Практическое задание №1
Многопоточная реализация солвера BiCGSTAB для СЛАУ с разреженной матрицей, заданной в формате CSR

Введение

Солверы систем линейных алгебраических уравнений (СЛАУ) широко применяются в суперкомпьютерном моделировании. Большие разреженные системы возникают в расчетах сеточными методами.

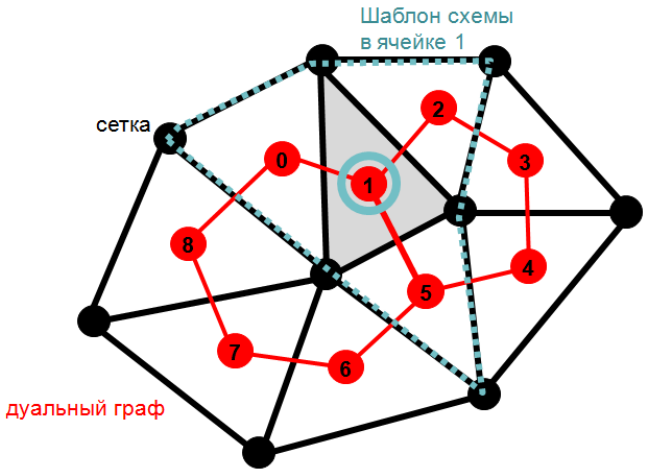
Метод BiCGSTAB хорошо подходит в качестве практического задания, поскольку он имеет достаточно простой алгоритм и использует всего три базовые операции линейной алгебры. При этом теория, на которой данный метод основывается, сложна для понимания.

Дискретная аппроксимация разностных операторов на пространственных сетках определяет топологию матрицы СЛАУ, т.е. её портрет. Портретом матрицы A называется множество пар индексов строк и столбцов (i,j), для которых соответствующий коэффициент $a_{ij} \neq 0$.

Как правило, портрет матрицы совпадает с графом связей расчетных ячеек сетки. Вершинам графа сопоставлены расчетные ячейки – узлы сетки или сеточные элементы сетки (в зависимости от того, где заданы сеточные функции). Наличие ребра между двумя вершинами графа означает, что шаблон численной схемы пространственной дискретизации, центрированный в одной из ячеек, включает другую ячейку.

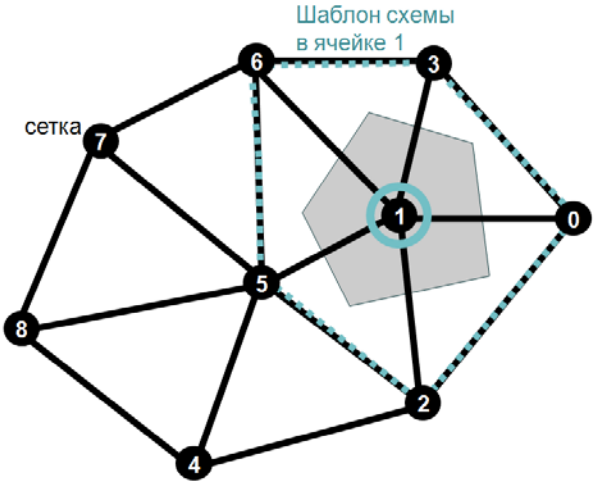
На рис. 1 показан пример двухмерной треугольной сетки и ее дуального графа, а также портрет матрицы, соответствующий этому примеру пространственной дискретизации с определением сеточных функций в элементах сетки. Аналогичный пример для случая, когда сеточные функции заданы в узлах, показан на рис. 2.

Значения коэффициентов матрицы определяются используемым численным методом. Способ их расчета в рамках данного задания нас не интересует. Портрет матрицы и значения ее коэффициентов являются входными данными для солвера.



	0	1	2	3	4	5	6	7	8
0	#	#							#
1	#	#	#			#			
2		#	#	#					
3			#	#	#				
4				#	#	#			
5		#			#	#	#		
6						#	#	#	
7							#	#	#
8	#							#	#

Рис 1. Пример треугольной сетки с определением сеточных функций в элементах (треугольниках). В этом примере шаблон схемы в ячейке состоит из этой ячейки и ее соседних ячеек, т.е. ячеек, имеющих с данной ячейкой общую грань. Справа показан портрет матрицы. Номера в узлах обозначают номера неизвестных в векторе. Номер элемента в сетке (красные кружки) соответствует номеру строки матрицы. Номера столбцов в строке узла соответствуют номерам соседних с ним элементов.



	0	1	2	3	4	5	6	7	8
0	#	#	#	#					
1	#	#	#	#		#	#		
2	#	#	#		#	#			
3	#	#		#			#		
4			#		#	#			#
5		#	#		#	#	#	#	#
6		#		#		#	#	#	
7						#	#	#	#
8					#	#		#	#

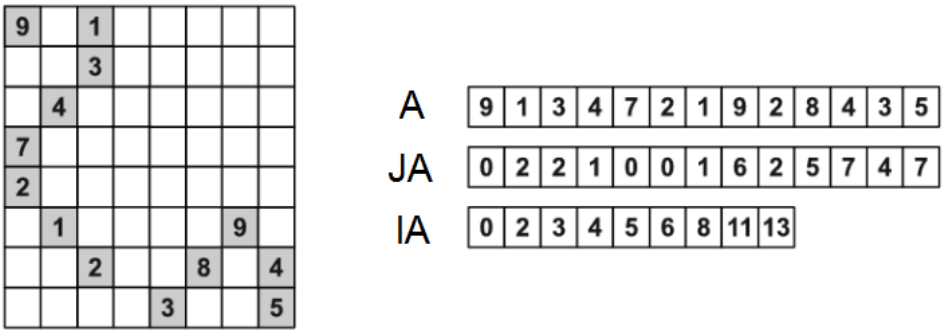
Рис 2. Пример треугольной сетки с определением сеточных функций в узлах сетки. В этом примере шаблон схемы в ячейке состоит из этой ячейки и ее соседних ячеек, т.е. ячеек, имеющих с данной ячейкой соединение ребром сетки. Номера в узлах обозначают номера неизвестных в векторе. Справа показан портрет матрицы. Номер узла в сетке соответствует номеру строки матрицы. Номера столбцов в строке узла соответствуют номерам соседних с ним узлов.

Формат разреженной матрицы

Предлагается использовать построчно-разреженный формат CSR – compressed sparse row. Разреженная матрица A размера NxN, в которой имеется M ненулевых коэффициентов, представляется в виде трех массивов:

- **int IA[N+1];** IA[i] содержит позицию начала данных i-й строки в массивах A и JA. IA[N+1] хранит значение M
- **int JA[M];** JA хранит номера столбцов ненулевых коэф. подряд по всем строкам
- **double A[M];** A хранит значения ненулевых коэффициентов подряд по всем строкам

Пример:



Доступ к ненулевым коэффициентам i-й строки:

```
for(int _j=IA[i]; _j<IA[i+1]; _j++){
      int j = JA[_j]; // номер столбца
      double a_ij = A[_j]; //значение коэффициента a_ij
      . . .
}
```

Генерация тестовой расчетной области

Для солвера СЛАУ портрет матрицы и значения ее коэффициентов являются входными данными.

Чтобы иметь возможность тестировать реализацию солвера, не прибегая к копированию объемных входных данных из реальных задач, предлагается сделать генератор расчетной области, которая будет определять портрет матрицы.

Суть данного действия - получить протрет матрицы.

Предлагается в качестве тестовой расчетной области использовать трехмерную декартову решетку, размерность которой, Nx, Ny, Nz, можно варьировать при тестировании.

Тут все полностью аналогично рис. 2, есть просто набор узлов, соединенных ребрами, только связи между узлами имеют регулярную структуру, что все сильно упрощает. Узлы могут адресоваться по трем индексам, I=0,...,Nx-1; J=0,...,Ny-1; K=0,...,Nz-1. Размер СЛАУ при этом будет

N = Nx*Ny*Nz.

Позиция в векторе в СЛАУ i=0,...,N-1 соответствует набору (I,J,K) индексов узла в решетке:

i = K*(Nx*Ny)+ J*Nx + I.

В i-й строке соответствующей матрицы будут следующие ненулевые позиции (т.е. набор номеров столбцов *Col(i)*):

i-Nx*Ny, если K>0;

i-Nx, если J>0;

i-1, если I>0;

i;

i+1, если I<Nx-1;

i+Nx, если J<Ny-1;

i+Nx*Ny, если K<Nk-1;

По решетке заданного размера необходимо сгенерировать описанный выше портрет матрицы в формате CSR и заполнить коэффициенты матрицы произвольными значениями, но так, чтобы матрица имела диагональное преобладание (сумма модулей внедиагональных элементов строки должна быть меньше диагонального элемента).

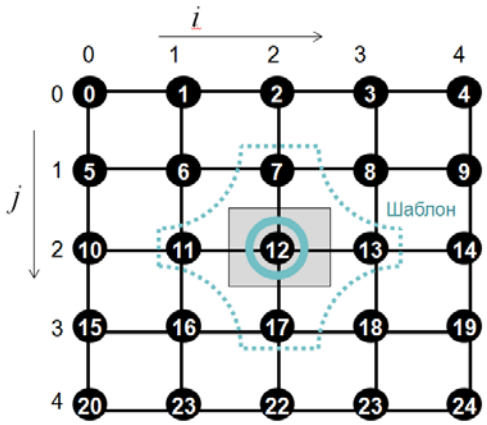
Например, для внедиагональных элементов строки можно взять синус от индексов строки и столбца, $a_{ij} = \sin(i + j + 1), i \neq j,\ \boldsymbol{j \in Col(i)}$

Набор индексов столбцов *Col(i)* для каждой строки i задан портретом матрицы!

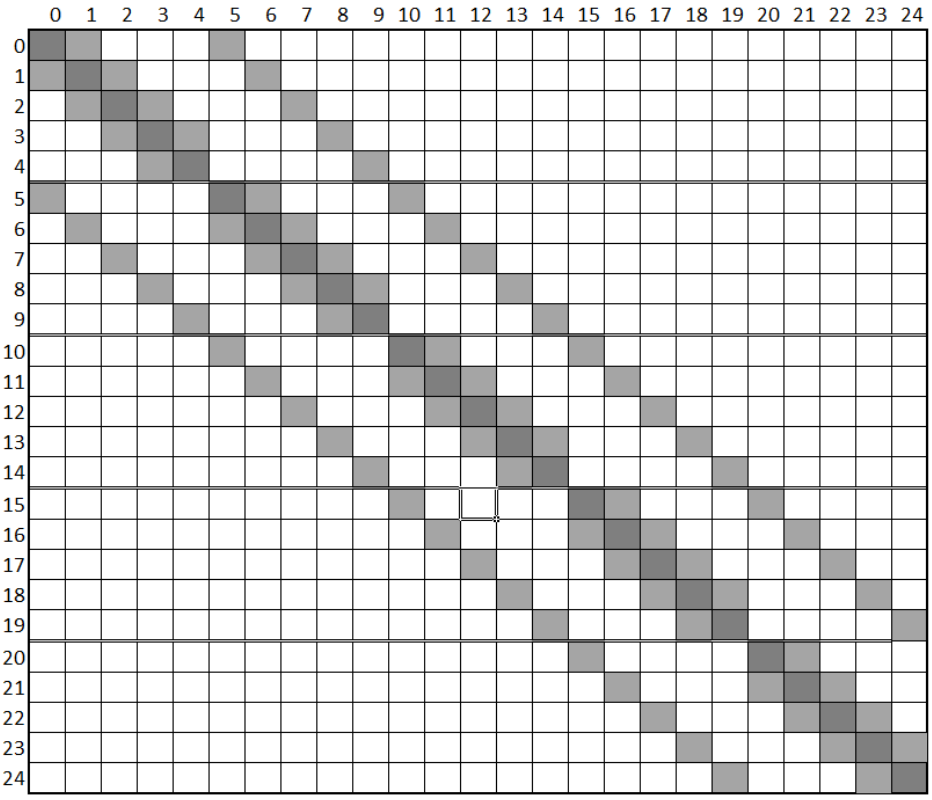
Диагональный элемент вычислить как сумму модулей внедиагональных элементов строки, домноженную на больший единицы коэффициент:

$$a_{ii} = 1.1 \sum_{j, j \neq i} |a_{ij}|$$

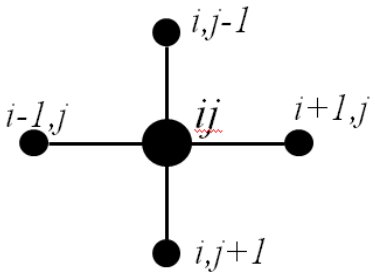
Для наглядности рассмотрим пример двухмерной решетки размера Nx*Ny, где Nx=Ny=5. Размер системы N=25.



Из топологии связей узлов в решетке следует портрет матрицы:



Шаблон схемы в данном случае выглядит так:



Алгоритм солвера СЛАУ BiCGSTAB

Один из вариантов алгоритм предобусловленного метода BiCGSTAB имеет следующий вид:

1. $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
2. Choose an arbitrary vector $\hat{\boldsymbol{r}}_0$ such that $(\hat{\boldsymbol{r}}_0, \boldsymbol{r}_0) \neq 0$, e.g., $\hat{\boldsymbol{r}}_0 = \boldsymbol{r}_0$
3. $\rho_0 = \alpha = \omega_0 = 1$
4. $\boldsymbol{v}_0 = \boldsymbol{p}_0 = \boldsymbol{0}$
5. For $i = 1, 2, 3, \ldots$
    1. $\rho_i = (\hat{\boldsymbol{r}}_0, \boldsymbol{r}_{i-1})$
    2. $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$
    3. $\boldsymbol{p}_i = \boldsymbol{r}_{i-1} + \beta(\boldsymbol{p}_{i-1} - \omega_{i-1}\boldsymbol{v}_{i-1})$
    4. $\boldsymbol{y} = \boldsymbol{K}^{-1}\boldsymbol{p}_i$
    5. $\boldsymbol{v}_i = \boldsymbol{A}\boldsymbol{y}$
    6. $\alpha = \rho_i/(\hat{\boldsymbol{r}}_0, \boldsymbol{v}_i)$
    7. $\boldsymbol{h} = \boldsymbol{x}_{i-1} + \alpha\boldsymbol{y}$
    8. If $\boldsymbol{h}$ is accurate enough then $\boldsymbol{x}_i = \boldsymbol{h}$ and quit
    9. $\boldsymbol{s} = \boldsymbol{r}_{i-1} - \alpha\boldsymbol{v}_i$
    10. $\boldsymbol{z} = \boldsymbol{K}^{-1}\boldsymbol{s}$
    11. $\boldsymbol{t} = \boldsymbol{A}\boldsymbol{z}$
    12. $\omega_i = (\boldsymbol{K}_1^{-1}\boldsymbol{t}, \boldsymbol{K}_1^{-1}\boldsymbol{s})/(\boldsymbol{K}_1^{-1}\boldsymbol{t}, \boldsymbol{K}_1^{-1}\boldsymbol{t})$
    13. $\boldsymbol{x}_i = \boldsymbol{h} + \omega_i\boldsymbol{z}$
    14. If $\boldsymbol{x}_i$ is accurate enough then quit
    15. $\boldsymbol{r}_i = \boldsymbol{s} - \omega_i\boldsymbol{t}$

В случае предобуславливателя Якоби матрица K – диагональная матрица, с диагональю из матрицы A.

Предлагается использовать следующий вариант алгоритма.

Входные данные:
N – число неизвестных в СЛАУ.
A – матрица системы размера NxN в формате CSR
BB – вектор правой части размера N
tol – критерий сходимости, отношение нормы невязки к норме правой части:
(невязка системы r = Ax-b)
maxit – максимальное число итераций

Выходные данные:
XX – вектор искомого решения размера N
nit – число итераций, <0 в случае ошибки

Внутренние данные солвера:
Матрицы:
DD – диагональная матрица размера NxN, имеющая обратную диагональ матрицы A: $d_{ii} = 1/a_{ii}$
Вектора:
PP, PP2, RR, RR2, TT, VV, SS, SS2 – вектора решателя СЛАУ размера N
Скалярные значения:
initres – начальная невязка системы, равная норме правой части
res – текущая норма невязки
mineps=1E-15 – минимальная абсолютная норма невязки
eps – норма невязки, соответствующая критерию tol
коэффициенты метода (указаны начальные значения):
Rhoi_1=1.0, alphai=1.0, wi=1.0, betai_1=1.0, Rhoi_2=1.0, alphai_1=1.0, wi_1=1.0;
RhoMin=1E-60 – минимальное значение коэффициентов метода
I – номер итерации.
info=0 – флаг режима отладки

операции:
dot(X,Y) – скалярное произведение
axpby(X,Y,a,b) – поэлементная операция X= aX+bY (линейная комбинация векторов)
SpMV(A,X,Y) - матрично-векторное произведение Y=AX

Алгоритм:

```
XX = 0;
RR = BB;
RR2 = BB;
initres = sqrt(dot(RR,RR));
eps = MAX(mineps, tol*initres);
res=initres;

for(I=0; I<maxit; I++){
  if(info) printf("It %d: res = %e tol=%e\n",
                  I,res, res/initres);
   if(res<eps) break;
   if(res>initres/mineps) return -1;

   if(I==0) Rhoi_1 = initres*initres;
   else Rhoi_1 = dot(RR2,RR);
   if(fabs(Rhoi_1)<RhoMin) return -1

   if(I==0) PP=RR;
   else{
       betai_1=(Rhoi_1*alphai_1)/(Rhoi_2*wi_1);
       axpby(PP, RR, betai_1, 1.0); // p=r+betai_1*(p-w1*v)
       axpby(PP, VV, 1.0, -wi_1*betai_1);
   }

   SpMV(DD,PP,PP2);

   SpMV(A,PP2,VV);

   alphai = dot(RR2,VV);
   if(fabs(alphai)<RhoMin) return -3;
   alphai = Rhoi_1/alphai;

   SS=RR;  // s=r-alphai*v
   axpby(SS, VV, 1.0, -alphai);

   SpMV(DD, SS, SS2);

   SpMV(A, SS2, TT);

   wi = dot(TT, TT);
   if(fabs(wi)<RhoMin) return -4;
   wi = dot(TT, SS) / wi;
   if(fabs(wi)<RhoMin) return -5;

   // x=x+alphai*p2+wi*s2
   axpby(XX, PP2,1.0,alphai);
   axpby(XX, SS2,1.0,wi);

   RR=SS; // r=s-wi*t
```

```
        axpby(RR, TT, 1.0, -wi);

        alphai_1=alphai;
        Rhoi_2=Rhoi_1;
        wi_1=wi;

        res = sqrt(dot(RR,RR));
    }
    if(info) printf("Solver_BiCGSTAB: outres: %g\n",res);
    return I;
```

Реализация

Выполнять реализацию предлагается следующим образом.

1. Сделать генератор матрицы для расчетной области, представленной трехмерной декартовой решеткой заданного размера Nx,Ny,Nz, проверить корректность заполнения матрицы. Заполнять коэффициенты, например, можно так: для внедиагональных элементов строки можно взять синус от индексов строки и столбца, $a_{ij} = \sin(i + j + 1), i \neq j,$ $\boldsymbol{j \in Col(i),}$ а диагональный элемент вычислить как сумму модулей внедиагональных элементов строки, домноженную на больший единицы коэффициент: $a_{ii} = 1.1 \sum_{j, j \neq i} |a_{ij}|$

2. Сделать исходные последовательные реализации операций dot, axpby, SpMV, а также операций копирования вектора и заполнения вектора константой (соответствует оператору присваивания в алгоритме выше).

3. Сделать проверочный тест, выполняющий операции dot, axpby, SpMV с матрицей, заполненной как в п. 1, и векторами, заполненными произвольными различающимися значениями, например X[i]=sin(i), Y[i]=cos(i). Проверочный тест для каждой операции должен выдать контрольные значения: для dot – просто результат, для axpby и SpMV контрольные значения по выходному вектору, например: сумма всех элементов, L2 норма (sqrt(dot(X,X)), C норма. Последующие параллельные реализации можно будет сравнивать с исходными последовательными реализациями по этим контрольным значениям для подтверждения корректности.

4. Сделать солвер BiCGSTAB на основе исходных операций. Проверить работоспособность солвера на сгенерированной матрице и правой части, заполненной, например, аналогично: B[i]=sin(i). Начальное значение вектора решения нулевое.

5. По желанию. Попробовать выполнить оптимизацию последовательных версий базовых операций, сохранив также исходные версии для сравнения.

Можно применить развертку циклов, SIMD инструкции и т.д. Сделать таймирование по всем базовым операциям, оценить улучшения.

6. Сделать многопоточные параллельные реализации базовых операций. Получить многопоточную корректно работающую версию солвера.

Реализовать все вышеописанное можно в виде программы, которая выполняет генерацию тестовой системы, решает ее солвером, выдает результат в виде таймирования по всем базовым операциям и по общему времени работы солвера, выдает затраченное число итераций и отношение нормы невязки к норме правой части.

Исходный код на C или C++, осторожно, новые стандарты (C++14 и тд) могут не поддерживаться на кластерах. Лучше что-то совместимое с GCC 4.8. Сборка make.

Аргументы с командной строки

nx=<int> ny=<int> nz=<int> размер решетки топологии для генерации матрицы размера NxN, N=nx·ny·nz

tol=<double> невязка относительно нормы правой части

maxit=<int> максимальное число итераций

nt=<int> число нитей

qa флаг тестирования базовых операций

Отчет

По результатам нужно будет подготовить отчет, в котором привести данные по исследованию производительности солвера. Для каждой из базовых операций и для всего алгоритма исследовать зависимость достигаемой производительности от размера системы N, построить графики GFLOPS от N. Измерить OpenMP ускорение для различных N. Оценить максимально достижимую производительность (TBP) с учетом пропускной способности памяти и сравнить с фактической производительностью. Оценить достигаемую скорость передачи данных.

Требования к отчету:

Титульный лист, содержащий
1.1 Название курса
1.2 Название задания
1.3 Фамилию, Имя, Отчество(при наличии)
1.4 Номер группы
1.5 Дата подачи

Содержание отчета:

2 Описание задания и программной реализации
2.1 Краткое описание задания
2.2 Краткое описание программной реализации - как организованы данные, какие функции реализованы (название, аргументы, назначение).
Просьба указывать, как программа запускается – с какими параметрами, с описанием этих параметров.
2.3 Описание опробованных способов оптимизации последовательных вычислений (по желанию)

3 Исследование производительности
3.1 Характеристики вычислительной системы:
описание одной или нескольких систем, на которых выполнено исследование (подойдет любой многоядерный процессор).
тип процессора, количество ядер, пиковая производительность, пиковая пропускная способность памяти.
по желанию - промерять и на своем декстопе/ноуте, и на кластере
Просьба указывать здесь или в следующих пунктах, как программа компилировалась (каким компилятором, с какими параметрами).
3.2 Результаты измерений производительности
3.2.1 Последовательная производительность
Для каждой из трех базовых операций и для всего алгоритма солвера исследовать зависимость достигаемой производительности от размера системы N, построить графики GFLOPS от N. *Нескольких N достаточно: N=1000, 10000, 100000, 1000000*
Для повышения точности измерений, замеры времени лучше производить, выполняя набор операций многократно в цикле, чтобы осреднить время измерений. Суммарное время измерений чтобы получалось порядка нескольких секунд.
Оценить выигрыш от примененной оптимизации (по желанию)

3.2.2. Параллельное ускорение

Измерить OpenMP ускорение для различных N для каждой из 3-х базовых операций и для всего алгоритма солвера: при фиксированном числе N варьируется число нитей и измеряется параллельное ускорение.

4 Анализ полученных результатов

4.1 Процент от пика

оценить для каждой из трех базовых операций, какой процент от пиковой производительности устройства составляет максимальная достигаемая в тесте производительность

4.2 Процент от достижимой производительности

аналогично оценить для каждой операции процент от максимально достижимой производительности с учетом пропускной способности памяти.

Приложение1: исходный текст программы в отдельном c/c++ файле

Требования к программе:

1 Программа должна использовать OpenMP или posix threads для многопоточного распараллеливания

2 Солвер должен корректно работать, т.е. показывать быструю сходимость.

Вспомогательная информация по заданию.

Получаемое ускорение сильно отличается в зависимости от матчасти.
Ускорение от оптимизации также сильно зависит от компилятора.
См. примеры ниже.

Пример тестирования операций и солвера на ноуте с 4-ядреным процессором, 2 канала памяти, Windows:

```
Testing BiCGSTAB solver for a 3D grid domain
N = 1000000 (Nx=100, Ny=100, Nz=100)
Aij = sin((double)i+j+1), i!=j
Aii = 1.1*sum(fabs(Aij))
Bi  = sin((double)i)
tol = 1.000000e-006

testing sequential ops:
Sequential ops timing:
dot   time= 0.450s GFLOPS=  1.78
axpy  time= 0.497s GFLOPS=  2.41
SpMV  time= 6.117s GFLOPS=  0.15
testing parallel ops for ntr=2:
dot   time= 0.307s GFLOPS=  2.61 Speedup=  1.47X
axpy  time= 0.439s GFLOPS=  2.73 Speedup=  1.13X
SpMV  time= 3.130s GFLOPS=  0.29 Speedup=  1.95X
testing parallel ops for ntr=4:
dot   time= 0.285s GFLOPS=  2.80 Speedup=  1.58X
axpy  time= 0.427s GFLOPS=  2.81 Speedup=  1.16X
SpMV  time= 2.287s GFLOPS=  0.40 Speedup=  2.67X

Testing solver:
Solver_BiCGSTAB: initres: 707.107; eps: 1e-006; N=1000000
Solver_BiCGSTAB: 0: res = 7.071068e+002 tol=1.000000e+000
Solver_BiCGSTAB: 1: res = 2.642244e+002 tol=3.736697e-001
Solver_BiCGSTAB: 2: res = 2.878979e+001 tol=4.071491e-002
Solver_BiCGSTAB: 3: res = 6.441312e+000 tol=9.109390e-003
Solver_BiCGSTAB: 4: res = 1.982630e+000 tol=2.803863e-003
Solver_BiCGSTAB: 5: res = 5.613354e-001 tol=7.938481e-004
Solver_BiCGSTAB: 6: res = 1.664901e-001 tol=2.354525e-004
Solver_BiCGSTAB: 7: res = 4.023986e-002 tol=5.690776e-005
Solver_BiCGSTAB: 8: res = 1.108974e-002 tol=1.568326e-005
Solver_BiCGSTAB: 9: res = 3.229193e-003 tol=4.566768e-006
Solver_BiCGSTAB: 10: res = 1.083402e-003 tol=1.532161e-006
Solver_BiCGSTAB: 11: res = 4.230738e-004 tol=5.983167e-007
Solver_BiCGSTAB: outres: 0.000423074 tol: 5.98317e-007
Solver finished in 11 iterations, res = 0.000423074 tol=5.98317e-007
```

Пример тестирования операций и солвера на 6-ядреном процессоре, 4 канала памяти, Linux:

```
cherepock@imm28:~/TEST$ cat /proc/cpuinfo | grep name | tail -n 1
model name      : Intel(R) Core(TM) i7-3960X CPU @ 3.30GHz

cherepock@imm28:~/TEST$ g++ -O3 -march=native -mtune=native -fopenmp solver.cpp
-o solver.px; ./solver.px

Testing BiCGSTAB solver for a 3D grid domain
N = 1000000 (Nx=100, Ny=100, Nz=100)
Aij = sin((double)i+j+1), i!=j
Aii = 1.1*sum(fabs(Aij))
Bi  = sin((double)i)
tol = 1.000000e-06

testing sequential ops:
Sequential ops timing:
dot   time= 0.378s GFLOPS=  2.11
axpy  time= 0.345s GFLOPS=  3.48
SpMV  time= 4.865s GFLOPS=  0.19
testing parallel ops for ntr=2:
dot   time= 0.199s GFLOPS=  4.02 Speedup=  1.90X
axpy  time= 0.184s GFLOPS=  6.53 Speedup=  1.88X
SpMV  time= 2.501s GFLOPS=  0.36 Speedup=  1.95X
testing parallel ops for ntr=4:
dot   time= 0.130s GFLOPS=  6.13 Speedup=  2.90X
axpy  time= 0.135s GFLOPS=  8.90 Speedup=  2.56X
SpMV  time= 1.680s GFLOPS=  0.54 Speedup=  2.90X
testing parallel ops for ntr=6:
dot   time= 0.100s GFLOPS=  8.01 Speedup=  3.79X
axpy  time= 0.103s GFLOPS= 11.60 Speedup=  3.33X
SpMV  time= 1.394s GFLOPS=  0.65 Speedup=  3.49X

Testing solver:
Solver_BiCGSTAB: initres: 707.107; eps: 1e-06; N=1000000
Solver_BiCGSTAB: 0: res = 7.071068e+02 tol=1.000000e+00
Solver_BiCGSTAB: 1: res = 2.642244e+02 tol=3.736697e-01
Solver_BiCGSTAB: 2: res = 2.878979e+01 tol=4.071491e-02
Solver_BiCGSTAB: 3: res = 6.441312e+00 tol=9.109390e-03
Solver_BiCGSTAB: 4: res = 1.982630e+00 tol=2.803863e-03
Solver_BiCGSTAB: 5: res = 5.613354e-01 tol=7.938481e-04
Solver_BiCGSTAB: 6: res = 1.664901e-01 tol=2.354525e-04
Solver_BiCGSTAB: 7: res = 4.023986e-02 tol=5.690776e-05
Solver_BiCGSTAB: 8: res = 1.108974e-02 tol=1.568326e-05
Solver_BiCGSTAB: 9: res = 3.229193e-03 tol=4.566768e-06
Solver_BiCGSTAB: 10: res = 1.083402e-03 tol=1.532161e-06
Solver_BiCGSTAB: 11: res = 4.230738e-04 tol=5.983167e-07
Solver_BiCGSTAB: outres: 0.000423074 tol: 5.98317e-07
Solver finished in 11 iterations, res = 0.000423074 tol=5.98317e-07
```

Пример простейшей оптимизации axpby, spmv (развертка, simd), результаты:

Windows, MS VS 2008, 4C

```
N = 125000 (Nx=50, Ny=50, Nz=50)
Sequential ops timing:
axpy  time= 0.043s GFLOPS=  3.50
SpMV  time= 0.757s GFLOPS=  0.86
Optimized:
axpy  time= 0.034s GFLOPS=  4.44
SpMV  time= 0.465s GFLOPS=  1.40

N = 1000000 (Nx=100, Ny=100, Nz=100)
Sequential ops timing:
axpy  time= 0.495s GFLOPS=  2.42
SpMV  time= 6.115s GFLOPS=  0.15
Optimized:
axpy  time= 0.479s GFLOPS=  2.50
SpMV  time= 3.727s GFLOPS=  0.24
```

Linux, 6C

```
N = 125000 (Nx=50, Ny=50, Nz=50)
Sequential ops timing:
axpy  time= 0.032s GFLOPS=  4.74
SpMV  time= 0.581s GFLOPS=  1.12
Optimized:
axpy  time= 0.032s GFLOPS=  4.67
SpMV  time= 0.365s GFLOPS=  1.78

N = 1000000 (Nx=100, Ny=100, Nz=100)
Sequential ops timing:
axpy  time= 0.353s GFLOPS=  3.40
SpMV  time= 4.878s GFLOPS=  0.19
Optimized:
axpy  time= 0.344s GFLOPS=  3.48
SpMV  time= 3.162s GFLOPS=  0.29
```

Пример организации тестирования базовой операции

```
// Testing basic operations
int Ntest = 20;
double taxpyseq=0.0, t;
const double axpyflop = Ntest*Ntest*N*3*1E-9;

printf("testing sequential ops:\n");
omp_set_num_threads(1);
for(int i=0; i<Ntest; i++){
    t = omp_get_wtime();
    for(int j=0; j<Ntest; j++) vec_expr(N, x, b, 1.00001, 0.99999);
    taxpyseq += omp_get_wtime() - t;
}

printf("Sequential ops timing: \n");
printf("axpy  time=%6.3fs GFLOPS=%6.2f\n", taxpyseq,  axpyflop/taxpyseq);

//parallel mode
const int NTR = omp_get_num_procs();
for(int ntr=2; ntr<=NTR; ntr+=2){
    for(int i=0; i<N; i++){ x[i]=0.0; b[i]=(i*i)%123; }
    printf("testing parallel ops for ntr=%d:\n", ntr);
    omp_set_num_threads(ntr);
    double taxpypar=0.0;
    for(int i=0; i<Ntest; i++){
        t = omp_get_wtime();
        for(int j=0; j<Ntest; j++) vec_expr(N, x, b, 1.00001, 0.99999);
        taxpypar += omp_get_wtime() - t;
    }

    printf("axpy  time=%6.3fs GFLOPS=%6.2f Speedup=%6.2fX \n",
           taxpypar, axpyflop/taxpypar, taxpyseq/taxpypar);
}
```