南開大學
Nankai University

计算机学院

并行程序设计第 4 次作业

高斯消去法的 Pthreads 并行化

姓名：丁屹

学号：2013280

专业：计算机科学与技术

2022 年 5 月 5 日

目录

1 问题描述 **2**

2 Pthreads 算法设计 **3**

2.1 测试用例的确定 3
2.2 实验环境和相关配置 3
2.3 算法设计 3
2.3.1 默认平凡算法 3
2.3.2 使用 Pthreads 动态创建线程并行化加速 4
2.3.3 使用 Pthreads 线程池和信号量同步并行化加速 5
2.3.4 使用 Pthreads 线程池和 barrier 栅栏同步并行化加速 9

3 实验结果分析 **10**

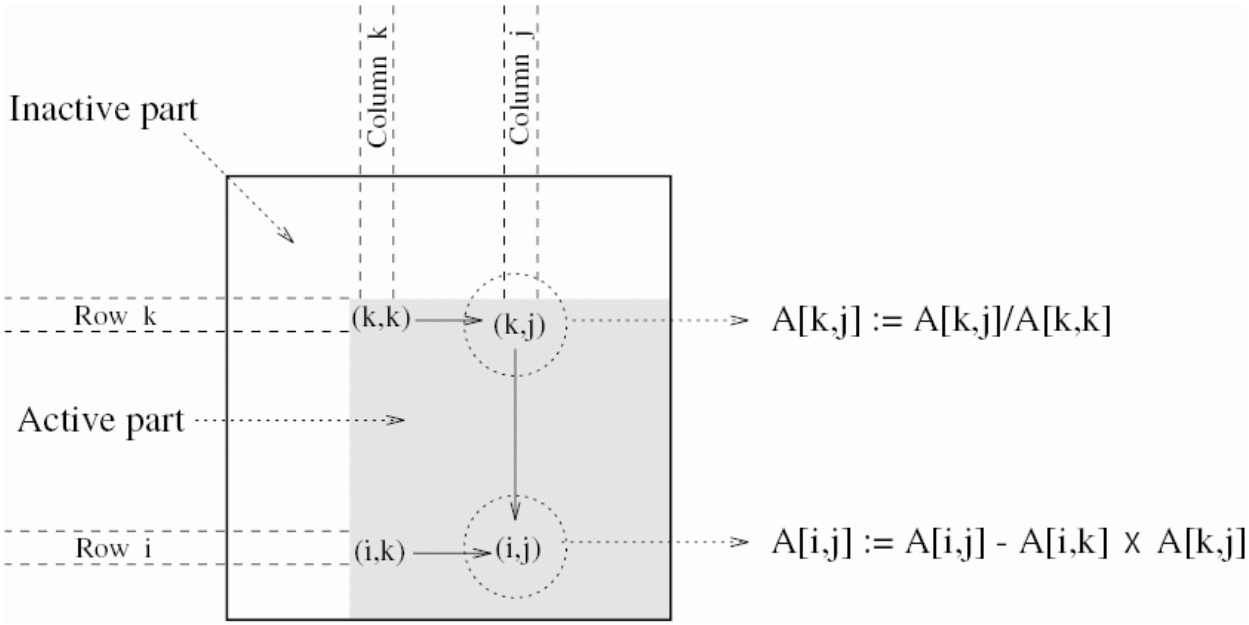

图 1.1: 高斯消去法示意图

1 问题描述

高斯消去的计算模式如图 1.1 所示，在第 k 步时，对第 k 行从 (k,k) 开始进行除法操作，并且将后续的 $k+1$ 至 N 行进行减去第 k 行的操作，串行算法如下面伪代码所示。

Algorithm 1 普通高斯消元算法伪代码

```
1:  function LU
2:      for k := 0 to n do
3:          for j := k + 1 to n do
4:              A[k, j] := A[k, j]/A[k, k]
5:          end for
6:          A[k, k] := 1.0
7:          for i := k + 1 to n do
8:              for j := k + 1 to n do
9:                  A[i, j] := A[i, j] − A[i, k] ∗ A[k, j]
10:             end for
11:             A[i, k] := 0
12:         end for
13:     end for
14: end function
```

观察高斯消去算法，注意到伪代码第 4，5 行第一个内嵌循环中的 $A[k,j] := A[k,j]/A[k,k]$ 以及伪代码第 8 9 10 行双层 for 循环中的 $A[i,j] := A[i,j]-A[i,k]\times A[k,j]$ 都是可以进行向量化的循环。可以通过 SIMD 扩展指令对这两步进行并行优化。

2 Pthreads 算法设计

源码链接：https://github.com/ArcanusNEO/Parallel-Programming/tree/master/4

2.1 测试用例的确定

由于测试数据集较大，不便于各个平台同步，所以采用固定随机数种子为 12345687 的 mt19937 随机数生成器。经过实验发现不同规模下，所有元素独立生成，限制大小在 $[0, 100]$，能够生成可以被正确消元的矩阵。

代码如下：

测试数据集生成器

```
uniform_real_distribution<float> dist(0, 100);
mt19937 mt(12345687);
int n;
istringstream iss(argv[1]);
iss >> n;
cout << n << endl;
for (int i = 1; i <= n; ++i)
  for (int j = 1; j <= n; ++j) cout << dist(mt) << " \n"[j == n];
```

2.2 实验环境和相关配置

实验在华为鲲鹏 ARM 集群平台和本地 Arch Linux x86_64 平台完成；

华为鲲鹏 ARM 集群平台使用毕昇的 clang++ 编译器，本地 Arch Linux x86_64 平台使用 GNU GCC 编译器；

使用 cmake 构建项目，编译开关如下：

```
set(CMAKE_CXX_FLAGS_RELEASE "-O3")
set(THREADS_PREFER_PTHREAD_FLAG ON)
find_package(Threads REQUIRED)
```

统一使用 1 管理线程 + 7 工作线程的 8 线程设计；

2.3 算法设计

2.3.1 默认平凡算法

使用一维数组模拟矩阵，避免改变矩阵大小时第二维不方便调整、必须设成最大值的问题，可以减少 cache 失效；

使用 $\#define\ matrix(i, j)\ arr[(i) * n + (j)]$ 宏，增强可读性；

平凡算法

```
#define matrix(i, j) arr[(i) * n + (j)]
void func(int& ans, float arr[], int n) {
```

Scale	Reperat times	x86 ordinary (s)	arm ordinary (s)
8 × 8	100	0.000001330460	0.000000525400
16 × 16	50	0.000001706920	0.000001666000
32 × 32	50	0.000003640080	0.000007127000
64 × 64	20	0.000015253300	0.000037566500
128 × 128	15	0.000098880800	0.000231574000
256 × 256	10	0.000716408500	0.001820356000
512 × 512	10	0.006722607300	0.014974396000
1024 × 1024	5	0.064893815400	0.135511226000
2048 × 2048	3	1.400074583333	1.101775523333
4096 × 4096	1	10.705585484000	13.088073440000

表 1: 所有平台平凡算法结果对比

```
  for (int k = 0; k < n; ++k) {
    for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      for (int j = k + 1; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
  }
#undef matrix
}
```

2.3.2 使用 Pthreads 动态创建线程并行化加速

动态创建线程 frame

```
#define matrix(i, j) arr[(i) *n + (j)]

#define MAX_SUB_THREAD 7

int     n;
float* arr;

struct thread_param_t {
  int k, t_id;
};

pthread_t       thread_handle[MAX_SUB_THREAD];
thread_param_t thread_param[MAX_SUB_THREAD];

```

```
  void* thread_func(void* param) {
    auto p    = (thread_param_t*) param;
    auto k    = p->k;
    auto t_id = p->t_id;
    int  i    = k + t_id + 1;
    for (int j = k + 1; j < n; ++j)
      matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
    matrix(i, k) = 0;
    pthread_exit(nullptr);
  }

  void func(int& ans, float arr[], int n) {
    ::n   = n;
    ::arr = arr;
    for (int k = 0; k < n; ++k) {
      for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
      matrix(k, k)     = 1.0;
      int worker_count = n - 1 - k;
      for (int offset = 0; offset < worker_count; offset += MAX_SUB_THREAD) {
        for (int t_id = 0, i = t_id + offset;
             i < worker_count && t_id < MAX_SUB_THREAD;
             ++t_id, i = t_id + offset) {
          thread_param[t_id] = {k, i};
          pthread_create(thread_handle + t_id, nullptr, thread_func,
                         thread_param + t_id);
        }
        for (int t_id = 0, i                                    = t_id + offset;
             i < worker_count && t_id < MAX_SUB_THREAD; ++t_id, i = t_id + offset)
          pthread_join(thread_handle[t_id], nullptr);
      }
    }
  }
  #undef matrix
}
```

2.3.3 使用 Pthreads 线程池和信号量同步并行化加速

线程池 + 信号量同步 + 主线程执行除法

```
#define matrix(i, j) arr[(i) *n + (j)]

#define MAX_SUB_THREAD 7

int    n;
float* arr;

struct thread_param_t {
  int t_id;
```

Scale	Reperat times	x86 dynamic (s)	arm dynamic (s)
8 × 8	100	0.000408343660	0.001069426000
16 × 16	50	0.001760123900	0.004892212000
32 × 32	50	0.007970805160	0.020191436600
64 × 64	20	0.031911559450	0.080842077000
128 × 128	15	0.130038008533	0.338396283333
256 × 256	10	0.515502232000	1.318216261000
512 × 512	10	2.105457739200	5.282467893000
1024 × 1024	5	8.558513793600	21.790754608000
2048 × 2048	3	33.120794944000	85.470935310000
4096 × 4096	1	139.134070615000	353.288384510000

表 2: 所有平台动态线程结果对比

```
};

sem_t             sem_main;
sem_t             sem_workerstart[MAX_SUB_THREAD];
pthread_t         handle[MAX_SUB_THREAD];
thread_param_t    param[MAX_SUB_THREAD];

void* thread_func(void* param) {
  auto p    = (thread_param_t*) param;
  auto t_id = p->t_id;
  for (int k = 0; k < n; ++k) {
    sem_wait(sem_workerstart + t_id);
    for (int i = k + 1 + t_id; i < n; i += MAX_SUB_THREAD) {
      for (int j = k + 1; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
    sem_post(&sem_main);
  }
  pthread_exit(nullptr);
}

void func(int& ans, float arr[], int n) {
  ::n   = n;
  ::arr = arr;
  sem_init(&sem_main, 0, 0);
  for (int i = 0; i < MAX_SUB_THREAD; ++i) sem_init(sem_workerstart + i, 0, 0);
  for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id) {
    param[t_id].t_id = t_id;
    pthread_create(handle + t_id, nullptr, thread_func, param + t_id);
  }
```

Scale	Reperat times	x86 semaphore (s)	arm semaphore (s)
8 × 8	100	0.000220605390	0.000414891100
16 × 16	50	0.000319653680	0.000546852800
32 × 32	50	0.000513098020	0.000823440200
64 × 64	20	0.000966301100	0.001490467500
128 × 128	15	0.001879047067	0.002734098000
256 × 256	10	0.003608040400	0.005964593000
512 × 512	10	0.011124012800	0.018504859000
1024 × 1024	5	0.050837437000	0.077263730000
2048 × 2048	3	1.160066842000	0.541380500000
4096 × 4096	1	11.558711337000	5.892652880000

表 3: 所有平台线程池 + 信号量同步 + 主线程执行除法结果对比

```
    for (int k = 0; k < n; ++k) {
      for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
      matrix(k, k) = 1.0;
      for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id)
        sem_post(sem_workerstart + t_id);
      for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id) sem_wait(&sem_main);
    }
    for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id)
      pthread_join(handle[t_id], nullptr);
    sem_destroy(&sem_main);
    for (int i = 0; i < MAX_SUB_THREAD; ++i) sem_destroy(sem_workerstart + i);
  }
  #undef matrix
}
```

线程池 + 信号量同步 + 工作线程执行除法

```
#define matrix(i, j) arr[(i) *n + (j)]

#define MAX_SUB_THREAD 7

int     n;
float* arr;

struct thread_param_t {
  int t_id;
};

sem_t           sem_leader;
sem_t           sem_div[MAX_SUB_THREAD - 1];
sem_t           sem_elim[MAX_SUB_THREAD - 1];
```

```
15  pthread_t       handle[MAX_SUB_THREAD];
16  thread_param_t  param[MAX_SUB_THREAD];
17
18  void* thread_func(void* param) {
19    auto p    = (thread_param_t*) param;
20    auto t_id = p->t_id;
21    for (int k = 0; k < n; ++k) {
22      if (t_id == 0) {
23        for (int j = k + 1; j < n; ++j)
24          matrix(k, j) = matrix(k, j) / matrix(k, k);
25        matrix(k, k) = 1.0;
26      } else sem_wait(sem_div + t_id - 1);
27      if (t_id == 0)
28        for (int i = 0; i < MAX_SUB_THREAD - 1; ++i) sem_post(sem_div + i);
29      for (int i = k + 1 + t_id; i < n; i += MAX_SUB_THREAD) {
30        for (int j = k + 1; j < n; ++j)
31          matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
32        matrix(i, k) = 0.0;
33      }
34      if (t_id == 0) {
35        for (int i = 0; i < MAX_SUB_THREAD - 1; ++i) sem_wait(&sem_leader);
36        for (int i = 0; i < MAX_SUB_THREAD - 1; ++i) sem_post(sem_elim + i);
37      } else {
38        sem_post(&sem_leader);
39        sem_wait(sem_elim + t_id - 1);
40      }
41    }
42    pthread_exit(nullptr);
43  }
44
45  void func(int& ans, float arr[], int n) {
46    ::n   = n;
47    ::arr = arr;
48    sem_init(&sem_leader, 0, 0);
49    for (int i = 0; i < MAX_SUB_THREAD - 1; ++i) {
50      sem_init(sem_div + i, 0, 0);
51      sem_init(sem_elim + i, 0, 0);
52    }
53    for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id) {
54      param[t_id].t_id = t_id;
55      pthread_create(handle + t_id, nullptr, thread_func, param + t_id);
56    }
57    for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id)
58      pthread_join(handle[t_id], nullptr);
59    sem_destroy(&sem_leader);
60    for (int i = 0; i < MAX_SUB_THREAD - 1; ++i) {
61      sem_destroy(sem_div + i);
62      sem_destroy(sem_elim + i);
63    }
```

Scale	Reperat times	x86 semaphore all (s)	arm semaphore all (s)
8 × 8	100	0.000295141480	0.000451200400
16 × 16	50	0.000439524060	0.000708521400
32 × 32	50	0.000699135760	0.001197251400
64 × 64	20	0.001294170900	0.001791016000
128 × 128	15	0.002380586333	0.004017182667
256 × 256	10	0.005267385000	0.009138907000
512 × 512	10	0.013941518400	0.020430503000
1024 × 1024	5	0.065091814800	0.088223814000
2048 × 2048	3	1.268976019000	0.604317156667
4096 × 4096	1	11.411054368000	5.209506790000

表 4: 所有平台线程池 + 信号量同步 + 工作线程执行除法结果对比

```
64  }
65  #undef matrix
```

2.3.4 使用 Pthreads 线程池和 barrier 栅栏同步并行化加速

线程池 + 栅栏同步 + 工作线程执行除法

```
1   #define matrix(i, j) arr[(i) *n + (j)]
2
3   #define MAX_SUB_THREAD 7
4
5   int     n;
6   float* arr;
7
8   struct thread_param_t {
9     int t_id;
10  };
11
12  pthread_barrier_t barrier_div;
13  pthread_barrier_t barrier_elim;
14  pthread_t         handle[MAX_SUB_THREAD];
15  thread_param_t    param[MAX_SUB_THREAD];
16
17  void* thread_func(void* param) {
18    auto p    = (thread_param_t*) param;
19    auto t_id = p->t_id;
20    for (int k = 0; k < n; ++k) {
21      if (t_id == 0) {
22        for (int j = k + 1; j < n; ++j)
23          matrix(k, j) = matrix(k, j) / matrix(k, k);
24        matrix(k, k) = 1.0;
25      }
```

```
26
27        pthread_barrier_wait(&barrier_div);
28
29        for (int i = k + 1 + t_id; i < n; i += MAX_SUB_THREAD) {
30          for (int j = k + 1; j < n; ++j)
31            matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
32          matrix(i, k) = 0.0;
33        }
34
35        pthread_barrier_wait(&barrier_elim);
36      }
37      pthread_exit(nullptr);
38    }
39
40    void func(int& ans, float arr[], int n) {
41      ::n   = n;
42      ::arr = arr;
43      pthread_barrier_init(&barrier_div, nullptr, MAX_SUB_THREAD);
44      pthread_barrier_init(&barrier_elim, nullptr, MAX_SUB_THREAD);
45
46      for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id) {
47        param[t_id].t_id = t_id;
48        pthread_create(handle + t_id, nullptr, thread_func, param + t_id);
49      }
50      for (int t_id = 0; t_id < MAX_SUB_THREAD; ++t_id)
51        pthread_join(handle[t_id], nullptr);
52
53      pthread_barrier_destroy(&barrier_div);
54      pthread_barrier_destroy(&barrier_elim);
55    }
56    #undef matrix
57  }
```

3 实验结果分析

观察比较表 1、2、3、4、5 可以发现使用线程池加信号量或者栅栏同步多线程的方式，在 arm 平台可以大幅提升性能，小数据量下效果微弱甚至不如平凡算法，但是大数据量有成倍提升。

4096 × 4096 数据规模下普通信号量同步用时 5.892652880000 s，平凡算法用时 13.088073440000 s，加速比 2.221083391；所有循环纳入线程回调函数用时 5.209506790000 s，加速比 2.51234406；使用栅栏同步可以继续获得性能提升，用时 5.017878950000 s，加速比 2.608287998。

然而，使用动态线程的方式处理会导致性能倒退，不如平凡算法：可以发现创建线程和销毁线程的开销巨大，造成性能下降。

比较表 1 两平台的平凡算法，可以发现本地 x86 平台单核性能领先于 arm 平台；但是比较表 3、4、5 中两平台可以发现，对于小数据量还是 x86 平台领先，但是 arm 平台在大数据量下表现更优，实现反超。在 x86 平台上使用信号量同步并将所有循环纳入线程回调函数或者栅栏同步时无助于性能提升。

Scale	Reperat times	x86 barrier (s)	arm barrier (s)
8 × 8	100	0.000356697420	0.000430526600
16 × 16	50	0.000631893380	0.000631723800
32 × 32	50	0.000695531360	0.000821696600
64 × 64	20	0.001590516150	0.001415785500
128 × 128	15	0.003217344067	0.003097213333
256 × 256	10	0.004891007500	0.005816473000
512 × 512	10	0.013557604400	0.017091460000
1024 × 1024	5	0.055628581000	0.082351834000
2048 × 2048	3	1.297545969000	0.503372826667
4096 × 4096	1	12.573134918000	5.017878950000

表 5: 所有平台线程池 + 栅栏同步 + 工作线程执行除法结果对比

Scale	Reperat times	x86 ordinary (s)	x86 dynamic (s)	x86 semaphore (s)	x86 semaphore all (s)	x86 barrier (s)
8 × 8	100	0.000001330460	0.000408343660	0.000220605390	0.000295141480	0.000356697420
16 × 16	50	0.000001706920	0.001760123900	0.000319653680	0.000439524060	0.000631893380
32 × 32	50	0.000003640080	0.007970805160	0.000513098020	0.000699135760	0.000695531360
64 × 64	20	0.000015253300	0.031911559450	0.000966301100	0.001294170900	0.001590516150
128 × 128	15	0.000098880800	0.130038008533	0.001879047067	0.002380586333	0.003217344067
256 × 256	10	0.000716408500	0.515502232000	0.003608040400	0.005267385000	0.004891007500
512 × 512	10	0.006722607300	2.105457739200	0.011124012800	0.013941518400	0.013557604400
1024 × 1024	5	0.064893815400	8.558513793600	0.050837437000	0.065091814800	0.055628581000
2048 × 2048	3	1.400074583333	33.120794944000	1.160066842000	1.268976019000	1.297545969000
4096 × 4096	1	10.705585484000	139.134070615000	11.558711337000	11.411054368000	12.573134918000

表 6: x86 平台所有结果对比

Scale	Reperat times	arm ordinary (s)	arm dynamic (s)	arm semaphore (s)	arm semaphore all (s)	arm barrier (s)
8 × 8	100	0.000000525400	0.001069426000	0.000414891100	0.000451200400	0.000430526600
16 × 16	50	0.000001666000	0.004892212000	0.000546852800	0.000708521400	0.000631723800
32 × 32	50	0.000007127000	0.020191436600	0.000823440200	0.001197251400	0.000821696600
64 × 64	20	0.000037566500	0.080842077000	0.001490467500	0.001791016000	0.001415785500
128 × 128	15	0.000231574000	0.338396283333	0.002734098000	0.004017182667	0.003097213333
256 × 256	10	0.001820356000	1.318216261000	0.005964593000	0.009138907000	0.005816473000
512 × 512	10	0.014974396000	5.282467893000	0.018504859000	0.020430503000	0.017091460000
1024 × 1024	5	0.135511226000	21.790754608000	0.077263730000	0.088223814000	0.082351834000
2048 × 2048	3	1.101775523333	85.470935310000	0.541380500000	0.604317156667	0.503372826667
4096 × 4096	1	13.088073440000	353.288384510000	5.892652880000	5.209506790000	5.017878950000

表 7: arm 平台所有结果对比

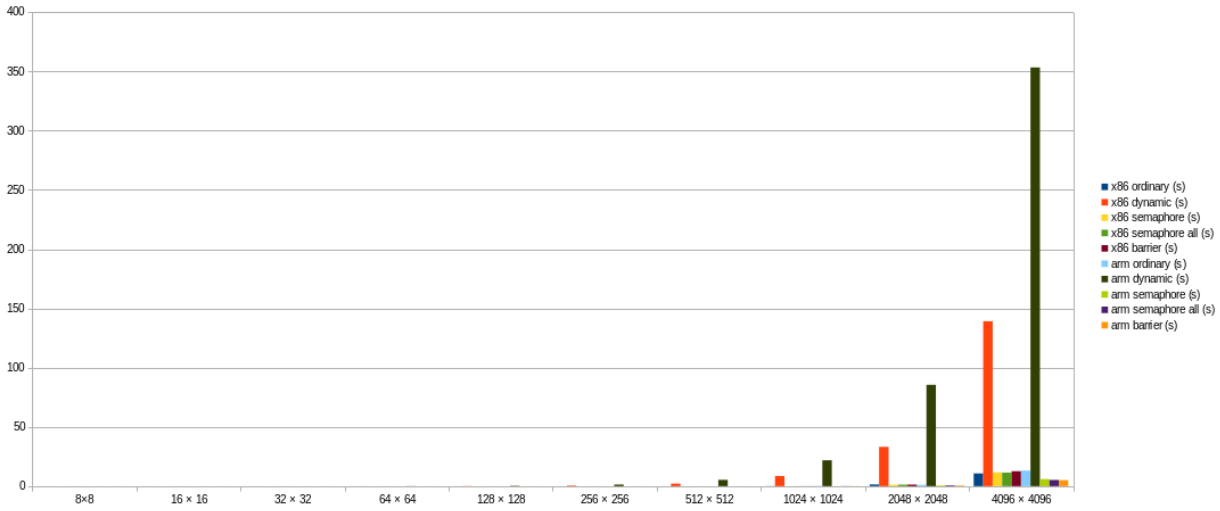

图 3.2: 所有平台所有结果对比柱状图

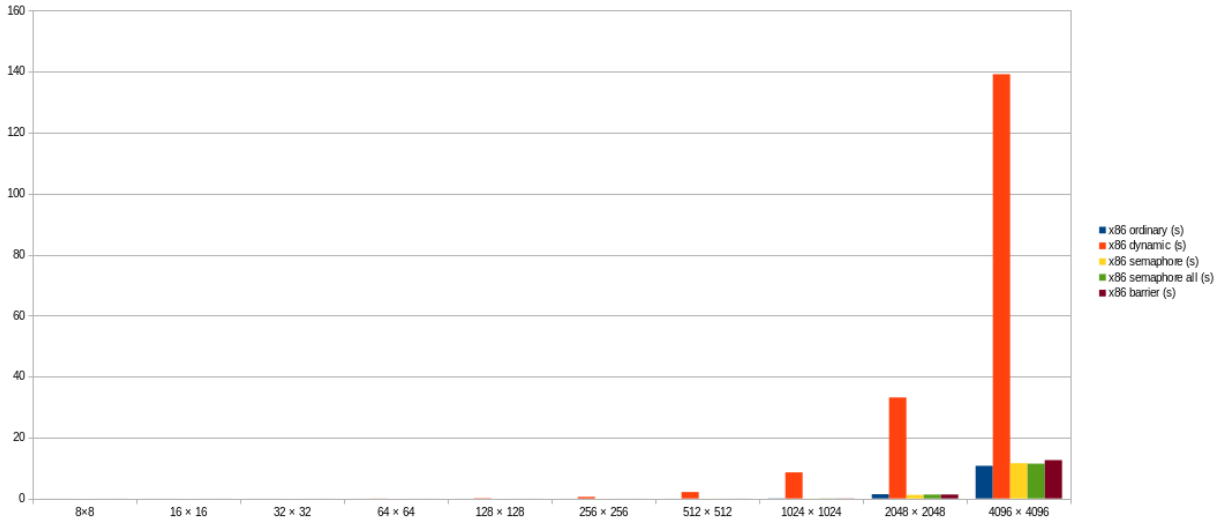

图 3.3: x86 平台所有结果对比柱状图

如图 3.5，对 barrier 使用 perf 进行分析可以发现并行度良好。

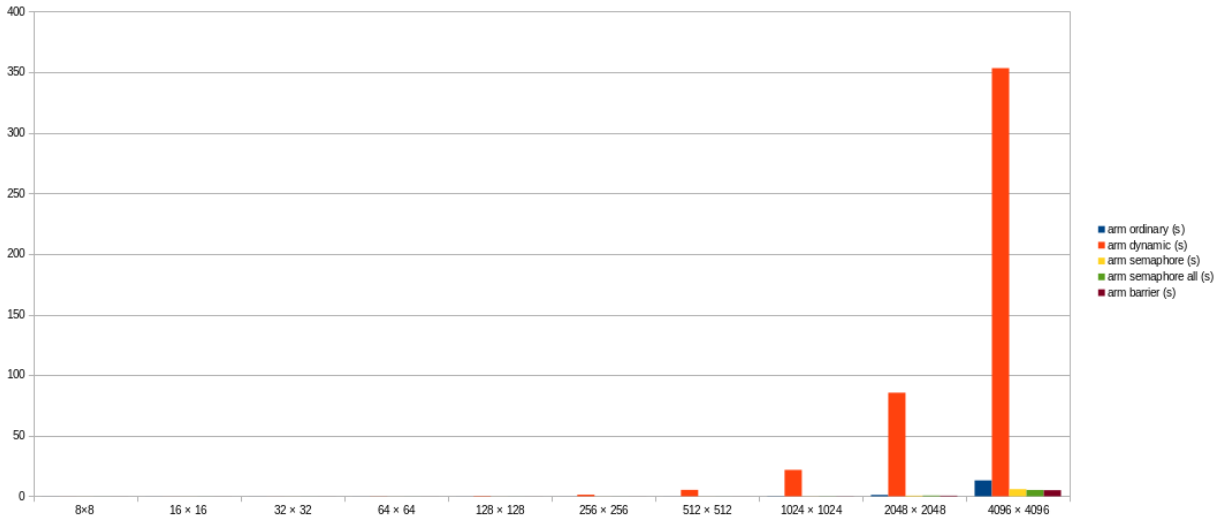

图 3.4: arm 平台所有结果对比柱状图

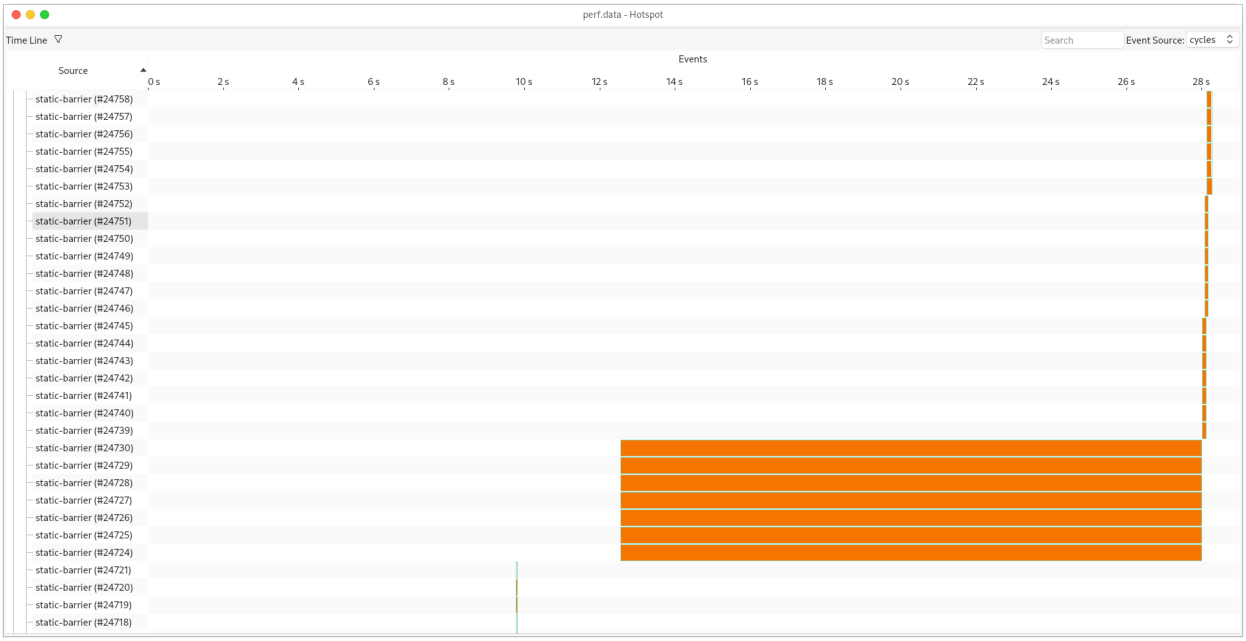

图 3.5: 对 barrier 使用 perf 分析