УДК 004.272.2 (075.8)

Карасик О. Н., Прихожий А. А.

Кооперативный планировщик потоков для многопоточных приложений Windows

Введение. Решение прикладных задач над данными большого размера за приемлемое время невозможно без использования параллельных вычислений на многопроцессорных (многоядерных) системах [1]. Эффективность распараллеливания задачи определяется рядом факторов: присутствием параллелизма в методах решения задачи; возможностью построения эффективного параллельного многопоточного алгоритма; наличием инструментальных программных средств, позволяющих реализовать многопоточный алгоритм на выбранной многопроцессорной (многоядерной) архитектуре [2]. Существует большое количество разнообразных инструментов (в том числе стандартизированных) и их реализаций для разработки многопоточных приложений, например: средства ОС Windows, OpenMP, Cilk Plus, Portable Operating System Interface Threads (POSIX Threads or PThreads), Threading Building Blocks (TBB), Open Computing Language (OpenCL), Message Passing Interface (MPI) и другие [2–5]. Выбор подходящего инструмента в конкретных условиях разработки зависит от таких факторов, как доступность, совместимость с выбранной операционной системой, учет особенностей решаемой задачи. В данной статье исследуется архитектура кооперативного планировщика выполнения потоков [6] и оценивается его производительность на параллельных многопоточных алгоритмах решения СЛАУ методом Гаусса [7, 8] и нахождения кратчайших путей на графе методом Флойда-Уоршелла [8–11].

Кооперативная многозадачность в Windows. Операционная система Windows является системной с вытесняющей многозадачностью [12], однако, начиная с версии Windows 7 (для рабочих станций) и Windows Server 2008 R2 (для серверных систем), она предоставляет программный интерфейс User-Mode Scheduling (UMS), позволяющий приложению, выполняющемуся под управлением вытесняющего планировщика, организовывать кооперативное выполнение потоков [13]. UMS включает три основных компонента: рабочий поток – *UmsWorkerThread*, поток планирования – *UmsSchedulerThread* и список разблокированных рабочих потоков, ожидавших завершения обработки запроса в ядре операционной системы – *UmsCompletionList*.

Поток планирования UMSST представляет собой поток операционной системы, выполняющийся под управлением вытесняющего планировщика и отвечающий за выполнение рабочих потоков UMSWT. Поток переключается в режим планирования вызовом процедуры *EnterUmsSchedulingMode*, в которую (в качестве параметров), передаются: указатель на процедуру планирования и указатель на выделенный список заблокированных потоков UMSCL. Процедура планирования вызывается операционной системой в следующих случаях: инициализация UMSST, сразу после вызова *EnterUmsSchedulingMode*; блокировка рабочего потока UMSWT в ожидании завершения обработки запроса в ядре операционной системы; добровольный возврат управления рабочим потоком UMSWT потоку планирования UMSST. Помимо обработки обратных вызовов со стороны операционной системы, в обязанности процедуры планирования входит выбор следующего рабочего потока UMSWT на выполнение. Выполнение осуществляется вызовом процедуры *ExecuteUmsThread*.

Рабочий поток UMSWT представляет собой поток операционной системы, проинициализированный специальным набором атрибутов, указывающих операционной системе, что данный поток должен являться рабочим потоком UMSWT. Передаваемые атрибуты содержат выделенный контекст UMSWT и указатель на список заблокированных потоков UMSCL. После инициализации, рабочий поток UMSWT помещается в указанный UMSCL и, начиная с этого момента, состояние его выполнения определяется UMSST, обслуживающим данный UMSCL. Возврат управления UMSWT выполняется вызовом процедуры *UmsThreadYield*.

Архитектура кооперативного многопоточного планировщика. Разработанный нами кооперативный многопоточный планировщик состоит из трех основных компонентов: потока планировщика (ППЛ), потока пользователя (ПП) и примитива синхронизации (ПС).

Планировщик выполняет:

- основные функции по управлению памяти (менеджер памяти): выделение памяти с выравниванием (aligned memory), выделение памяти с ближайшего к логическому процессору узла NUMA, буферизация больших блоков памяти;
- функции создания / уничтожения и взаимодействия ППЛ, ПП и ПС.

При инициализации планировщик выделяет каждому логическому процессору системы:

- по одному списку заблокированных рабочих потоков UMSCL;
- по одной очереди готовых потоков пользователя (ОГПП);
- по одному ППЛ, включающему в себя выделенные на этот процессор список UMSCL и очередь ОГПП.

Кроме вышеперечисленного, планировщик включает в себя массив выделенных ППЛ, массив ПП и массив ПС. Архитектура кооперативного многопоточного планировщика изображена на рисунке 1, где объекты операционной системы показаны заливкой.

Поток планировщика ППЛ выполняет обработку: обратных вызовов со стороны операционной системы, запросов на прямую передачу управления между ПП, а также запросов на блокировку ПП в ожидании наступления события, представленного ПС. ППЛ включает в себя UMSST, процедуру планирования *UmsSchedulerProc*, очередь готовых к выполнению ПП (ОГПП) и поле, характеризующее текущее состояние ППЛ.

ППЛ инициализируется ядром планировщика, при этом на каждый логический процессор назначается один ППЛ. Во время инициализации состояние ППЛ переключается по цепочке: *Created* (создан, выделены ресурсы) → *Initializing* (проходит инициализация ядром планировщика) → *Initialized* (инициализация выполнена) → *Executing* (выполняется процедура планирования). В дальнейшем ППЛ всегда находится в одном из четырех состояний: «выполняется» - *Executing*, «простаивает» – *WaitingWrIdle* (в данный момент нет доступных к выполнению ПП), «выполняет ПП» – *WaitingTaskExecuting* (в данный момент выполняется ПП) и «уничтожен» – *Terminated* (ППЛ уничтожен).

Помимо непосредственного выполнения ПП и обработки обратных вызовов со стороны операционной системы, ППЛ реализует процесс прямой передачи управления между двумя ПП и процессы блокировки-разблокировки ПП с использованием ПС.

Процесс прямой передачи управления между ПП реализован следующим образом: $ПП_1$ добровольно возвращает управление ППЛ при помощи вызова процедуры *UmsThreadYield* и через её параметр *SchedulerParam* передает информацию о $ПП_2$, которому необходимо передать управление. Получив управление, ППЛ находит затребованный $ПП_2$ и приступает к его выполнению посредством вызова процедуры *ExecuteUmsThread*. Процессы блокировки-разблокировки ПП с помощью примитива синхронизации подробно рассмотрены в [6].

Карасик О. Н., *к. т. н., ведущий инженер иностранного производственного унитарного предприятия «ИССОФТ СОЛЮШЕНЗ».*
Беларусь, 220034, г. Минск, ул. Чапаева, 5, корпус 1.
Прихожий А. А., *д. т. н., профессор кафедры «Программное обеспечение информационных систем и технологий» Белорусского национального технического университета.*
Беларусь, 220013, г. Минск, пр-т Независимости, 65.

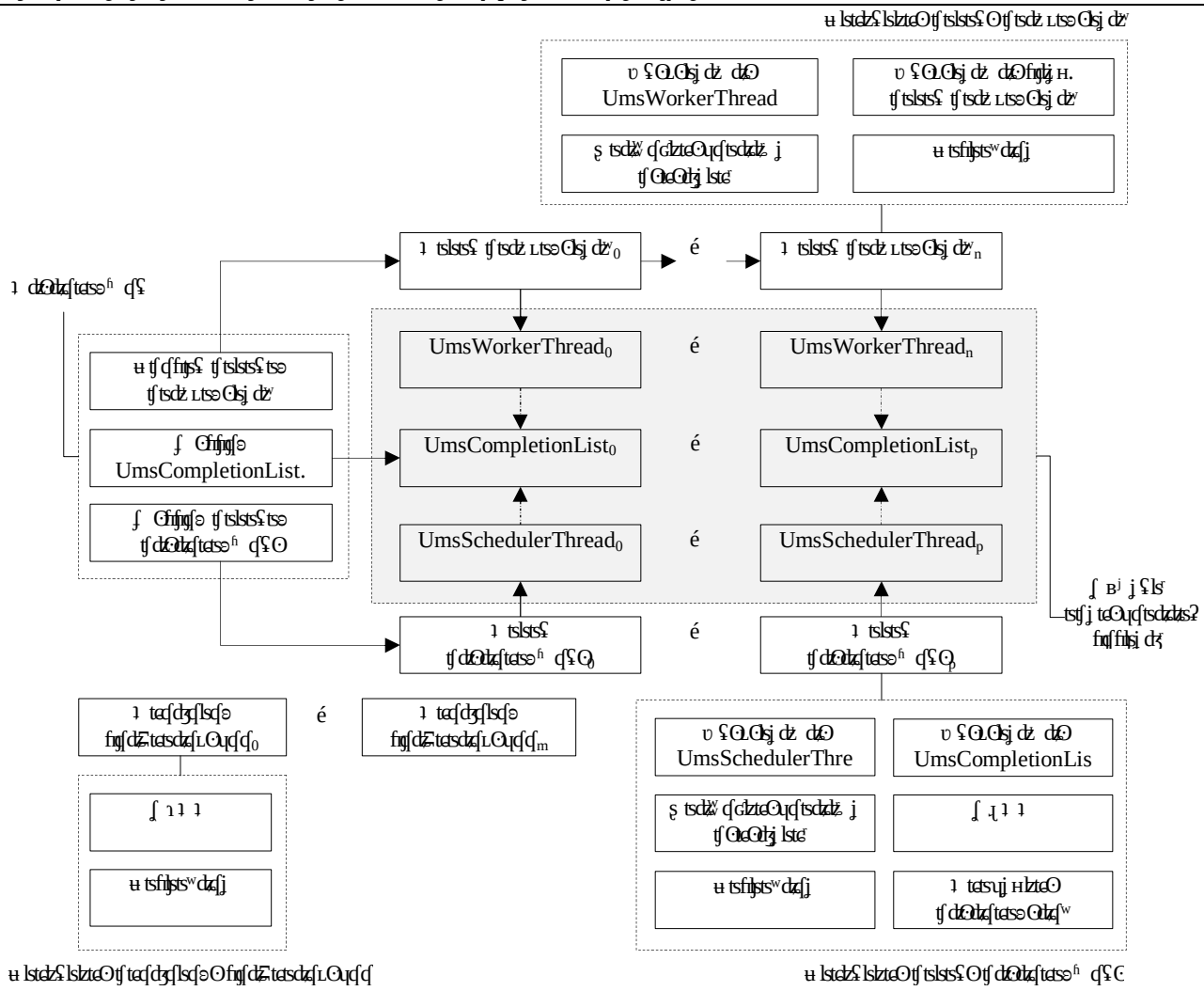


Рисунок 1 – Схематическое изображение архитектуры кооперативного многопоточного планировщика

Поток пользователя ПП выполняет пользовательскую процедуру и включает в себя UMSWT, а также поле, характеризующее текущее состояние потока. ПП инициализируется по запросу пользователя, который включает в себя указатель на пользовательскую процедуру, а также номер логического процессора, на котором процедура должна выполняться. Во время инициализации ПП переходит из состояния в состояние: *Created* (создан, выделены ресурсы) → *Initializing* (проходит инициализация) → *Initialized* (инициализация завершена) → *Ready* (поток перемещен в ОЗПП и готов к выполнению). Все остальное время ПП находится в одном из пяти конечных состояний: «готов» (состояние *Ready*); «выполняется» (состояние *Executing*); «заблокирован» (состояние *WaitingWrStopped*) в случае, когда ПП добровольно приостановил выполнение в ожидании прямой передачи управления; «ожидает» (состояние *WaitingWrBlocked*) в случае, когда ПП заблокирован в ожидании уведомления от ПС; «ожидает» (состояния *WaitingWrSystemTrap* и *WaitingWrSystemCall*) в случае, когда ПП заблокирован на время выполнения системного вызова; «уничтожен» (состояние *Terminated*). Во время перехода между основными состояниями, ПП переключается в промежуточные состояния: *StandBy* – когда ПП выбран для выполнения (на каждом логическом процессоре только один ПП может находиться в этом состоянии), и *WaitingWrYielded* – когда ПП добровольно вернул управление ППЛ, однако цель добровольной передачи (остановка, передача управления другому ПП, блокировка с использованием ПС) еще не определена.

Примитив синхронизации. Примитив синхронизации (ПС) представляет событие и применяется для синхронизации работы двух и более ПП. Использование ПС позволяет приостановить выполнение ПП до наступления события, а затем возобновить, без использования механизмов операционной системы. ПС инициализируется пользователем и не привязан к какому-либо логическому процессору, ППЛ или ПП.

ПС включает очередь заблокированных пользовательских потоков (ОЗПП), в которую помещаются заблокированные ПП в ожидании наступления события, и включает поле, характеризующее текущее состояние, в котором находится ПС.

В каждый момент времени ПС находится в одном из восьми возможных состояний: *Reset* (не установлен), *Signaled* (установлен), *Signaling* (устанавливается), *SignalingActSignal* (устанавливается-установить), *SignalingActReset* (устанавливается-сбросить), *Joining* (присоединяется), *JoiningActSignal* (присоединяется-установить) и *JoiningActReset* (присоединяется-сбросить). Состояния *Reset* и *Signaled* являются конечными состояниями, т. е. по завершении любой операции ПС переводится в одно из этих двух состояний, остальные состояния являются промежуточными и необходимы для выполнения параллельных операций блокировки / разблокировки ПП. Матрица переходов состояний ПС, и операции, вызывающие переходы, показаны на рисунке 2.

Чтобы показать роль каждого состояния, рассмотрим важнейшие ситуации, возникающие в процессе функционирования ПС.

	R	S	SG	SR	SS	JG	JR	JS
R			f_s			f_j		
S	f_r		f_s					
SG		f_s		f_s, f_{cr}	f_s, f_{cs}			
SR	-							
SS			-					
JG	-						f_j, f_{cr}	f_j, f_{cs}
JR	-							
JS			-					

Условные обозначения:

S – ПС в состоянии *Signaling*;

R – ПС в состоянии *Reset*;

SG – ПС в состоянии *Signaling*, ОЗПП не пуст;

SR – ПС в состоянии *Signaling*, ОЗПП не пуст, ПП заблокированы;

SS – ПС в состоянии *Signaling*, ОЗПП пуст, ПП заблокированы;

J – ПС в состоянии *Joining*;

JR – ПС в состоянии *Joining*, ПП заблокированы;

JS – ПС в состоянии *Joining*, ПП заблокированы, ОЗПП пуст.

Условные обозначения:

f_r – ПП$_1$ переключает ПС из состояния *Reset* в состояние *Signaling*;

f_s – ПП$_1$ переключает ПС из состояния *Signaling* в состояние *Signaled*;

f_j – ПП$_1$ переключает ПС из состояния *Joining* в состояние *Joining*;

f_{cr} – ПП$_1$ переключает ПС из состояния *Signaling* в состояние *SignalingActReset*;

f_{cs} – ПП$_1$ переключает ПС из состояния *Signaling* в состояние *SignalingActSignal*;

f_{cj} – ПП$_1$ переключает ПС из состояния *Joining* в состояние *JoiningActSignal*.

Рисунок 2 – Матрица переходов состояний примитива синхронизации

Ситуация 1. ПС находится в состоянии *Reset*, следовательно, событие, представляемое ПС, не наступило. Поток ПП$_1$ хочет уведомить о наступлении этого события, для этого он выполняет процедуру уведомления, которая должна перевести ПС в состояние *Signaled* или разблокировать один из заблокированных ПП. Выполнение процедуры уведомления выглядит следующим образом:

Шаг 1. ПП$_1$ переключает ПС из состояния *Reset* в состояние *Signaling*, означающее, что в данный момент поток выполняет процедуру уведомления.

Шаг 2. ПП$_1$ проверяет ОЗПП на наличие заблокированных ПП. Если ОЗПП не содержит заблокированных ПП, то ПП$_1$ переходит к шагу 3а, иначе к шагу 3б.

Шаг 3а. Так как ОЗПП не содержит заблокированных ПП, то ПП$_1$ переключает ПС из состояния *Signaling* в состояние *Signaled*, означающее, что представляемое ПС событие наступило, при этом процедура завершает выполнение.

Шаг 3б. Так как ОЗПП содержит один или более заблокированных ПП, то ПП$_1$ извлекает один заблокированный ПП из ОЗПП и перемещает его в ОГПП (одновременно переключая этот ПП в состояние *Ready*) соответствующего ППЛ. После этого переключает ПС в состояние *Reset* и завершает выполнение процедуры.

Ситуация 2. Представим, что одновременно с ПП$_1$, из предыдущего примера, еще один поток ПП$_2$ хочет уведомить о наступлении того же самого события, тогда выполнение процедуры уведомления ПС будет выглядеть следующим образом:

Шаг 1. ПП$_2$ не удается переключить ПС из состояния *Reset* в состояние *Signaling,* поскольку ПС уже находится в состоянии *Signaling*.

Шаг 2. Поскольку ПП$_2$ не знает, выполнил ли уже ПП$_1$ проверку ОЗПП или нет, то он переключает ПС из состояния *Signaling* в состояние *SignalingActSignal*, которое означает, что по завершении процедуры, если ПП$_1$ не разблокировал ни один ПП, он должен переключить ПС обратно в состояние *Signaling* и провести повторную проверку ОЗПП.

Ситуация 3. Представим, что ПП$_2$ из предыдущего примера хотел не уведомить, а наоборот сбросить уведомление о наступлении события. Процедура сброса выглядит точно так же, как и описанная выше процедура уведомления, за исключением того, что при выполнении второго шага ПП$_2$ переключит ПС в состояние *SignalingActReset*, означающее, что по завершении процедуры ПП$_1$ обязан переключить ПС в состояние *Reset* вне зависимости от того, были разблокированы ПП или нет.

Блокировка ПП выполняется похожим образом, за тем лишь исключением, что операция блокировки выполняется ППЛ. Рассмотрим две ситуации.

Ситуация 4. ПП$_3$ добровольно вернул управление ППЛ$_1$ с целью выполнения блокировки в ожидании наступления события, представляемого ПС. ПС находится в состоянии *Reset*. Выполнение процедуры блокировки выглядит следующим образом:

Шаг 1. ППЛ$_1$ переключает ПС из состояния *Reset* в состояние *Joining*, означающее, что в данный момент ППЛ выполняет процедуру блокировки ПП.

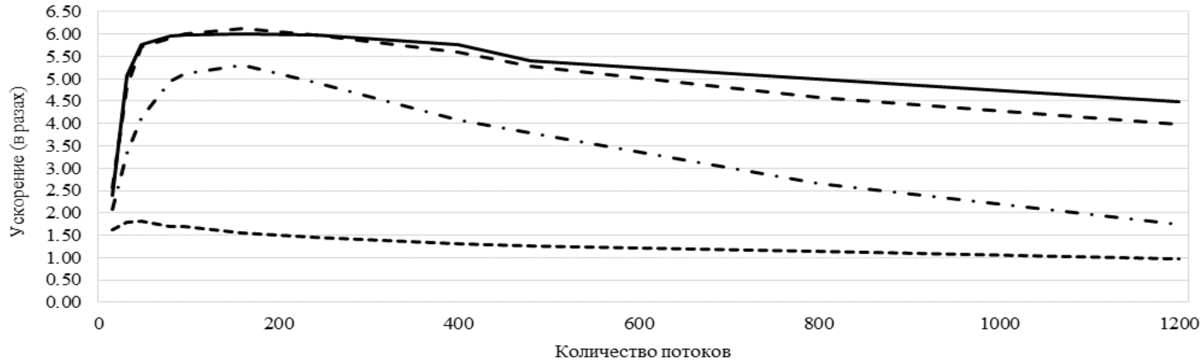
Шаг 2. ППЛ$_1$ помещает ПП$_3$ в ОЗПП.

Шаг 3. ППЛ$_1$ переключает ПС обратно в состояние *Reset*.

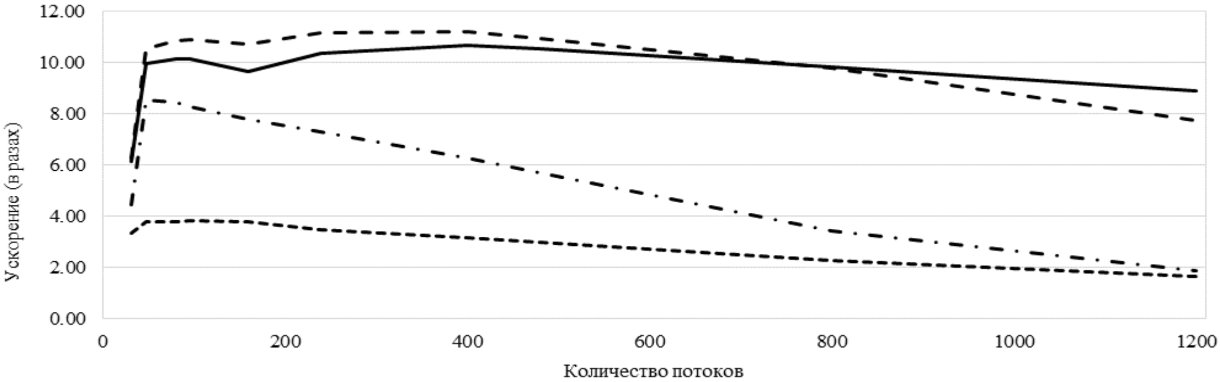
Ситуация 5. Представим, что одновременно с процессом блокировки ПП$_3$, другой поток ПП$_1$ хочет уведомить о наступлении этого события, тогда выполнение процедуры уведомления выглядит следующим образом:

Шаг 1. ПП$_1$ не удается переключить ПС из состояния *Reset* в состояние *Signaling,* поскольку ПС уже находится в состоянии *Joining*.

Шаг 2. ПП$_1$ переключает ПС из состояние *Joining* в состояние *JoiningActSignal*, которое означает, что по завершении процедуры блокировки ППЛ$_1$ должен переключить ПС в состояние *Signaling* и выполнить проверку ОЗПП на возможность разблокировки заблокированных ПП.



а) Intel Core i5-3450



б) Intel Xeon E5520

Рисунок 3 – Ускорение в разах алгоритмов μ1к (сплошная), μ2к (пунктирная) и алгоритмов μ1 (штрихпунктирная), μ2 (точечная) над лучшим однопоточным приложением в серии из тысячи программных прогонов в зависимости от числа потоков для СЛАУ размером 2400

Таблица 1 – Диапазоны разброса времени выполнения алгоритмов μ1, μ2, μ1к и μ2к на двух многоядерных системах

	Первая многоядерная система (Intel Core i5-3450, 4 ядра)			Вторая многоядерная система (2 x Intel Xeon E5520, 8 ядер)		
Алгоритм	От (сек)	До (сек)	Ширина (сек)	От (сек)	До (сек)	Ширина (сек)
μ1 (160 потоков)	0.89	1.05	0.16	1.09	1.85	0.76
μ2 (48 потоков)	2.49	3.19	0.70	2.61	2.95	0.34
μ1к (160 потоков)	0.80	1.35	0.55	0.969	1.005	0.044
μ2к (160 потоков)	0.79	0.99	0.20	0.912	0.952	0.040

Таблица 2 – Диапазоны разброса времени выполнения алгоритмов БФУ и КПБПА

Многоядерная система (Intel Core i5-3450, 4 ядра)			
Алгоритм	От (сек)	До (сек)	Ширина (сек)
БФУ (размер блока 120x120)	0.836	1.176	0.340
КПБПА (размер блока 120x120)	0.655	0.755	0.100

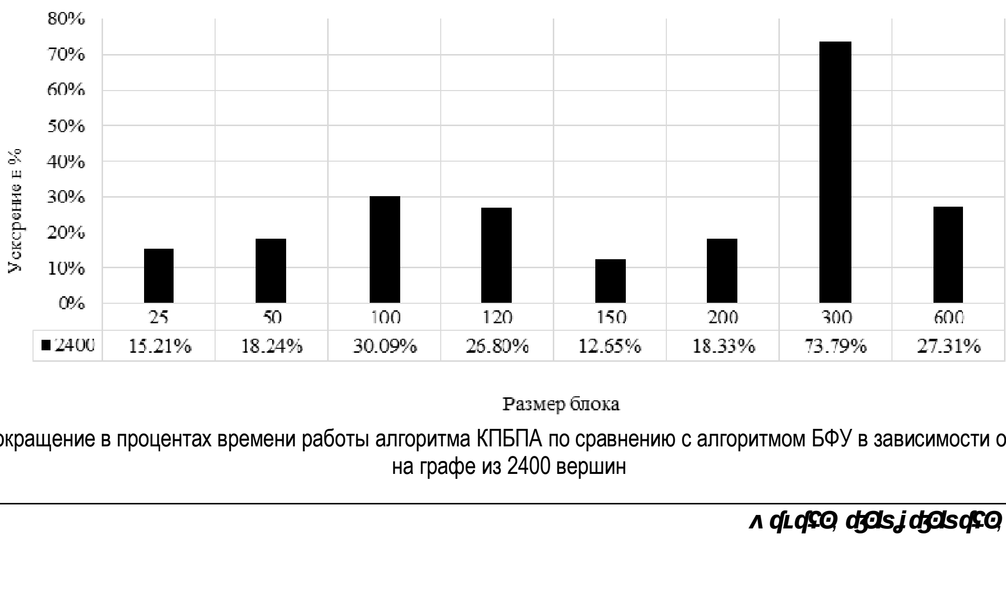


Рисунок 4 – Сокращение в процентах времени работы алгоритма КПБПА по сравнению с алгоритмом БФУ в зависимости от размера блока на графе из 2400 вершин

Экспериментальные результаты. В экспериментах использованы две многоядерные системы. Первая оснащена двумя 4 ядерными процессорами Intel Xeon E5520. Каждое ядро работает с частотой 2.26 GHz, оснащено многоуровневой кэш-памятью (L1 – 64KB, L2 – 256 KB) и технологией Hyper-Threading. Каждый процессор имеет разделяемую ядрами кэш-память третьего уровня L3 емкостью 8MB, а также имеет доступ к локальной и удаленной памяти с использованием технологии NUMA. Управление многоядерной системой осуществляет ОС Windows Server 2012 R2 64. Вторая многоядерная система оснащена одним 4-ядерным процессором Intel Core i5-3450 с частотой 3.10 ГГц (архитектура IvyBridge), имеющим многоуровневую кэш-память (локальный уровень L1 емкостью 256 KB, локальный уровень L2 емкостью 1.0 MB и разделяемый уровень L3 емкостью 6.0 MB), 16 GB ОЗУ, и управляется ОС Windows 10 Professional версии 1809.

Реализация планировщика и потоковых алгоритмов. Кооперативный многопоточный планировщик реализован на языке C/C++ в виде динамически подключаемой библиотеки (.dll) с использованием компилятора Visual C++ 14.1. Известные алгоритмы $\mu 1$ и $\mu 2$ решения СЛАУ методом Гаусса [10, 14] реализованы стандартными средствами операционной системы Windows: потоками (threads) и примитивом синхронизации «автоматическое событие» (AutoResetEvent). Предложенные кооперативные алгоритмы $\mu 1\kappa$ и $\mu 2\kappa$ [7, 8] реализованы с использованием разработанной библиотеки планировщика. Исходные коды алгоритмов $\mu 1$, $\mu 2$, $\mu 1\kappa$ и $\mu 2\kappa$ написаны на языке C/C++ и откомпилированы в исполняемые файлы (.exe) посредством компилятора Visual C++ 14.1. Известный блочно-параллельный алгоритм Флойда – Уоршелла (БФУ) [15-17] реализован посредством директив OpenMP для позадачного параллелизма (task-based). Предложенный кооперативный потоковый блочно-параллельный алгоритм КПБПФ [9] реализован на базе библиотеки кооперативного планировщика. Исходные коды алгоритмов БФУ и КПБПА написаны на языке C/C++ и скомпилированы компилятором Intel Compiler 18, настроенным на максимальную оптимизацию (OV3) с опциями дополнительной оптимизации для экспериментальной среды (архитектура процессора IvyBridge с применением векторизации Intel AVX).

Результаты экспериментов. С целью доказательства эффективности разработанного кооперативного многопоточного планировщика проведены эксперименты по решению двух прикладных задач: решения СЛАУ методом Гаусса и поиска кратчайших путей на графе методом Флойда – Уоршелла. На рисунках 3, *а* и 3, *б* представлены результаты для решения СЛАУ размером 2400 переменных, полученные в серии из 1000 прогонов каждого из четырех алгоритмов $\mu 1$, $\mu 2$, $\mu 1\kappa$ и $\mu 2\kappa$ на различных количествах потоков. Результаты многопоточных реализаций сопоставлены с результатами однопоточных реализаций на двух многоядерных системах. На первой многоядерной системе однопоточное приложение показало наилучший результат в 4.91 сек. и показало 10.43 сек. на второй многоядерной системе. На первой многоядерной системе, кооперативные алгоритмы $\mu 1\kappa$ и $\mu 2\kappa$ дали максимальное ускорение в 6.00 и в 6.12 раз, что превосходит максимальное ускорение, полученное алгоритмами $\mu 1$ и $\mu 2$ (5.30 и 1.82 раз соответственно). На второй многоядерной системе, кооперативные алгоритмы $\mu 1\kappa$ и $\mu 2\kappa$ продемострировали максимальное ускорение в 10.68 и 11.23 раз, что значительно превосходит ускорение, полученное известными алгоритмамим $\mu 1$ и $\mu 2$ (8.47 и 3.83 раз соответственно).

В дополнение к наилучшим средним результатам, полученным каждым из алгоритмов $\mu 1$, $\mu 2$, $\mu 1к$, $\mu 2к$ решения СЛАУ на обеих многоядерных системах, был оценен и проанализирован разброс времени выполнения в серии из 1000 прогонов. В таблице 1 приведены диапазоны разброса времени выполнения алгоритмов, показывающие превосходство новых $\mu 1к$ и $\mu 2к$ алгоритмов над известными $\mu 1$, $\mu 2$. Диапазоны для $\mu 1к$ и $\mu 2к$ на обоих многоядерных системах смещены в сторону уменьшения времени, при этом они незначительно пересекаются только с диапазоном алгоритма $\mu 1$ на первой многоядерной системе, а в остальных случаях не пересекаются.

На рисунке 4 представлены результаты для времени поиска кратчайших путей на графе, полученные в серии из 1000 прогонов программ на первой многоядерной системе, оснащенной процессором Intel Core i5-3450, для обоих алгоритмов БФУ и КПБПА на графе из 2400 вершин в зависимости от размера блока. Кооперативный алгоритм КПБПА продемонстрировал ускорение на всех размерах блока. Для наименьшего времени выполнения КПБПА показал ускорение в 26.8 % (0.842 против 0.664 секунд) при размере блока 120x120. Максимальное ускорение получено для размера блока 300x300. Разброс времени выполнения БФУ и КПБПА представлен в таблице **Ошибка! Источник ссылки не найден.**, которая убедительно демонстрирует, что кооперативный потоковый блочно-параллельный алгоритм, выполненный с использованием разработанного кооперативного планировщика, значительно превосходит известный блочно-параллельный алгоритм БФУ, выполненный с использованием известного вытесняющего планировщика. Ни один прогон из 1000 не привел к выигрышу алгоритма БФУ у алгоритма КПБПА.

Заключение. При решении прикладных задач, оперирующих данными большого размера, производительность многоядерной системы сильно зависит от операционной системы и встроенного в нее системного программного обеспечения, с одной стороны, а также от способа построения параллельных многопоточных алгоритмов, решающих прикладные задачи и выполняющих вычисления над большим объемом данных, с другой стороны. В статье предложен и экспериментально исследован кооперативный многопоточный планировщик, снижающий по сравнению с вытесняющим планировщиком операционной системы процессорное время, затрачиваемое на управление выполнением потоков, и позволяющий создавать многопоточные алгоритмы таким образом, что потоки, выполняющиеся на одном ядре, сами передают управление друг другу, а потоки, выполняющиеся на разных ядрах, взаимодействуют друг с другом посредством предложенного примитива синхронизации, ускоряющего процессы блокировки-разблокировки потоков. Проведенные вычислительные эксперименты показали высокую производительность планировщика на кооперативных блочно-параллельных алгоритмах решения систем линейных алгебраических уравнений методом Гаусса и на кооперативных блочно-параллельных алгоритмах нахождения кратчайших путей на графе методом Флойда-Уоршелла. Планировщик может быть использован для эффективного параллельного решения многих других задач, декомпозированных на части.

СПИСОК ЦИТИРОВАННЫХ ИСТОЧНИКОВ

1. Prihozhy, A. A. Analysis, transformation and optimization for high performance parallel computing / A. A. Prihozhy – Minsk : BNTU, 2019. – 229 p.
2. Richter, J. M. Windows via C/C++ / J. M. Richter, C. Nasarre. – 5th ed. – Microsoft Press, 2007. – 848 p.
3. OpenMP. OpenMP [Электронный ресурс]. – Режим доступа : https://www.openmp.org. – Дата доступа : 24.02.2020.
4. ISO/IEC/IEEE 9945:2009 Information technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7 [Электронный ресурс]. – Режим доступа : https://www.iso.org/standard/50516.html. – Дата доступа : 24.02.2020.
5. MPI Forum. MPI Forum [Электронный ресурс]. – Режим доступа : www.mpi-forum.org. – Дата доступа : 24.02.2020.
6. Карасик, О. Н. Усовершенствованный планировщик кооперативного выполнения потоков на многоядерной системе / О. Н. Карасик, А. А. Прихожий // Системный анализ и прикладная математика. – 2017. – № 1. – С. 4–11.
7. Прыхожы, А. А. Кааператыўныя блочна-паралельныя алгарытмы рашэння задач на шмат'ядравых сістэмах / А. А. Прыхожы, А. М. Карасік // Сістэмны аналіз і прыкладная інфарматыка. – 2015. – № 2. – С. 10–18.
8. Прихожий, А. А. Кооперативная модель оптимизации выполнения потоков на многоядерной системе / А. А. Прихожий, О. Н. Карасик // Системный анализ и прикладная информатика. – 2014. – № 4. – С. 13–20.

9. Карасик, О. Н. Потоковый блочно-параллельный алгоритм поиска кратчайших путей на графе / О. Н. Карасик, А. А. Прихожий // Доклады БГУИР. – 2018. – № 2. – С. 77–84.
10. Прихожий, А. А. Исследование методов реализации многопоточных приложений на многоядерных системах / А. А. Прихожий, О.Н. Карасик // Информатизация образования. – 2014. – № 1. – С. 43–62.
11. Прихожий, А. А. Кооперативная потоковая модель решения задач большой размерности на многоядерных системах / А. А. Прихожий, О. Н. Карасик // Big Data and Advanced Analytics. – 2018. – № 4. – С. 381–386.
12. Probert, D. Dave Probert: Inside Windows 7 - User Mode Scheduler (UMS) [Электронный ресурс]. / D. Probert. – Режим доступа : https://channel9.msdn.com/shows/Going+Deep/Dave-Probert-Inside-Windows-7-User-Mode-Scheduler-UMS/. – Дата доступа : 01.11.2019.
13. Microsoft. User-Mode Scheduling [Электронный ресурс]. – Режим доступа : https://docs.microsoft.com/en-us/windows/desktop/ProcThread/user-mode-scheduling. – Дата доступа : 24.02.2020.
14. Howe, J. Parallel Gaussian Elimination. [Электронный ресурс]. – J. Howe, S. Bratcher. – Режим доступа : http://www.cse.ucsd.edu/classes/fa98/cse164b/Projects/PastProjects/LU. – Дата доступа : 01.11.2018.
15. Venkataraman, G. Blocked All-Pairs Shortest Paths Algorithm / G. Venkataraman, S. Sahni, S. Mukhopadhyaya // Journal of Experimental Algorithmics (JEA). – 2003. – Vol. 8. – P. 857–874.
16. Tang, P. Rapid development of parallel blocked all-pairs shortest paths code for multi-core computers / P. Tang // IEEE SOUTHEASTCON 2014. – Lexington, KY, USA: IEEE, 2014. – P. 1–7.
17. Park, J. Optimizing graph algorithms for improved cache performance / J. Park, M. Penner, V. K. Prasanna // IEEE Transactions on Parallel and Distributed Systems. – 2004. – Vol. 15. – №. 9. – P. 769–782.

Поступила в редакцию 13.03.2020

KARASIK A. N., PRIHOZHY A. A. Cooperative multi-thread scheduler for efficient solving applied tasks on multi-core systems

An architecture of the cooperative multi-thread scheduler for thread execution on a multi-core system that runs OS Windows is proposed. The architecture is implemented using the User Mode Scheduling mechanism, which allows the user application to organize cooperative thread execution. The architecture under development a user thread for executing user code, a new synchronization primitive for organizing the interaction of user threads on different cores, a control transfer mechanism between user threads on the same core. The architecture allows a programmer to implement cooperative multi-threaded algorithms to accelerate the solution of large-scale problems on multi-core systems.