РАСПАРАЛЛЕЛИВАНИЕ И ОПТИМИЗАЦИЯ ПРОГРАММ С ПОМОЩЬЮ WEB-УСКОРИТЕЛЯ ОРС

Е.В. Алымова, Е.Н. Кравченко, Р.И. Морылев, М.В. Юрушкин, Б.Я. Штейнберг

Аннотация

В работе представлен высокоуровневый конвертор, предназначенный для ускорения программ. Ускорение достигается за счет нескольких технологий, разработанных на мехмате Южного федерального университета. Конвертор снабжен веб-интерфейсом и эффективность разработанных технологий демонстрируется в сети Интернет.

1. **Введение**

В работе представлен ускоритепль высокоуровневых программ (конвертор) с веб-интерфейсом [1]. Конвертор для ускорения программ использует несколько оригинальных технологий. Проект может быть использован для обучения параллельному программированию и для распараллеливания и оптимизации небольших исследовательских или промышленных программ.

Проблемы создания программного обеспечения, адекватно использующего возможности архитектуры современных и проектируемых в настоящее время вычислительных систем отмечаются в [2], [3]. Технологии ускорения программ, используемые в данном проекте, могут лечь в основу распараллеливающих компиляторов для современных высокопроизводительных компьютеров.

Данный проект основан на ДВОР (Диалоговом высокоуровневом оптимизирующем распараллеливателе программ), который, в свою очередь, является развитием ОРС (Открытой распараллеливающей системы) [4].

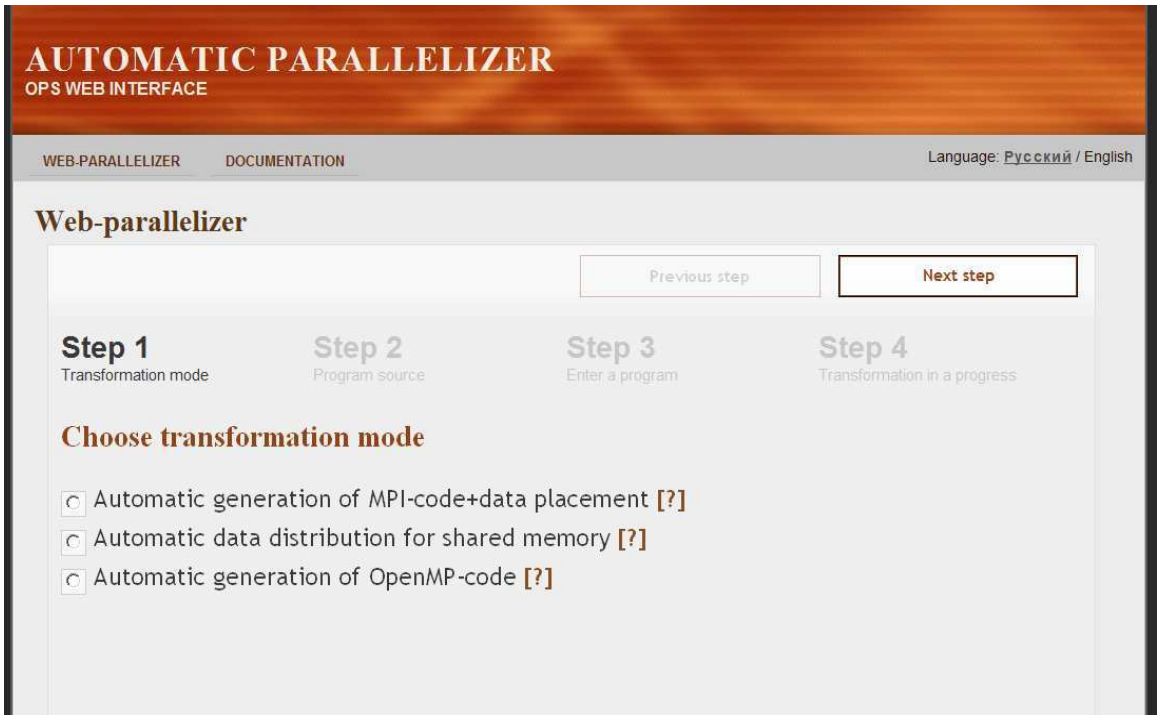

Рис. 1 Главная страница Web-Ускорителя

Проект представляет собой высокоуровневый конвертор. На вход Web-Ускорителя принимаются программы на языке С, соответствующие стандарту С99. Программа должна состоять из одного файла. Если входная программа включает нестандартные заголовочные файлы, то перед загрузкой следует включить их исходный код в текст программы с помощью препроцессора. Для использования технологий ускорения следует в начале программы писать директивы компиляции в виде прагм. Каждая из разработанных технологий ускорения эффективна на некотором классе программ. Эти классы программ выглядят узкими. Но среди программ, требующих больших объемов вычислений, для которых и нужны ускорения, эти классы являются достаточно представительными. Реализованы следующие возможности ускорения программ:

1. Автоматическое распределение данных в оперативной памяти для оптимального использования кэш-памяти;

2. Автоматическое распараллеливание программы с автоматическим размещением массивов в распределенной памяти с перекрытиями для минимизации пересылок данных;

3. Автоматическое распараллеливание программ на общую память (OpenMP).
На сайт выложены демонстрационные примеры, иллюстрирующие работу системы.

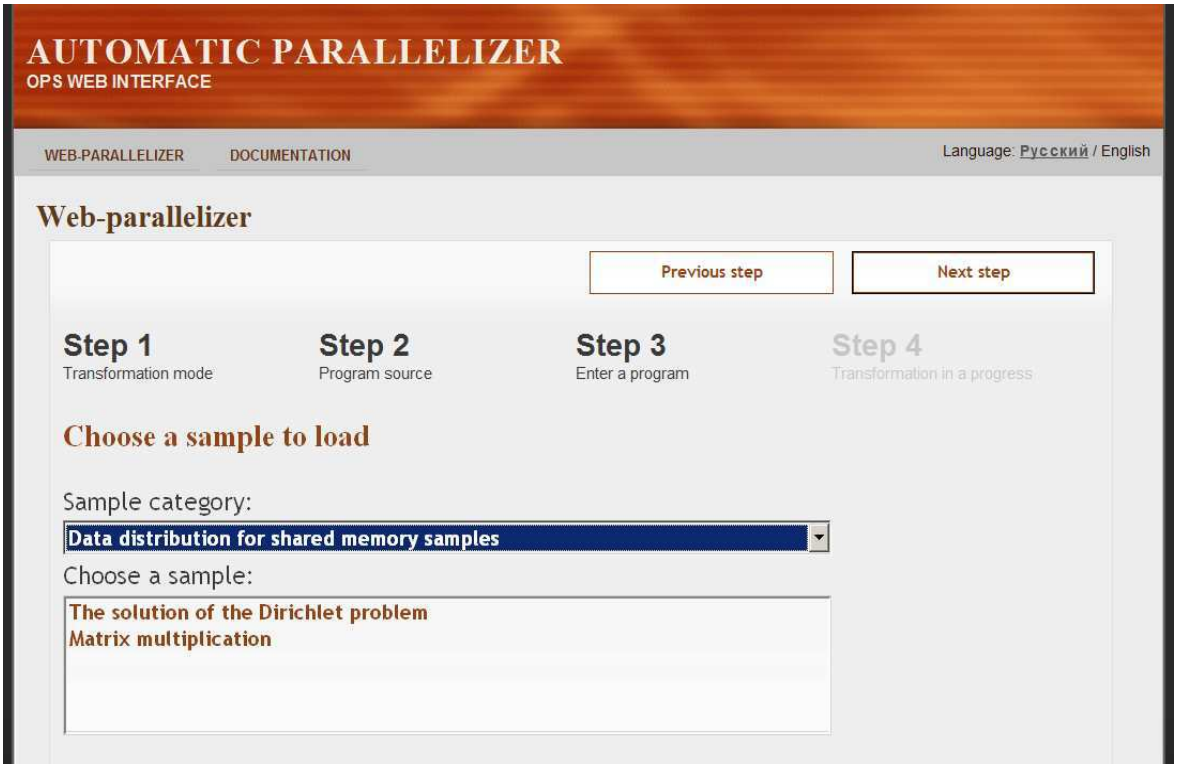

Рис. 2 Библиотека демонстрационных примеров.

2. Блочно-аффинные размещения данных

Практически все реализованные технологии используют специальные размещения данных в памяти. В основе размещения массивов в распределенной памяти лежат блочно-аффинные размещения данных [5], [6]. Приведем основные понятия о таких размещениях.

Размещением m-мерного массива X[1..N_1; ...; 1..N_m] будем называть функцию, которая каждому набору индексов I=(I_1, ..., I_m), (1 <= I_1 <= N_1, ..., 1 <= I_m <= N_m) ставит в соответствие номер модуля памяти, в котором должен находиться элемент массива X(I_1; ..., I_m).

Пусть натуральные числа p, d_1, ..., d_m, целые неотрицательные числа q_1, ..., q_m, r_1, ..., r_m и целые числа s_0, s_1, ..., s_m зависят только от m-мерного массива X. Блочно-аффинное по модулю p размещение m-мерного массива X с перекрытиями - это такое размещение, при котором элемент X(I_1 + t_1, ..., I_m + t_m) для любых целых t_i, i = 1..m, удовлетворяющих условию -q_i <= I_i <= r_i, находится в модуле памяти с номером u=(]I_1/d_1[* s_1 + ... +]I_m/d_m[* s_m + s_0) mod p.

Величины перекрытий определяются числами q_1, ..., q_m, r_1, ..., r_m.

3. Автоматическое распределение данных для оптимизации использования кэш-памяти

Реализовано расширение языка Си (C99) операциями работы с памятью [7]. Оно позволяет производить распределение матриц по строкам, по столбцам и по блокам. Данные способы распределения матриц обобщаются на массивы произвольной размерности. Произведены численные эксперименты, в которых сравнивалась производительность различных блочных алгоритмов (перемножение матриц, возведение матрицы в квадрат, LU-разложение матрицы) при различных способах распределения массивов. Распределение массивов по блокам повышает производительность программы до 80% по сравнению со стандартными методами распределения массивов, используемых компиляторами. Для проведения экспериментов использовался компилятор Intel C/C++ v12.1.

Расширение реализовано в виде директивы компилятора, в которой указываются параметры разбиения (размеры блоков) и количество процессоров на целевой архитектуре.

#pragma ops distribute data(<Имя размещаемого массива>, <dimi>, P, <di>)

dimi - список, верхние границы массива;
P - количество процессоров (равно 1 в случае оптимизации под кэш-память);
di - список, размеры блока;

Директиву нужно вставлять после инициализации размещаемого массива и до начала работы с ним. Все параметры директивы должны быть константами времени компиляции.

Пример 1. Реализация алгоритма ускорения блочного умножения матриц (A=B*C) с использованием директивы распределения данных. A, B, C - матрицы размера N×N. d - размер блока. double**A, **B, **C;

```
...
#pragma ops distribute data(A, N, N, 1, d, d)
#pragma ops distribute data(B, N, N, 1, d, d)
#pragma ops distribute data(C, N, N, 1, d, d)

for (bi = 0; bi < blockCount; bi++)
for (bj = 0; bj < blockCount; bj++)
for (bk = 0; bk < blockCount; bk++)
        for (i = 0; i < d; i++)
        for (j = 0; j < d; j++)
        for (k =0; k <d; k++)
                C[bi*d+i][bj*d+j] += A[bi*d+i][bk*d+k]*B[bk*d+k][bj*d+j];
```

Конец примера 1.

Блочное распределение массивов не накладывает никаких синтаксических ограничений на входную программу, т.к. сами операции после распределения и их порядок следования не изменяются. Для того, чтобы программа ускорилась после применения блочного распределения массивов, необходимо, чтобы индексные выражения, встречающиеся во вхождениях распределяемых массивов, имели вид (di*d1+i, dj*d2+j), где (d1, d2) – размер блока; 0<= i < d1, 0 <= j < d2. Реализовано специальное преобразование, которое проверяет входную программу на соответствие индексов во вхождениях распределяемых массивов требуемому виду, и, если необходимо, преобразует их.

Например, программа содержащая блок

```
 for (i = 0; i < N1; i++)
 for (j = 0; j < N2; j++)
  A[i][j] = i + j;
```

будет автоматически преобразована к следующему виду:

```
for (di = 0; di < N1/d1; di++)
for (dj = 0; dj < N2/d2; dj++)
 for (i = 0; i < d1; i++)
 for (j = 0; j < d2; j++)
  A[di*d1+i][dj*d2+j] = (di*d+i) + (dj*d+j);
```

4. Автоматическая генерация MPI-кода с размещением данных

Для эффективного распараллеливания последовательного алгоритма на вычислительную систему с распределенной памятью необходимо учитывать множество факторов. Одним из таких факторов является размещение данных. Размещение данных в вычислительной системе с распределенной памятью позволяет, во-первых, минимизировать объем используемой памяти, а во-вторых, минимизировать накладные расходы на выполнение пересылок данных.

В Диалоговом высокоуровневом оптимизирующем распараллеливателе (ДВОР) размещение данных реализовано с помощью расширения языка Си [8]. Данные размещаются в соответствии с некоторым блочно-аффинным размещением с перекрытиями [9]. Для автоматического распараллеливания с размещением данных средствами ДВОР исходная программа должна удовлетворять ряду требований. Одним из самых существенных требований является запрет на передачу распределяемых данных подпрограмме в качестве параметра.

Для правильной работы преобразования необходимо после строки подключения заголовочного файла mpi.h включить следующий фрагмент исходного кода:

```
int __MPI_COMM_WORLD = MPI_COMM_WORLD;
int __MPI_COMM_SELF  = MPI_COMM_SELF;

int __MPI_CHAR = MPI_CHAR;
int __MPI_INTEGER1 = MPI_INTEGER1;
int __MPI_INTEGER2 = MPI_INTEGER2;
int __MPI_INTEGER4 = MPI_INTEGER4;
```

```
int __MPI_INTEGER8 = MPI_INTEGER8;

int __MPI_REAL4 = MPI_REAL4;
int __MPI_REAL8 = MPI_REAL8;
```

Расширение языка Си

Для автоматической генерации параллельного MPI кода с размещением данных в ДВОР в язык Си были добавлены дополнительные директивы:

```
#pragma ops distribute <storage_rank>
#pragma ops ignore
#pragma ops single_access <rank>
#pragma ops distribute data(<parameters>)
```

Директива #pragma ops distribute <storage_rank>

Данная директива должна быть помещена непосредственно перед объявлением динамического массива. Директива указывает ДВОР, что помеченный ею динамический массив является распределенным. При этом память под помеченный массив выделяется только на узле с номером <rank>.

Пример 2. Использование директивы #pragma ops distribute <storage_rank>. Массивы X и Y являются распределенными. Память под массив X выделяется только на узле с номером 0, под массив Y - на узле с номером 1.

```
...
#pragma ops distribute 0
int *X;
#pragma ops distribute 1
int **Y;
...
```

Конец примера 2.

Директива #pragma ops ignore

Данная директива должна быть помещена непосредственно перед объявлением переменной. Директива указывает ДВОР, что изменение значений помеченной ею переменной не должно учитываться при распараллеливании.

Пример 3. Использование директивы #pragma ops ignore. Изменение значений переменных i и j игнорируется при распараллеливании.

```
...
#pragma ops ignore
int i, j;
...
```

Конец примера 3.

Директива #pragma ops single_access <rank>

Данная директива должна быть помещена непосредственно перед оператором в программе. Директива указывает ДВОР, что помеченный ею оператор после распараллеливания должен выполняться только процессом с номером <rank>.

Пример 4. Использование директивы #pragma ops single_access. В результате автоматического распараллеливания оператор, помеченный директивой #pragma ops single_access будет выполняться только процессом с номером 1.

```
...
#pragma ops distribute 1
double *Y;
...
#pragma ops single_access 1
{
        Y = (double*)malloc(10*sizeof(double));
        (i = 0; i < 10; i = i + 1)
        {
                Y[i] = 0;
        }
```

```
}
...
```
Конец примера 4.

Директива #pragma ops distribute data(<parameters>)

<parameters> представляет из себя строку вида:
"X, SR, GR, P, m, dim_1, ..., dim_m, d_1, ..., d_m, s_0, s_1, ..., s_m, t_1, ..., t_m",
где:
X - имя размещаемого массива
SR - признак необходимости рассылки данных
GR - признак необходимости сбора данных
P - число процессов
m - размерность массива X
dim_1, ..., dim_m - размеры массива X по соответствующей координате
d_1, ..., d_m, s_0, s_1, ..., s_m - параметры блочно-аффинного размещения данных по модулю P
t_1, ..., t_m - константы удовлетворяющие условию: после выполнения оператора, помеченного директивой #pragma ops distribute data() актуальный элемент массива X(I_1, …, I_m) хранится в процессе с номером u=(](I_1 - t_1)/d_1[* s_1 + ... +](I_m - t_m)/d_m[* s_m + s_0) mod P.

Данная директива должна быть помещена непосредственно перед оператором в программе. Директива указывает ДВОР, что в процессе выполнения помеченного ею оператора распределенный массив X должен быть размещен в соответствии с блочно-аффинным размещением данных по модулю P с перекрытиями, определяемым параметрам d_1, ..., d_m, s_0, s_1, ..., s_m. При этом величина перекрытий определяется автоматически.

Пример 5. Использование директивы #pragma ops distribute data(<parameters>).

В результате автоматического распараллеливания внутри оператора, помеченного директивой #pragma ops distribute data(X, 1, 1, 2, 1, 10, 5, 1, 0, 0), массив X будет размещен в соответствии с блочно-аффинным размещением данных с перекрытиями (d_0 = 5, s_0 = 1, s_1 = 0, q_0 = 0, r_0 = 1). ...

```
#pragma ops distribute
int *X;


...
#pragma ops distribute data(X, 1, 1, 2, 1, 10, 5, 1, 0, 0)
{
        ...
        ... X[i] ...
        ... X[i+1] ...
        ...
}
```
Конец примера 5.

Директива #pragma ops parallel_loop_nesting

Данная директива должна быть помещена непосредственно перед оператором цикла FOR в программе. Директива указывает ДВОР, что итерации помеченного цикла следует выполнять параллельно. При этом на узле с номером rank выполняются итерации с номерами из диапазона [rank*N/size, (rank + 1)*N/size - 1], где size - число процессов, N - исходное число итераций.

Пример 6. Использование директивы #pragma ops parallel_loop_nesting. В результате автоматического распараллеливания итерации помеченного цикла будут выполняться параллельно.

```
...
#pragma ops parallel_loop_nesting
for (i = 0; i < 10000; i = i + 1)
{
        ...
}
...
```
Конец примера 6.

5. Автоматическая генерация OpenMP кода

На основе анализатора распараллеливаемости циклов ДВОР реализована автоматическая генерация OpenMP кода [10]. Данный генератор находит в тексте программы все циклы, которые можно выполнять параллельно с помощью OpenMP и вставляет в код соответствующие прагмы.

Кроме определения возможности параллельного выполнения используются следующие вспомогательные виды анализа и расстановка соответствующих клауз OpenMP:

нахождение приватизируемых переменных (private(list));

определение редукции (reduction(op,list)).

Пример 7. Перемножение матриц

Исходное гнездо циклов:

```
for (i=0; i<N; i++)
   for(j=0; j<N; j++)
     for (k=0; k<N; k++)
       C[i][j] += A[i][k] * B[k][j];
```

Сгенерированный код:

```
#pragma omp parallel for
 for (i = 0; i < 1000; i = i + 1)
 {
   #pragma omp parallel for
   for (j = 0; j < 1000; j = j + 1)
   {
     for (k = 0; k < 1000; k = k + 1)
     {
       C[i][j] = C[i][j] + A[i][k] * B[k][j];
     }
   }
 }
```

Конец примера 7.

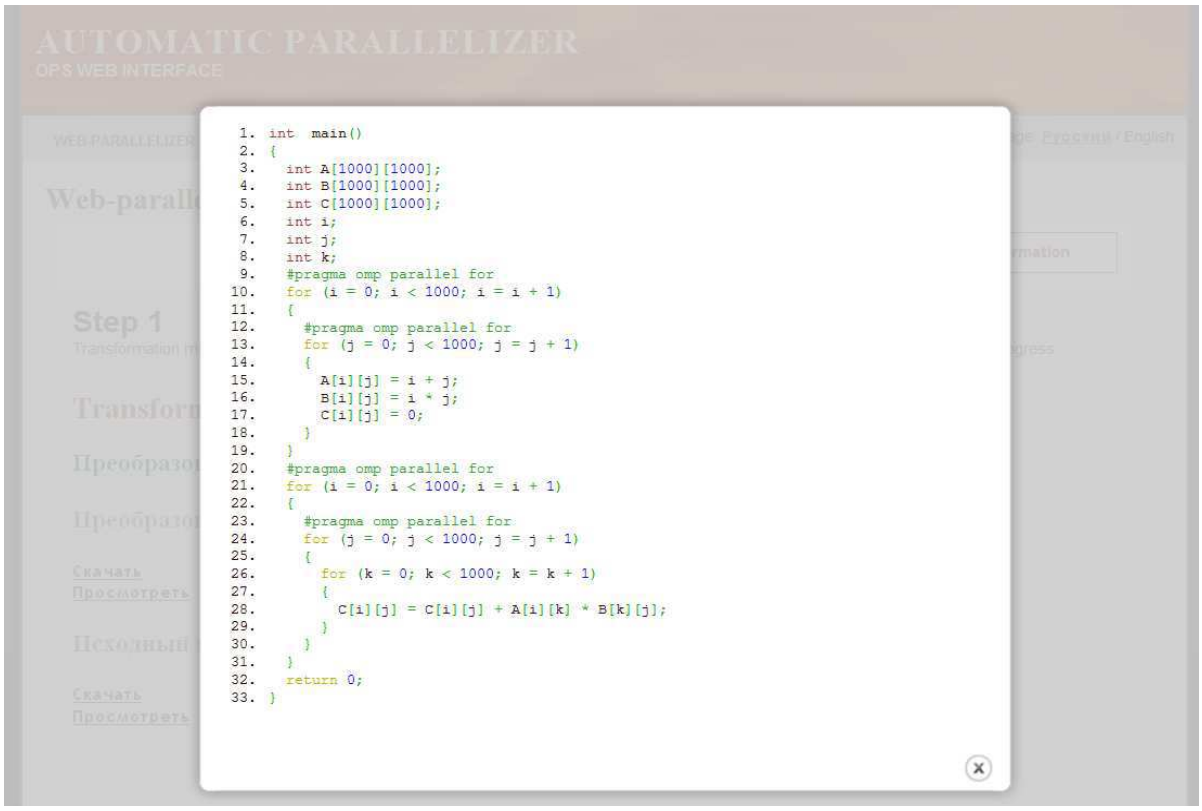

Рис. 3 Результат, полученный при автоматической генерации OpenMP-кода для задачи умножения матриц.

6. Дальнейшее развитие

Предполагаются следующие направления развития данного веб-распараллеливателя:

a) Расширение класса оптимизируемых программ за счет преобразования программных конструкций к шаблонному виду, в котором производится оптимизация (распараллеливание).
b) Оптимизирующее распараллеливание программ на вычислительные системы с распределенной памятью с учетом топологии соединия процессорных элементов, в том числе и с иерархией памяти.
c) Автоматическое преобразование последовательных программ (без специальных директив) в программы с описанными в статье директивами..
d) Автоматическое распараллеливание программ на параллельные, графические и другие ускорители.

ЛИТЕРАТУРА:

1. Веб-интерфейс автоматического распаралелливателя программ http://ops.opsgroup.ru/ Дата обращения 11 мая 2012 г.
2. Mycroft A. Programming Language Design and Analysis motivated by Hardware Evolution (invited Presentation) http://www.cl.cam.ac.uk/~am21/papers/sas07final.pdf Дата обращения 11 мая 2012 г.
3. Галушкин А.И. Стратегия развития современных суперкомпьютеров на пути к экзафлопным вычислениям. Приложение к журналу «Информационные технологии», №2, 2012 г.
4. Оптимизирующая распараллеливающая система www.ops.rsu.ru Дата обращения 11 мая 2012 г.
5. Штейнбер Б.Я. Оптимизация размещения данных в параллельной памяти. Ростов-на-Дону, изд-во Южного федерального университета, 2010, 255 с.
6. Штейнберг Б.Я. Блочно-аффинные размещения данных в параллельной памяти. «Информационные технологии», М.: из-во «Новые технологии», №6, 2010 г., с. 36-41.
7. Юрушкин М.В. Реализация алгоритмов распределения данных под общую и распределенную память в системе ДВОР. Современные проблемы математического моделирования. Труды XIV молодежной конференции-школы с международным участием. Южный федеральный университет, (ЮФУ, г. Ростов-на-Дону), Южно-Российский региональный центр информатизации ЮФУ, (ЮГИНФО ЮФУ, г. Ростов-на-Дону), Институт прикладной математики им. М.В. Келдыша РАН (ИПМ РАН, г. Москва), 12-17 сентября 2011 г., пос. Дюрсо, с. 396-400.
8. Кравченко Е.Н. Автоматическая генерация MPI-кода с размещением данных в диалоговом высокоуровневом автоматическом распараллеливателе. Современные проблемы математического моделирования. Труды XIV молодежной конференции-школы с международным участием. Южный федеральный университет, (ЮФУ, г. Ростов-на-Дону), Южно-Российский региональный центр информатизации ЮФУ, (ЮГИНФО ЮФУ, г. Ростов-на-Дону), Институт прикладной математики им. М.В. Келдыша РАН (ИПМ РАН, г. Москва), 12-17 сентября 2011 г., пос. Дюрсо, с. 193-197.
9. Гервич Л.Р., Штейнберг Б.Я. Параллельное итерационное умножение ленточной матрицы на вектор. Труды научной школы И.Б. Симоненко. Сборник статей. Ростов-на-Дону, изд.-во ЮФУ, 2010. с. 58-66.
10. Штейнберг Б.Я., Алымова Е.В., Баглий А.П., Гуда С.А., Кравченко Е.Н., Морылев Р.И., Нис З.Я., Петренко В.В., Скиба И.С., Шаповалов В.Н., Штейнберг О.Б. Особенности реализации распараллеливающих преобразований программ в ДВОР. РАСО’2010/ Труды международной конференции «Параллельные вычисления и задачи управления». М., 26-28 октября 2010 г., ИПУ РАН, с.787-854