Отчёт по курсу «Высокопроизводительные параллельные вычисления на кластерных системах».

Владислав Соврасов
аспирант гр. 2-о-051318

1 Постановка задачи

Требуется получить параллельную версию солвера MIDACO [1], которая бы эффективно работала как на распределённых системах, так и на системах с общей памятью.

MIDACO (Mixed Integer Distributed Ant Colony Optimization) предназначен для решения глобальной оптимизации как с дискретными, так и с непрерывными параметрами.

Будем рассматривать задачу глобальной оптимизации в непрерывном многомерном пространстве:

$$\varphi(y^*) = \min\{\varphi(y) : y \in D\}, D = \{y \in \mathbf{R}^N : a_i \leqslant x_i \leqslant b_i, 1 \leqslant i \leqslant N\}$$

В качестве тестовых задач для измерения производительности рассматривались 100 четырёхмерных задач вида (1), полученных генератором GKLS [2].

2 Реализация

Авторы MIDACO предлагают использовать довольно простую схему распараллеливания (рис. 1): интерфейс их метода устроен таким образом, что на каждой итерации позволяет получить заданное количество точек, в которых необходимо вычислить значение целевой функции. Вычисление во всех точках могут быть проведены параллельно. В практических задачах глобальной оптимизации целевая функция, как правило, достаточно трудоёмка, поэтому данная схема имеет смысл.

Вместе с исходным кодом MIDACO предоставляются примеры распараллеливания вычисления целевой функции как на распределённой, так и на общей памяти. Они имеют ряд недостатков:

- нет примера, сочетающего в себе смешанную модель распараллеливания на общей + распределённой памяти;
- все примеры написаны на языке C таким образом, что в них нет чётко выделенного интерфейса для солвера, который бы обеспечивал удобство использования;
- в распределённой версии для передачи многомерных точек на другие узлы используются MPI-операции Send/Receive, в то время как описанная схема идеально подходит под использование Scatter/Gather и может быть эффективнее реализована с их помощью. Эта проблема особенно актуальна в режиме смешанного распараллеливания, когда на каждый узел вместо одной точки будут пересылаться несколько.

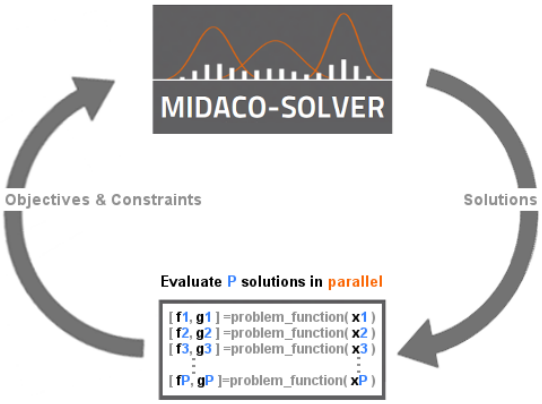

Рис. 1: Схема распараллеливания в MIDACO

Учитывая все перечисленные недостатки публично доступной параллельной версии MIDACO, была подготовлена собственная реализация, предоставляющая более удобный C++ интерфейс к солверу, реализующая параллельное вычисление целевой функции на различных узлах и на различных процессорах в рамках одного узла, использующая при этом эффективные групповые операции Scatter/Gather. Часть исходного кода приведена в секции 4.

3 Результаты

Все вычислительные эксперименты проводились на узлах суперкомпьютера «Лобачевский», каждый узел которого имеет 16 вычислительных ядер. В экспериментах было задействовано до 12 узлов. Таким образом, максимальное количество задействованных вычислительных ядер достигало 192.

В каждом эксперименте решались 100 четырёхмерных задач, полученных генератором GKLS. Измерялось количество обращений к целевой функции и время решения каждой задачи. С целью имитации трудоёмких целевых функций в вычисление каждой функции вносилась искусственная задержка, равная в различных экспериментах 0.1, 0.5 или 1мс. Задержка реализована в виде дополнительной нагрузки по вычислению некоторых элементарных функций. Объём вычислений подбирался так, чтобы они занимали заданное время.

Во всех экспериментах считается, что тестовая задача решена, если метод оптимизации провёл очередное испытание y^k в δ-окрестности глобального минимума y^*, т.е. $\left\|y^k - y^*\right\| \leqslant \delta = 0.01 \left\|b - a\right\|$, где a и b — левая и правая границы гиперкуба из (1). Если указанное соотношение не выполнено до истечения лимита на количество испытаний, то задача считается нерешённой. Максимальный лимит на количество испытаний установлен раным $250 \cdot 10^3$

На рис. 2 приведён график ускорения по времени ($S_p = \frac{t_1}{t_p}$) при различном количестве узлов и потоков на один узел, а также всех рассматриваемых значениях задержки в целевой функции. Из графика видно, что ускорение достигает максимальных значений при задержке 1мс, т.к. накладные расходны на передачу данных становятся меньше по сравнению с временем вычисления целевых функций. Использование 16 ядер на каждом узле также приносит значительное ускорение, по сравнению с использованием одного ядра на узел. При увеличении количества узлов ускорение линейно нарастает.

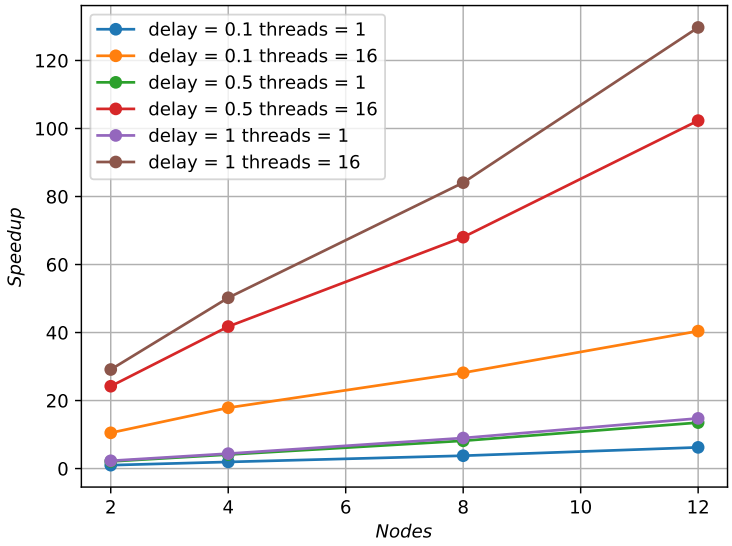

Рис. 2: Графики полученного ускорения по узлам в зависимости от трудоёмкости вычисления целевой функции

В таблице 1 приведено количество итераций метода в каждой из конфигураций, количество решённых задач, а также ускорения по времени S_p и по итерациям $S_p^i = \frac{iters_p}{iters_1}$. S_p^i является верхней границей для S_p, т.к. в лучшем случае каждая параллельная итерация занимает столько же времени, сколько последовательная. Параллельный метод делает меньше итераций, за счёт того, что на каждой из них проводит больше испытаний, однако ускорение по итерациям может быть в некоторых случаях меньше количества вычислительных устройств из-за избыточности по испытаниям, характерной для методов параллельной глобальной оптимизации. Согласно таблице 1, ускорение по времени довольно близко к ускорению по итерациям при малом количестве вычислительных устройств. С ростом количества узлов и потоков S_p начинает заметно отставать от S_p^i, однако при этом сохраняется приемлемое соотношение между ними. Также стоит заметить, что метод оптимизации, реализованный в MIDACO, с ростом количества испытаний на итерацию решает меньше задач за отведённое число испытаний. Эта ситуация происходит при использовании схем распараллеливания (4, 16), (8, 16) и (12, 16).

Таблица 1: Показатели ускорения по времени и по итерациям при задержке 1мс

Узлы, потоки	S_p^i	S_p		Итерации	Решено задач
1, 1	72068	132.5s		72068	71
1, 16	16.7	14.4		4304	70
2, 1	2.7	2.3		27128	73
2, 16	32.0	29.1		2254	73
4, 1	4.5	4.4		15980	75
4, 16	57.0	50.2		1264	66
8, 1	10.3	9.0		9853	83
8, 16	93.1	84.1		774	57
12, 1	16.0	14.8		6022	86
12, 16	183.2	129.7		393	53

4 Исходный код

Полная версия кода доступна по ссылке `https://github.com/sovrasov/midaco-mpi-cpp`.

```
#include <mpi.h>
#include <algorithm>
#include "midaco_mpi.hpp"

#include <midaco_core.h>
#include <omp.h>

MidacoSolution solve_midaco_mpi(const IGOProblem<double>* problem,
                                const MidacoMPIParameters& params,
                                std::function<bool(const double*)> external_stop)
{
  MidacoSolution solution;

  int proc, nprocs;
  MPI_Status status;
  MPI_Comm_rank( MPI_COMM_WORLD, &proc );
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );

  long int o,n,ni,m,me,maxeval,maxtime,printeval,save2file,iflag,istop;
  long int liw,lrw,lpf,i,iw[5000],p=1; double rw[20000],pf[20000],param[13];
  char key[] = "MIDACO_LIMITED_VERSION___[CREATIVE_COMMONS_BY-NC-ND_LICENSE]";

  o  = 1; /* Number of objectives */
  n  = problem->GetDimension(); /* Number of variables (in total) */
  ni = 0; /* Number of integer variables (0 <= ni <= n)    */
  m  = problem->GetConstraintsNumber(); /* Number of constraints (in total) */
  me = 0; /* Number of equality constraints (0 <= me <= m) */

  double* f = new double[o*params.numThreads];
  double* g = new double[m*params.numThreads];
```

```
31    double* x = new double[n*params.numThreads];
32    double* xl = new double[n];
33    double* xu = new double[n];
34
35    problem->GetBounds(xl, xu);
36    std::copy_n(xl, n, x);
37
38    maxeval = params.maxEvals;
39    maxtime = 60*60*24;
40    printeval = 1000;
41    save2file = 0;
42
43    param[ 0] =  0.0;  /* ACCURACY  */
44    param[ 1] =  params.seed;  /* SEED      */
45    param[ 2] =  0.0;  /* FSTOP     */
46    param[ 3] =  0.0;  /* ALGOSTOP  */
47    param[ 4] =  0.0;  /* EVALSTOP  */
48    param[ 5] =  params.focus;  /* FOCUS     */
49    param[ 6] =  0.0;  /* ANTS      */
50    param[ 7] =  0.0;  /* KERNEL    */
51    param[ 8] =  0.0;  /* ORACLE    */
52    param[ 9] =  0.0;  /* PARETOMAX */
53    param[10] =  0.0;  /* EPSILON   */
54    param[11] =  0.0;  /* BALANCE   */
55    param[12] =  0.0;  /* CHARACTER */
56
57    long int num_points = params.numThreads * nprocs;
58    p = nprocs;
59
60    if (proc == 0)
61    {
62      double *xxx,*fff,*ggg;
63      /* Allocate arrays for parallelization */
64      xxx = new double[params.numThreads*p*n];
65      fff = new double[params.numThreads*p*o];
66      ggg = new double[params.numThreads*p*m];
67      /* Store starting point x in xxx array */
68      for(int c=0; c<p*params.numThreads; c++)
69      {
70        std::copy_n(x, n, xxx + c*n);
71      }
72      lrw=sizeof(rw)/sizeof(double);
73      lpf=sizeof(pf)/sizeof(double);
74      liw=sizeof(iw)/sizeof(long int);
75      /* Print midaco headline and basic information */
76      midaco_print(1,printeval,save2file,&iflag,&istop,&*f,&*g,&*x,&*xl,&*xu,
77                   o,n,ni,m,me,&*rw,&*pf,maxeval,maxtime,&*param,num_points,&*key);
78      int n_evals = 0;
79      while(istop==0) /*~~~ Start of the reverse communication loop ~~~*/
```

```
80      {
81          for (int c=2; c<=p; c++) /* Send iterates X for evaluation */
82          {
83            for(int i=0; i<params.numThreads; i++)
84              if (external_stop(xxx + (c-1)*n*params.numThreads + i*n))
85                istop = 1;
86          }
87          MPI_Scatter(xxx, n*params.numThreads, MPI_DOUBLE, x,
88                      n*params.numThreads, MPI_DOUBLE, 0, MPI_COMM_WORLD);
89
90          #pragma omp parallel for num_threads(params.numThreads)
91          for (unsigned t = 0; t < params.numThreads; t++)  {
92            for (int i = 0; i < m; i++)
93              g[t*m + i] = problem->Calculate(xxx + t*n, i);
94            f[t*o] = problem->Calculate(xxx + t*n, m);
95            if (external_stop(xxx + t*n))
96            #pragma omp atomic write
97              istop = 1;
98          }
99
100         /* Collect results F & G */
101         MPI_Gather(f, o*params.numThreads, MPI_DOUBLE, fff, o*params.numThreads,
102                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
103         MPI_Gather(g, m*params.numThreads, MPI_DOUBLE, ggg, m*params.numThreads,
104                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
105
106         n_evals += p*params.numThreads;
107         /* Call MIDACO */
108         midaco(&num_points,&o,&n,&ni,&m,&me,&*xxx,&*fff,&*ggg,&*xl,&*xu,&iflag,
109                &istop,&*param,&*rw,&lrw,&*iw,&liw,&*pf,&lpf,&*key);
110         /* Call MIDACO printing routine */
111         midaco_print(2,printeval,save2file,&iflag,&istop,&*fff,&*ggg,&*xxx,&*xl,&*xu,
112                      o,n,ni,m,me,&*rw,&*pf,maxeval,maxtime,&*param,num_points,&*key);
113         /* Send istop to slave */
114         for (int c=2; c<=p; c++)
115         {
116           MPI_Send( &istop,1, MPI_INTEGER, c-1,4, MPI_COMM_WORLD);
117         }
118     }
119
120     solution.optValues = std::vector<double>(ggg, ggg + m);
121     solution.optValues.push_back(*fff);
122     solution.optPoint = std::vector<double>(xxx, xxx + n);
123     solution.calcCounters = std::vector<int>(m + 1, n_evals);
124     delete[] xxx;
125     delete[] fff;
126     delete[] ggg;
127   }
128   else
```

```
129     {
130       istop = 0;
131       while(istop<=0)
132       {
133         MPI_Scatter(nullptr, n*params.numThreads, MPI_DOUBLE, x,
134           n*params.numThreads, MPI_DOUBLE, 0, MPI_COMM_WORLD);
135
136         #pragma omp parallel for num_threads(params.numThreads)
137         for (unsigned t = 0; t < params.numThreads; t++)  {
138           for (int i = 0; i < m; i++)
139             g[t*m + i] = problem->Calculate(x + t*n, i);
140           f[t*o] = problem->Calculate(x + t*n, m);
141         }
142
143         MPI_Gather(f, o*params.numThreads, MPI_DOUBLE, nullptr, 0,
144                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
145         MPI_Gather(g, m*params.numThreads, MPI_DOUBLE, nullptr, 0,
146                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
147         MPI_Recv( &istop,1,MPI_INTEGER,0,4,MPI_COMM_WORLD, &status );
148       }
149     }
150
151     delete[] f;
152     delete[] g;
153     delete[] x;
154     delete[] xl;
155     delete[] xu;
156
157     return solution;
158   }
```

Список литературы

[1] http://www.midaco-solver.com

[2] http://wwwinfo.deis.unical.it/yaro/GKLS.html