МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ

(государственный университет)

ФАКУЛЬТЕТ РАДИОТЕХНИКИ И КИБЕРНЕТИКИ

КАФЕДРА ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

РЕАЛИЗАЦИЯ НАБОРА ПРАГМ ВЕКТОРИЗАЦИИ И ФУНКЦИОНАЛЬНОГО ПАРАЛЛЕЛИЗМА В КОМПИЛЯТОРЕ "ЭЛЬБРУС"

Выпускная квалификационная работа бакалавра

Студент: В. В. Антонов

Научный руководитель: С. М. Зотов

Распараллеливание

- **Векторизация**

  single instruction multiple data (SIMD)
- **Функциональный параллелизм**

  исполнение разного кода на разных потоках

В работе рассматривается стандарт OpenMP

- *совокупность прагм (директив) компилятора, библиотечных процедур и переменных окружения*
- *для систем с общей памятью*

Цель работы

Поддержка стандарта OpenMP в компиляторе «Эльбрус» путём реализации:

- прагм векторизации (`pragma omp simd`);
- прагм функционального параллелизма (`pragma omp task`, `pragma omp taskwait`);
- прагм функционального параллелизма в библиотеке runtime-поддержки.

Прагма векторизации (`simd`)

Описание

Прагма `simd` - подсказка компилятору, что следующий за прагмой цикл может быть векторизован.

```
#pragma omp simd [clause[[,] clause] ...] new-line
for-loops
```

Реализация прагмы векторизации (`simd`) стандарта OpenMP основана на применении прагмы векторизации (`vector`) компилятора «Эльбрус».

Поэтому были реализованы следующие опции (clause):

- `private;`
- `lastprivate;`
- `reduction.`

Прагма векторизации (`simd`)

Реализованные опции

- `private(list)`
  задание списка переменных, локального для данного потока
- `lastprivate(list)`
  аналогичен `private`, но переменные из списка на главном потоке присваиваются значениям переменных из последнего исполнившегося потока
- `reduction(reduction-identifier:list)`
  значение каждой переменной из списка на главном потоке определяется как результат последовательного применения операции `reduction-identifier` ко всем локальным переменным каждого потока и к значению данной переменной на главном потоке до прагмы

Прагма векторизации (simd)

Использование

`#pragma omp simd`

- векторизация цикла

`#pragma omp for simd`

- векторизация цикла
- разрезание цикла по итерациям

`#pragma omp parallel for simd`

- векторизация цикла
- разрезание цикла по итерациям
- создание потоков

```
void sum(int n, int* a, int* b, int* c)
{
    int i;
    #pragma omp parallel for simd num_threads(4)
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

Прагма векторизации (`simd`)

Реализация в компиляторе

Поиск прагмы `simd`

1) проход по всем процедурам во внутреннем представлении компилятора (EIR)
2) поиск прагм векторизации у каждой процедуры в EIR
3) создание хэш-таблиц операций `call`, содержащих прагму векторизации

Прагма векторизации (`simd`)

Реализация в компиляторе

Разбор прагмы

1) анализ правильности структуры прагмы в EIR
2) выдача предупреждения о неподдерживаемых опциях
3) переход на нужный вариант прагмы: `simd`, `for simd`, `parallel for simd`
4) определение точки в EIR, куда нужно вставить прагму `vector`
5) создание элемента прагмы `vector` в EIR и подстановка в найденную точку
6) изменение исходной прагмы в EIR
7) обработка опций, поддерживаемых только в прагме `simd`

Прагма векторизации (simd)

Реализация в компиляторе

Тестирование

```
short *A, *B, *C;

void f()
{
    ...
    #pragma omp simd
    for (i = 0; i < 1000; i++)
        C[i] = A[i] + B[i];
}

int main()
{
    ...
    for (i = 0; i < 100000000; i++)
        f();
    ...
}
```

Тестирование на 2-ядерном ВК "Монокуб" с тактовой частотой 500 МГц

Прагмы функционального параллелизма

Постановка задачи

- Разработка алгоритмов реализации функционального параллелизма для расширения возможностей библиотеки runtime-поддержки.
- Определение преобразования прагм функционального параллелизма стандарта OpenMP во внутреннем представлении компилятора.

Из полного списка прагм OpenMP для реализации были определены прагмы `task` и `taskwait`, формирующие базис для разработки приложений с использованием функционального параллелизма.

Прагмы функционального параллелизма

Общие понятия

- В текущей версии реализации стандарта OpenMP в компиляторе «Эльбрус» распараллеливание обеспечивается прагмой `parallel`.
  - данная прагма создаёт параллельный регион из нескольких потоков (threads)
  - каждый поток исполняет один и тот же код – структурный блок, расположенный следом за прагмой (**параллелизм данных**)
- Прагма `task`, описанная в стандарте OpenMP, задаёт **функциональный параллелизм -** структурный блок, расположенный следом за этой прагмой, исполняется одним из параллельных потоков**.** Т.о. разные потоки исполняют разный код.
- Прагма `task` использует потоки, созданные прагмой `parallel` и поэтому имеет смысл только внутри параллельного региона. Если же она используется вне параллельного региона, то структурный блок будет исполняться последовательно.

Прагмы функционального параллелизма

Общие понятия

- Модель функционального параллелизма в стандарте OpenMP основана на работе потоков с задачами.
- Задача (task) – особый экземпляр кода и его данных.

Структура задачи (task)

- Различают явную и неявную задачи.
- Явная задача генерируется потоком, встретившим прагму `task`.
- После генерации задачи поток ставит её в очередь задач на исполнение.
- Каждый поток имеет свою очередь задач.
- Поток занимается исполнением задач из очереди в точках планирования – местах в программе, в которых происходит синхронизация потоков или создание задачи.
- Во время исполнения некоторой задачи поток может сгенерировать ещё одну задачу. Сгенерированная задача называется дочерней задачей, а исполняемая - родителем.
- Очередь задач хранит иерархию задач.

Прагмы функционального параллелизма

Общие понятия

- В начале очереди всегда хранится неявная задача. Неявная задача генерируется для каждого потока в исходной последовательной программе при встрече прагмы `parallel`. Код неявной задачи совпадает с кодом структурного блока прагмы `parallel`.

- В начальном элементе очереди указатель на родителя `null`.

Элемент очереди

Прагмы функционального параллелизма

Описание

Прагма `task`:

- определяет явную задачу;
- задаёт точку планирования.

```
#pragma omp task [clause[[,] clause] ...] new-line
structured-block
```

Реализуемые опции:

- `if(`*`scalar-expression`*`)` – *если* `false`*, то исполняется сразу*
- `final(`*`scalar-expression`*`)` – *если* `true`*, то генерируется финальная задача – задача, требующая безотлагательного исполнения*
- `default(shared | none)` – *определяет все переменные задачи разделяемыми или требует указания атрибутов совместного использования внутри задачи*
- `firstprivate(`*`list`*`)` – *задаёт список переменных, локальный для данного потока, с инициализацией значениями из главного потока*
- `shared(`*`list`*`)` – *задаёт список разделяемых переменных*

Прагма `taskwait`:

- устанавливает ожидание завершения дочерних задач текущей задачи;
- задаёт точку планирования.

```
#pragma omp taskwait newline
```

Прагмы функционального параллелизма

Пример

```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

Прагмы функционального параллелизма

Пример

```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

потоки (1):

- master_thread

очереди задач:

- отсутствует

Прагмы функционального параллелизма

Пример

```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

Создан ещё один поток (`thread_1`).

Созданы 2 очереди задач, в каждую из которых помещены неявные задачи.

код **`master_thread_implicit_task`** и **`thread_1_implicit_task`** :

```
#pragma omp master
{
    int x = 1, y = 2;
    #pragma omp task
        task1(x);
    #pragma omp task
        task2(y);
}
```

потоки (2):

- `master_thread`
- `thread_1`

очереди задач (2):

`master_thread_implicit_task`	
null	null

`thread_1_implicit_task`	
null	null

Прагмы функционального параллелизма

Пример

```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

Поток `thread_1`, исполнив `thread_1_implicit_task`, перешёл в точку синхронизации.

потоки (2):

- `master_thread`
- `thread_1`

очереди задач (2):

master_thread_implicit_task	
null	null

thread_1_implicit_task	
null	null

Прагмы функционального параллелизма

Пример

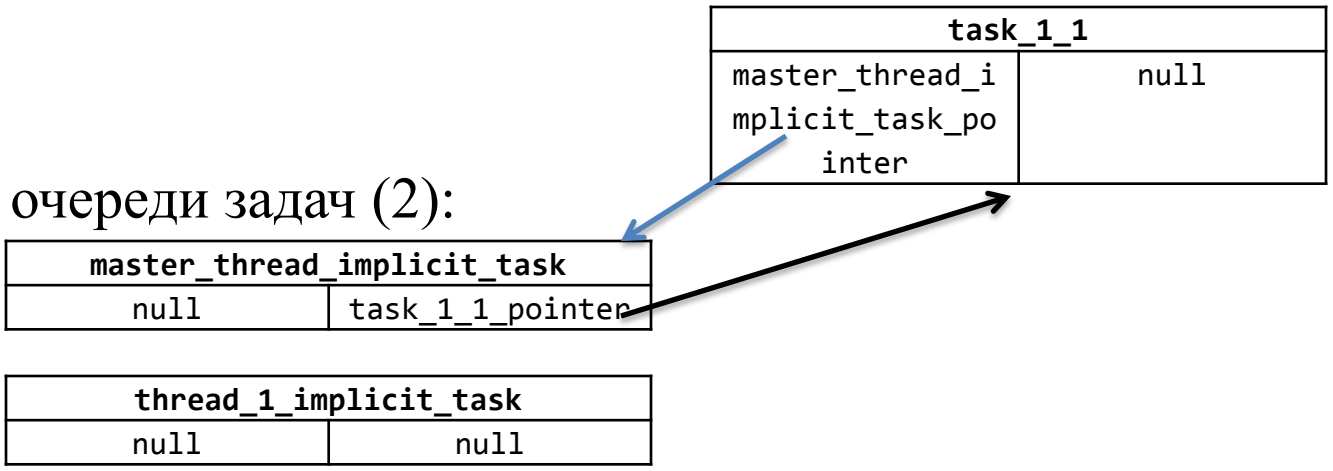
```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

Поток `master_thread` сгенерировал явную задачу `task_1_1`, состоящую из функции `task1()`.

потоки (2):

- `master_thread`
- `thread_1`

очереди задач (2):



Прагмы функционального параллелизма

Пример

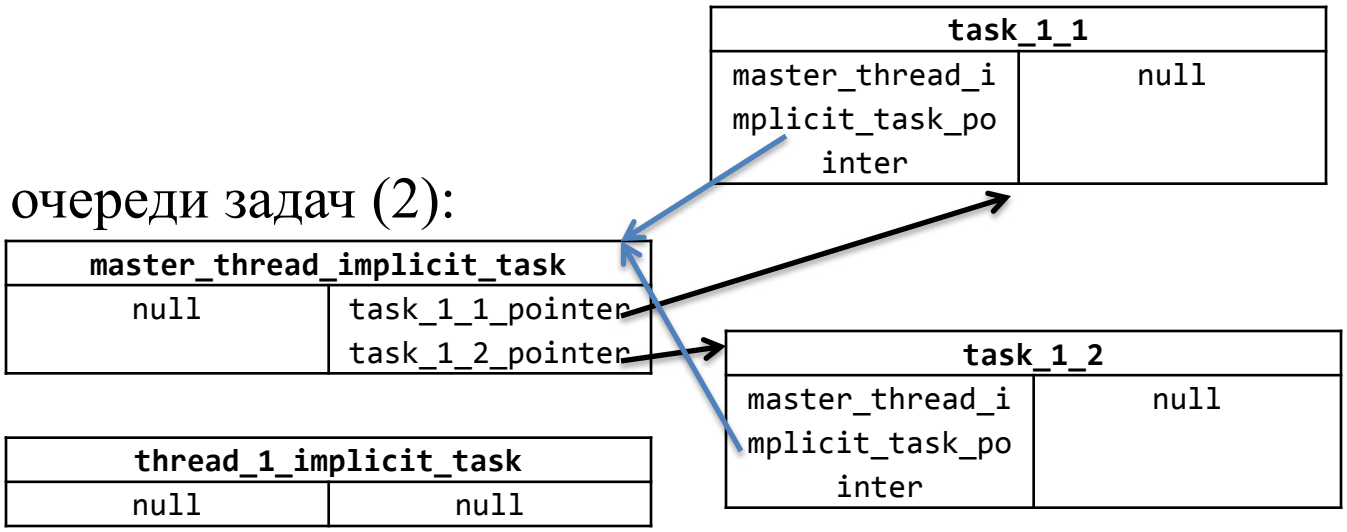
```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

Поток `master_thread` сгенерировал явную задачу `task_1_2`, состоящую из функции `task2()`.

потоки (2):

- `master_thread`
- `thread_1`

очереди задач (2):



Прагмы функционального параллелизма

Пример

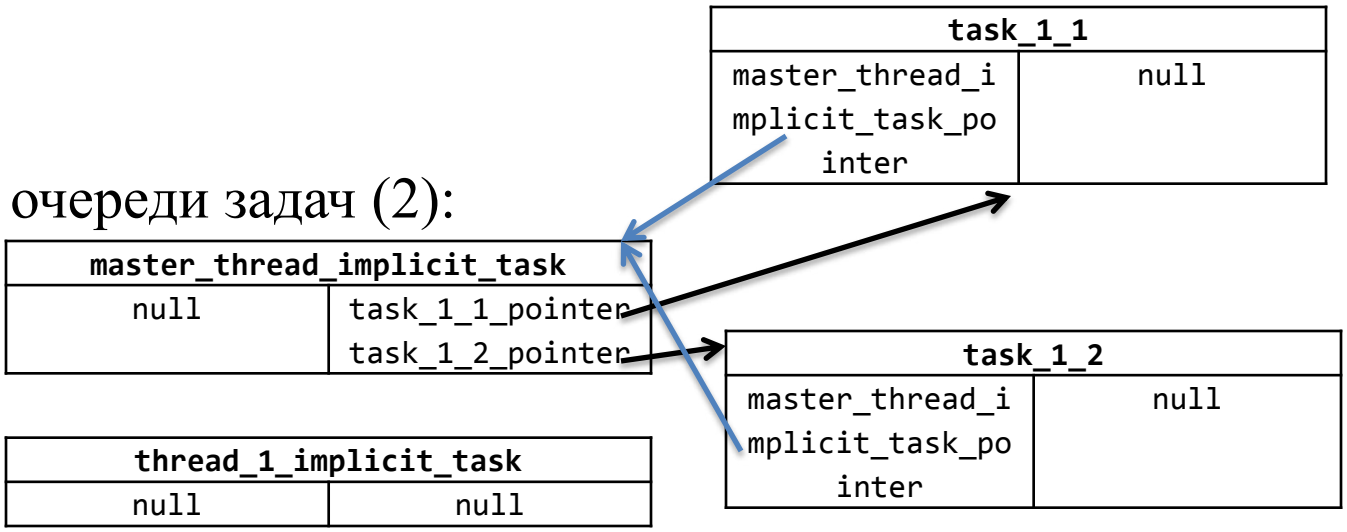
```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

Поток `master_thread`, исполнив `master_thread_implicit_task`, перешёл в точку синхронизации.

потоки (2):

- `master_thread`
- `thread_1`

очереди задач (2):



Прагмы функционального параллелизма

Алгоритм выбора задач из очереди на исполнение

Прагмы функционального параллелизма

Пример

```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

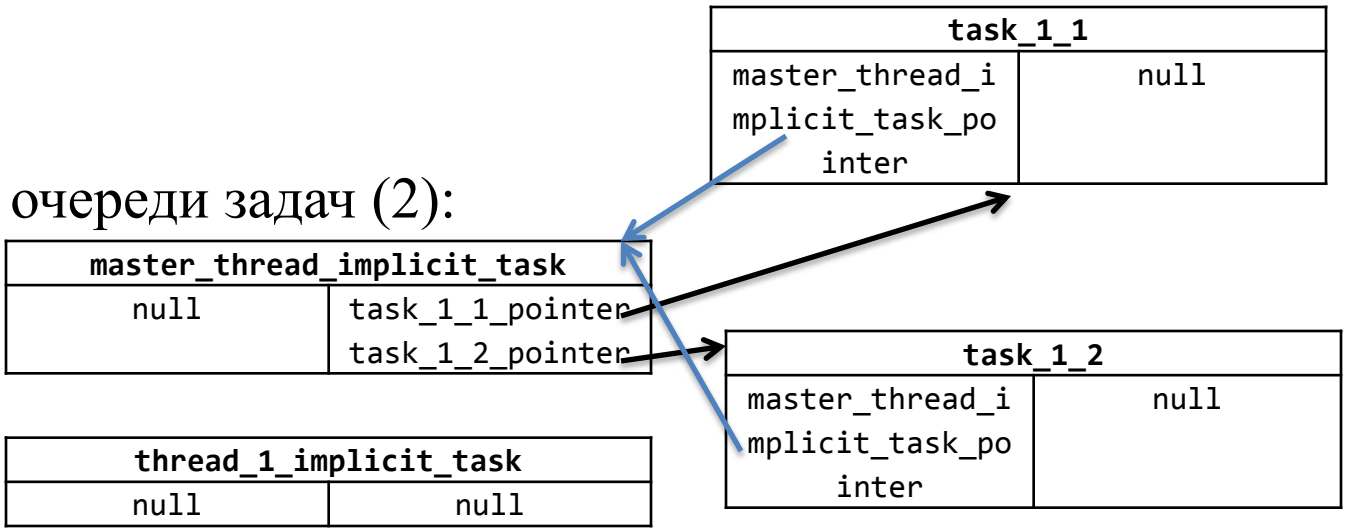
Поток `thread_1`, находясь в точке синхронизации, начал исполнять задачу из очереди потока `master_thread - task_1_1.`

Поток `master_thread` в это время дошёл до точки синхронизации, увидел, что в его очереди осталась только задача `task_1_2` и начал её исполнение.

потоки (2):

- `master_thread`
- `thread_1`

очереди задач (2):



Прагмы функционального параллелизма

Пример

```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

Когда очереди всех потоков оказались пустыми, потоки завершают свою работу, очереди задач больше не нужны.

потоки (2):

- master_thread
- thread_1

очереди задач (2):

master_thread_implicit_task	
null	null

thread_1_implicit_task	
null	null

Прагмы функционального параллелизма

Пример

```
void task1(int num)
{...}
void task2(int num)
{...}
...
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        int x = 1, y = 2;
        #pragma omp task
            task1(x);
        #pragma omp task
            task2(y);
    }
}
...
```

потоки (1):

- master_thread

очереди задач:

- отсутствует

Прагмы функционального параллелизма

Новые функции библиотеки runtime-поддержки

- `__omp_create_task()`
  - создаёт явную задачу и помещает её в очередь;
  - исполняет задачу из очереди в случае переполнения очереди;
  - возвращает значение типа `bool`: `true`, если исполнение идёт в параллельном регионе; `false` иначе.
- `__omp_taskwait()`
  - просматривает очередь задач и ожидает завершения дочерних задач данной задачи;
  - выбирает и исполняет задачу из очереди во время ожидания.

Прагмы функционального параллелизма

Модифицируемые функции библиотеки runtime-поддержки

```
#pragma omp parallel
{
    region;
}
```



```
{
    setup data;
    __omp_parallel_start (par_function, &data);
    par_function (&data);
    __omp_parallel_end ();
}
```

- `par_function (&data)` – функция параллельного региона
- `data` – данные параллельного региона

```
#pragma omp barrier
```



```
__omp_barrier();
```

Нужно модифицировать следующие функции библиотеки runtime-поддержки:

- `__omp_parallel_start()`
- `__omp_parallel_end()`
- `__omp_barrier()`

Прагмы функционального параллелизма

Модифицируемые функции библиотеки runtime-поддержки

- `__omp_parallel_start()`
  - создание структуры очередей задач;
  - создание неявных задач и занесение их в очереди.
- `__omp_parallel_end()`
  - проверка очередей задач: если очереди не пустые, исполнить задачу, иначе - перейти к основному алгоритму.
- `__omp_barrier()`
  - проверка очередей задач: если очереди не пустые, исполнить задачу, иначе – продолжить выполнение.

Прагмы функционального параллелизма

Преобразование в компиляторе новых прагм

```
{
    [context 1]
    #pragma omp task
    {
        task_body;
    }
    [context 2]
}
```

```
void body_function(void *data)
{
    get data;
    task_body;
}
...
{
    [context 1]
    if (!__omp_create_task(&body_function, &data, final))
        body_function(&data);
    [context 2]
}
```

```
#pragma omp taskwait
```

```
__omp_taskwait();
```

- `body_function(…)` – генерируемая компилятором функция, состоящая из кода блока прагмы `task` и принимающая на вход указатель на данные
- `__omp_create_task(…)`, `__omp_taskwait()` – новые функции библиотеки runtime-поддержки

Результаты

- В компилятор «Эльбрус» внедрена прагма векторизации (`simd`) стандарта OpenMP в сочетании со следующими опциями: `private`, `lastprivate` и `reduction`. Прагма может быть использована в различных комбинациях с другими прагмами.
- Получено 17% ускорение тестовой программы с использованием прагмы векторизации (`simd`) в сравнении с аналогичной программой без использования данной прагмы.
- Определено преобразование прагм функционального параллелизма стандарта OpenMP во внутреннем представлении компилятора "Эльбрус".
- Предложен алгоритм реализация функционального параллелизма для библиотеки runtime-поддержки компилятора «Эльбрус».