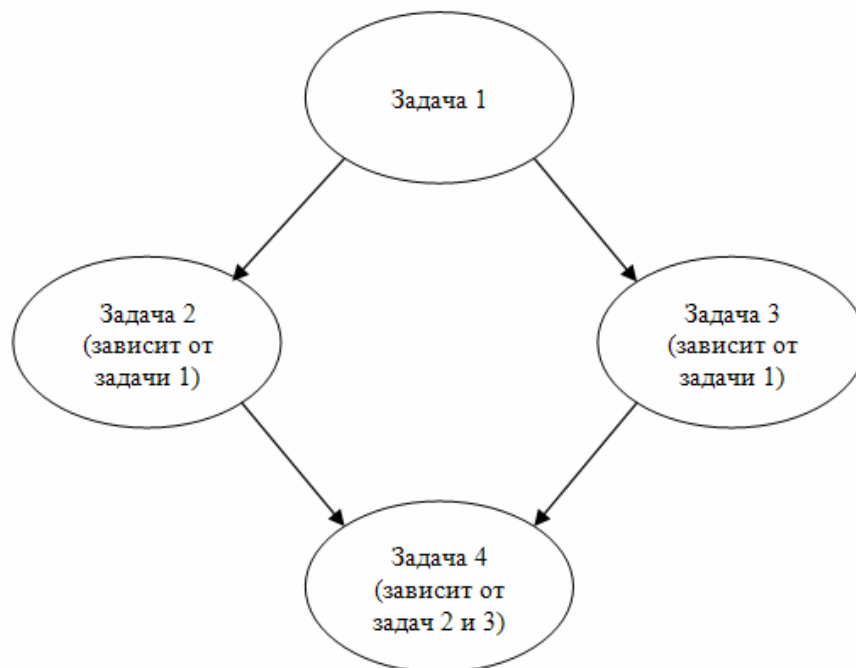
- Создать задание с описанием вычислительных ресурсов, необходимых для его выполнения,
- Создать задачу. Задача определяется при помощи той или иной команды, выполнение которой приводит к запуску на кластере последовательных или параллельных программ. Например, параллельная задача описывается при помощи команды **mpiexec.exe** с соответствующими параметрами (список узлов для ее запуска, имя параллельной программы, аргументы командной строки программы и др.),
- Добавить задачу к созданному ранее заданию.

Выделяют два особых вида заданий:

- **Параметрическое множество задач (parametric sweep)** – одна и та же программа (последовательная или параллельная), несколько экземпляров которой запускается (возможно, одновременно) с разными входными параметрами и разными файлами вывода,



- **Поток задач (task flow)** – несколько задач (возможно, одна и та же программа с разными входными параметрами) запускаются в определенной последовательности. Последовательность запуска объясняется, например, зависимостью некоторых задач последовательности от результатов вычислений предыдущих.



Далее в лабораторной работе на примерах будет показано, как компилировать и запускать последовательные и параллельные задачи в CCS 2003. Кроме того, будут приведены примеры параметрического множества заданий и потока заданий.

Упражнение 1 – Компиляция программы для запуска в CCS 2003

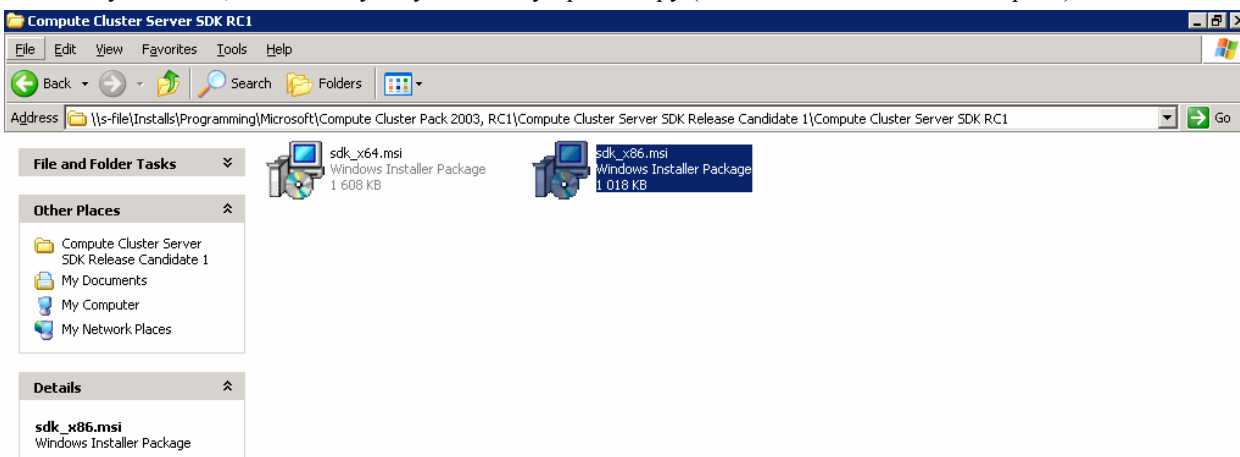
Как было отмечено в предыдущем пункте, Microsoft Compute Cluster Server 2003 позволяет управлять ходом выполнения как последовательных, так и параллельных задач. При этом параллельные MPI задачи не обязательно должны быть скомпилированы для MS MPI (хотя в случае MPI использование реализации от Microsoft является предпочтительным). Кроме того, возможно использование других технологий поддержки параллельного программирования (например, программирование с использованием OpenMP).

Данный пункт описывает только компиляцию параллельных программ для MS MPI в Microsoft Visual Studio 2005.

Задание 1 - Установка Microsoft Compute Cluster Pack SDK

Для компиляции параллельных программ, работающих в среде MS MPI, необходимо установить **SDK (Software Development Kit)** – набор интерфейсов и библиотек для вызова MPI-функций:

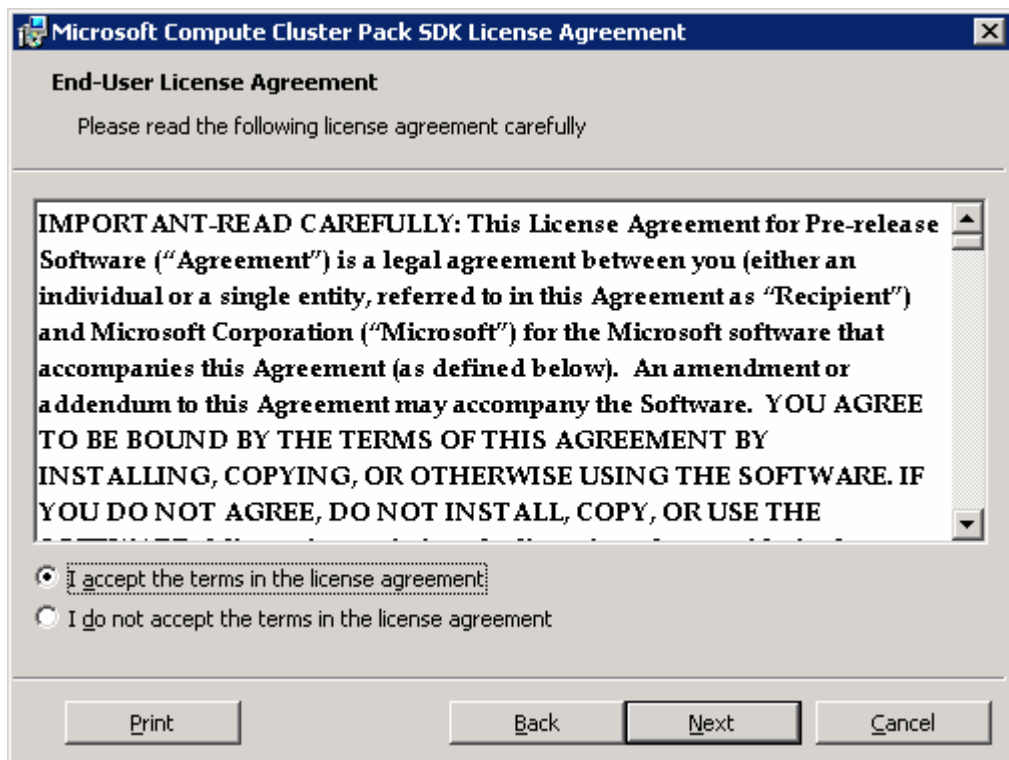
- Откройте директорию со скачанной версией SDK (описание процедуры скачивания можно найти в лабораторной работе “Установка Microsoft Compute Cluster Server 2003”) и запустите программу установки, соответствующую Вашему процессору (32-битная или 64-битная версия)



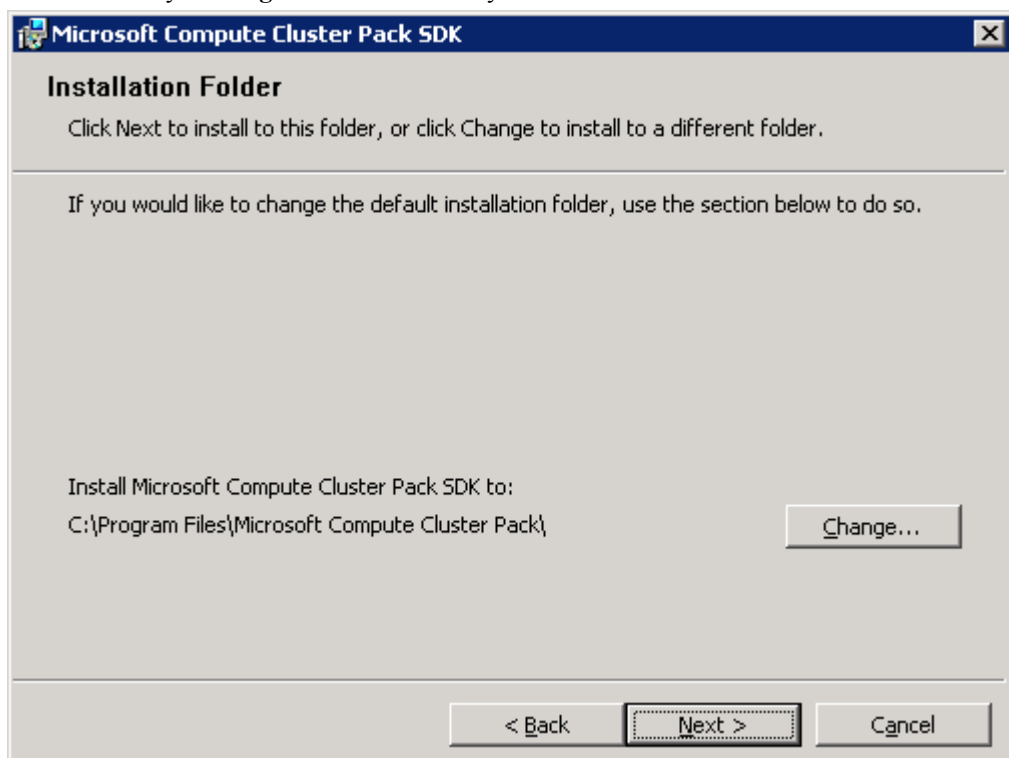
- В открывшемся окне нажмите кнопку “**Next**” для начала установки



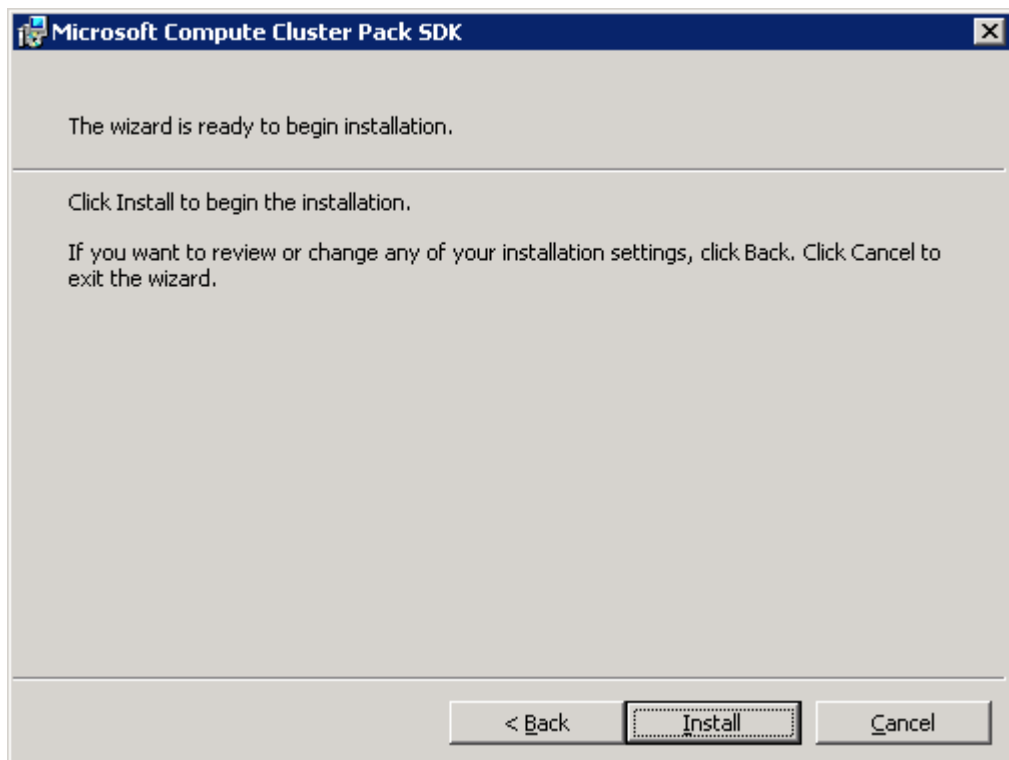
- Внимательно прочитайте лицензионное соглашение. Выберите пункт "**I accept the terms in the license agreement**" в случае согласия с лицензионным соглашением об использовании системы CCS 2003 и нажмите кнопку "**Next**"



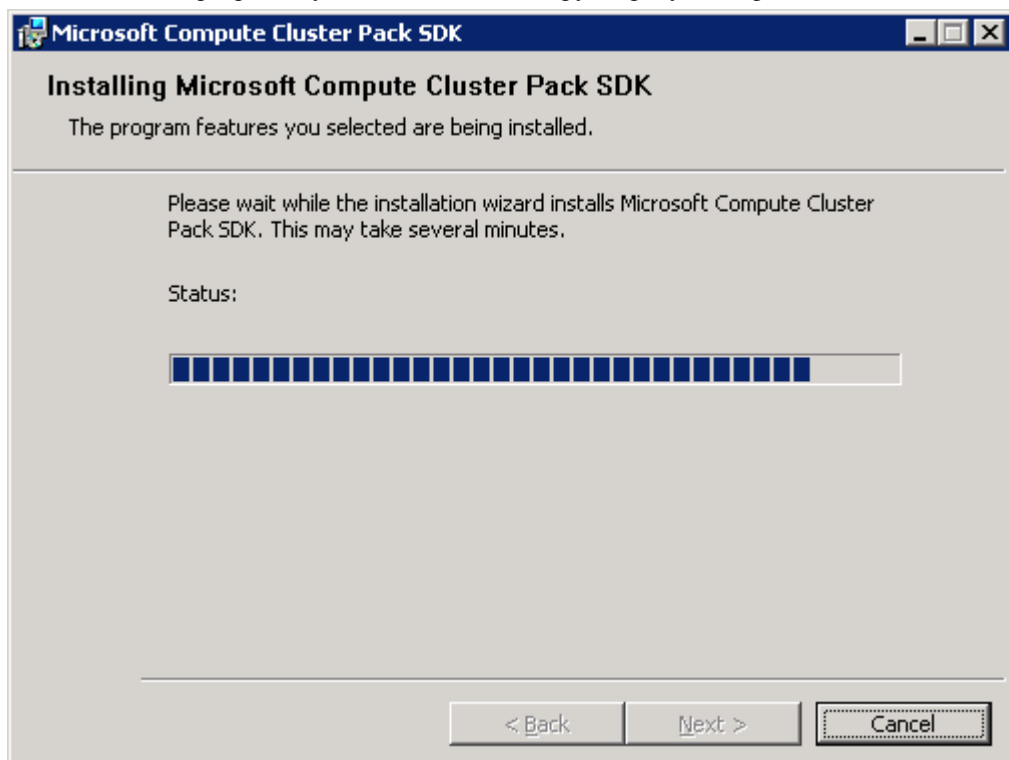
- Выберите директорию, в которую будет установлен SDK. Для изменения стандартной директории нажмите кнопку "Change". Нажмите кнопку "Next"



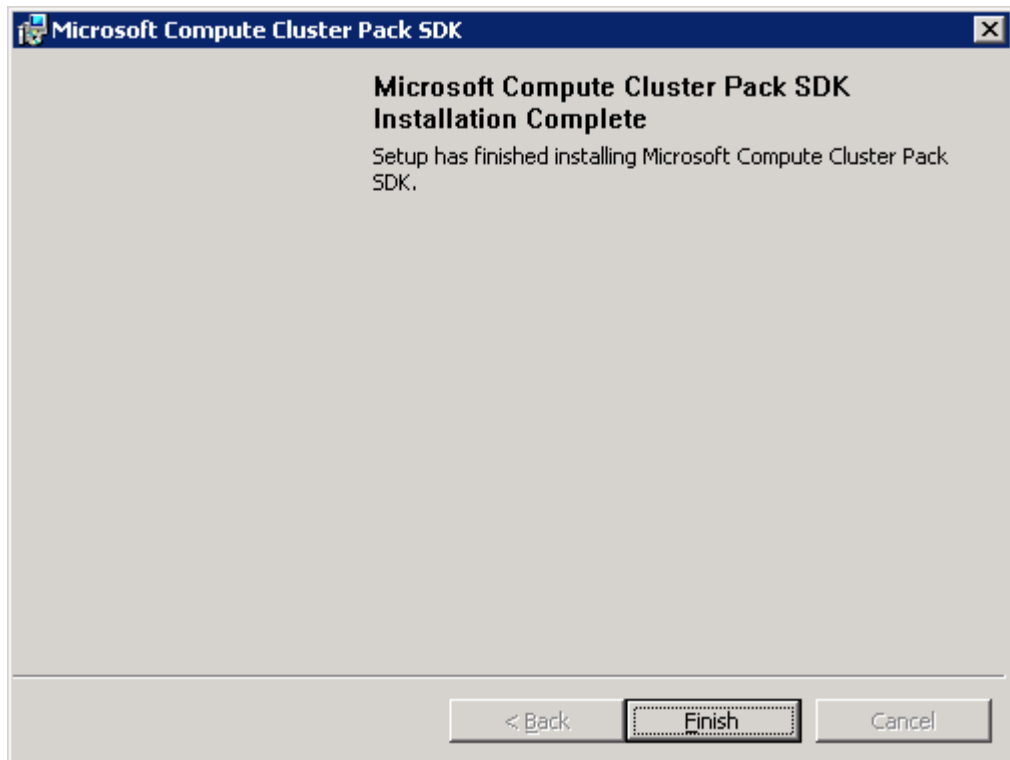
- В открывшемся окне нажмите кнопку "Install" для начала установки SDK



- Дождитесь пока программа установки SDK скопирует требуемые файлы



- По окончании копирования необходимых файлов нажмите кнопку "**Finish**"

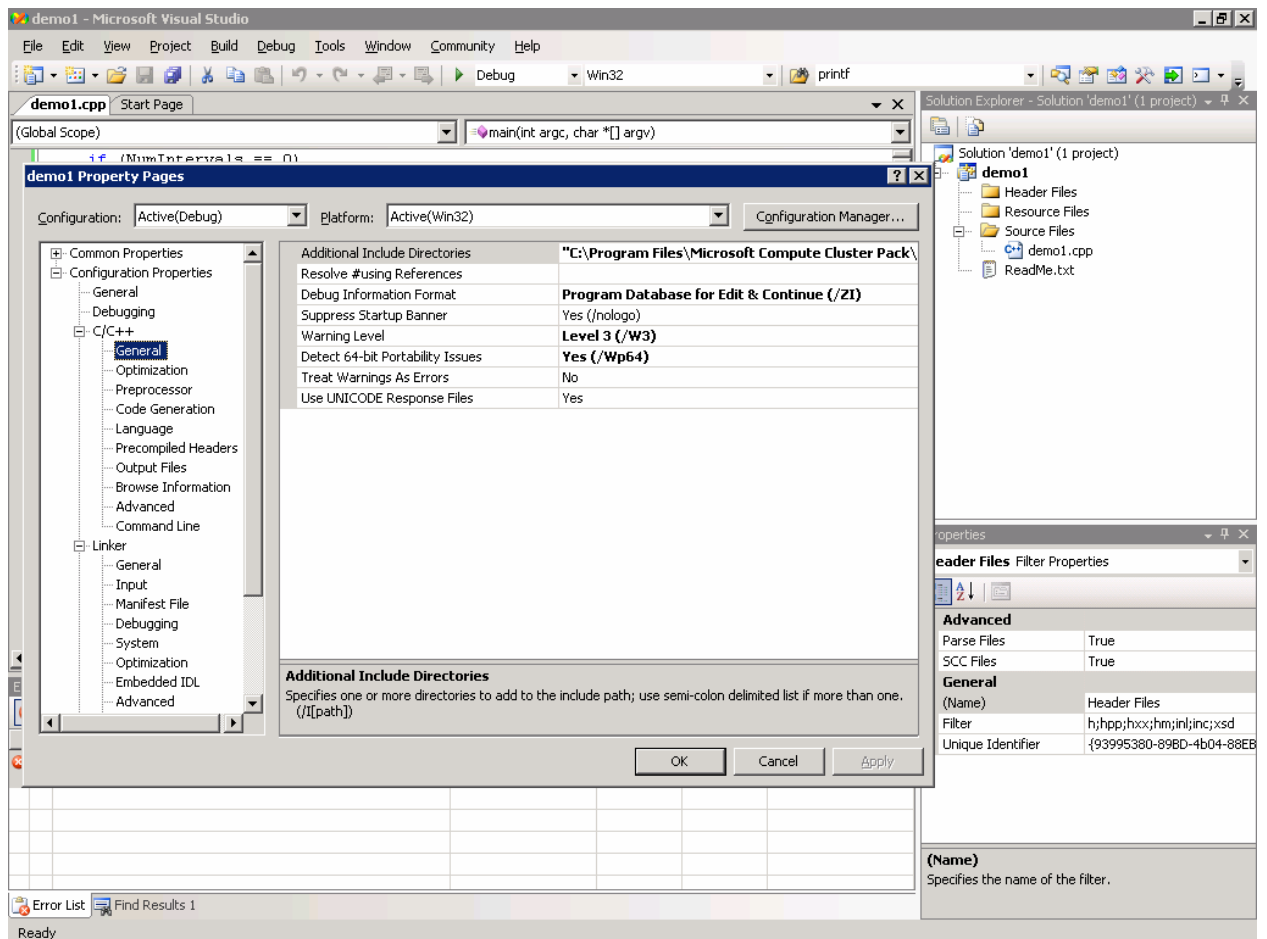


- Поздравляем! Установка Microsoft Compute Cluster Server 2003 SDK завершена.

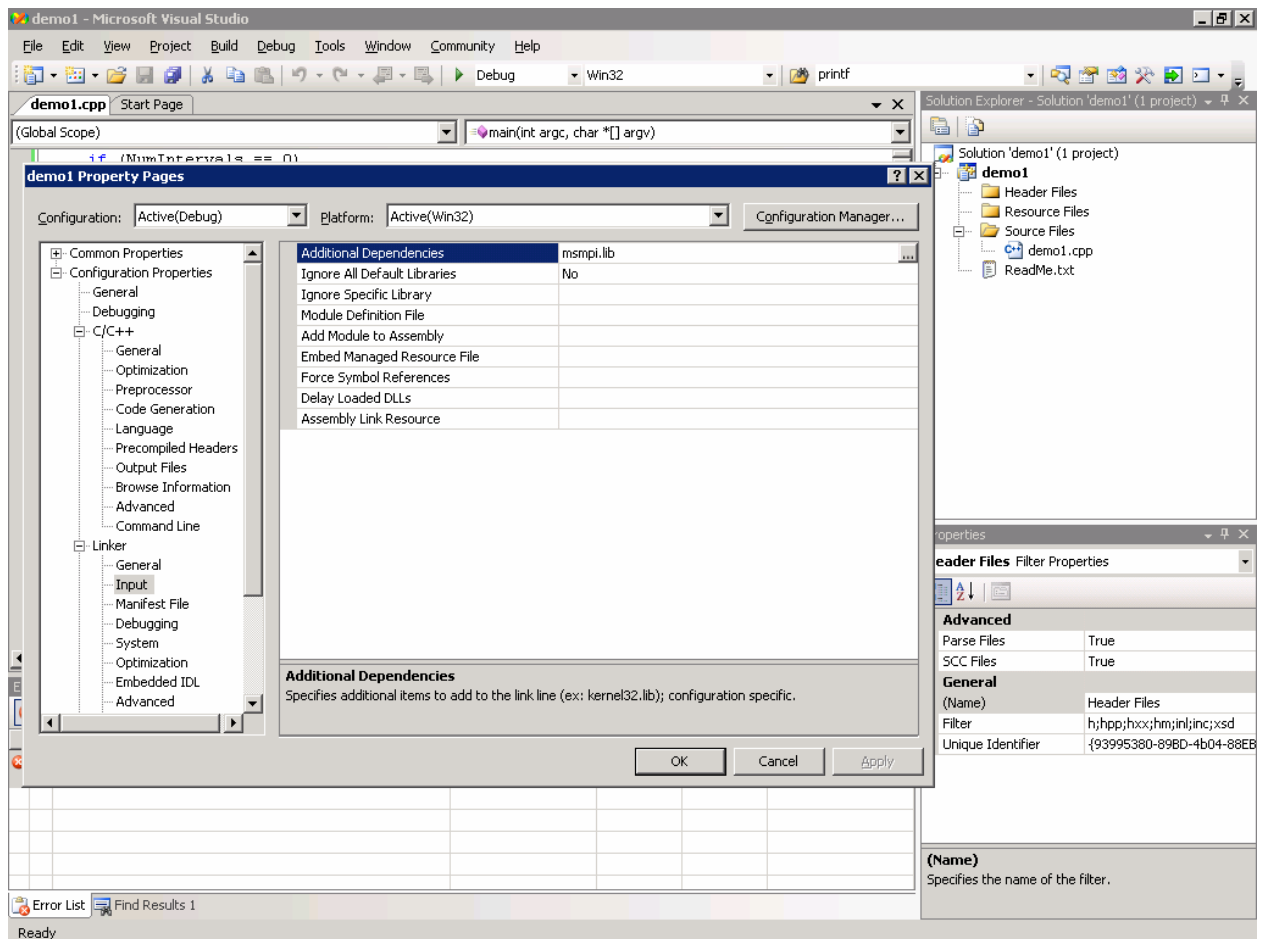
Задание 2 – Настройка интегрированной среды разработки Microsoft Visual Studio 2005

Для того, чтобы скомпилировать Вашу программу для использования в среде MS MPI, необходимо изменить следующие настройки проекта по умолчанию в Microsoft Visual Studio 2005:

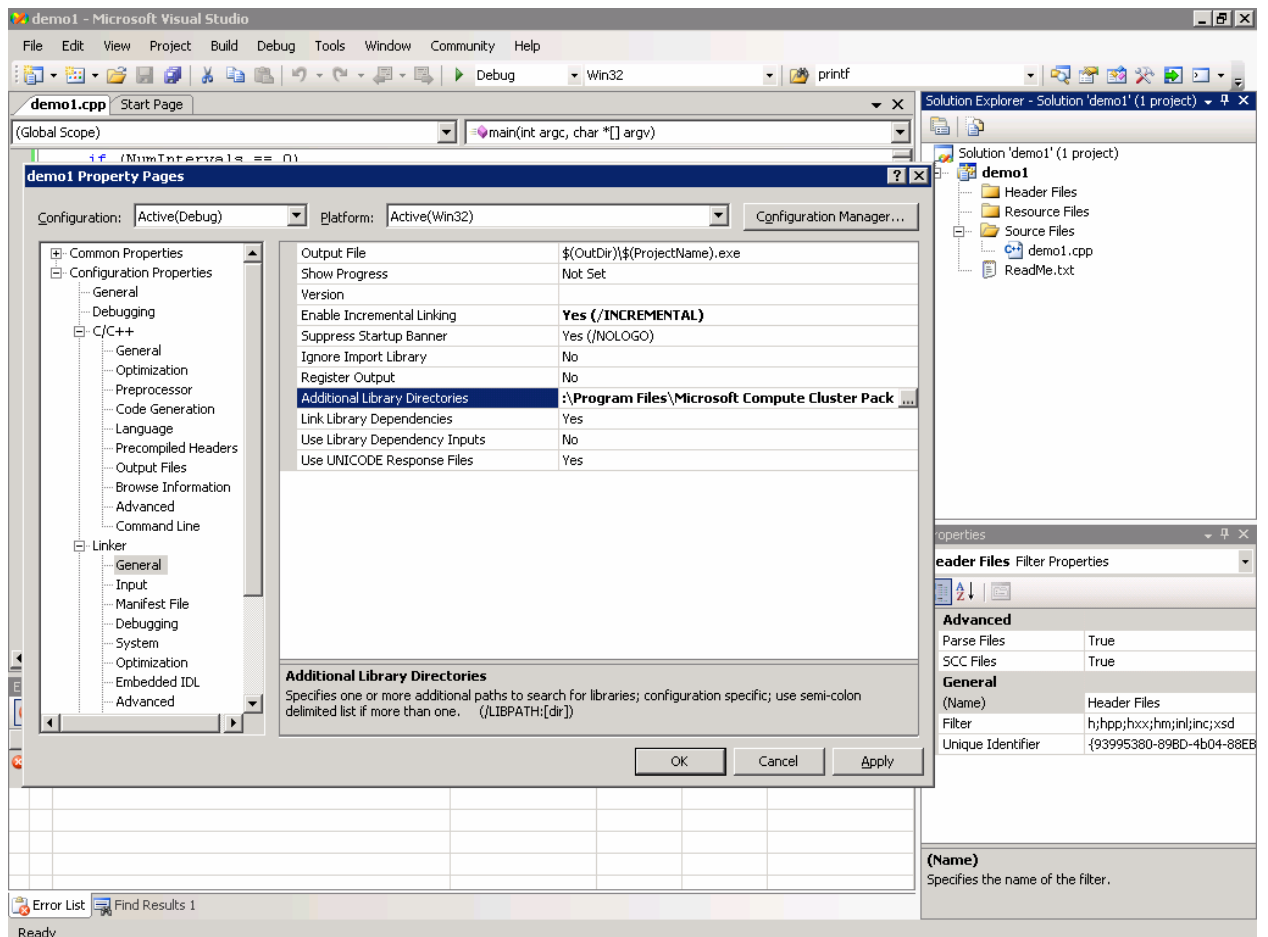
- **Путь до заголовочных файлов объявлений MPI.** Выберите пункт меню **Project->Project Properties**. В пункте **Configuration Properties->C++->General->Additional Include Directories** введите путь до заголовочных файлов MS MPI: **<Директория установки CCS SDK>\Include**



- Библиотечный файл с реализацией функций MPI. Выберите пункт меню **Project->Project Properties**. В пункте **Configuration Properties->C++->Linker->Input->Additional Dependencies** введите название библиотечного файла **msmpi.lib**



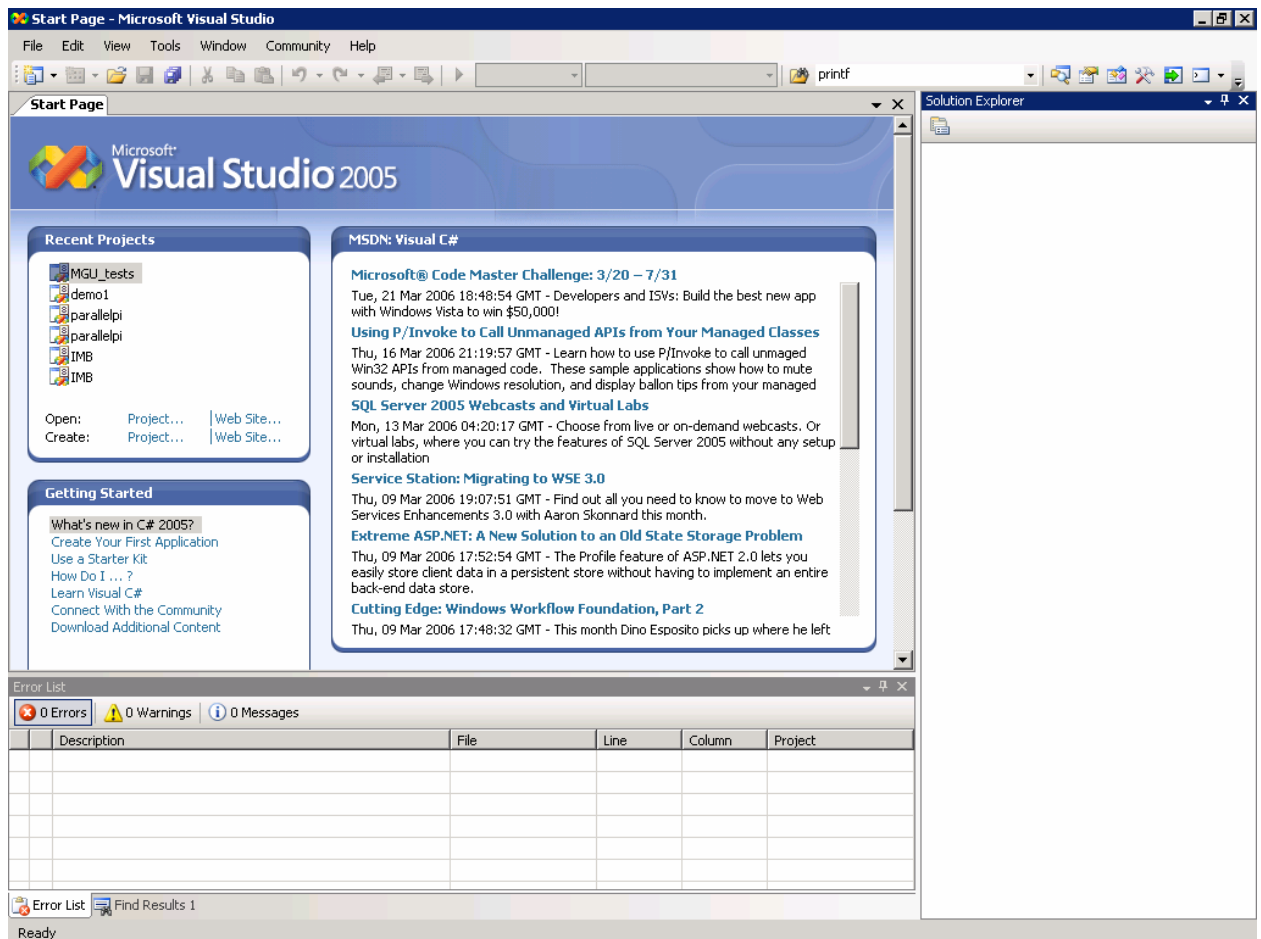
- **Путь до библиотечного файла msmpi.lib.** Выберите пункт меню **Project->Project Properties**. В пункте **Configuration Properties->C++->Linker->General->Additional Library Directories** введите путь до библиотечного файла **msmpi.lib**: **<Директория установки CCS SDK>\Lib\i386** или **<Директория установки CCS SDK>\Lib\AMD64** в зависимости от используемой Вами архитектуры процессоров.



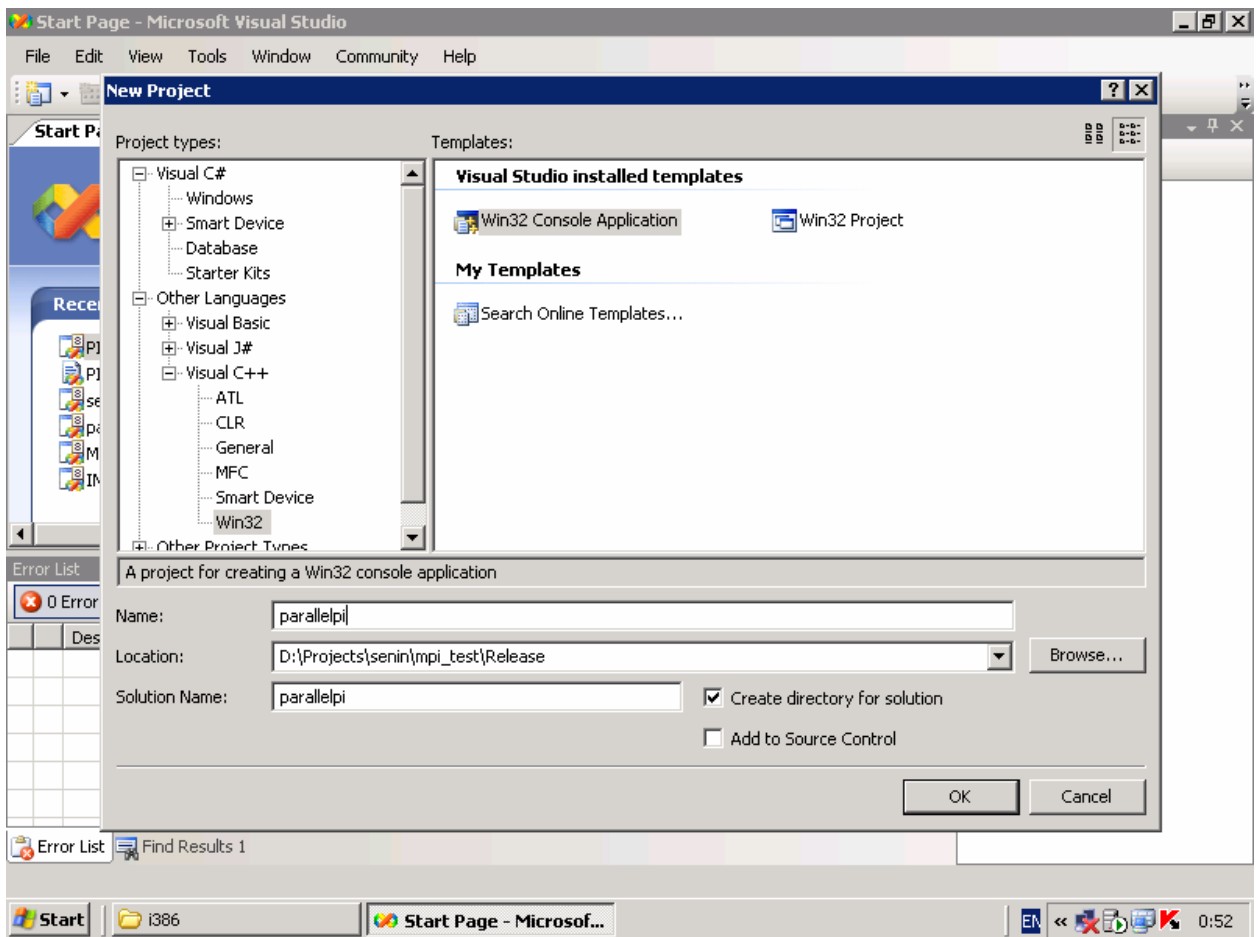
Задание 3 – Компиляция параллельной программы в Microsoft Visual Studio 2005

В качестве примера параллельной программы для этого задания будет использоваться параллельный алгоритм вычисления числа π . В данной работе рассматриваются только технические вопросы использования Microsoft Compute Cluster Server 2003; описание алгоритма и вопросы его реализации рассмотрены в лабораторной работе "Параллельный метод вычисления числа π ". В данном задании будут рассмотрены вопросы использования Visual Studio 2005 для компиляции параллельной MPI программы для использования в среде MS MPI:

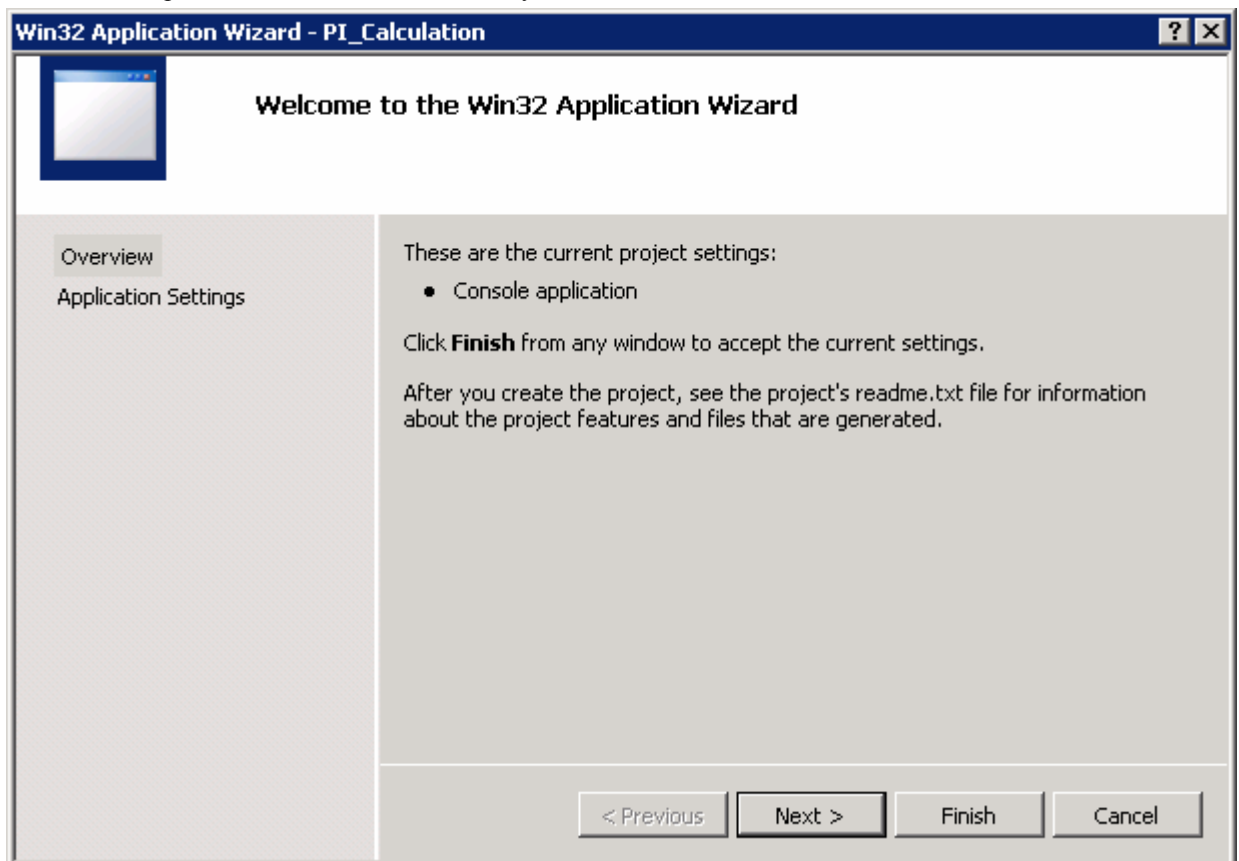
- Запустите **Microsoft Visual Studio 2005**



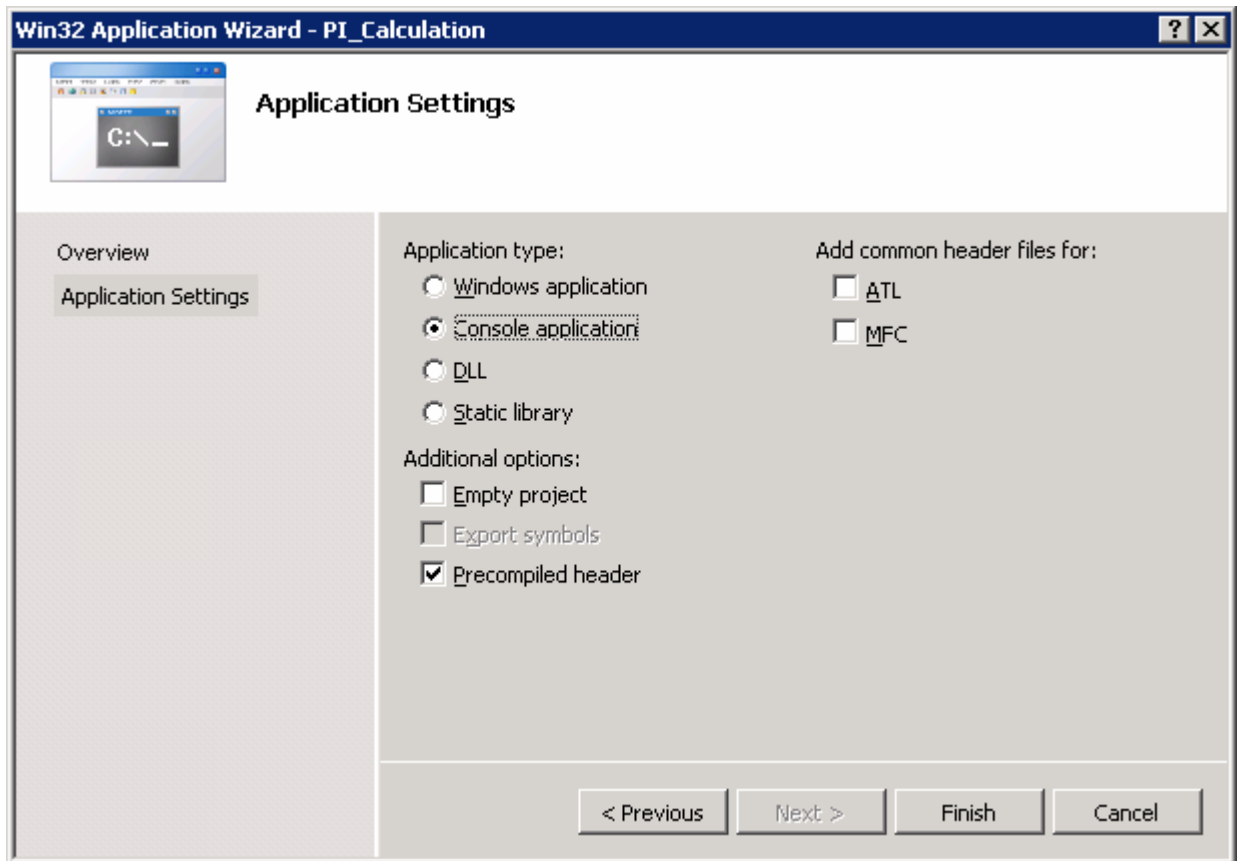
- Создайте новый проект: выберите пункт меню **File->New->Project**. В окне выбора нового проекта выберите **консольное Win32 приложение (Other Languages->Visual C++->Win32->Win32 Console Application)**, введите имя проекта в поле "Name" (например, "parallelp1") и убедитесь, что путь до проекта выбран правильно (поле "Location"). Нажмите кнопку "OK" для выбора остальных настроек создаваемого проекта



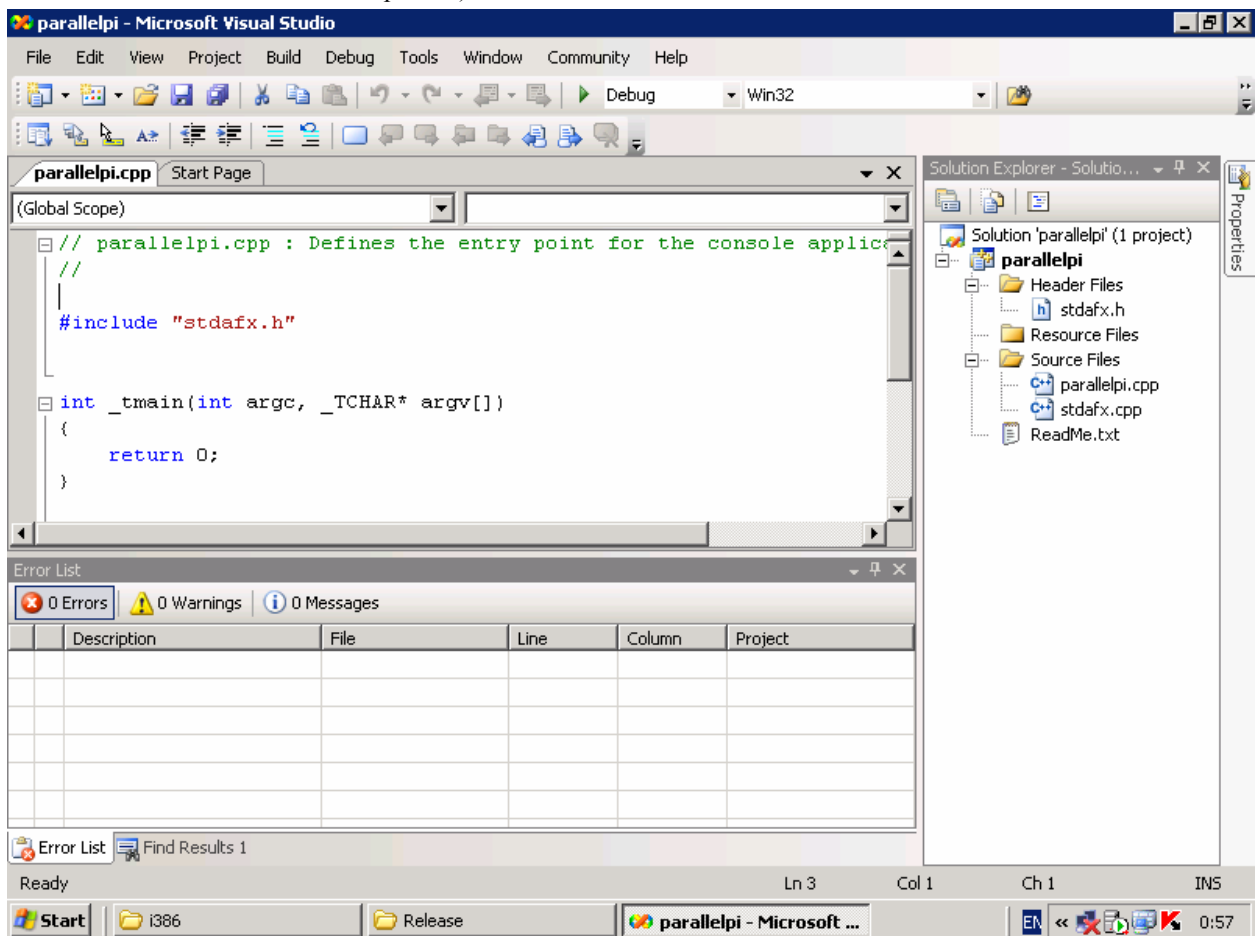
- В открывшемся окне нажмите кнопку “Next”



- В открывшемся окне выберите настройки проекта (можно оставить все настройки по умолчанию). Нажмите кнопку “Finish”



- В окне **Solution Explorer** щелкните 2 раза на файле **parallepi.cpp** (имя файла совпадает с введенным названием проекта)



- Удалите содержимое файла и замените его следующим (см. лабораторную работу "Параллельный метод вычисления числа Пи"):

```

#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

void main(int argc, char *argv[]) {
    int    NumIntervals    = 0;    // num intervals in the domain [0,1]
    double IntervalWidth  = 0.0;  // width of intervals
    double IntervalLength = 0.0;  // length of intervals
    double IntrvlMidPoint = 0.0;  // x mid point of interval
    int    Interval       = 0;    // loop counter
    int    done           = 0;    // flag
    double MyPI           = 0.0;  // storage for PI approximation results
    double ReferencePI    = 3.141592653589793238462643; // value for comparison
    double PI;
    char   processor_name[MPI_MAX_PROCESSOR_NAME];
    char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
    int    numprocs;
    int    MyID;
    int    namelen;
    int    proc = 0;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
    MPI_Get_processor_name(processor_name,&namelen);

    all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

    MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
               all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
    if (MyID == 0) {
        for (proc=0; proc < numprocs; ++proc)
            printf("Process %d on %s\n", proc, all_proc_names[proc]);
    }

    IntervalLength = 0.0;
    if (MyID == 0) {
        if (argc > 1) {
            NumIntervals = atoi(argv[1]);
        }
        else {
            NumIntervals = 100000;
        }
        printf("NumIntervals = %i\n", NumIntervals);
    }

    // send number of intervals to all procs
    MPI_Bcast(&NumIntervals, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (NumIntervals != 0)
    {
        //approximate the value of PI
        IntervalWidth  = 1.0 / (double) NumIntervals;

        for (Interval = MyID+1; Interval <= NumIntervals; Interval += numprocs){
            IntrvlMidPoint = IntervalWidth * ((double)Interval - 0.5);
            IntervalLength += (4.0 / (1.0 + IntrvlMidPoint*IntrvlMidPoint));
        }
        MyPI = IntervalWidth * IntervalLength;
    }
}

```



```

// Calculating the sum of all local alues of MyPI
MPI_Reduce(&MyPI, &PI, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

//report approximation
if (MyID == 0) {
    printf("PI is approximately %.16f, Error is %.16f\n",
        PI, fabs(PI - ReferencePI));
}
}

MPI_Finalize();
}

```

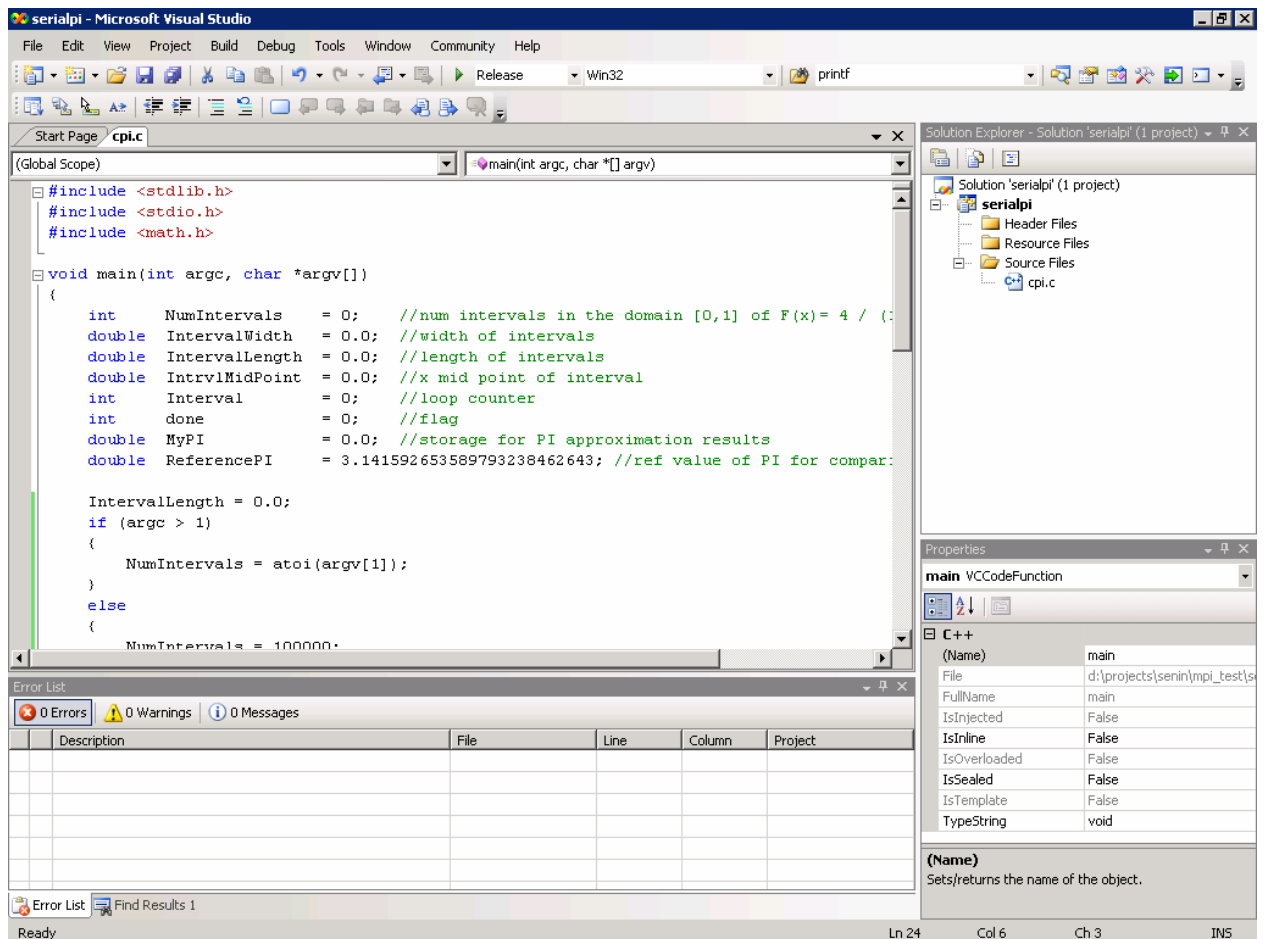
- Выполните настройки проекта Visual Studio 2005 для компиляции MPI части проекта в соответствии с указаниями пункта “**Настройка интегрированной среды разработки Microsoft Visual Studio 2005**”,
- Выполните команду **Build->Rebuild Solution** для компиляции и линковки проекта,
- Поздравляем! Компиляция параллельной программы для MS MPI успешно завершена.

Упражнение 2 – Запуск последовательной задачи

Последовательной называется задача, которая используется для своей работы ресурсы только одного процессора. Порядок компиляции последовательной программы (а также параллельной программы с использованием технологии OpenMP) для использования на кластере под управлением Compute Cluster Server 2003 не отличается от обычного и не требует использования дополнительных библиотек. В данном задании рассматривается порядок запуска последовательной задачи на кластере.

Запуск программы через графический пользовательский интерфейс

- Откройте проект последовательного алгоритма вычисления числа Пи (**serialpi**), поставляющийся вместе с лабораторной работой "Параллельный метод вычисления числа Пи", и скомпилируйте программу в конфигурации **Release**



- Откройте **Computer Cluster Job Manager (Start->All Programs->Microsoft Compute Cluster Pack->Compute Cluster Job Manager)** для запуска программы на кластере. Если Вы установили клиентскую часть **Compute Cluster Pack** на Вашу рабочую станцию, то постановку задач в очередь можно выполнять непосредственно с Вашего компьютера, иначе необходимо зайти по **Remote Desktop Connection** на головной узел кластера или любой другой узел с установленной клиентской частью

Job Queue at s-cw-head

ID	Name	Priority	Submitted By	Status	Submit Time	Pending Reason
1	hostname	Normal	CCAM\Senin	Finished	21.06.2006 4:10:14	
2	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:19:50	
3	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:24:24	
4	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:25:23	
5	imb	Normal	CCAM\Senin	Finished	21.06.2006 5:13:20	

Select a job to view its tasks

Name	Status	Task Id	Command Line	Processors	End Time	Failure Message

Ready

- В открывшемся окне менеджера заданий выберите пункт меню **File->Submit Job** для постановки нового задания в очередь
- В окне постановки задания в очередь введите имя задания (поле “**Job Name**”), при необходимости измените приоритет задания (пользовательские задания с большим приоритетом будут выполнены раньше заданий с меньшим приоритетом). Перейдите на вкладку “**Processors**”

Submit Job Serial Pi computing

General | Processors | Tasks | Licenses | Advanced

Serial Pi computing

Job Name: Serial Pi computing

Project Name:

Priority: Normal

Submitted By: N/A

Submitted on: N/A

Status: Not Submitted

Save As Template... Submit Cancel

- На вкладке **Processors** введите **минимальное** и **максимальное** числа процессоров, необходимых для выполнения задания (в нашем случае максимальное необходимое число процессоров – один, так как задача последовательная). Под максимальным здесь понимается оптимальное для задания число процессоров (именно столько будет выделено в случае низкой загрузки кластера). Гарантируется, что задание не начнет выполняться, если на кластере доступно менее минимального числа процессоров. Дополнительно можно ввести предполагаемое время работы задания (это поможет планировщику эффективнее распределить системные ресурсы) – панель **Estimate run time for this job**. Если Вы хотите, чтобы для задания вычислительные ресурсы оставались зарезервированными в течение указанного времени, даже после того, как все задачи задания завершили свою работу, то поставьте галочку около пункта **“Run job until end of run time or until canceled”**. Таким образом, Вы сможете запускать новые задачи в рамках задания даже после того, как все задачи, указанные первоначально, выполнены. Перейдите на вкладку **“Tasks”** для добавления в задание новых задач

Submit Job Serial Pi computing

General Processors Tasks Licenses Advanced

Processors required for this job

Processors available in this cluster: 60

Minimum required: 1

Maximum required: 1

Estimate run time for this job

Days: 0 Hours: 0 Minutes: 1

Run job until end of run time or until canceled.

This option lets you run extra tasks after running all tasks already listed in the job if there is time left.

Save As Template... Submit Cancel

- Введите имя задачи (поле “**Task Name**”) и команду, которую необходимо выполнить (поле “**Command Line**”) – имя программы и параметры командной строки. Программу необходимо разместить на сетевом диске, доступном со всех узлов кластера. Нажмите кнопку “**Add**” для добавления задачи в задание

Submit Job Serial Pi Calculation [X]

General | Processors | **Tasks** | Licenses | Advanced

Task Command Line

Task Name:

Command Line:

Use job's allocated processors

Minimum Processors: Maximum Processors:

This job contains the following tasks:

Order	Command Line	Processors
-------	--------------	------------

[What is it?](#)

Task Summary

- Добавленная задача появится в списке задач текущего задания (список “**This job contains the following tasks**”). Выделите ее в списке и нажмите кнопку “**Edit**” для редактирования дополнительных параметров задачи

Submit Job Serial Pi Calculation

General | Processors | **Tasks** | Licenses | Advanced

Task Command Line

Task Name:

Command Line:

Use job's allocated processors

Minimum Processors: Maximum Processors:

This job contains the following tasks:

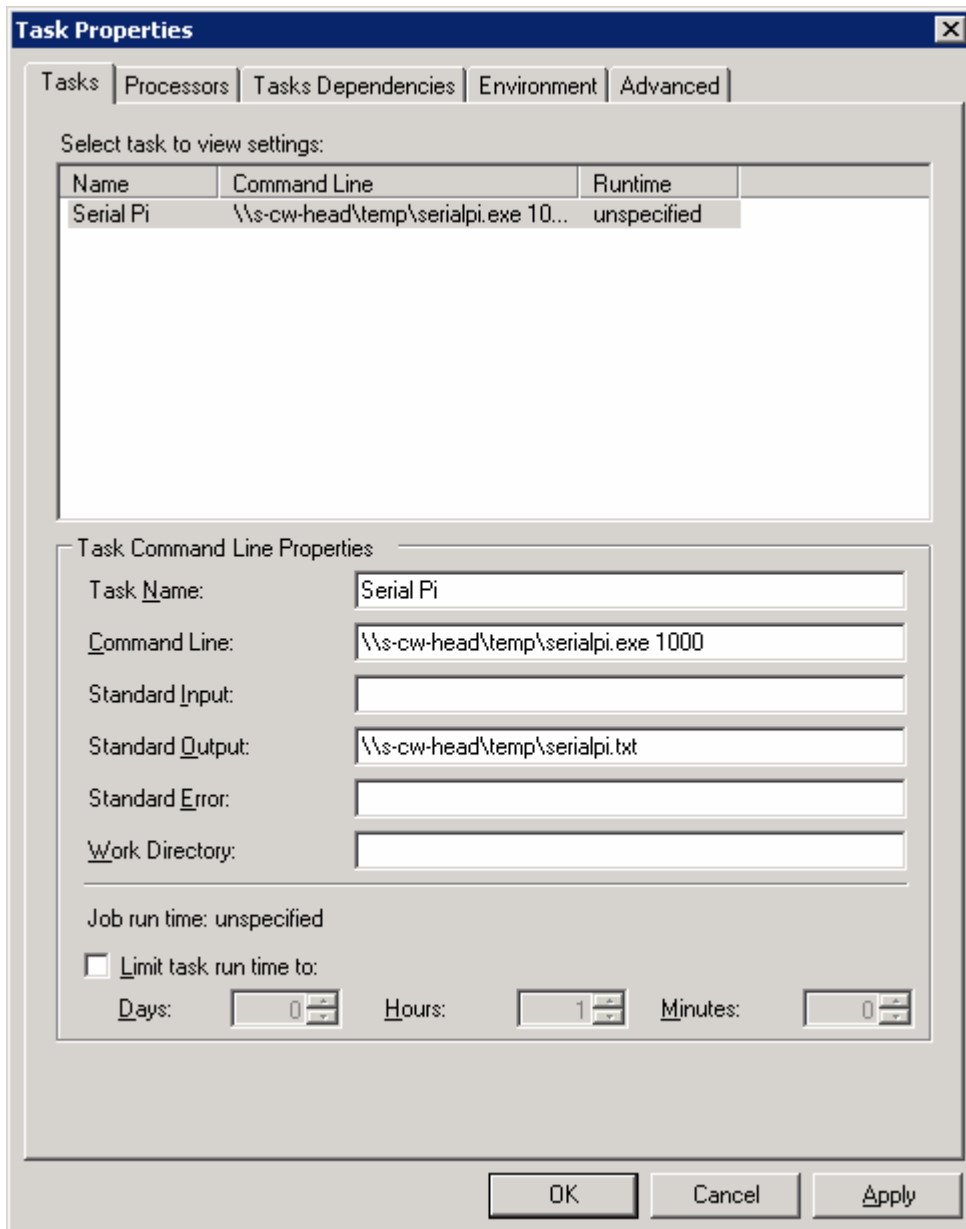
Order	Command Line	Processors
1	\\s-cw-head\temp\serialpi.exe 1000	1

[What is it?](#)

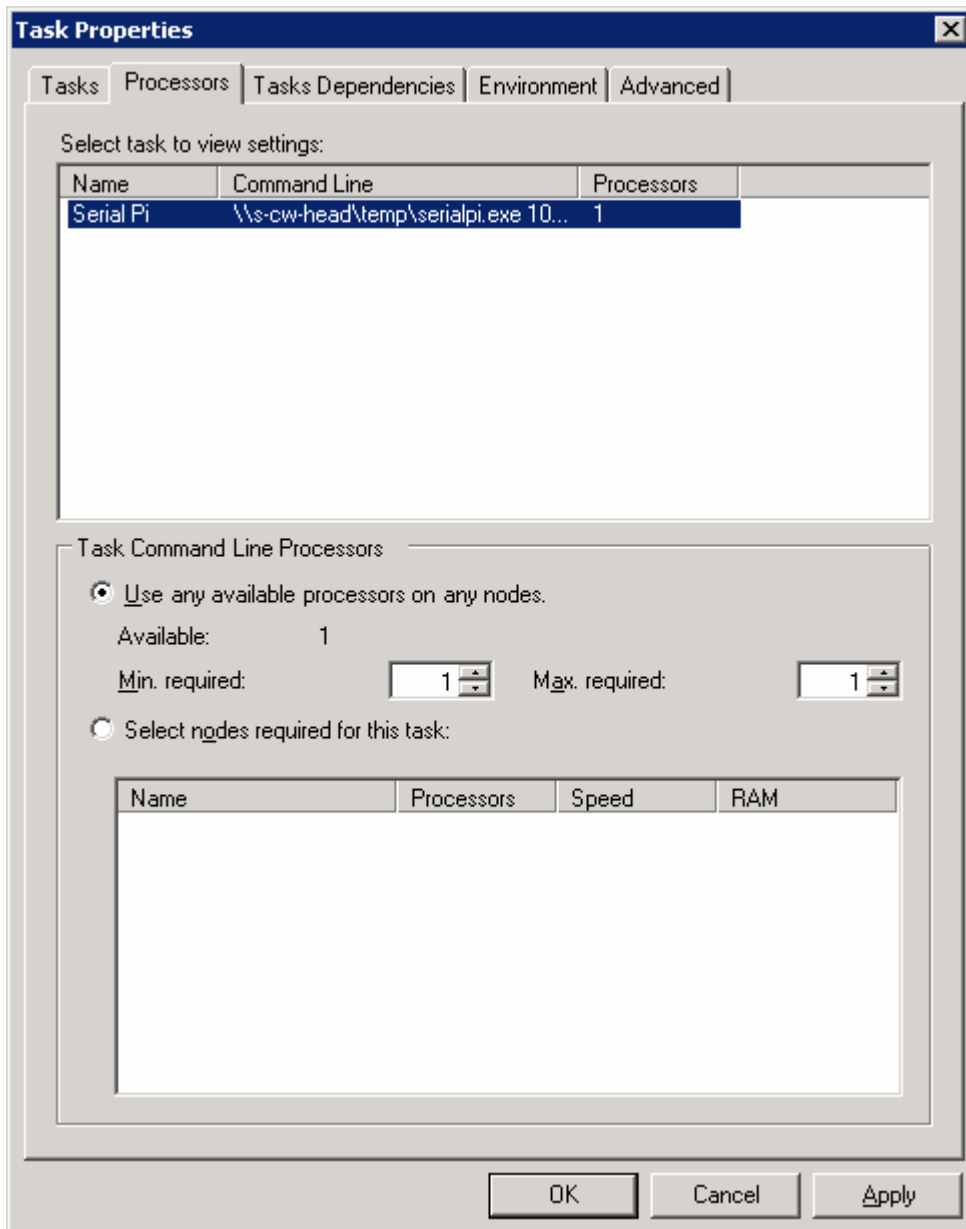
Task Summary

Name :Serial Pi
 Command Line :\\s-cw-head\temp\serialpi.exe 1000
 Standard Input :
 Standard Output :
 Standard Error :
 Work Directory :
 Number of processors requested :1
 RunTime :Infinite
 Preceding tasks(dependent tasks) :
 Exclusive :False

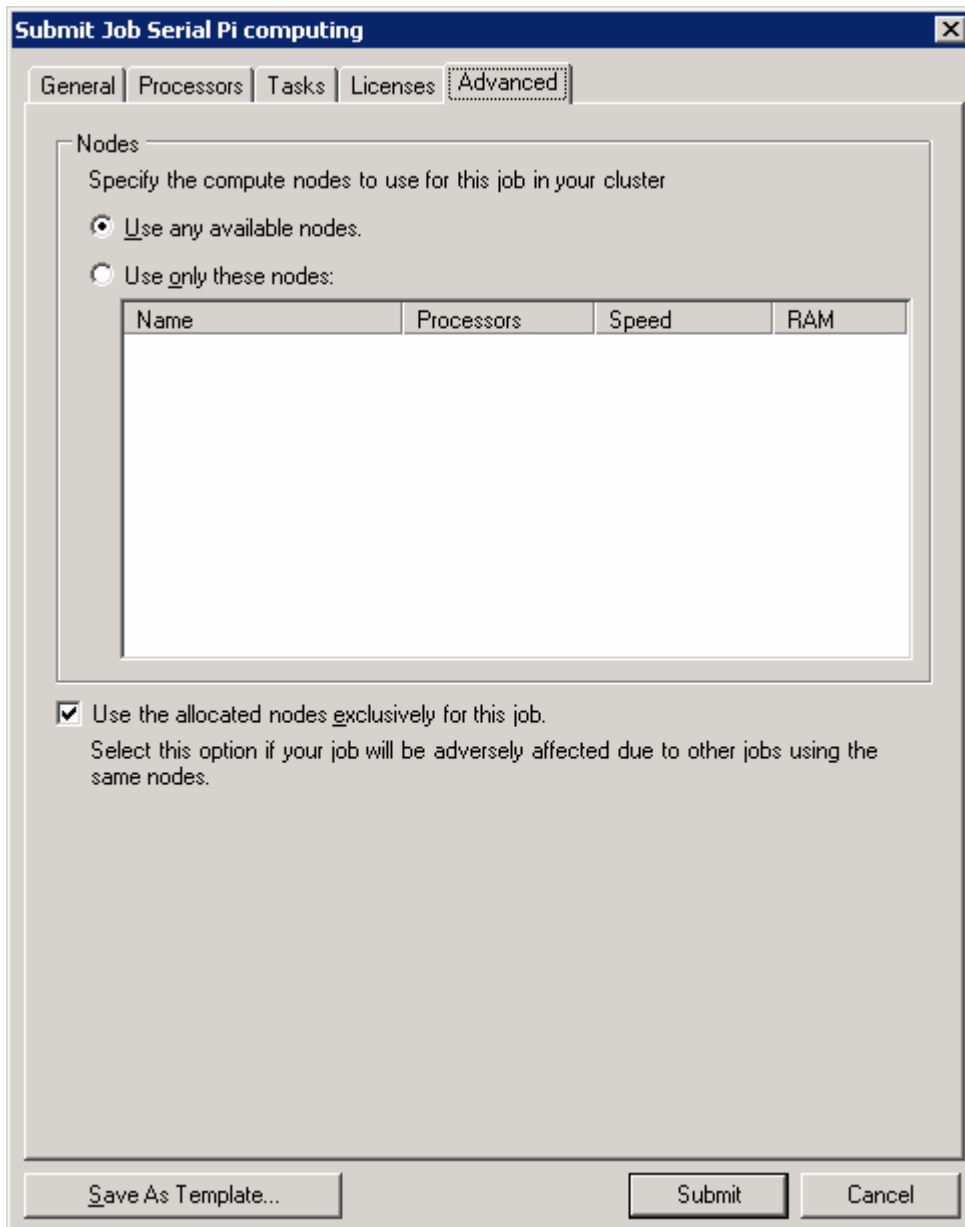
- В открывшемся окне введите файл, в который будет перенаправлен стандартный поток вывода консольного приложения (поле **“Standard Output”**). Кроме того, можно указать файл стандартного потока ввода (поле **“Standard Input”**), файл стандартного потока ошибок (поле **“Standard Error”**), рабочую директорию запускаемой программы (поле **“Work Directory”**) и ограничение по времени на продолжительность выполнения задачи (суммарное время выполнения задач не должно превышать оценки времени выполнения задания) – поле **“Limit task run time to”**. Выберите вкладку **“Processors”**,



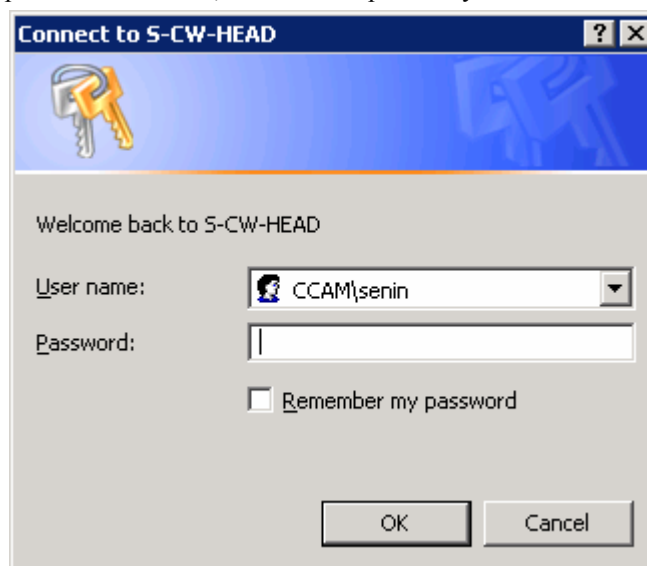
- На вкладке “**Processors**” в верхнем списке (“**Select task to view settings**”) выделите задачу, настройки которой Вы хотите изменить, и укажите минимальное и максимальное числа процессоров для выбранной задачи (поля “**Min. required**” и “**Max. required**”) в случае, если Вы хотите, чтобы планировщик выбрал узлы для запуска автоматически (“**Use any available processors on any nodes**”). Если Вы хотите вручную указать узлы для запуска, выберите пункт “**Select nodes required for this task**” и поставьте флажки около требуемых узлов в нижнем списке. Так как задача последовательная, то для ее выполнения требуется только 1 процессор. На этом настройка параметров задачи закончена. Нажмите “**OK**” для сохранения внесенных изменений и возвращения к настройкам задания



- Перейдите на вкладку “**Advanced**” и выберите пункт “**Use any available nodes**” для автоматического выбора узлов для задания. Если Вы хотите вручную указать узлы, на которых будет выполняться задание, выберите пункт “**Use only these nodes**”. Учтите, что в том случае, если Вы вручную указали узлы для запуска задачи, то они также должны быть выбраны для всего задания (в случае неавтоматического распределения). Установите флаг “**Use the allocated nodes exclusively for this job**” для того, чтобы запретить запуск нескольких заданий на одном узле. Нажмите кнопку “**Submit**” для добавления задания в очередь



- Введите имя и пароль пользователя, имеющего право запуска задач на кластере, и нажмите “OK”



- Задание появится в очереди. По окончании выполнения его состояние изменится на “**Finished**”

The screenshot shows a window titled "Job Queue at s-cw-head" with a menu bar (File, View, Job, Help) and a "Show:" dropdown set to "All Jobs". Below is a table of jobs:

ID	Name	Priority	Submitted By	Status	Submit Time	Pending Reason
1	hostname	Normal	CCAM\Senin	Finished	21.06.2006 4:10:14	
2	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:19:50	
3	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:24:24	
4	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:25:23	
5	imb	Normal	CCAM\Senin	Failed	21.06.2006 5:13:20	
6	Serial Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 1:59:22	
7	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 2:00:46	
8	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 3:50:22	
9	Serial Pi Calculation	Normal	CCAM\Senin	Finished	25.06.2006 4:08:39	

Below the job list is a section titled "Tasks for Serial Pi Calculation" with a sub-table:

Name	Status	Task Id	Command Line	Processors	End Time	Failure Message
Serial Pi	Finished	1	\\s-cw-head\temp\serialpi...	1	25.06.2006 ...	

The status bar at the bottom of the window reads "Ready".

- В файле, указанном в настройках задачи для перенаправления стандартного потока вывода, содержится результат работы программы

The screenshot shows a Notepad window titled "serialpi.txt - Notepad" with a menu bar (File, Edit, Format, View, Help). The text content is:

```
NumIntervals = 1000
PI is approximately 3.1415927369231227, Error is 0.0000000833333296
```

Запуск программы с использованием шаблона

Если в будущем Вы захотите еще раз запустить задачу последовательного вычисления числа Пи (например, с другими параметрами), то Вам будет полезна функция сохранения всех параметров задания, запущенного ранее, в xml файл с возможностью последующего быстрого создания копии:

- Откройте **Computer Cluster Job Manager (Start->All Programs->Microsoft Compute Cluster Pack->Compute Cluster Job Manager)** и дважды щелкните левой кнопкой мыши по заданию, параметры которого Вы хотите сохранить в xml-файл

Job Queue at s-cw-head

File View Job Help Show: All Jobs

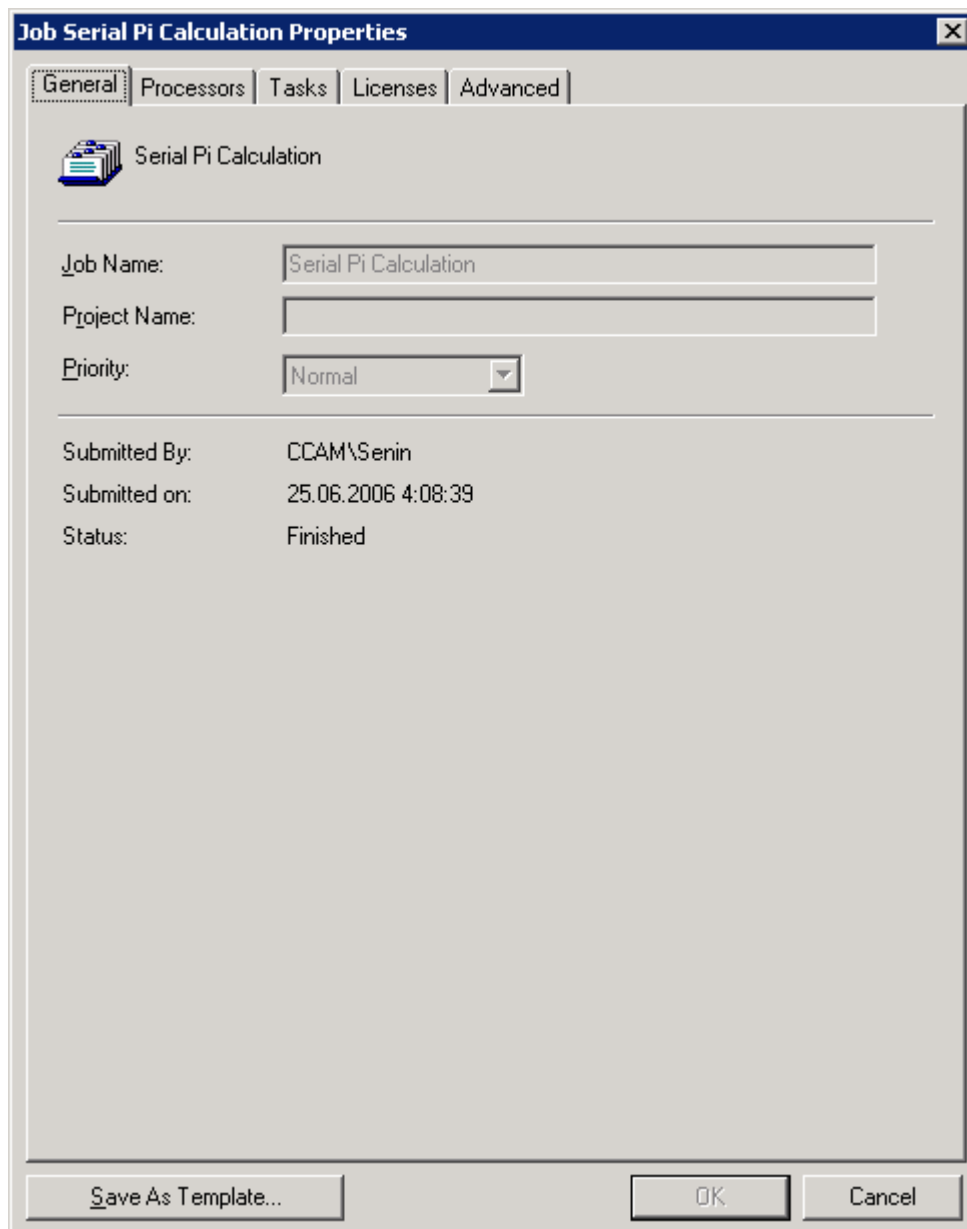
ID	Name	Priority	Submitted By	Status	Submit Time	Pending Reason
1	hostname	Normal	CCAM\Senin	Finished	21.06.2006 4:10:14	
2	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:19:50	
3	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:24:24	
4	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:25:23	
5	imb	Normal	CCAM\Senin	Failed	21.06.2006 5:13:20	
6	Serial Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 1:59:22	
7	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 2:00:46	
8	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 3:50:22	
9	Serial Pi Calculation	Normal	CCAM\Senin	Finished	25.06.2006 4:08:39	

Tasks for SerialPi Calculation

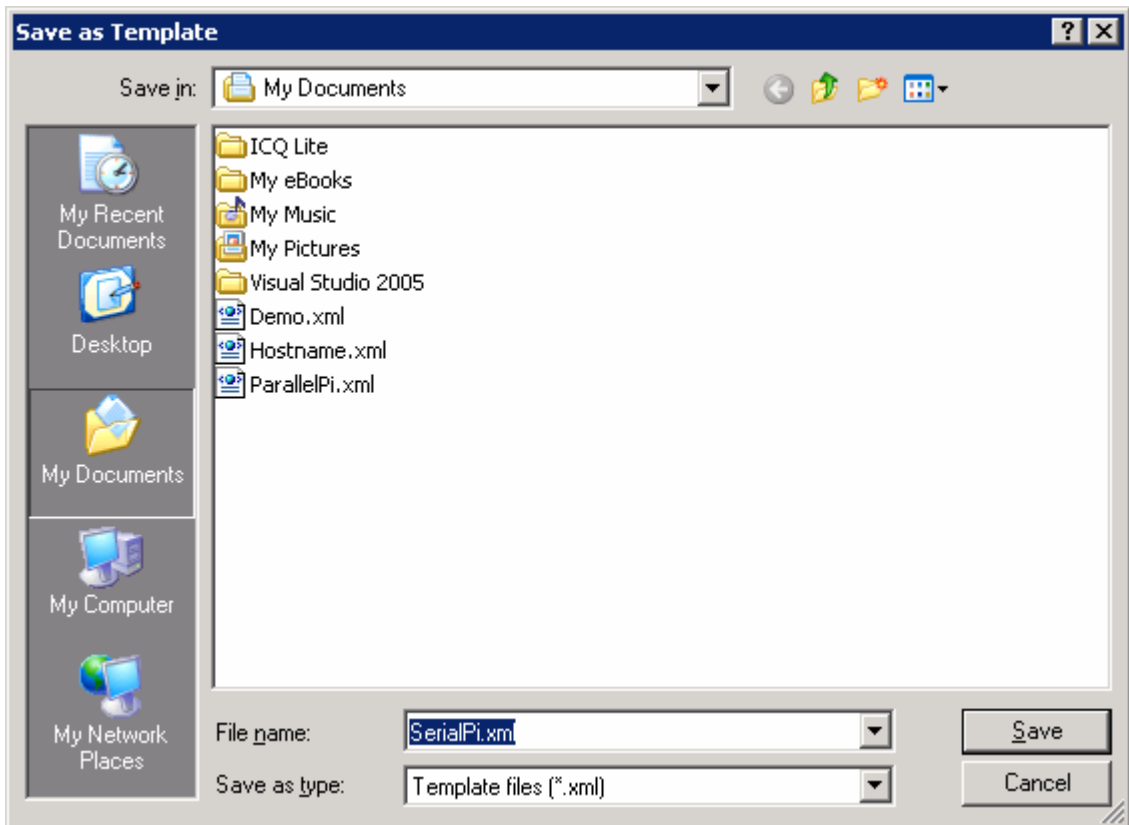
Name	Status	Task Id	Command Line	Processors	End Time	Failure Message
Serial Pi	Finished	1	\\s-cw-head\temp\serialpi...	1	25.06.2006 ...	

Ready

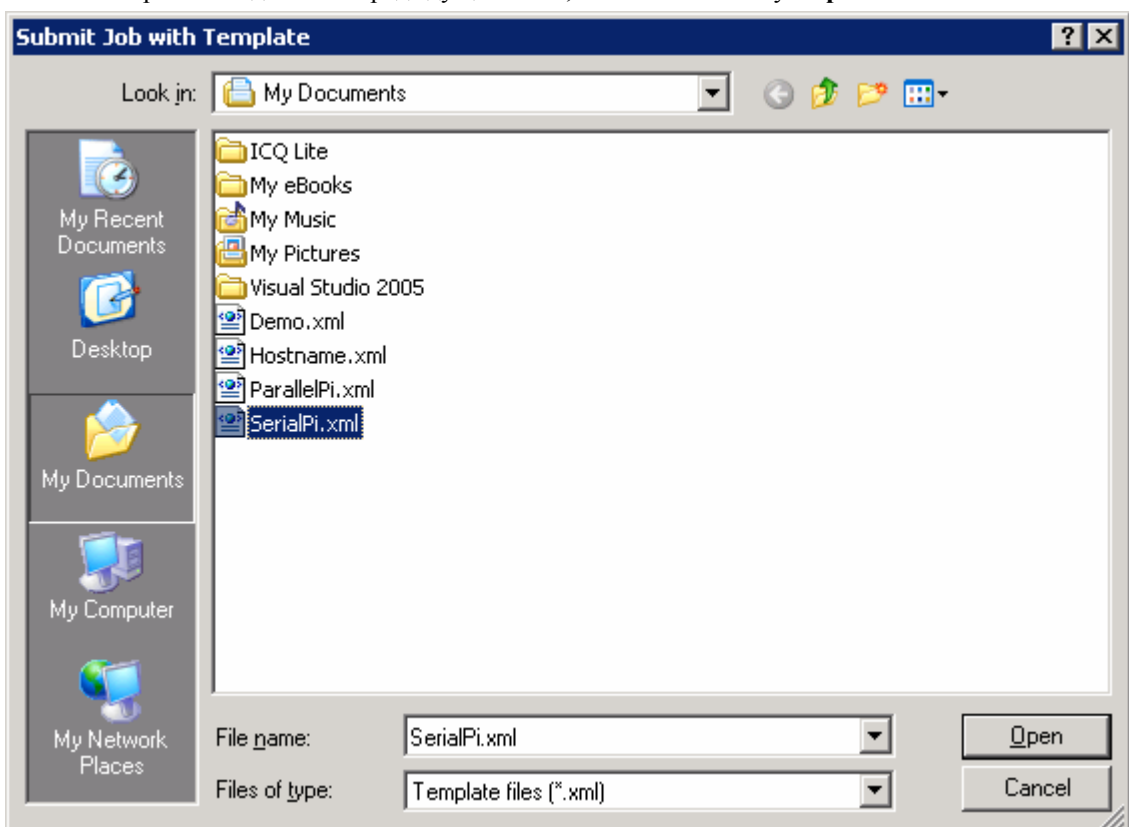
- В открывшемся окне нажмите кнопку “**Save As Template**” для сохранения параметров задания в файл



- В открывшемся окне выберите директорию, в которую следует сохранить файл и введите его имя. Нажмите кнопку “**Save**” для сохранения задания в файл

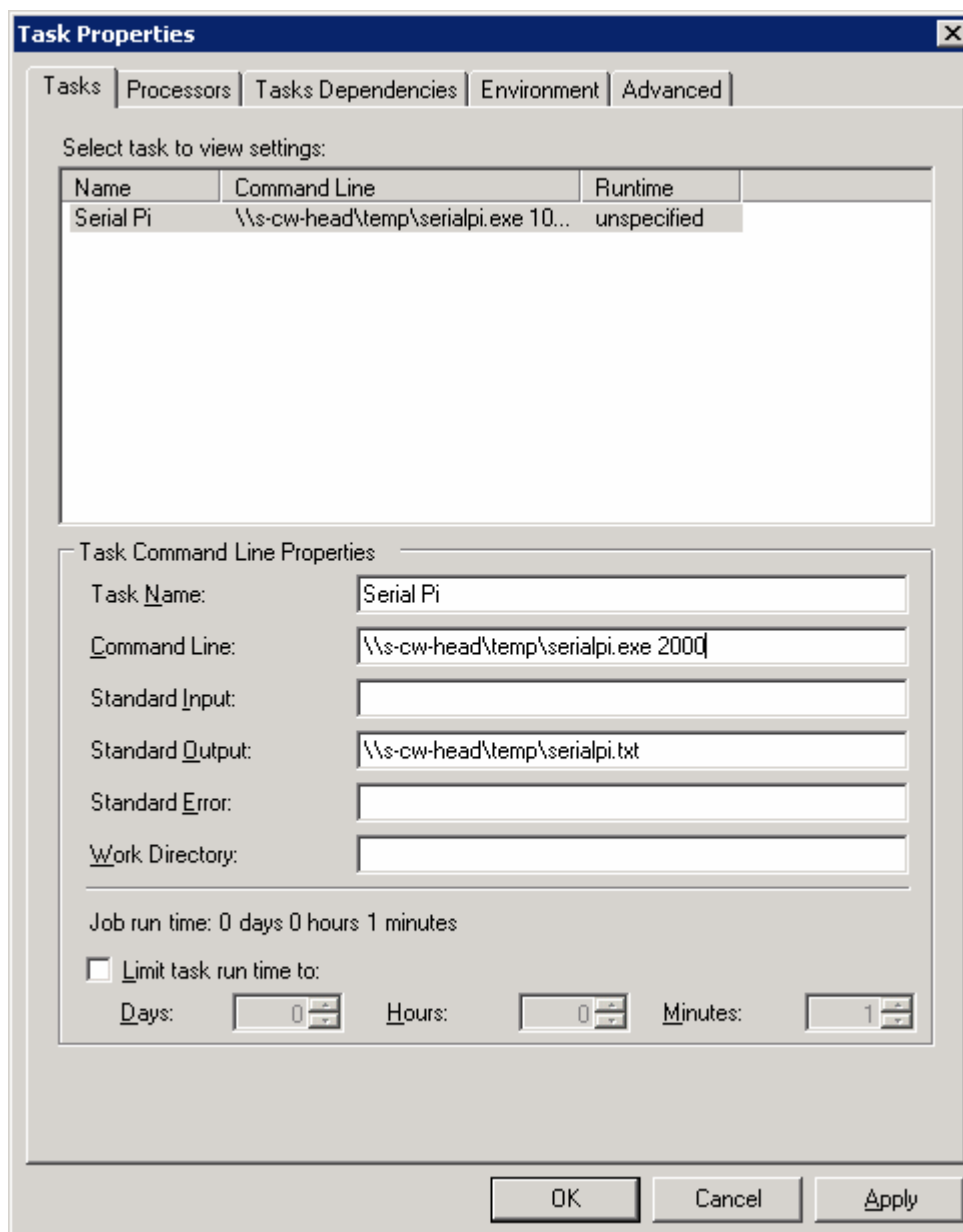


- Для создания задания с использованием шаблона в окне **Compute Cluster Job Manager** выберите пункт меню **File->Submit Job with Template...** В окне выбора шаблона выделите файл, в который было сохранено задание на предыдущем шаге, и нажмите кнопку **“Open”**



- Откроется окно добавления задания в очередь. При этом параметры задания и всех входящих в него задач будут теми же, что и у задания, на основе которого был создан шаблон. Вы можете изменять интересующие параметры задания, оставляя другие неизменными, и экономя, таким образом, время на редактировании. Например, Вы можете увеличить число разбиений отрезка интегрирования при вычислении числа Пи из прошлого примера (используемый алгоритм вычисления числа Пи

сводится к численному вычислению определенного интеграла), оставив остальные параметры теми же.



Запуск программы из командной строки

Часто бывает удобней управлять ходом выполнения заданий из командной строки. Microsoft Compute Cluster Server 2003 имеет в своем составе текстовые утилиты, предоставляющие полный контроль за ходом выполнения заданий на кластере.

В данной лабораторной работе мы покажем, как из командной строки запускать последовательные задачи. Запуск параллельных задач, а также создание параметрического множества и потока задач может быть также осуществлен из командной строки. Дополнительная информация о командах и их параметрах содержится в документации, поставляемой с Microsoft Compute Cluster Pack.

Для запуска последовательного алгоритма вычисления числа Пи выполните следующие действия:

- Откройте командный интерпретатор (**Start->Run**, введите команду **"cmd"** и нажмите **"ввод"**),
- Для создания нового задания введите команду **"job new /jobname:SerialPiCL /scheduler:s-cw-head"** (не забудьте заменить параметры команды на соответствующие Вашему случаю), где параметр **"jobname"** – имя добавляемого задания, **"scheduler"** – имя головного узла кластера. Команда напечатает идентификатор созданного задания, с которым Вы будете работать далее

```
C:\ Command Prompt
D:\Projects\senin\mpi_test\IMB_K\release>job new /jobname:SerialPiCL /scheduler:
s-cw-head
Job queued, ID: 26
D:\Projects\senin\mpi_test\IMB_K\release>_
```

- Для добавления новой задачи в задание введите команду “**job add 26 /numprocessors:1 /scheduler:s-cw-head /stdout://s-cw-head/temp/serialpi.txt /workdir://s-cw-head/temp/ serialpi.exe 1000**” (не забудьте заменить параметры команды на соответствующие Вашему случаю). Здесь число “26” – идентификатор задания, возвращенный на предыдущем шаге. Параметр “**numprocessors**” задает число процессоров, необходимое данной задаче (для задания минимального и максимального числа процессоров необходимо использовать формат “**/numprocessors:x-y**”, где **x** – минимальное число процессоров, **y** – максимальное число процессоров). Параметр “**stdout**” задает файл, в который будет перенаправлен стандартный поток вывода. Параметр “**workdir**” задает директорию по умолчанию для запускаемого приложения. После параметров указывается, собственно, команда и аргументы командной строки

```
C:\ Command Prompt
D:\Projects\senin\mpi_test\IMB_K\release>job new /jobname:SerialPiCL /scheduler:
s-cw-head
Job queued, ID: 26
D:\Projects\senin\mpi_test\IMB_K\release>job add 26 /numprocessors:1 /scheduler:
s-cw-head /stdout://s-cw-head/temp/serialpi.txt /workdir://s-cw-head/temp/ seria
lpi.exe 1000
Task 26.1 added.
D:\Projects\senin\mpi_test\IMB_K\release>_
```

- Для начала планирования задания введите команду “**job submit /id:26 /scheduler:s-cw-head**” (не забудьте заменить параметры команды на соответствующие Вашему случаю). Введите пароль пользователя, под которым Вы зарегистрированы в системе. На вопрос о том, следует ли запомнить Ваш пароль, чтобы не приходилось вводить его далее, введите “**n**” для отказа


```

C:\ Command Prompt
D:\Projects\senin\mpi_test\IMB_K\release>job new /jobname:SerialPiCL /scheduler:
s-cw-head
Job queued, ID: 26
D:\Projects\senin\mpi_test\IMB_K\release>job add 26 /numprocessors:1 /scheduler:
s-cw-head /stdout://s-cw-head/temp/serialpi.txt /workdir://s-cw-head/temp/ seria
lpi.exe 1000
Task 26.1 added.
D:\Projects\senin\mpi_test\IMB_K\release>job submit /id:26 /scheduler:s-cw-head
Enter the password for 'CCAM\senin' to connect to 'S-CW-HEAD':
Remember this password? (Y/N)n
Job 26 has been submitted.
D:\Projects\senin\mpi_test\IMB_K\release>_

```

- Ваша задача была добавлена в очередь, и планировщик начал осуществлять планирование ее запуска. Вы можете посмотреть ее состояние в программе **Job Manager** или введя команду “**job list /scheduler:s-cw-head /all**” (не забудьте заменить параметры команды на соответствующие Вашему случаю).

Упражнение 3 – Запуск параллельного задания

В задании 3 упражнения 1 была скомпилирована параллельная программа вычисления числа Пи для MS MPI. Запустим теперь ее на кластере под управлением Microsoft Compute Cluster Server 2003:

- Откройте **Computer Cluster Job Manager (Start->All Programs->Microsoft Compute Cluster Pack->Compute Cluster Job Manager)** для запуска программы на кластере,

ID	Name	Priority	Submitted By	Status	Submit Time	Pending Reason
1	hostname	Normal	CCAM\Senin	Finished	21.06.2006 4:10:14	
2	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:19:50	
3	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:24:24	
4	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:25:23	
5	imb	Normal	CCAM\Senin	Finished	21.06.2006 5:13:20	
6	Serial Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 1:59:22	
7	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 2:00:46	
8	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 3:50:22	
9	Serial Pi Calculation	Normal	CCAM\Senin	Finished	25.06.2006 4:08:39	

Select a job to view its tasks						
Name	Status	Task Id	Command Line	Processors	End Time	Failure Message

- В открывшемся окне менеджера заданий выберите пункт меню **File->Submit Job** для постановки нового задания в очередь,
- В окне постановки задания в очередь введите имя задания (поле “**Job Name**”). Перейдите на вкладку “**Processors**”

Submit Job Parallel Pi computing

General | Processors | Tasks | Licenses | Advanced

Parallel Pi computing

Job Name: Parallel Pi computing

Project Name:

Priority: Normal

Submitted By: N/A

Submitted on: N/A

Status: Not Submitted

Save As Template... Submit Cancel

- На вкладке “**Processors**” введите минимальное и максимальное числа процессоров, необходимых для выполнения задания (например, 10 и 20 соответственно). Перейдите на вкладку “**Tasks**” для добавления в задание новых задач

Submit Job Parallel Pi computing

General **Processors** Tasks Licenses Advanced

Processors required for this job

Processors available in this cluster: 60

Minimum required: 10

Maximum required: 20

Estimate run time for this job

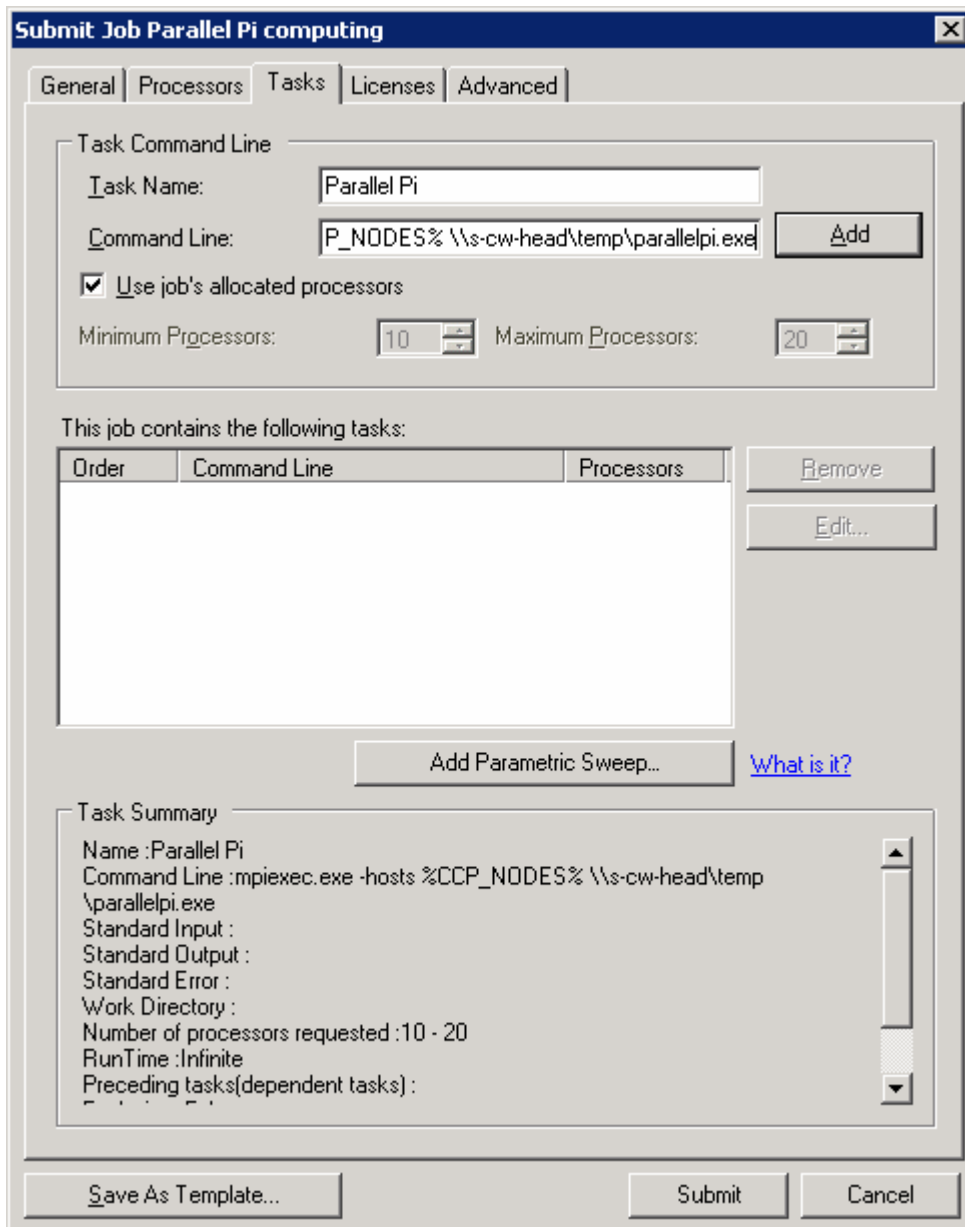
Days: 0 Hours: 1 Minutes: 0

Run job until end of run time or until canceled.

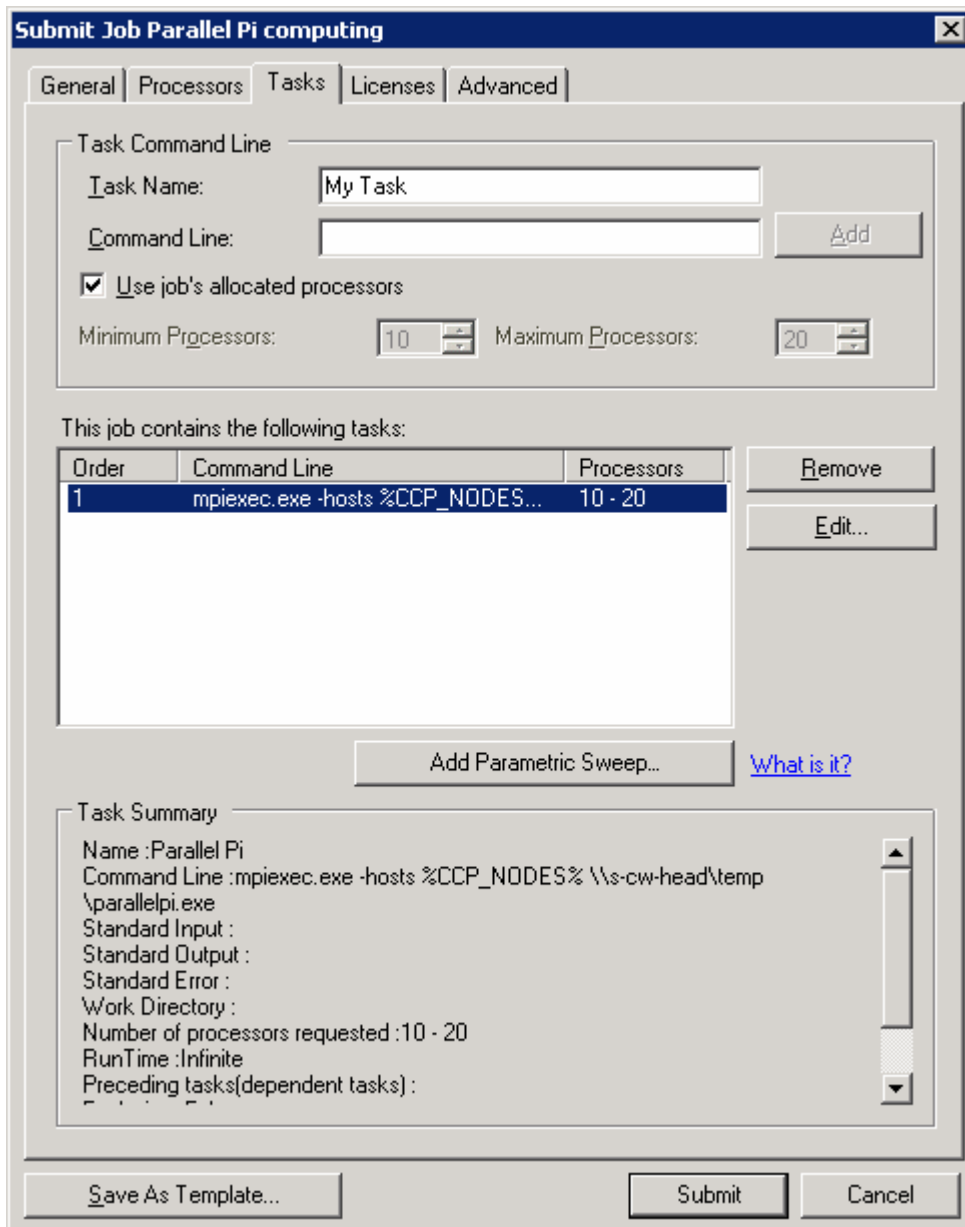
This option lets you run extra tasks after running all tasks already listed in the job if there is time left.

Save As Template... Submit Cancel

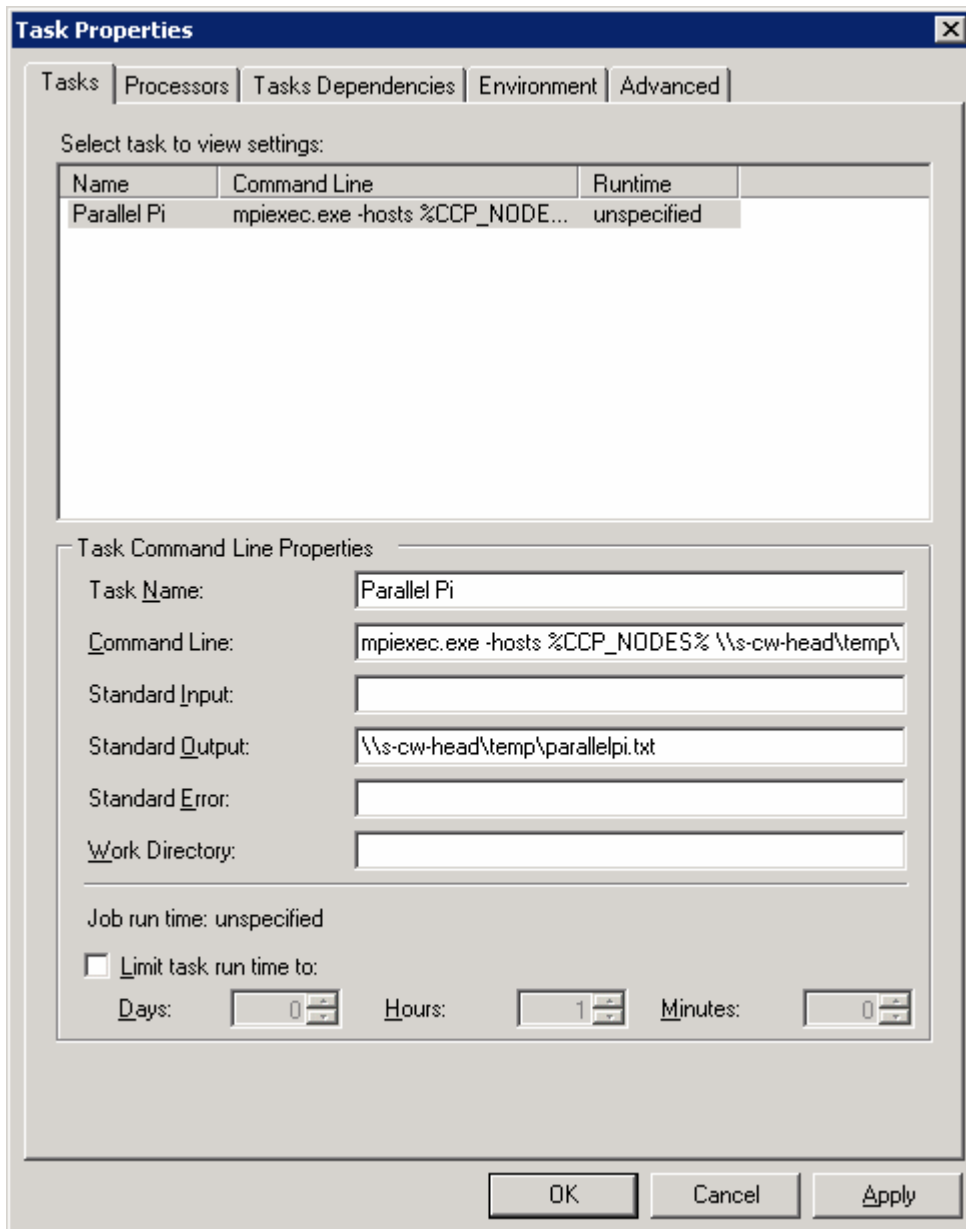
- Введите имя задачи (поле “**Task Name**”) и команду, которую необходимо выполнить (поле “**Command Line**”). Запуск задач, разработанных для MS MPI, необходимо осуществлять с использованием специальной утилиты **mpiexec.exe**, которая принимает в качестве параметров имя параллельной программы, список узлов, на которых произойдет запуск, и параметры запускаемой программы. Список узлов задается параметром “**-hosts**”. При этом в случае, если узлы были выделены планировщиком автоматически, список узлов будет содержать переменная окружения **CCP_NODES**, значение которой и следует передавать в качестве параметра утилите. Пример команды для запуска параллельной программы: “**mpiexec.exe -hosts %CCP_NODES \\s-cw-head\temp\parallempi.exe**”. Нажмите кнопку “**Add**” для добавления задачи в задание



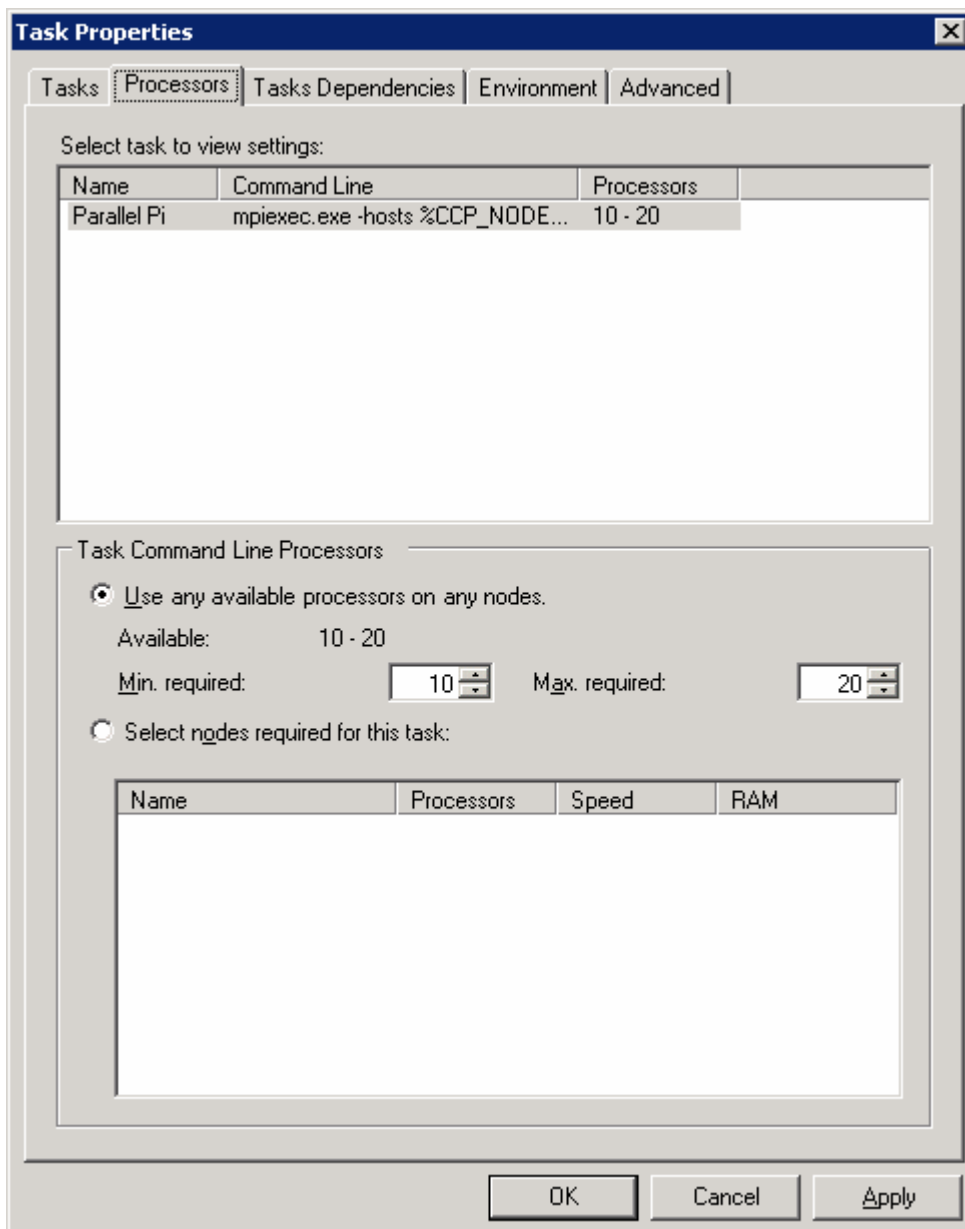
- Добавленная задача появится в списке задач текущего задания (список “**This job contains the following tasks**”). Выделите ее в списке и нажмите кнопку “**Edit**” для редактирования дополнительных параметров задачи



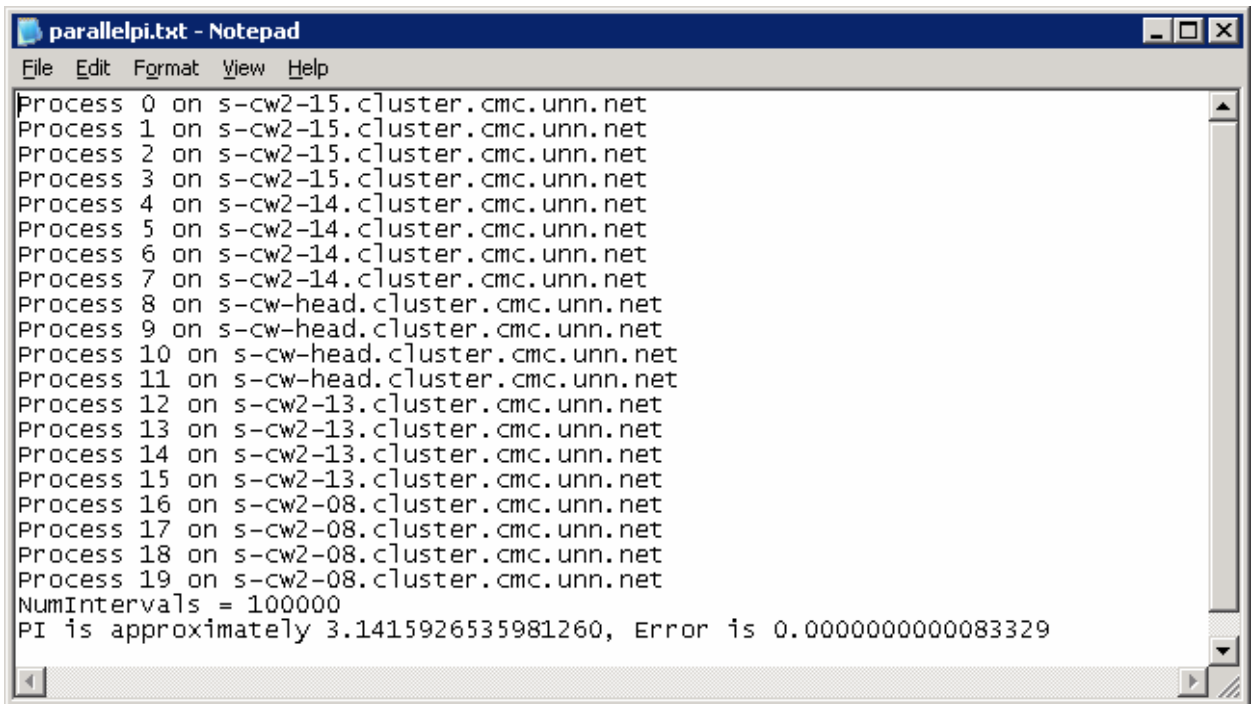
- В открывшемся окне введите путь до файла, в который будет перенаправлен стандартный поток вывода консольного приложения (поле “**Standard Output**”). Выберите вкладку “**Processors**”



- На вкладке “**Processors**” в верхнем списке (“**Select task to view settings**”) выделите задачу, настройки которой Вы хотите изменить, и укажите минимальное и максимальное числа процессоров для выбранной задачи (поля “**Min. required**” и “**Max. required**”). Нажмите “**OK**” для сохранения внесенных изменений и возвращения к настройкам задания



- Нажмите кнопку “**Submit**” для добавления задания в очередь. В окне запроса пароля введите имя и пароль пользователя, имеющего право запуска задач на кластере, и нажмите “**OK**”. Задание появится в очереди **Job Manager**. По окончании работы его состояние изменится на “**Finished**”. В файле, указанном в настройках задачи для перенаправления стандартного потока вывода, содержится результат работы программы



```
parallelpi.txt - Notepad
File Edit Format View Help
Process 0 on s-cw2-15.cluster.cmc.unn.net
Process 1 on s-cw2-15.cluster.cmc.unn.net
Process 2 on s-cw2-15.cluster.cmc.unn.net
Process 3 on s-cw2-15.cluster.cmc.unn.net
Process 4 on s-cw2-14.cluster.cmc.unn.net
Process 5 on s-cw2-14.cluster.cmc.unn.net
Process 6 on s-cw2-14.cluster.cmc.unn.net
Process 7 on s-cw2-14.cluster.cmc.unn.net
Process 8 on s-cw-head.cluster.cmc.unn.net
Process 9 on s-cw-head.cluster.cmc.unn.net
Process 10 on s-cw-head.cluster.cmc.unn.net
Process 11 on s-cw-head.cluster.cmc.unn.net
Process 12 on s-cw2-13.cluster.cmc.unn.net
Process 13 on s-cw2-13.cluster.cmc.unn.net
Process 14 on s-cw2-13.cluster.cmc.unn.net
Process 15 on s-cw2-13.cluster.cmc.unn.net
Process 16 on s-cw2-08.cluster.cmc.unn.net
Process 17 on s-cw2-08.cluster.cmc.unn.net
Process 18 on s-cw2-08.cluster.cmc.unn.net
Process 19 on s-cw2-08.cluster.cmc.unn.net
NumIntervals = 100000
PI is approximately 3.1415926535981260, Error is 0.0000000000083329
```

Упражнение 4 – Запуск множества задач

В данном упражнении идет речь о запуске параметрического множества задач (parametric sweep) внутри одного задания. Под параметрическим множеством задач понимается серия запусков одной и той же программы с разными параметрами. В качестве примера можно привести запуск серии из нескольких сотен экспериментов по вычислению числа Пи для исследования скорости сходимости метода к точному решению. В качестве примера программы для данного упражнения мы будем использовать программу параллельного вычисления числа Пи:

- Откройте **Computer Cluster Job Manager (Start->All Programs->Microsoft Compute Cluster Pack->Compute Cluster Job Manager)** для создания параметрического множества задач

Job Queue at s-cw-head

File View Help Show: All Jobs

ID	Name	Priority	Submitted By	Status	Submit Time	Pending Reason
1	hostname	Normal	CCAM\Senin	Finished	21.06.2006 4:10:14	
2	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:19:50	
3	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:24:24	
4	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:25:23	
5	imb	Normal	CCAM\Senin	Finished	21.06.2006 5:13:20	
6	Serial Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 1:59:22	
7	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 2:00:46	
8	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 3:50:22	
9	Serial Pi Calculation	Normal	CCAM\Senin	Finished	25.06.2006 4:08:39	
10	Parallel Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 22:13:00	
11	Parallel Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 22:15:36	
12	Parallel Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 22:16:49	
13	Parallel Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 22:17:33	
14	Parallel Pi computing	Normal	CCAM\Senin	Finished	26.06.2006 0:29:12	
15	Parallel Pi computing	Normal	CCAM\Senin	Finished	26.06.2006 0:31:30	

Select a job to view its tasks

Name	Status	Task Id	Command Line	Processors	End Time	Failure Message

Ready

- В открывшемся окне менеджера заданий выберите пункт меню **File->Submit Job** для постановки нового задания в очередь,
- В окне постановки задания в очередь введите имя задания (поле “**Job Name**”). Перейдите на вкладку “**Processors**”

Submit Job Parallel Pi parametric sweep

General Processors Tasks Licenses Advanced

Parallel Pi parametric sweep

Job Name: Parallel Pi parametric sweep

Project Name:

Priority: Normal

Submitted By: N/A

Submitted on: N/A

Status: Not Submitted

Save As Template... Submit Cancel

- На вкладке “**Processors**” введите минимальное и максимальное числа процессоров, необходимых для выполнения задания (например, 10 и 20 соответственно). Перейдите на вкладку “**Tasks**” и нажмите кнопку “**Add parametric Sweep**” на вкладке для добавления в задание множества новых задач

Submit Job Parallel Pi parametric sweep

General Processors Tasks Licenses Advanced

Processors required for this job

Processors available in this cluster: 60

Minimum required: 10

Maximum required: 20

Estimate run time for this job

Days: 0 Hours: 1 Minutes: 0

Run job until end of run time or until canceled.

This option lets you run extra tasks after running all tasks already listed in the job if there is time left.

Save As Template... Submit Cancel

- В окне добавления параметрического множества задач введите имя, которое будет присвоено каждой новой задаче (поле “Name”). Введите команду для задач, используя звездочку (“*”) как параметр аргументов командной строки. Символ “*” для каждой конкретной команды будет заменен целым числом, пределы изменения которого указываются в полях “Index Start” и “Index End”. Для нашей задачи индекс (число отрезков численного интегрирования) может изменяться, например, от 50 до 100. Таким образом, команда может быть следующей: “**mpiexec.exe -hosts %CCP_NODES% \\s-cw-head\temp\parallempi.exe ***”. Укажите файлы, в которые будет перенаправлен стандартный поток вывода, используя “*” в качестве параметра. Например: “**\\s-cw-head\temp\parallempi*.txt**”. Нажмите “OK” для добавления множества задач в задание

Add Parametric Sweep

Create Task

Name: Parallel Pi

Command Line (Use * to represent index if desired):
NODES% \\s-cw-head\temp\parallelpi.exe *

Index Start: 50 Index End: 100 Index Skip: 0

Use Standard Input (Use * to represent index if desired)
Input File Location

Collect Standard Output (Use * to represent index if desired)
Output File Location: \\s-cw-head\temp\parallelpi*.txt

Collect Standard Error (Use * to represent index if desired)
Error File Location

Assign Work Directory
Work Directory

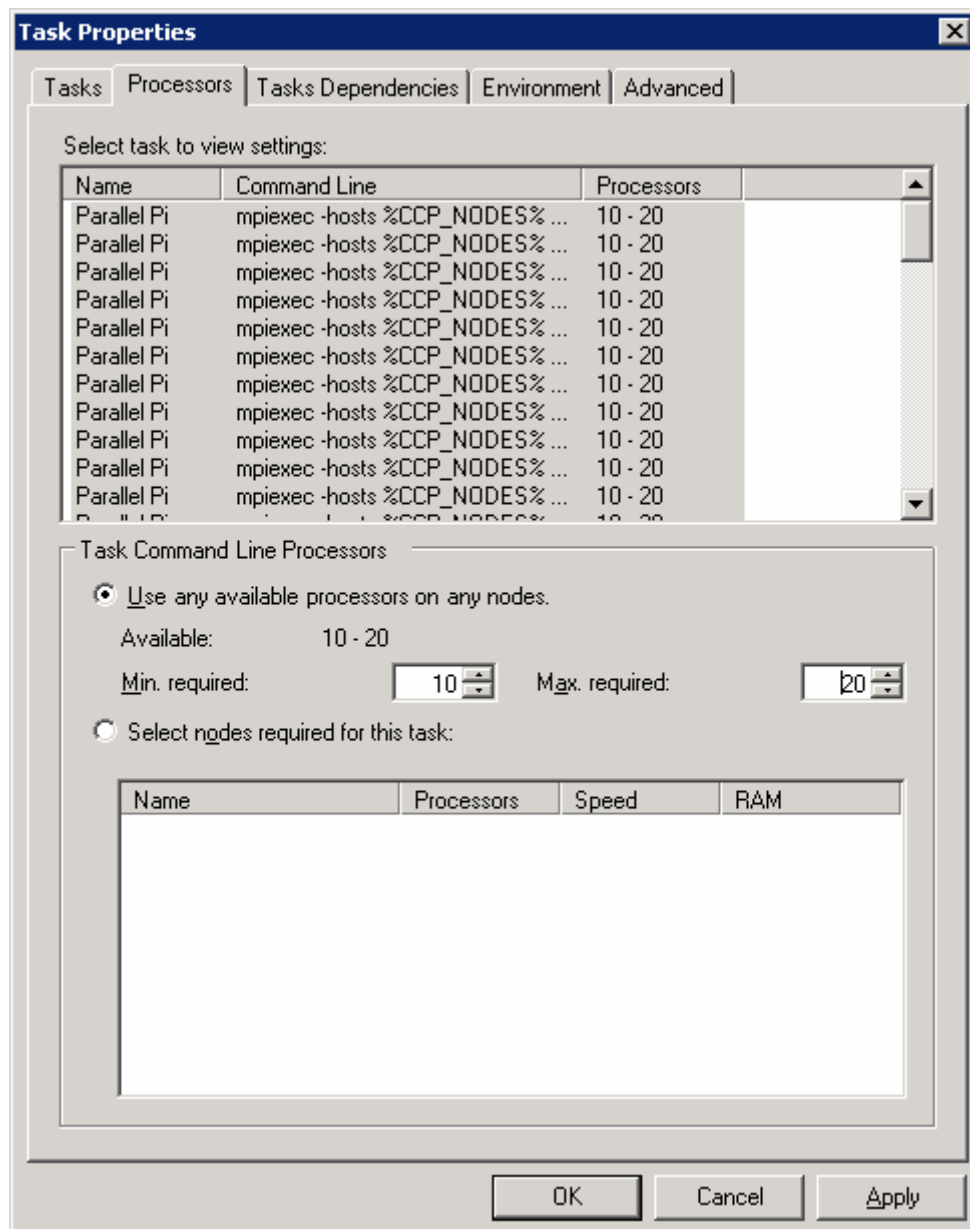
Preview Task

Command Line	Standard Output
Parallel Pi	
mpirun -hosts %CCP_NODES% \\s-cw-head\temp\parallelpi.exe 50	\\s-cw-head\ter
mpirun -hosts %CCP_NODES% \\s-cw-head\temp\parallelpi.exe 51	\\s-cw-head\ter
mpirun -hosts %CCP_NODES% \\s-cw-head\temp\parallelpi.exe 52	\\s-cw-head\ter

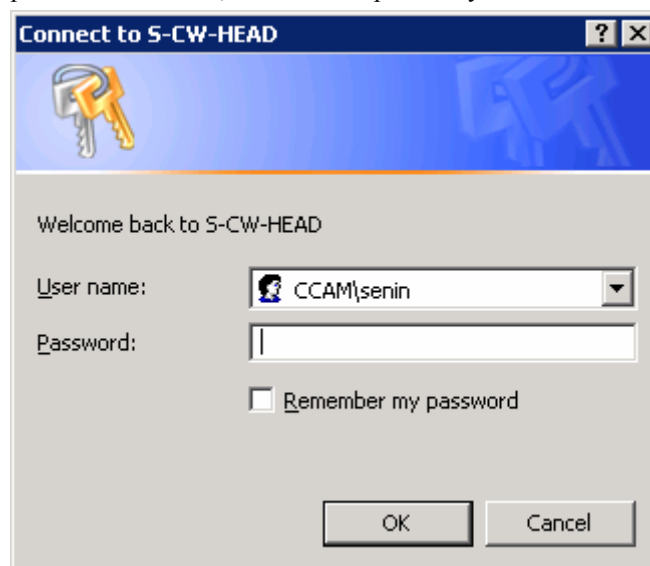
Extension Length: 1

Add Cancel

- В окне настройки задания выделите все входящие в него задачи (для выделения нескольких задач используйте клавишу “Shift”), нажмите кнопку “Edit” для указания числа процессоров, требуемых для задач. В открывшемся окне перейдите на вкладку “Processors”, выберите пункт “Use any available processors on any nodes” и укажите, например, в качестве минимального числа процессоров 10, в качестве максимального – 20. Нажмите “OK”. В открывшемся окне нажмите кнопку “Submit” для добавления задания в очередь



- Введите имя и пароль пользователя, имеющего право запуска задач на кластере, и нажмите “OK”



- В открывшемся окне **Job Manager** появится новое задание. Выделив его, Вы получите возможность наблюдать за ходом выполнения его задач в нижнем списке. Когда задание выполнится, его состояние изменится на **“Finished”**

The screenshot shows the 'Job Queue at s-cw-head' window. The main table lists jobs with columns: ID, Name, Priority, Submitted By, Status, Submit Time, and Pending Reason. Job 16, 'Parallel Pi parametric sweep', is highlighted and has a status of 'Running'.

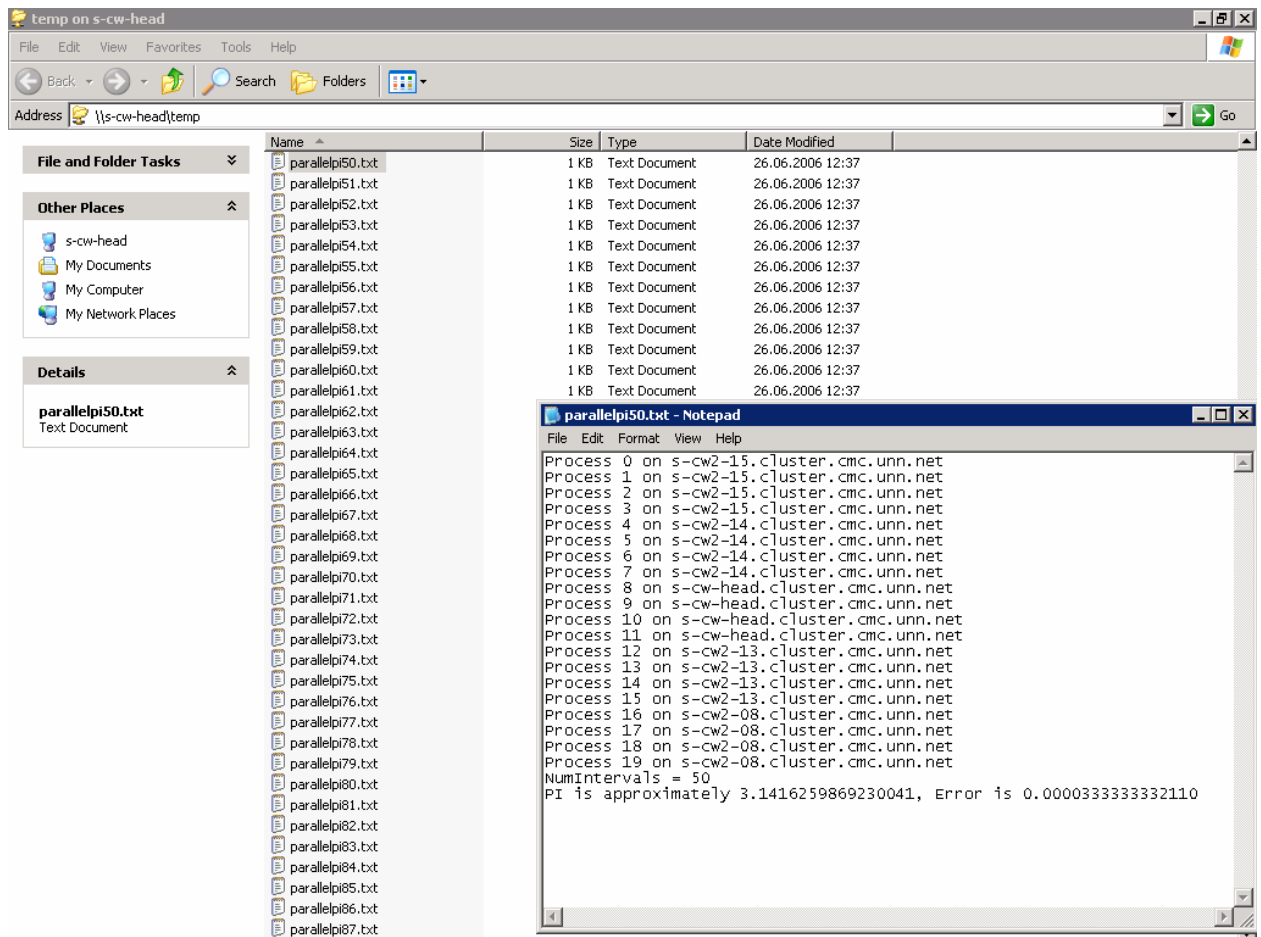
ID	Name	Priority	Submitted By	Status	Submit Time	Pending Reason
1	hostname	Normal	CCAM\Senin	Finished	21.06.2006 4:10:14	
2	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:19:50	
3	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:24:24	
4	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:25:23	
5	imb	Normal	CCAM\Senin	Finished	21.06.2006 5:13:20	
6	Serial Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 1:59:22	
7	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 2:00:46	
8	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 3:50:22	
9	Serial Pi Calculation	Normal	CCAM\Senin	Finished	25.06.2006 4:08:39	
10	Parallel Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 22:13:00	
11	Parallel Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 22:15:36	
12	Parallel Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 22:16:49	
13	Parallel Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 22:17:33	
14	Parallel Pi computing	Normal	CCAM\Senin	Finished	26.06.2006 0:29:12	
15	Parallel Pi computing	Normal	CCAM\Senin	Finished	26.06.2006 0:31:30	
16	Parallel Pi parametric sweep	Normal	CCAM\Senin	Running	26.06.2006 1:37:01	

Below the main table is a section titled 'Tasks for ParallelPi parametric sweep' with columns: Name, Status, Task Id, Command Line, Processors, End Time, and Failure Message. It shows 13 tasks, with task 2 currently 'Running' and others 'Queued'.

Name	Status	Task Id	Command Line	Processors	End Time	Failure Message
Parallel Pi	Finished	1	mpiexec -hosts %CCP_NO...	10 - 20	26.06.2006 ...	
Parallel Pi	Running	2	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	3	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	4	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	5	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	6	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	7	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	8	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	9	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	10	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	11	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	12	mpiexec -hosts %CCP_NO...	10 - 20	N/A	
Parallel Pi	Queued	13	mpiexec -hosts %CCP_NO...	10 - 20	N/A	

A tooltip from 'Compute Cluster Job Manager' is visible, stating: 'A new job 16 is submitted. Please click here for detailed information.'

- Вы можете просмотреть результаты выполнения заданий, в файлах, которые Вы указали для сохранения перехваченного потока вывода



Упражнение 5 – Запуск потока задач

Поток задач используется тогда, когда для выполнения некоторой задачи в составе задания необходимы результаты работы других задач, что выдвигает требования к порядку их выполнения. Эти требования удобно задавать в виде ациклического ориентированного графа, в котором каждая вершина представляет собой задачу, а стрелка выражает зависимость вершины-потомка от вершины-родителя. В этом случае порядок выполнения задач определяется следующим простым правилом: ни одна задача не начнет выполняться до тех пор, пока не будут выполнены все задачи, соответствующие на графе зависимостей ее родителям.

Примером может служить следующий граф зависимостей задач.



Реализуем граф зависимостей в CCS 2003:

- Откройте **Computer Cluster Job Manager (Start->All Programs->Microsoft Compute Cluster Pack->Compute Cluster Job Manager)**


ID	Name	Priority	Submitted By	Status	Submit Time	Pending Reason
1	hostname	Normal	CCAM\Senin	Finished	21.06.2006 4:10:14	
2	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:19:50	
3	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:24:24	
4	imb	Normal	CCAM\Senin	Failed	21.06.2006 4:25:23	
5	imb	Normal	CCAM\Senin	Finished	21.06.2006 5:13:20	
6	Serial Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 1:59:22	
7	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 2:00:46	
8	Serial Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 3:50:22	
9	Serial Pi Calculation	Normal	CCAM\Senin	Finished	25.06.2006 4:08:39	
10	Parallel Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 22:13:00	
11	Parallel Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 22:15:36	
12	Parallel Pi computing	Normal	CCAM\Senin	Failed	25.06.2006 22:16:49	
13	Parallel Pi computing	Normal	CCAM\Senin	Finished	25.06.2006 22:17:33	
14	Parallel Pi computing	Normal	CCAM\Senin	Finished	26.06.2006 0:29:12	
15	Parallel Pi computing	Normal	CCAM\Senin	Finished	26.06.2006 0:31:30	
16	Parallel Pi parametric sweep	Normal	CCAM\Senin	Finished	26.06.2006 1:37:01	

Name	Status	Task Id	Command Line	Processors	End Time	Failure Message

- В открывшемся окне менеджера заданий выберите пункт меню **File->Submit Job**,
- В окне постановки задания в очередь введите имя задания (поле **“Job Name”**). Перейдите на вкладку **“Processors”**

Submit Job Example of task flow [X]

General | Processors | Tasks | Licenses | Advanced

 Example of task flow

Job Name:

Project Name:

Priority: ▼

Submitted By: N/A

Submitted on: N/A

Status: Not Submitted

- На вкладке “**Processors**” введите минимальное и максимальное числа процессоров, необходимых для выполнения задания (например, 5 и 10 соответственно). Перейдите на вкладку “**Tasks**” для добавления в задание новых задач

Submit Job Example of task flow

General Processors Tasks Licenses Advanced

Processors required for this job

Processors available in this cluster: 60

Minimum required: 5

Maximum required: 10

Estimate run time for this job

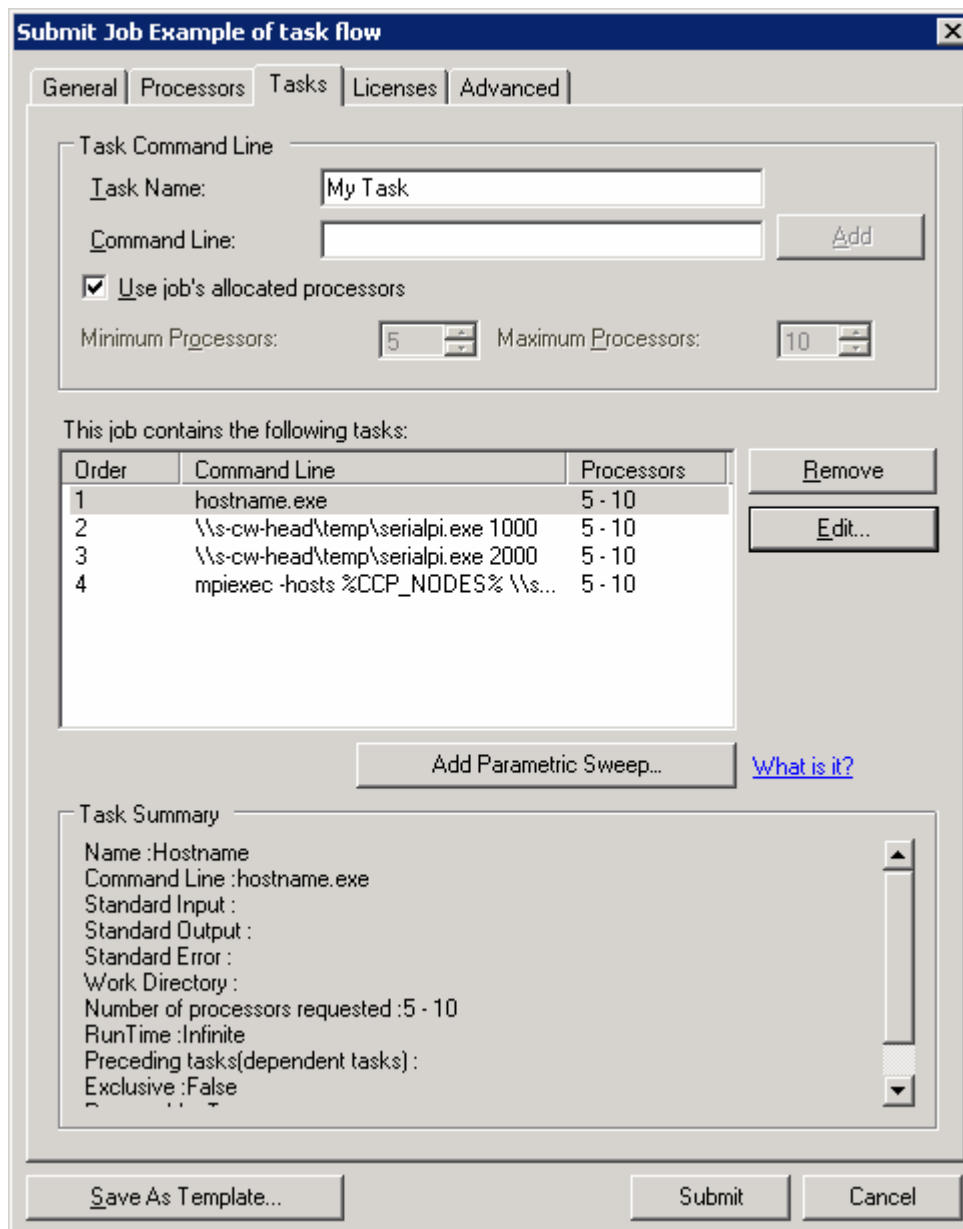
Days: 0 Hours: 1 Minutes: 0

Run job until end of run time or until canceled.

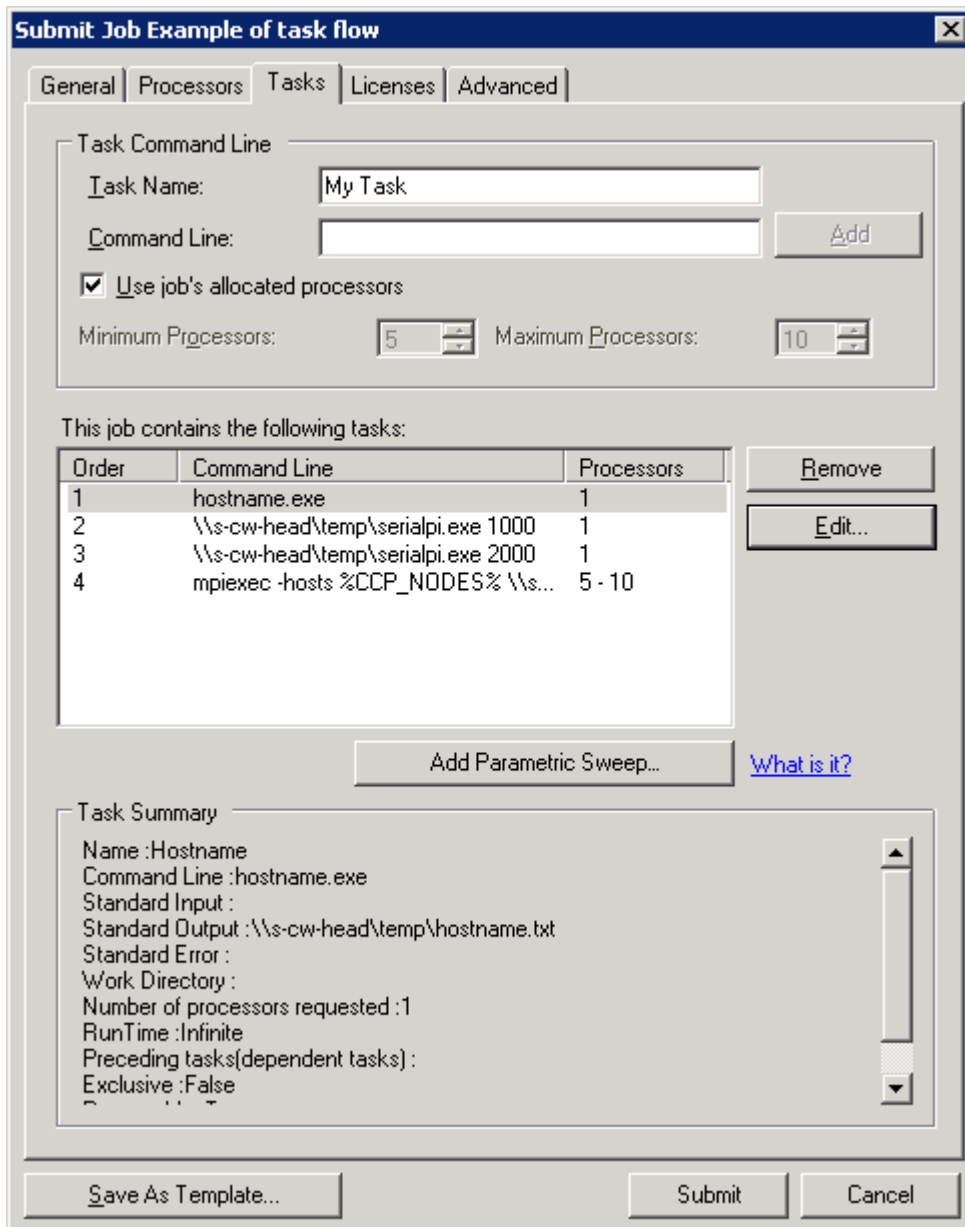
This option lets you run extra tasks after running all tasks already listed in the job if there is time left.

Save As Template... Submit Cancel

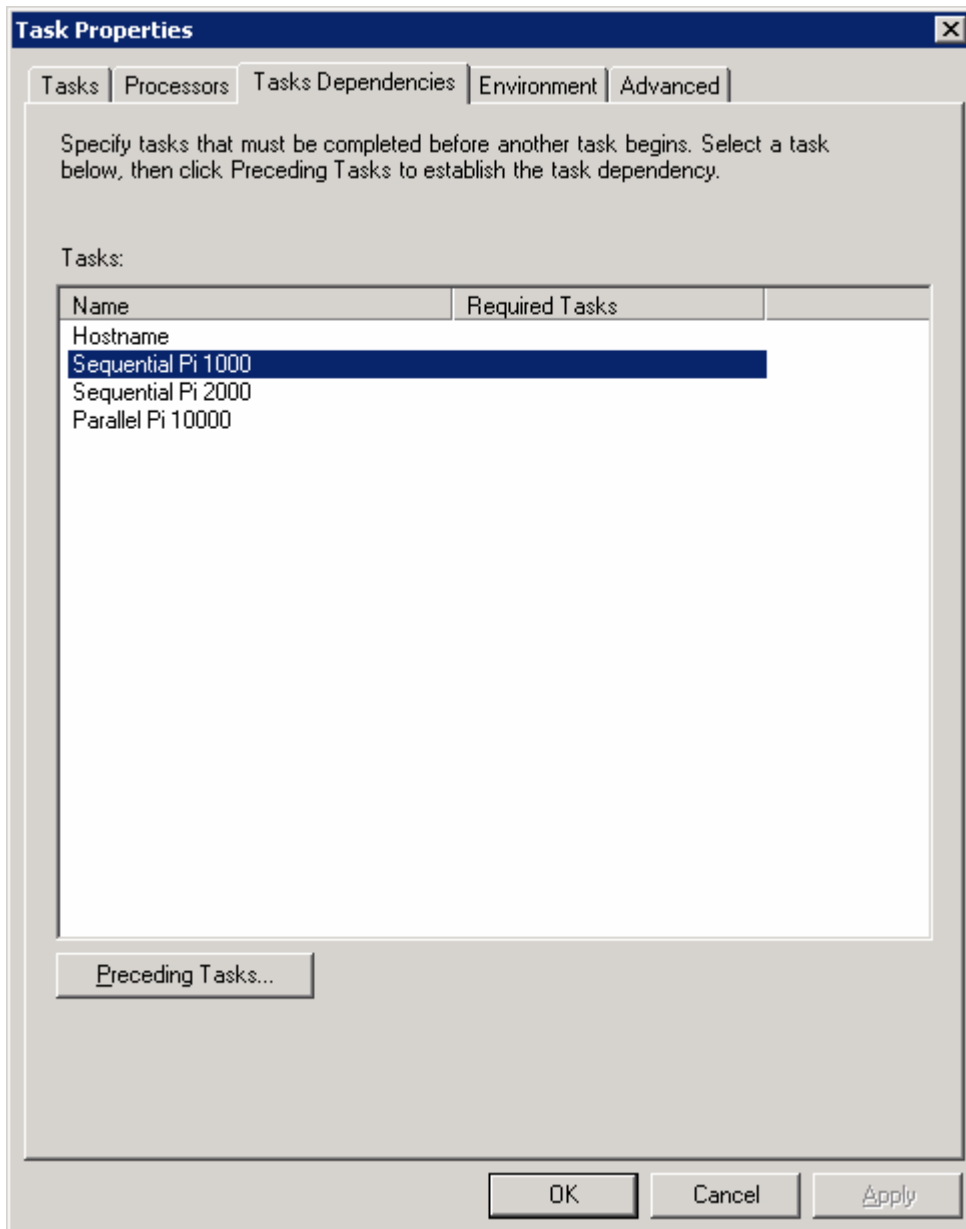
- Последовательно добавьте в задание следующие 4 задачи:
 - Задачу с именем **“Hostname”** и командой **“hostname.exe”**,
 - Задачу с именем **“Sequential Pi 1000”** и командой **“\\s-cw-head\temp\serialpi.exe 1000”** (замените путь до исполняемого файла программы на существующий),
 - Задачу с именем **“Sequential Pi 2000”** и командой **“\\s-cw-head\temp\serialpi.exe 2000”** (замените путь до исполняемого файла программы на существующий),
 - Задачу с именем **“Parallel Pi 10000”** и командой **“mpixec -hosts %CCP_NODES% \\s-cw-head\temp\parallepi.exe 10000”** (замените путь до исполняемого файла программы на существующий)



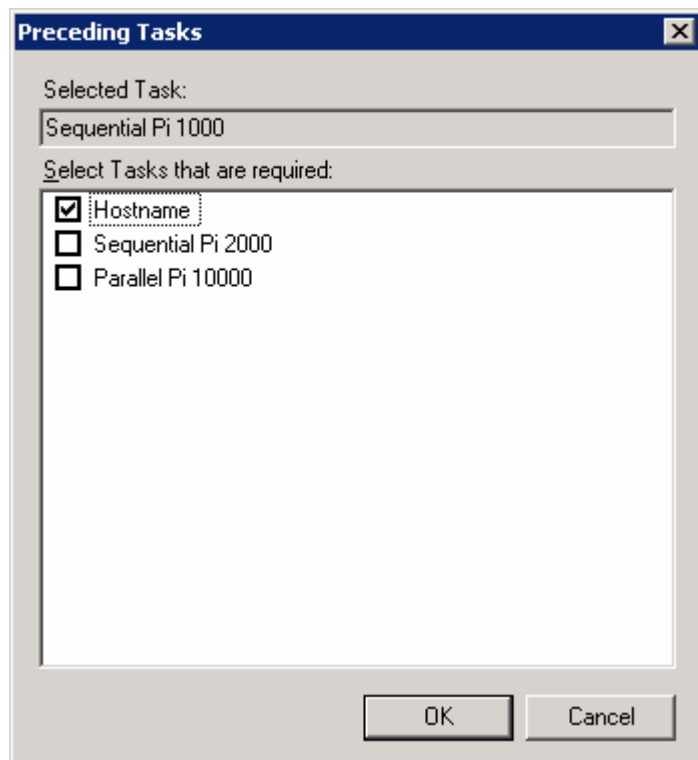
- Установите дополнительные свойства задач:
 - Для задач “**Hostname**”, “**Sequential Pi 1000**” и “**Sequential Pi 2000**” установите максимальное необходимое число процессоров в 1, установите для каждой из 3 задач файл для переправления стандартного потока вывода,
 - Для задачи “**Parallel Pi 10000**” установите минимальное и максимальное числа процессоров в 5 и 10 соответственно, установите файл для переправления стандартного потока вывода,
- Перейдите на вкладку “**Tasks Dependencies**” свойств задач (для перехода к свойствам задач нажмите кнопку “**Edit**” на вкладке “**Tasks**” окна “**Submit Job**”)



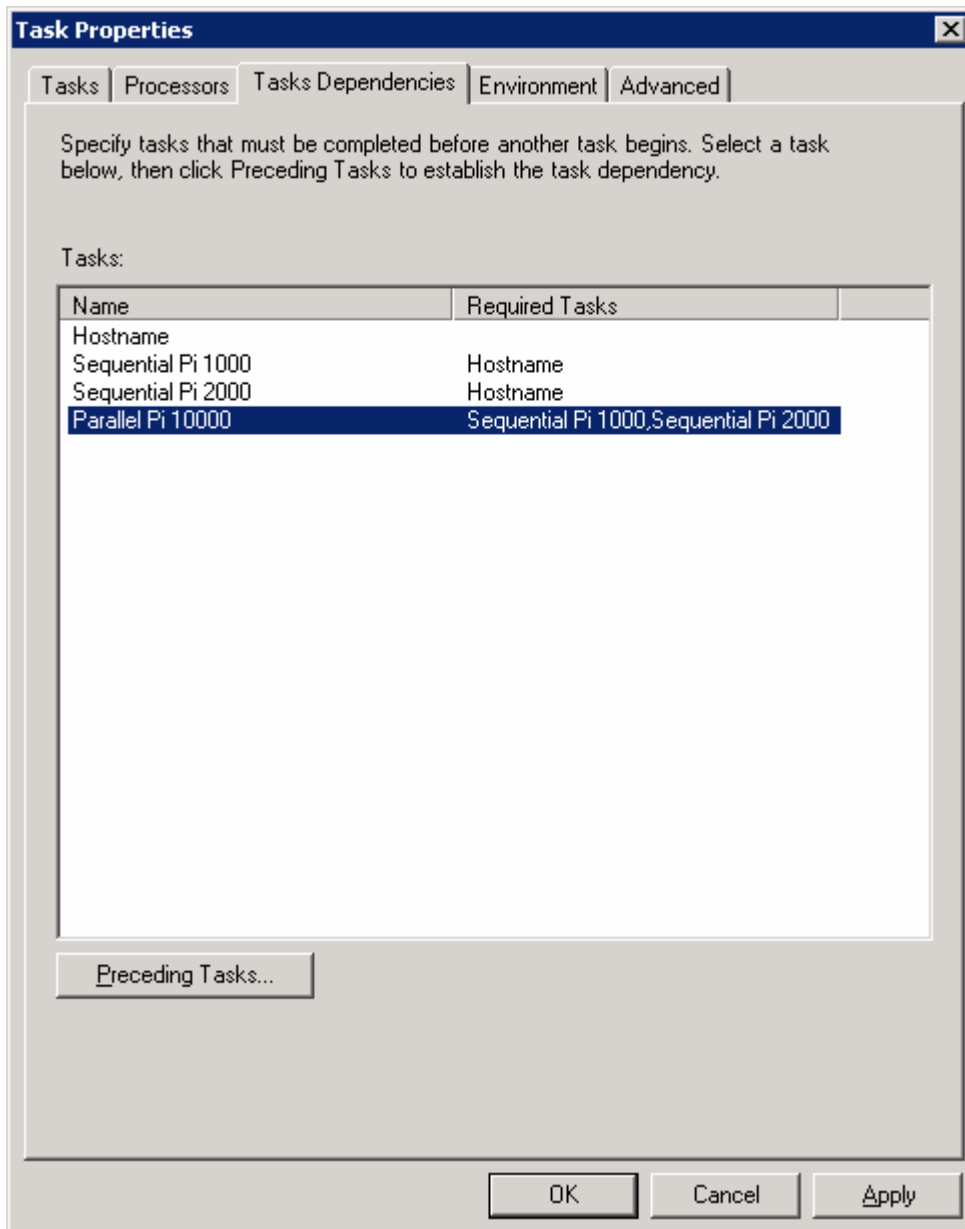
- Выделите задачу “**Sequential Pi 1000**” и нажмите кнопку “**Preceding Tasks**” для указания задач, от которых зависит рассматриваемая задача



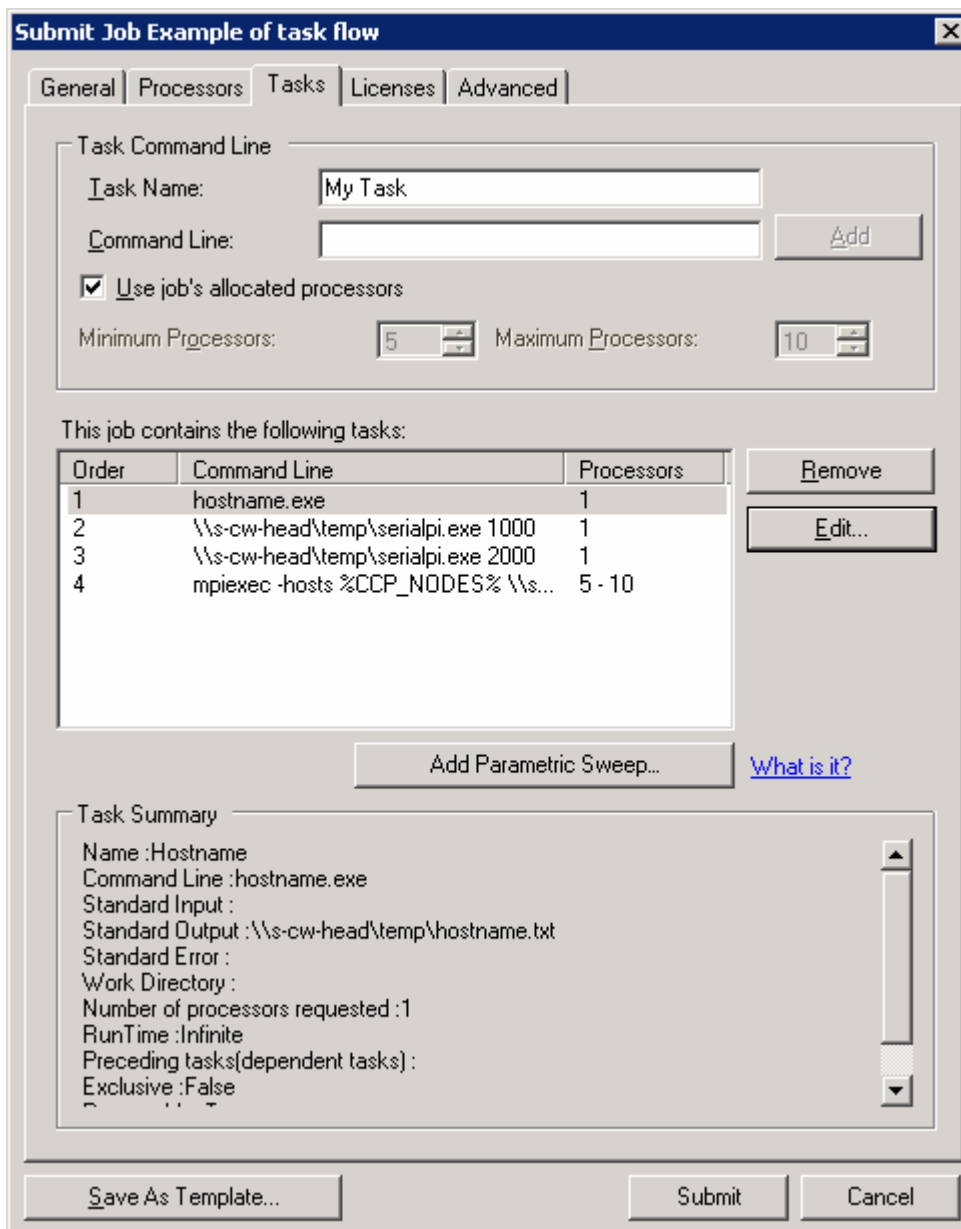
- В открывшемся окне поставьте флажок около задачи “**Hostname**”. Нажмите “**OK**”



- Установить для задачи **“Sequential Pi 2000”** зависимость от **“Hostname”**. Установите для задачи **“Parallel Pi 10000”** зависимость от задач **“Sequential Pi 1000”** и **“Sequential Pi 2000”**. Нажмите **“OK”** для сохранения внесенных изменений



- В окне “**Submit Job**” нажмите кнопку “**Submit**” для добавления задания в очередь



- Введите имя и пароль пользователя с правами запуска заданий на кластере,
- Планировщик CCS 2003 сначала запустит задачу “**Hostname**”, затем параллельно задачи “**Sequential Pi 1000**” и “**Sequential Pi 2000**” и только потом задачу “**Parallel Pi 10000**”.

Дополнительное упражнение. Задача определения характеристик сети передачи данных

Особенностью разработки программ для кластерных систем является необходимость учитывать не только характеристики отдельных компьютеров (прежде всего это производительность процессора и скорость памяти), но и характеристики сети передачи данных между ними. Чаще всего эти характеристики используются для построения теоретических оценок времени работы алгоритмов, что позволяет предсказывать время работы программ в зависимости от размера входных данных. Получение характеристик сети – отдельная задача, решаемая запуском специальных тестовых программ на конкретном имеющемся оборудовании. Необходимость проведения тестов для каждого конкретного кластера объясняется тем, что данные, предоставляемые поставщиком аппаратного обеспечения, могут сильно отличаться в зависимости от используемого программного обеспечения, настроек кластера или особенностей взаимодействия оборудования разных моделей и изготовителей.

Описание характеристик, определяющих производительность сети

Основными характеристиками, определяющими производительность сети, являются **латентность** и **пропускная способность**. Латентностью (задержкой) называется время, затрачиваемое аппаратной и

программной частью на обработку запроса отправки сетевого сообщения. То есть это время с момента поступления команды на пересылку информации до начала ее непосредственной передачи. Обычно латентность указывается в **микросекундах (мкс)**.

Пропускной способностью сети называется количество информации, передаваемое между узлами сети за единицу времени. Обычно пропускная способность сети указывается в **мегабайтах в секунду (Мбайт/сек)** или **мегабитах в секунду (Мбит/сек)**.

Общая характеристика алгоритма

Идея алгоритма определения характеристик сети, используемого в тестах данной работы, состоит в последовательной пересылке между 2 узлами сообщений различной длины, используя средства установленной реализации MPI, и измерения времени, затрачиваемого на пересылку. Имея эти данные, пропускную способность можно определить, поделив длину переданного сообщения на затраченное на передачу время. Для минимизации погрешности передачу повторяют N раз и усредняют результат. При этом оценка пропускной способности обычно увеличивается с ростом длины сообщения, стремясь к некоторому предельному значению. Обычно именно предельное значение (или значение, полученное при пересылке большого сообщения) целесообразно использовать как оценку пропускной способности.

За латентность обычно принимается время, затрачиваемое для пересылки сообщений нулевой длины.

В данной лабораторной работе рассматриваются 2 тестовые программы: **Intel MPI Benchmark (IMB)** и **набор тестов, разработанный в НИВЦ МГУ**.

Компиляция программы

Последнюю версию тестового пакета IMB в составе Intel Cluster Tools Вы можете скачать на сайте Intel по адресу <http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219848.htm> . Для того, чтобы скомпилировать IMB для ОС Windows, Вам придется самостоятельно создать проект в Microsoft Visual Studio 2005 аналогично проектам, создание которых было описано в данной работе. Либо Вы можете воспользоваться заготовкой проекта, которая входит в состав лабораторной работы (папка **IMB_2_3**).

Скачать тесты НИВЦ МГУ можно по адресу <http://parallel.ru/ftp/tests/mpi-bench-suite.zip> . Для компиляции тестов в ОС Windows Вам также придется самостоятельно создать проект в Microsoft Visual Studio 2005, либо воспользоваться заготовкой проекта, входящей в состав лабораторной работы (папка **MGU_tests**).


Выполнение программы

Тесты необходимо запускать на 2 узлах сети по одному процессу на каждом узле. Таким образом, для запуска тестов в CCS 2003 необходимо указать в качестве требований для задания суммарное число процессоров на 2 узлах и выбрать эти узлы вручную:

- Откройте **Job Manager**. Выполните команду меню **File->Submit Job...** Укажите имя задачи и перейдите на вкладку **“Processors”**

Submit Job Network test [X]

General | Processors | Tasks | Licenses | Advanced

 Network test

Job Name:

Project Name:

Priority: ▼

Submitted By: N/A

Submitted on: N/A

Status: Not Submitted

- Укажите в качестве требования к заданию суммарное число процессоров на 2 вычислительных узлах, на которых Вы хотите произвести запуск (например, 8 в случае использования 2 4х процессорных узлов). Перейдите на вкладку “**Tasks**”

Submit Job Network test

General Processors Tasks Licenses Advanced

Processors required for this job

Processors available in this cluster: 60

Minimum required: 8

Maximum required: 8

Estimate run time for this job

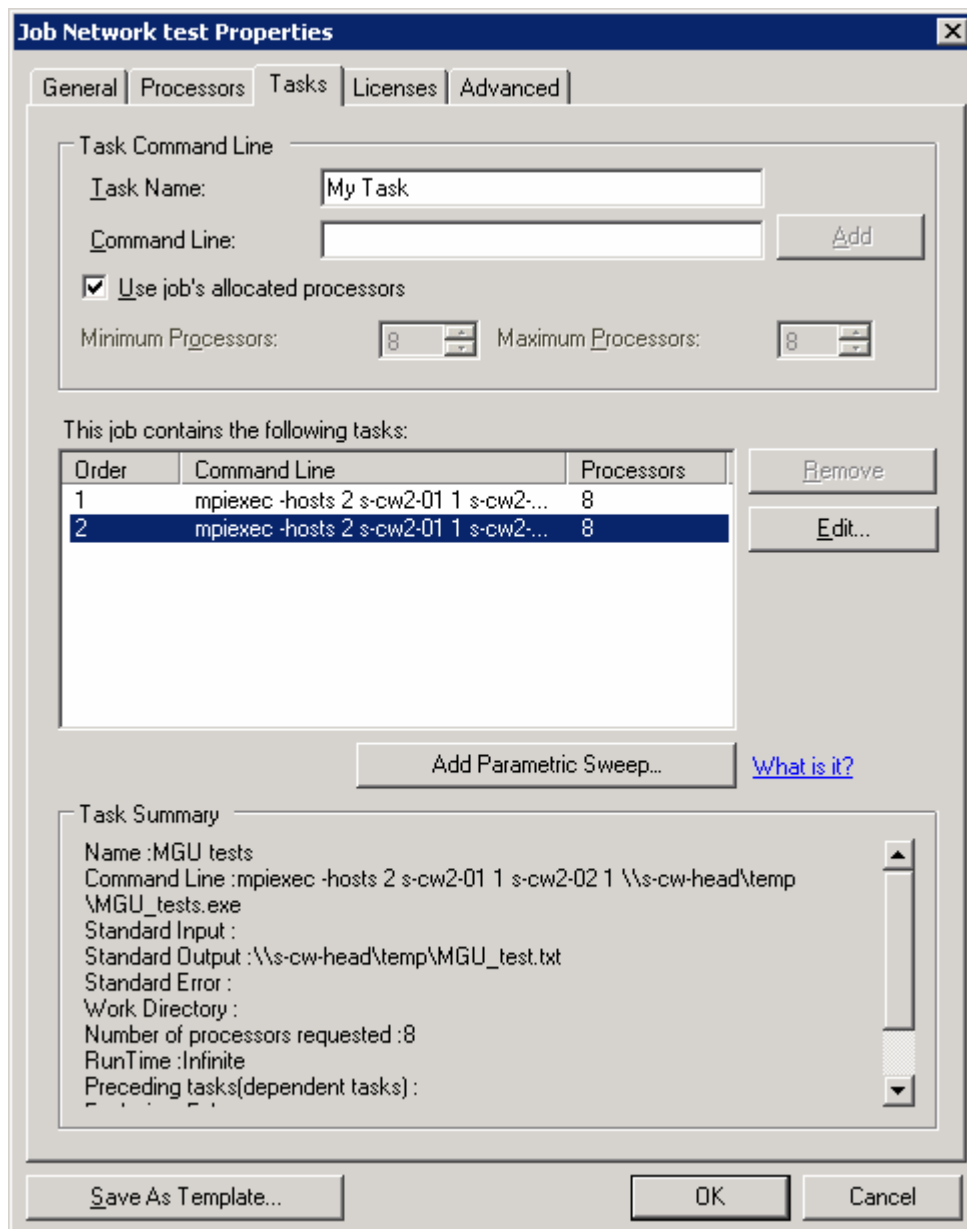
Days: 0 Hours: 1 Minutes: 0

Run job until end of run time or until canceled.

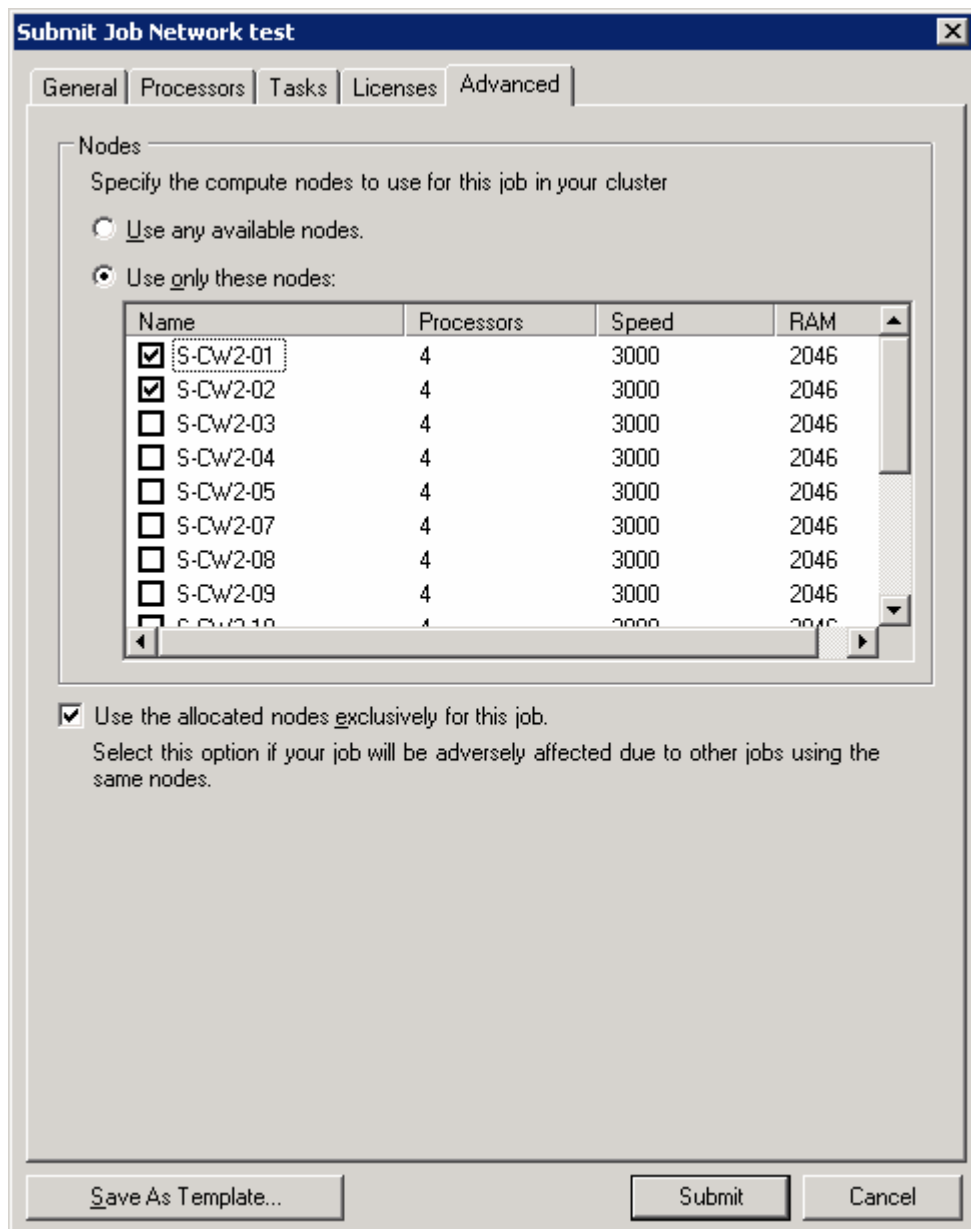
This option lets you run extra tasks after running all tasks already listed in the job if there is time left.

Save As Template... Submit Cancel

- Добавьте в задание две задачи: “`mpixec -hosts 2 s-cw2-01 1 s-cw2-02 1 \\s-cw-head\temp\imb.exe`”, “`mpixec -hosts 2 s-cw2-01 1 s-cw2-02 1 \\s-cw-head\temp\MGU_tests.exe`” (не забудьте заменить параметры команды на соответствующие Вашему случаю). В командах должны быть указаны именно те узлы, на которых планируется запуск. Параметр “hosts” имеет следующий формат: “**n node1 m1 node2 m2 ... noden mn**”. Использовать переменную окружения `CCP_NODES` в данном случае нельзя, так как на каждом узле должен быть запущен только 1 процесс. Укажите для задач файлы для перенаправления стандартного потока вывода. Перейдите на вкладку “Advanced”



- Выберите пункт “Use only these nodes” и поставьте флажки около тех узлов, которые были указаны в командах на предыдущем шаге. Нажмите кнопку “Submit” для добавления задания в очередь и введите имя и пароль пользователя, имеющего права запуска заданий на кластере



- Файлы, в которые был переправлен поток вывода, содержат результаты работы тестов. Важно отметить, что ИМВ проводит много различных тестов, но нас для получения характеристик сети интересует только первый из них, **PingPong**, так как он передает данные между двумя узлами сети используя блокирующие функции MPI_Send и MPI_Recv, что, вероятно, наилучшим образом подходит для оценки характеристик сети.

```
IMB_test.txt - Notepad
File Edit Format View Help
#-----
# Benchmarking PingPong
# #processes = 2
#-----
#
#bytes #repetitions      t[usec]  Mbytes/sec
#-----
0      1000      131.62   0.00
1      1000      130.68   0.01
2      1000      130.47   0.01
4      1000      128.91   0.03
8      1000      127.64   0.06
16     1000      124.63   0.12
32     1000      124.85   0.24
64     1000      123.77   0.49
128    1000      123.78   0.99
256    1000      125.24   1.95
512    1000      128.76   3.79
1024   1000      129.25   7.56
2048   1000      137.03   14.25
4096   1000      254.61   28.34
8192   1000      364.53   42.73
16384  1000      665.65   81.16
32768  1000      1291.61  161.51
65536  640       880.04   108.13
131072 320       1993.63  249.16

MGU_test.txt - Notepad
File Edit Format View Help
Size(b) Transfer (MB/sec)

Iteration 0
[0 -- 1] Latency: 127.272 microseconds (at 20 times)
1024      6.878
2048      8.065
3072     11.79
4096     15.71
5120     20.11
6144     22.67
7168     25.65
8192     26.35
9216     24.59
10240    26.89
11264    29.62
12288    32.4
13312    34.92
14336    37.72
15360    40.29
16384    42.98
17408    44.91
```

Контрольные вопросы

- Дайте определения терминам задание (job) и задача (task). В чем основные отличия?
- Какие основные настройки Microsoft Visual Studio 2005 необходимо произвести при компиляции параллельной программы для использования в среде MS MPI?
- В чем особенность запуска параллельных задач (скомпилированных для MS MPI) на кластере?
- Что такое параметрическое множество задач (parametric sweep)? Что такое поток задач (task flow)?
- Какие характеристики, определяющие производительность сети, Вы знаете? Дайте их определения.

Лабораторная работа №6
**ПРОВЕРКА КОРРЕКТНОСТИ СИНХРОНИЗАЦИИ В
ПАРАЛЛЕЛЬНЫХ АЛГОРИТМАХ**

1. ПОИСК КРИТИЧЕСКИХ ДАННЫХ В ПАРАЛЛЕЛЬНЫХ АЛГОРИТМАХ

Возможность модификации данного в граф-модели вычислительного процесса определяется специальным признаком – **способом использования** данного. Способ использования определяется формально как функция $H(\alpha|d)$, использующая два операнда:

α - объект граф-модели, в качестве которого может выступать отдельный актор, подграф агрегата (ветвь) или целый агрегат;

d – данное предметной области.

Функция $H(\alpha|d)$ показывает, каким образом данное d используется объектом α .

Вводится следующая семантика для значений функции H :

$H(\alpha|d) = 0$ - d не используется объектом α

$H(\alpha|d) = 1$ - d используется для чтения объектом α

$H(\alpha|d) = 2$ - d используется для записи объектом α

$H(\alpha|d) = 3$ - d – критическое данное (конфликт совместного использования данного d в объекте α)

Определение: Данное d является **критическим**, если оно используется несколькими вершинами граф-модели, которые в процессе вычислений могут выполняться одновременно. При этом хотя бы одна из этих вершин помечена объектом, использующим данное d для записи.

Под одновременным исполнением мы понимаем пересечение отрезков времени, в течение которых исполняются интересные нас вершины.

Возникает задача выделения из множества вершин граф-модели подмножества таких вершин, которые могут выполняться одновременно и используют для записи одни и те же данные. При решении этой задачи как минимум необходимо первоначально сформировать списки одновременно исполняющихся вершин. Будем называть такие вершины *параллельными*.

Параллельные вершины однозначно должны принадлежать различным параллельным ветвям, однако данное требование не является достаточным - в программе могут присутствовать последовательные участки, содержащие параллельные ветви. Следовательно, чтобы ответить на вопрос, являются ли две вершины параллельными, необходимо внимательно изучить структуру граф-модели, проверяя принадлежность вершин различным параллельным ветвям, и рассматривая взаимное расположение этих ветвей друг относительно друга.

Опираясь на понятие способа использования, задачу поиска критических данных можно сформулировать следующим образом: зная способ использования данных в каждой вершине, найти такие данные, для которых способ использования графом модели в целом равен 3.

Рассматриваемый ниже алгоритм поиска критических данных решает задачу именно в такой формулировке. Он основан на построении формального описания граф-модели, которое позволяет обнаруживать в ней критические данные на основе информации о способе использования данных каждой вершиной и взаимном расположении этих вершин.

Рассмотрим возможные варианты взаимного расположения вершин в графе модели.

Простейший случай – передача управления из одной вершины модели в другую. Введем двуместную **операцию следования** над способами использования данных в вершинах, соединенных дугой управления. Обозначим эту операцию символом " Δ ". Результатом операции будет способ использования данных для подграфа, состоящего из двух вершин и соединяющей их дуги. Пример подграфа приведен на рисунке 1, а).

Очевидно, что при отсутствии критических данных в вершинах, такой подграф не приведет к их возникновению. Вершины подграфа выполняются строго последовательно, поэтому конфликтов при доступе к данным нет.

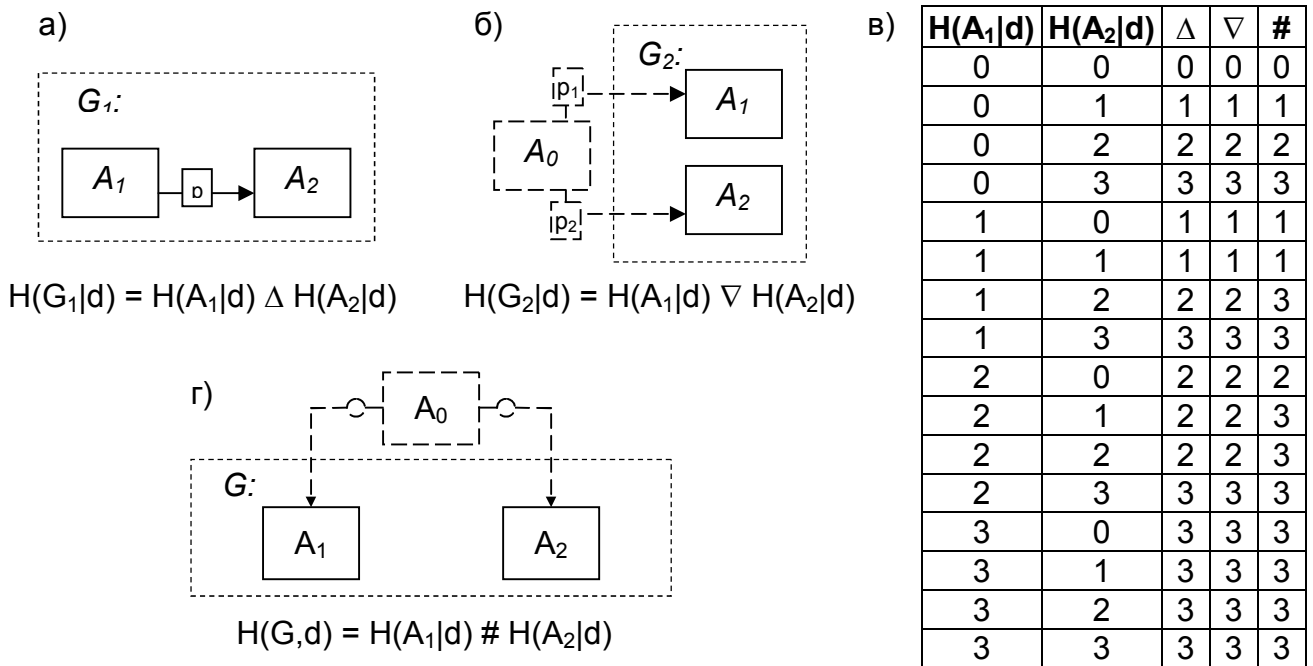


Рисунок 1 – Определение операций над способами использования данных для различных вариантов взаимного расположения вершин

Критические данные в таком подграфе могут быть только в том случае, когда они уже присутствуют в одной из его вершин, то есть способ использования некоторого данного в одной из вершин подграфа равен 3. Для определения результата операции следования при других значениях способа использования данных учтем то обстоятельство, что критические данные возникают только при использовании данных для записи. Если рассматриваемый нами подграф будет исполняться одновременно с другой параллельной ветвью, имеющей с ним общие

данные, то при использовании этих данных для записи в любой из вершин A_1 или A_2 , может возникнуть конфликтная ситуация. Следовательно, результат операции следования должен быть равен 2, если хотя бы одна из вершин подграфа использует некоторое данное для записи. Результаты операции следования для любых значений способов использования данных приведены в таблице истинности на рисунке 1, в).

Рассмотрим подграф G_2 , изображенный на рисунке 1, б). Этот подграф обозначает ветвление, то есть развитие вычислительного процесса по одному из нескольких путей. Он состоит из вершин, в которые входят последовательные дуги, исходящие из одной и той же вершины (на рисунке 1, б) эта вершина изображена пунктиром). Введем для таких подграфов **операцию ветвления** над способами использования данных в вершинах, принадлежащих различным направлениям развития вычислений. Обозначим ее символом " ∇ ". Формула вычисления способа использования данных в подграфе G_2 приведена на рисунке 1, б). Таблица истинности для операции ветвления приведена на рисунке 1, в).

Как видно из таблицы, результат операций следования и ветвления совпадает. Операция ветвления определена на основе рассуждений, приведенных выше для операции следования. Тем не менее, введение операции ветвления удобно при построении формул для вычисления способа использования данных в агрегатах со сложной структурой.

Подграф, содержащий параллельные вершины, изображен на рисунке 1, г). Он состоит из вершин, в которые ведут параллельные дуги, исходящие из одной вершины. Для таких подграфов введем **операцию параллельного исполнения** над способами использования данных и обозначим ее символом " $\#$ ".

Таблица истинности операции параллельного исполнения приведена на рисунке 1, в). Операция порождает критические данные, если один из ее операндов равен 3, а также в случае, когда оба операнда больше нуля, и хотя бы один из операндов равен 2. Другими словами, критическое данное возникает при совместном использовании некоторого данного параллельно исполняющимися вершинами, когда хотя бы одна из этих вершин использует данное для записи.

Используя введенные операции, определим **алгебру над способами использования данных** в граф-модели вычислительного процесса.

Определение: Алгеброй над способами использования данных назовем систему $\Lambda = (H; \Omega)$, где

$H = \{H(A_1|d_1), \dots, H(A_1|d_l), \dots, H(A_n|d_1), \dots, H(A_n|d_l)\}$ – множество, состоящее из способов использования данных предметной области в вершинах граф-модели,

$\Omega = \{\Delta, \nabla, \#\}$ – совокупность операций над способами использования данных.

Правильно построенные формулы в введенной алгебре определим следующим образом.

Элементарной правильно построенной формулой является способ

использования $H(A_i|d)$ данного d в некоторой вершине A_i граф-модели, помеченной оператором f_i , а не агрегатом. Кроме того, если α, β - правильно построенные формулы, то следующие формулы будут также правильно построенными:

- 1) (α)
- 2) $\alpha \Delta \beta$
- 3) $\alpha \nabla \beta$
- 4) $\alpha \# \beta$

- других правильно построенных формул нет

Для граф-моделей с произвольной структурой, не содержащих иерархии и кратных дуг, алгебра над способами использования данных позволяет строить формулу для вычисления способа использования каждого данного произвольным объектом модели (актором, подграфом граф-модели или целым агрегатом). Если для какого-то данного результат вычисления его способа использования агрегатом равен 3, то в агрегате возможен конфликт совместного использования этого данного.

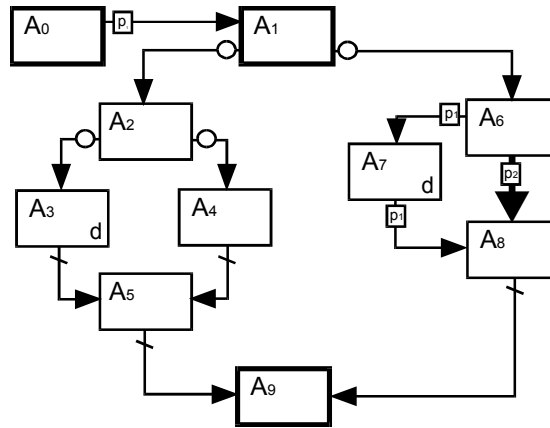
Свойства операций следования, ветвления и параллельного исполнения приведены ниже (для сокращения записи способ использования данного d в вершине A - $H(A|d)$ - заменим обозначением самой вершины A и опустим «пустые» вершины):

- 1) $A \Delta B = B \Delta A$ (коммутативность операции следования)
- 2) $A \Delta B \Delta A = A \Delta B$ (поглощение операции следования)
- 3) $A \Delta (B \Delta C) = (A \Delta B) \Delta C$ (ассоциативность операции следования)
- 4) $A \Delta B = A \nabla B$ (эквивалентность операций следования и ветвления)
- 5) $A \# B = B \# A$ (коммутативность операции параллельного исполнения)
- 6) $A \# (B \# C) = (A \# B) \# C$ (ассоциативность операции параллельного исполнения)
- 7) $A \# A \# A = A \# A$ (поглощение операции параллельного исполнения)
- 8) $A \# A \neq A$
- 9) $A \# B \# A \neq A \# B$
- 10) $A \# (B \Delta C) = (A \# B) \Delta (A \# C)$, $A \# (B \nabla C) = (A \# B) \nabla (A \# C)$
(дистрибутивность операции $\#$ относительно Δ и ∇)
- 11) $A \Delta (B \# C) \neq (A \Delta B) \# (A \Delta C)$

$$12) (A\#B) \Delta (A\#C) \Delta (B\#C) = A\#B\#C$$

Следует обратить внимание на свойство 2 (поглощение операции следования). Это свойство упрощает построение формул для граф-моделей, содержащих циклы. В таких моделях дуги, осуществляющие возврат управления в цикле, могут не учитываться.

Рассмотрим пример, иллюстрирующий применение формул над способами использования данных. На рисунке 2 приведена граф-модель и соответствующая ей формула.



$$H(G,d) = A_0 \Delta (A_1 \Delta ((A_2 \Delta (A_3 \# A_4) \Delta A_5) \# (A_6 \Delta (A_7 \nabla A_8)))) \Delta A_9$$

Рисунок 2 - Пример граф-модели и соответствующей ей формулы над способами использования данных

Пусть вершины A3 и A7 производят запись одного и того же данного d. Другие вершины не используют это данное:

$$H(A_3, d)=2, H(A_7, d)=2, H(A_i, d)=0, \forall i, i \neq 3, i \neq 7. \quad (1)$$

Подставим в формулу рисунка 2 значения способов использования данного d в вершинах граф-модели:

$$H(G,d) = 0 \Delta (0 \Delta ((0 \Delta (2 \# 0) \Delta 0) \# (0 \Delta (2 \nabla 0)))) \Delta 0 \quad (2)$$

Вычисление формулы 2 показывает, что способ использования данного d для граф-модели в целом равен 3 (критическое данное) (рисунок 3).

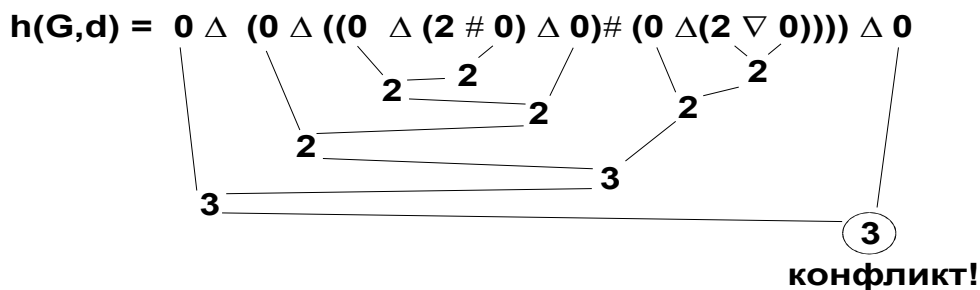


Рисунок 3 - Вычисление способа использования данного d граф-моделью

2 ПРОВЕРКА КОРРЕКТНОСТИ СИНХРОНИЗАЦИИ ГРАФ-МОДЕЛИ

Если в граф-модели присутствуют критические данные, разработчику модели необходимо позаботиться об устранении конфликта путем переименования данных или введения в модель дуг синхронизации.

Вместе с тем, при построении дуг синхронизации разработчик может допустить ошибку. Поэтому необходимо иметь средства проверки корректности синхронизации в графической модели параллельного алгоритма.

2.1 Метод проверки корректности синхронизации граф-модели

Описываемый ниже метод проверяет, исключают ли введенные в модель дуги синхронизации конфликт совместного использования данных. Поскольку синхронизация должна быть корректной при любом наборе входных данных, метод не учитывает содержимого предикатов, которыми помечены дуги управления в граф-модели. Переход по любой дуге управления считается возможным.

Несколько изменим семантику граф-модели, добавив к каждой дуге управления параллельную ей дугу синхронизации, соединяющую те же вершины, условно положив, что при передаче управления от одной вершины к другой, передается сообщение о разрешении запуска очередного оператора.

Такое изменение позволяет привести семафорный предикат каждой вершины граф-модели к универсальному виду. Для произвольной вершины A_i семафорный предикат будет иметь следующий вид:

$$R(A_i) = (\bigvee_j b_{j,i}) \wedge (r(b_{k_1,i}, \dots, b_{k_M,i})), \quad j = j_1, \dots, j_n \quad (3)$$

где j_1, \dots, j_n – номера вершин, из которых исходят дуги управления, входящие в A_i , k_1, \dots, k_M – номера вершин, из которых исходят дуги синхронизации, входящие в A_i , а $r(b_{k_1,i}, \dots, b_{k_M,i})$ – логическая функция.

Принятие семафорным предикатом значения истинности в результате его вычисления можно считать условием запуска оператора вершины на исполнение. Действительно, для того, чтобы началось вычисление семафорного предиката некоторой вершины, необходимо, чтобы эта вершина получила управление по дуге управления. После этого единственным условием запуска оператора вершины на исполнение является истинность семафорного предиката данной вершины.

Условие запуска произвольной вершины A_i в рекурсивном виде можно записать так:

$$R'(A_i) = (\bigvee_{j=j_1}^{j_n} (b_{j,i} \wedge R'(A_j))) \wedge (r((R'(A_{k_1}) \wedge b_{k_1,i}), \dots, (R'(A_{k_M}) \wedge b_{k_M,i}))),$$
$$R'(A_0) = \text{«истина»}, \quad (4)$$

где j_1, \dots, j_n – номера вершин, из которых исходят дуги управления, входящие в A_i , k_1, \dots, k_M – номера вершин, из которых исходят дуги синхронизации, входящие в A_i , а $r(b_{k_1,i}, \dots, b_{k_M,i})$ – логическая функция.

Назовем условие запуска оператора вершины на исполнение, определяемое выражением (4), ее обобщенным семафорным предикатом.

Можно доказать следующее

Утверждение: Критическое данное d , используемое вершинами A_1, \dots, A_n , не приводит к конфликту тогда и только тогда, когда для любых двух вершин $A_i, A_j \in \{A_1, \dots, A_n\}$, обобщенный семафорный предикат одной вершины содержит сообщение от другой вершины.

Метод проверки корректности синхронизации множества вершин $A' = \{A_{i_1}, \dots, A_{i_n}\}$, использующих некоторое критическое данное d , заключается в построении обобщенного семафорного предиката для каждой из вершин A_{i_1}, \dots, A_{i_n} и анализе содержимого полученных семафорных предикатов. Если существует такая пара вершин $\{A_i, A_j\} \subset A'$, для которой семафорный предикат вершины A_i не содержит сообщения от вершины A_j , и при этом семафорный предикат вершины A_j не содержит сообщения от вершины A_i , то конфликт совместного использования данного d вершинами A_{i_1}, \dots, A_{i_n} не разрешен.

Рассмотренный метод используется для проверки корректного использования критических данных в модели, в которой дуги синхронизации соединяют вершины, не входящие в отличные друг от друга циклы. Это означает, что во время исполнения оператора одной из вершин, соединенных дугой синхронизации, другая вершина не может вновь получить управление. Данное ограничение выполнимо, так как дугу синхронизации всегда можно вынести за пределы цикла, например, проведя ее от вершины, следующей за циклом или предшествующей ему. В таком случае может лишь снизиться быстродействие вычислительного процесса, так как синхронизироваться (ожидать друг друга) будут более крупные его блоки.

2.2 Метод поиска тупиков в граф-модели параллельного алгоритма

Введение в модель дуг синхронизации позволяет избавиться от критических данных, но влечет за собой опасность возникновения в модели тупиков.

Определение. **Тупиком** (взаимоблокировкой, дедлоком, клинчем) будем называть такое состояние вычислительного процесса, при котором он еще не завершен, но ни один из его операторов не может начать исполнение.

Тупики возникают, когда параллельные ветви вычислительного процесса бесконечно долго ожидают появления или освобождения ранее захваченных ресурсов, которые никогда не появятся или не будут освобождены. В граф-модели технологии ГСП в качестве таких ресурсов выступают сообщения, пересылаемые между вершинами в соответствии с дугами синхронизации. Если

несколько вершин граф-модели будут ожидать сообщения, которые никогда не будут сформированы, возникнет тупик.

Пример граф-модели, содержащей тупик, приведен на рисунке 4.

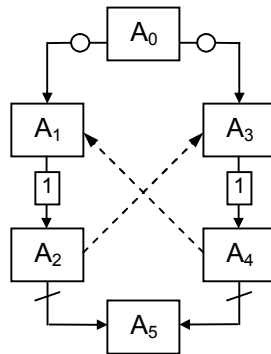


Рисунок 4 - Пример граф-модели, содержащей тупик

По завершении оператора начальной вершины управление передается вершинам A_1 и A_3 , принадлежащим различным параллельным ветвям. Семафорный предикат вершины A_1 содержит сообщение от вершины A_4 , а семафорный предикат вершины A_3 – сообщение от вершины A_2 . Эти сообщения никогда не поступят в почтовый ящик, так как вершины A_2 и A_4 , в которых они формируются, сами получают управление после завершения вершин A_1 и A_3 соответственно. Таким образом, после передачи управления вершинам A_1 и A_3 , ни в одной из вершин не сможет начаться исполнение оператора. Возникнет тупик.

Для поиска тупиков в граф-модели параллельного вычислительного процесса воспользуемся понятием состояния модели параллельного вычислительного процесса как множества вершин, в которых происходят вычисления в момент времени t : $S_t = \{ A_{i_0}, \dots, A_{i_{k-1}} \}$.

Так как в каждой параллельной ветви одновременно может выполняться не более одной вершины, значение k не может превышать числа параллельных ветвей в модели (обозначим его через K) за вычетом мастер-ветви: $0 \leq k \leq K-1$.

Построим **покрывающее дерево**, определяющее множество возможных состояний модели параллельного вычислительного процесса.

Покрывающее дерево будем строить по следующим правилам.

В каждой вершине дерева разместим по одному состоянию модели, которое описывается множеством $S = \{ A_{i_0}, \dots, A_{i_{k-1}} \}$.

Корневая вершина покрывающего дерева содержит множество $S_0 = \{ A_0 \}$, где A_0 – начальная вершина модели. S_0 соответствует начальному состоянию модели в момент времени $t = 0$.

Произвольная вершина покрывающего дерева (содержащая состояние S_i) может иметь потомков, описывающих состояния, в которые способна перейти модель из состояния S_i . Очевидно, что число таких состояний

$$Ns_i \leq \sum_{j=1}^{k_i} M_j,$$

где M_j – количество исходящих дуг из вершины A_j

k_i – количество элементов множества S_i .

Так как M_j конечно, а $k_i \leq K-1$, где K – конечно, то для модели без циклов покрывающее дерево содержит конечное число вершин.

Отметим, что перечень потомков, которые могут быть добавлены к некоторой вершине i , зависит не только от содержимого множества S_i , но и от содержимого вершин, предшествующих вершине i , то есть расположенных ближе к корневой вершине покрывающего дерева. Это обусловлено тем, что возможность запуска некоторой вершины модели (т.е. перехода в новое состояние) определяется значением предиката, помечающего дугу, входящую в данную вершину, а также истинностью семафорного предиката этой вершины. При поиске тупиков нас интересуют все возможные пути развития вычислительного процесса, поэтому будем полагать, что переход по любой дуге модели возможен. Для этого достаточно подобрать соответствующий набор входных данных модели. Тогда возможность изменения состояния модели путем перехода по некоторой дуге определятся истинностью семафорного предиката вершины, в которую ведет эта дуга. Семафорный предикат использует сообщения от других вершин модели, отправляемых данной вершине. Наличие этих сообщений при нахождении модели в состоянии S_i зависит от того, какие вершины завершили вычисления на момент перехода модели в состояние S_i , т.е. от предшествующих ему состояний.

В качестве примера на рисунке 5, б) изображено покрывающее дерево, описывающее множество возможных состояний граф-модели, приведенной на рисунке 5, а).

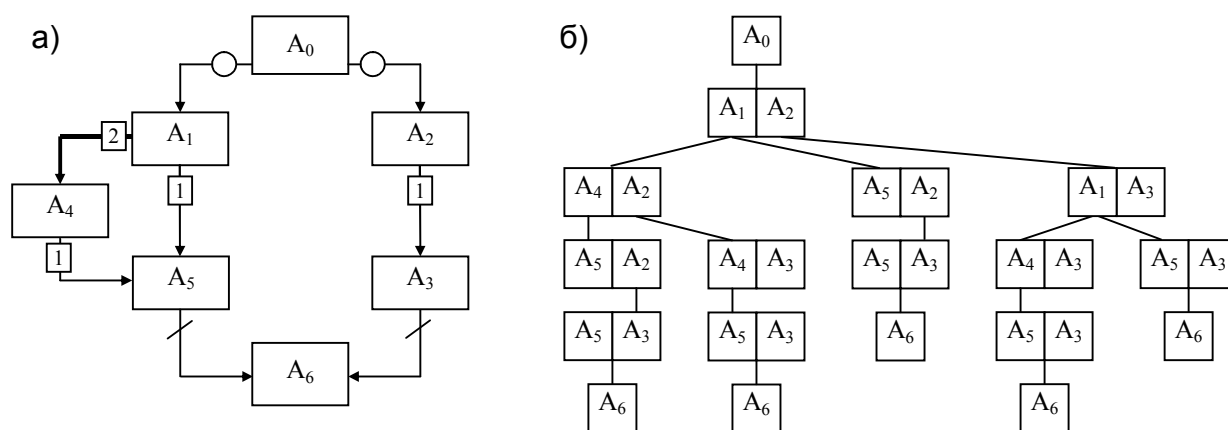


Рисунок 5 - Пример покрывающего дерева, соответствующего граф-модели параллельного вычислительного процесса

2. ПРОВЕРКА КОРРЕКТНОСТИ СИНХРОНИЗАЦИИ В СИСТЕМЕ ГСП PGRAPH

Описанные выше методы реализованы в системе ГСП PGRAPH.

Пункт «Поиск критических данных» меню «Данные» позволяет получить список критических данных, а также увидеть вершины агрегата, в которых они используются.

Проверка корректности введенных пользователем дуг синхронизации осуществляется выбором пункта «Проверка корректности синхронизации» меню «Данные».

Пункт «Поиск тупиков» при наличии в параллельном алгоритме условия для взаимной блокировки позволяет увидеть вершины, в которых остановится вычислительный процесс в случае возникновения тупика.

3. Лабораторная работа №6

Цель работы: Проверка корректности синхронизации в параллельных алгоритмах.

3.1 Задание на самостоятельную работу

1. Получить задание у преподавателя.
2. Используя операции следования, ветвления и параллельного исполнения, построить формулу над способами использования данных.
3. С помощью вычислений по построенной формуле найти критические данные в параллельном алгоритме в соответствии с заданием.
4. Проверить результаты вычислений, построив граф-модель алгоритма в системе PGRAPH и воспользовавшись средством поиска критических данных системы PGRAPH.
5. Добавить в граф-модель дуги синхронизации для разрешения конфликта совместного использования критических данных.
6. Проверить правильность введенной в алгоритм синхронизации, используя метод проверки корректности синхронизации.
7. Проверить модель на наличие взаимных блокировок с помощью метода поиска тупиков.
8. Сравнить полученные результаты с результатами работы средств проверки корректности синхронизации системы PGRAPH.
9. Составить отчет по результатам работы.

3.2 Содержание отчета

1. Постановка задачи.
2. Формула над способами использования данных для граф-модели в соответствии с заданием. Результаты вычислений по построенной формуле.
3. Обобщенные семафорные предикаты для вершин, использующих общие данные.
4. Покрывающее дерево для граф-модели в соответствии с заданием.
5. Результаты сравнения произведенных вычислений с результатами работы средств проверки корректности синхронизации системы PGRAPH.
6. Выводы по работе.

3.3 Контрольные вопросы

1. Что такое критическое данное?
2. Запишите таблицу истинности для операции параллельного исполнения.
3. Что такое обобщенный семафорный предикат?
4. Что такое взаимная блокировка? Опишите условия ее возникновения.
5. Что такое покрывающее дерево?
6. Опишите правила построения покрывающего дерева и способ поиска тупиков с его использованием.

СПИСОК ЛИТЕРАТУРЫ

1. Коварцев А.Н. Автоматизация разработки и тестирования программных средств. - Самар. гос. аэрокосм. ун-т., Самара, 1999. - 150 с.
2. Коварцев А.Н., Жидченко В.В. Методы и средства визуального параллельного программирования. Автоматизация программирования. Электронный учебник. - Самара, СГАУ, 2010.
3. Коварцев А.Н. Численные методы: курс лекций для студентов заоч. формы обучения. - Самара: СГАУ, 2000. – 177 с.
4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. - М.: Вильямс, 2003. - 512 с.