Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ имени Н.Э.БАУМАНА
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

Лабораторная работа №2

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

Выполнил
студент группы ИУ9-51Б
Лисов Алексей

Москва, 2024

1 Условие

Написать программу, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A – матрица размером N×N, x и b – векторы длины N. Тип элементов – double. Программу распараллелить с помощью MPI с раз- резанием матрицы A по строкам на близкие по размеру, возможно не одина- ковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. Векторы x и b дублируются в каждом MPI-процессе. Уделить внимание тому, чтобы при запуске программы на различном числе MPI-процессов решалась одна и та же задача (исходные данные заполнялись одинаковым образом). Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные, параметры N и e подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Также параметр N разрешено подобрать таким образом, что- бы он нацело делился на на 1, 2, 4, 8 и 16. На основании полученных результатов сделать вывод о целесообразности использования программы.

2 Код решения

Имя Go файла: main.go Содержимое Go файла:

```
//Имя Go файла: main.go
//Содержимое Go файла:
package main

import (
        "context"
        "flag"
        "fmt"
        "log"
        "time"

        "github.com/BaldiSlayer/rprp/lab2/internal/generator"
        "github.com/BaldiSlayer/rprp/lab2/internal/pariter"
        "github.com/emer/empi/mpi"
)
```

```
const (
        eps = 1e-5
)

var (
        size = 20000
)

func init() {
        flag.IntVar(&size, "n", size, "size of matrix")
}

func main() {
        startTime := time.Now()

        flag.Parse()

        mpi.Init()
        defer mpi.Finalize()

        comm, errC := mpi.NewComm(nil)
        if errC != nil {
                log.Fatalf("can't init mpi comm %e", errC)
        }

        start, end, buff := initMpiMatrix(comm)

        err := comm.BcastF64(0, buff)
        if err != nil {
                log.Fatalf("can't broadcast buf %e", err)
        }

        baseMatrixChunk := generator.GenChunkMatrix(start, end, size)
        freeConst := generator.GenCheckFreeVector(baseMatrixChunk.RawMatrix().Rows
        solver := pariter.NewSolverWithVecSeparation(comm, baseMatrixChunk, freeCo

        ctx, cancel := context.WithTimeout(context.Background(), time.Second*60)
        defer cancel()

        s, err := solver.FindSolution(ctx)
```

```
	if err != nil {
		log.Fatalf("%e", err)
	}

	fullRes := make([]float64, size)

	fmt.Println(s.RawVector().Data)

	err = comm.AllGatherF64(fullRes, s.RawVector().Data)
	if err != nil {
		log.Fatalf("can't record res data %e", err)
	}

	//fmt.Println(fullRes)

	log.Println("Program time: ", time.Since(startTime))
}

func initMpiMatrix(comm *mpi.Comm) (int, int, []float64) {
	start, end := pariter.GetMpiChunkParams(comm, size)

	if comm.Rank() == 0 {
		return start, end, generator.GenRandomVector(size)
	}

	return start, end, make([]float64, size)
}

----------------------

//Имя Go файла: main.go
//Содержимое Go файла:
package main

import (
	"context"
	"log"
	"time"

	"github.com/BaldiSlayer/rprp/lab2/internal/generator"
```

```
        "github.com/BaldiSlayer/rprp/lab2/internal/iterations"
        "gonum.org/v1/gonum/mat"
)

const (
        size    = 10000
        epsilon = 1e-5
)

func main() {
        matrixA := generator.GenMatrix(size)
        bData := generator.GenVectorConstN(size)

        b := mat.NewVecDense(size, bData)

        xData := generator.GenRandomVector(size)
        x := mat.NewVecDense(size, xData)

        ctx, cancel := context.WithTimeout(context.Background(), time.Second*60)
        defer cancel()

        startTime := time.Now()

        result, err := iterations.SimpleIteration(ctx, matrixA, b, x, epsilon)
        if err != nil {
                log.Fatalf("ошибка SimpleIteration: %e", err)
        }

        log.Println("Program time:", time.Since(startTime))

        printResult(matrixA, result, b, x, epsilon)
}

func printResult(A *mat.Dense, result, b, x *mat.VecDense, epsilon float64) {
        log.Println("Матрица A:\n", mat.Formatted(A))
        log.Println("Вектор b:", b)
        log.Println("Начальное приближение x:", x)
        log.Println("Желаемая точность:", epsilon)
        log.Println("Результат:\n", mat.Formatted(result))
}
```

```
----------------------

//Имя Go файла: parallel_iteration.go
//Содержимое Go файла:
package pariter

import (
        "context"
        "errors"
        "math"

        "github.com/emer/empi/mpi"
        "gonum.org/v1/gonum/mat"
)

type mpiObj struct {
        comm  *mpi.Comm
        rank  int
        start int
        end   int
}

type matrixChunk struct {
        baseMatrixChunk *mat.Dense
        freeConst       *mat.VecDense
        size            int
}

type SolverWithVecSeparation struct {
        mo  *mpiObj
        mc  *matrixChunk
        tau float64
        eps float64
}

func NewSolverWithVecSeparation(
        comm *mpi.Comm, baseMatrixChunk *mat.Dense, freeConst *mat.VecDense, size i
) *SolverWithVecSeparation {
        return &SolverWithVecSeparation{
```

```
                mo:  newMpiObj(comm, size),
                mc:  newMatrixChunk(baseMatrixChunk, freeConst, size),
                eps: eps,
                tau: float64(1) / float64(size),
        }
}

func newMpiObj(comm *mpi.Comm, size int) *mpiObj {
        rowStart, rowEnd := GetMpiChunkParams(comm, size)

        return &mpiObj{
                rank:  comm.Rank(),
                start: rowStart,
                end:   rowEnd,
                comm:  comm,
        }
}

func GetMpiChunkParams(comm *mpi.Comm, size int) (int, int) {
        chunkSize := size / comm.Size()

        return comm.Rank() * chunkSize, (comm.Rank() + 1) * chunkSize
}

func newMatrixChunk(baseMatrixChunk *mat.Dense, freeConst *mat.VecDense, size int)
        return &matrixChunk{
                baseMatrixChunk: baseMatrixChunk,
                freeConst:       freeConst,
                size:            size,
        }
}

func (s *SolverWithVecSeparation) FindSolution(ctx context.Context) (*mat.VecDense
        res := mat.NewVecDense(s.mo.end-s.mo.start, nil)
        buff := make([]float64, s.mc.size)
        iteration := 0

        err := s.iterateLoop(ctx, res, &buff, &iteration)

        return res, err
```

```
}

func (s *SolverWithVecSeparation) iterateLoop(
        ctx context.Context,
        res *mat.VecDense,
        buff *[]float64,
        iteration *int,
) error {
        metric := math.MaxFloat64
        prev := metric

        for metric > s.eps {
                select {
                case <-ctx.Done():
                        return errors.New("context deadline")
                default:
                }

                *iteration++
                chunk := s.iterateSolution(res).RawVector().Data

                err := s.mo.comm.AllGatherF64(*buff, chunk)
                if err != nil {
                        return err
                }

                tmp := mat.NewVecDense(s.mc.size, *buff)

                res.SubVec(res, tmp.SliceVec(s.mo.start, s.mo.end))

                prev = metric

                metric, err = s.calcMetric(res)
                if err != nil {
                        return err
                }

                if prev < metric {
                        s.tau *= -1
                }
```

```
        }

        return nil
}

func (s *SolverWithVecSeparation) iterateSolution(chunk *mat.VecDense) *mat.VecDens
        buff := make([]float64, s.mc.size)
        _ = s.mo.comm.AllGatherF64(buff, chunk.RawVector().Data)

        tmp := mat.NewVecDense(s.mo.end-s.mo.start, nil)

        tmp.MulVec(s.mc.baseMatrixChunk, mat.NewVecDense(s.mc.size, buff))
        tmp.SubVec(tmp, s.mc.freeConst)
        tmp.ScaleVec(s.tau, tmp)

        return tmp
}

type fraction struct {
        numerator   float64
        denominator float64
}

func (s *SolverWithVecSeparation) calcMetric(chunk *mat.VecDense) (float64, error)
        buff := make([]float64, s.mc.size)

        err := s.mo.comm.AllGatherF64(buff, chunk.RawVector().Data)
        if err != nil {
                return 0, err
        }

        tmp := mat.NewVecDense(s.mo.end-s.mo.start, nil)

        tmp.MulVec(s.mc.baseMatrixChunk, mat.NewVecDense(s.mc.size, buff))
        tmp.SubVec(tmp, s.mc.freeConst)

        frac := getChunkMetricFraction(s, tmp)

        sumFrac := make([]float64, 2)
```

```
        err = s.mo.comm.AllReduceF64(mpi.OpSum, sumFrac, []float64{frac.numerator,
        if err != nil {
                return 0, err
        }

        return getMetricValue(sumFrac), nil
}

func getChunkMetricFraction(s *SolverWithVecSeparation, tmp *mat.VecDense) fractio
        numerator, denominator := float64(0), float64(0)

        for _, v := range tmp.RawVector().Data {
                numerator += v * v
        }

        for _, v := range s.mc.freeConst.RawVector().Data {
                denominator += v * v
        }

        return fraction{
                numerator:   numerator,
                denominator: denominator,
        }
}

func getMetricValue(frac []float64) float64 {
        return math.Sqrt(frac[0]) / math.Sqrt(frac[1])
}

----------------------

//Имя Go файла: matrix_generator.go
//Содержимое Go файла:
package generator

import (
        "math/rand"

        "gonum.org/v1/gonum/mat"
)
```

```
func GenMatrix(size int) *mat.Dense {
	A := mat.NewDense(size, size, nil)

	for i := 0; i < size; i++ {
		A.Set(i, i, 2.0)

		for j := 0; j < size; j++ {
			if i != j {
				A.Set(i, j, 1.0)
			}
		}
	}

	return A
}

// GenVectorConstN создает вектор, где все значения равны n+1
func GenVectorConstN(n int) []float64 {
	bData := make([]float64, n)

	for i := 0; i < n; i++ {
		bData[i] = float64(n + 1)
	}

	return bData
}

func GenChunkMatrix(start, end, size int) *mat.Dense {
	rows := end - start
	res := mat.NewDense(rows, size, make([]float64, rows*size))

	for i := start; i < end; i++ {
		for j := 0; j < size; j++ {
			if i == j {
				res.Set(i-start, j, 2.0)
			} else {
				res.Set(i-start, j, 1.0)
			}
		}
```

```
        }

        return res
}

func GenCheckFreeVector(matrixChunkRowsSize, size int) *mat.VecDense {
        resBuf := make([]float64, matrixChunkRowsSize)

        for i := 0; i < matrixChunkRowsSize; i++ {
                resBuf[i] = float64(size + 1)
        }

        return mat.NewVecDense(matrixChunkRowsSize, resBuf)
}

func GenRandomVector(size int) []float64 {
        res := make([]float64, size)

        for i := 0; i < size; i++ {
                res[i] = float64(rand.Intn(size))
        }

        return res
}

----------------------

//Имя Go файла: simple_iteration.go
//Содержимое Go файла:
package iterations

import (
        "context"
        "errors"
        "log"
        "math"

        "gonum.org/v1/gonum/mat"
)
```

```
func SimpleIteration(ctx context.Context, A *mat.Dense, b, x0 *mat.VecDense, epsilo
        n, _ := A.Dims()
        x := mat.NewVecDense(n, nil)

        t := getTau(n)
        prevNormDiff := 0.0

        var iteration int
        for {
                select {
                case <-ctx.Done():
                        return nil, errors.New("context deadline exceeded")
                default:
                }

                calculateMatrix(A, x, x0, b, t)
                iteration++

                if completionCriterion(x, x0, b, n, iteration, epsilon, &prevNormDi
                        return x, nil
                }

                x0.CopyVec(x)
        }
}

func calculateMatrix(A *mat.Dense, x, x0, b *mat.VecDense, t float64) {
        // Compare A * x0
        x.MulVec(A, x0)

        // Compare A * x0 - b
        x.SubVec(x, b)

        // Compare t * (A * x0 - b)
        x.ScaleVec(t, x)

        // Compare x0 + t * (A * x0 - b)
        x.AddVec(x0, x)
}
```

```
func completionCriterion(x, x0, b *mat.VecDense, n, i int, epsilon float64, prevNor
        diff := mat.NewVecDense(n, nil)
        diff.SubVec(x, x0)
        normDiff := mat.Norm(diff, 2)
        normB := mat.Norm(b, 2)
        if normDiff/normB < epsilon {
                log.Printf("Завершено после %d итераций\n", i)
                return true
        }

        if *prevNormDiff == 0 {
                *prevNormDiff = normDiff
        } else {
                // Если изменение  меньше определенного порога (например, 1e-8), т
                if math.Abs(normDiff/normB-(*prevNormDiff)/normB) < 1e-8 {
                        log.Printf("Завершено после %d итераций\n", i)
                        return true
                }
                *prevNormDiff = normDiff
        }

        return false
}

func getTau(N int) float64 {
        return -0.1 / float64(N)
}
```

---

3 Результаты

Характеристики устройства

- Процессор - Apple M2 Pro
- Оперативная память - 16 Гб
- Операционная система - MacOS Sonoma 14.5

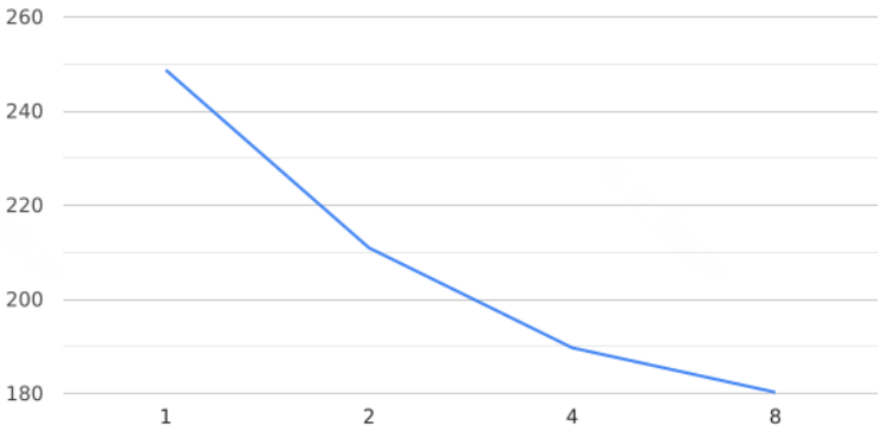

Рис. 1: График

4 Вывод

Каждый процесс создает полные копии векторов b и x, что приводит к значительному увеличению общего объема используемой памяти, особенно при большом количестве процессов. Этот подход неэффективен с точки зрения памяти, так как все процессы используют одни и те же данные. Более того, с увеличением числа процессов время выполнения также возрастает, поскольку значительная часть времени уходит на передачу полных векторов между процессами. Таким образом, данная программа будет эффективной только при работе с очень большими данными, когда затраты на обмен информацией окажутся ниже, чем затраты на вычисления.