本报告评估了Julia语言在储能规划工具内的报告。本项目中涉及大量计算，涵盖变量多，算法迭代循环次数多。计算速度是其中成败关键之一。Julia 是一种兼备python和matlab 的一种小众语言，NERL 采用Julia 作为其REOPT工具的核心语言。项目之初会借鉴其经验，故而对其性能做出测试。

测试平台

Processor Intel(R) Xeon(R) w9-3495X 1.90 GHz

Installed RAM 256 GB (255 GB usable)

System type 64-bit operating system, x64-based processor

Edition Windows 11 Pro for Workstations

Version 22H2

Installed on ‎10/‎4/‎2024

OS build 22621.4317

Experience Windows Feature Experience Pack 1000.22700.1041.0

软件平台

|  |  |
| --- | --- |
| 编译器 | VS code |
| Julia 版本 | 1.8.3 |
| OS | Windows |
| CPU 核心数 | 56核心 112线程 |
| 测试时间 | 16 Oct 2024 |

代码

using Distributed

# 添加进程

for j in 1:61

    rmprocs(workers())

    addprocs(j)

    # 定义要执行的函数

    @everywhere function slow\_square(x,y)

        #sleep(1)  # 模拟长时间任务

        ans=0

        y=Float64(y)

        for i in 1:10000\*x

            ans=ans+i^1.2+y^6

        end

        #print(x,"  ")

        return ans

    end

    @time begin

    # 使用 pmap 将 slow\_square 应用到 1 到 10 的每个元素

        results = pmap(slow\_square, 1000:1500,1000:1500)

        println(j)

    end

#println("pmap 并行计算结果: ", results)

end

测试逻辑

测试系统最高支持的并行线程数量。并记录其线程计算时间。

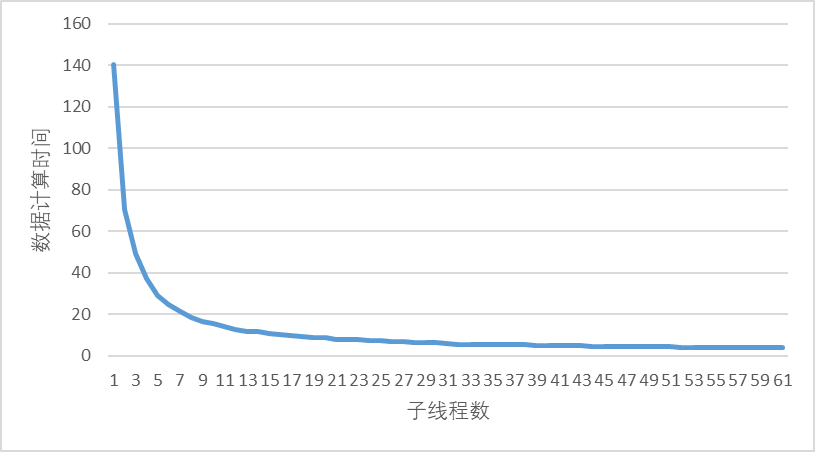
测试结果

并行线程支持的最大数量是61。超过61 程序会崩溃。该数字远少于系统提供的112线程数。查阅资料并未找到依据。Julia采用1个主线程+0或若干子线程的工作模式。61非 形式，故而建议在实际应用中采用<61的子线程数。考虑采用32，16，8，4的线程数确保稳定性。

单次创建线程所需时间为*4.747299 seconds (71.31 k allocations: 10.286 MiB)*

*运行数据所需时间如下*

|  |  |
| --- | --- |
| 子线程数 | 运行数据计算所需时间 |
| 1 | 140.247817 seconds (25.61 k allocations: 948.461 KiB) |
| 2 | 70.441373 seconds (25.39 k allocations: 922.836 KiB) |
| 3 | 49.286552 seconds (37.03 k allocations: 1.501 MiB, 0.04% gc time, 0.02% compilation time) |
| 4 | 37.391017 seconds (25.24 k allocations: 916.883 KiB) |
| 5 | 29.120293 seconds (25.26 k allocations: 923.195 KiB) |
| 6 | 24.694904 seconds (25.23 k allocations: 921.664 KiB) |
| 7 | 21.100860 seconds (25.25 k allocations: 923.008 KiB) |
| 8 | 18.638303 seconds (25.23 k allocations: 923.523 KiB) |
| 9 | 16.622064 seconds (25.23 k allocations: 921.977 KiB) |
| 10 | 15.363107 seconds (25.23 k allocations: 923.680 KiB) |
| 11 | 13.987786 seconds (25.25 k allocations: 924.961 KiB) |
| 12 | 12.811431 seconds (25.26 k allocations: 924.086 KiB) |
| 13 | 11.886954 seconds (25.26 k allocations: 924.555 KiB) |
| 14 | 11.633971 seconds (25.27 k allocations: 929.477 KiB) |
| 15 | 10.668288 seconds (25.27 k allocations: 935.477 KiB) |
| 16 | 10.093425 seconds (25.30 k allocations: 934.852 KiB) |
| 17 | 9.646589 seconds (25.30 k allocations: 930.336 KiB) |
| 18 | 9.158760 seconds (25.30 k allocations: 931.352 KiB) |
| 19 | 8.854581 seconds (25.31 k allocations: 938.805 KiB) |
| 20 | 8.563014 seconds (25.31 k allocations: 933.664 KiB) |
| 21 | 7.923295 seconds (25.32 k allocations: 941.133 KiB) |
| 22 | 7.805936 seconds (25.32 k allocations: 932.961 KiB) |
| 23 | 7.590929 seconds (25.36 k allocations: 926.461 KiB) |
| 24 | 7.281086 seconds (25.36 k allocations: 937.852 KiB) |
| 25 | 7.500795 seconds (25.38 k allocations: 928.930 KiB) |
| 26 | 7.027862 seconds (25.39 k allocations: 930.070 KiB) |
| 27 | 6.787767 seconds (25.39 k allocations: 949.133 KiB) |
| 28 | 6.413111 seconds (25.40 k allocations: 952.227 KiB) |
| 29 | 6.379535 seconds (25.39 k allocations: 950.977 KiB) |
| 30 | 6.161528 seconds (28.36 k allocations: 1000.727 KiB) |
| 31 | 6.026902 seconds (34.80 k allocations: 1.074 MiB) |
| 32 | 5.633686 seconds (35.09 k allocations: 1.093 MiB) |
| 33 | 5.620245 seconds (35.49 k allocations: 1.284 MiB) |
| 34 | 5.369084 seconds (35.95 k allocations: 1.702 MiB) |
| 35 | 5.627324 seconds (36.07 k allocations: 1.101 MiB) |
| 36 | 5.283953 seconds (36.51 k allocations: 1.089 MiB) |
| 37 | 5.258760 seconds (36.75 k allocations: 1.109 MiB) |
| 38 | 5.219272 seconds (37.12 k allocations: 1.230 MiB) |
| 39 | 4.900815 seconds (37.66 k allocations: 1.798 MiB) |
| 40 | 4.887378 seconds (37.85 k allocations: 1.888 MiB) |
| 41 | 4.888670 seconds (38.05 k allocations: 1.156 MiB) |
| 42 | 4.680589 seconds (38.50 k allocations: 1.141 MiB) |
| 43 | 4.691093 seconds (38.69 k allocations: 1.129 MiB) |
| 44 | 4.624887 seconds (39.02 k allocations: 1.209 MiB) |
| 45 | 4.631523 seconds (39.60 k allocations: 1.945 MiB) |
| 46 | 4.368357 seconds (39.85 k allocations: 1.191 MiB) |
| 47 | 4.359270 seconds (40.22 k allocations: 1.348 MiB) |
| 48 | 4.347226 seconds (40.55 k allocations: 2.020 MiB) |
| 49 | 4.294488 seconds (41.01 k allocations: 2.097 MiB) |
| 50 | 4.311680 seconds (41.25 k allocations: 1.193 MiB) |
| 51 | 4.268058 seconds (41.56 k allocations: 1.183 MiB) |
| 52 | 4.126945 seconds (42.23 k allocations: 2.124 MiB) |
| 53 | 3.983642 seconds (42.64 k allocations: 2.201 MiB) |
| 54 | 3.982178 seconds (42.63 k allocations: 1.218 MiB) |
| 55 | 3.969886 seconds (43.18 k allocations: 1.212 MiB) |
| 56 | 3.952869 seconds (43.49 k allocations: 1.739 MiB) |
| 57 | 3.913393 seconds (43.90 k allocations: 2.298 MiB) |
| 58 | 3.937786 seconds (44.19 k allocations: 1.230 MiB) |
| 59 | 3.905426 seconds (44.71 k allocations: 1.949 MiB) |
| 60 | 3.855922 seconds (45.06 k allocations: 2.344 MiB) |
| 61 | 3.842177 seconds (45.49 k allocations: 2.370 MiB) |



和python 相比，Julia 计算效率x 8

|  |  |  |  |
| --- | --- | --- | --- |
| 子线程数 | Julia 运行数据计算所需时间 | Python 运行数据计算所需时间 | |
| 1 | 140.247817 | 1122.48 | seconds |
| 2 | 70.441373 | 600.68 | seconds |
| 3 | 49.286552 | 404.23 | seconds |
| 4 | 37.391017 | 304.7 | seconds |
| 5 | 29.120293 | 247.8 | seconds |
| 6 | 24.694904 | 205.77 | seconds |
| 7 | 21.10086 | 171.42 | seconds |
| 8 | 18.638303 | 158.18 | seconds |
| 9 | 16.622064 | 139.89 | seconds |
| 10 | 15.363107 | 129.46 | seconds |
| 11 | 13.987786 | 119.89 | seconds |
| 12 | 12.811431 | 111.23 | seconds |
| 13 | 11.886954 | 103.04 | seconds |
| 14 | 11.633971 | 92.54 | seconds |
| 15 | 10.668288 | 92.24 | seconds |
| 16 | 10.093425 | 83.83 | seconds |
| 17 | 9.646589 | 82.8 | seconds |
| 18 | 9.15876 | 74.08 | seconds |
| 19 | 8.854581 | 74.71 | seconds |
| 20 | 8.563014 | 74.68 | seconds |
| 21 | 7.923295 | 66.45 | seconds |
| 22 | 7.805936 | 64.69 | seconds |
| 23 | 7.590929 | 64.17 | seconds |
| 24 | 7.281086 | 63.87 | seconds |
| 25 | 7.500795 | 64.25 | seconds |
| 26 | 7.027862 | 56.48 | seconds |
| 27 | 6.787767 | 56.01 | seconds |
| 28 | 6.413111 | 55.87 | seconds |
| 29 | 6.379535 | 55.23 | seconds |
| 30 | 6.161528 | 54.7 | seconds |
| 31 | 6.026902 | 55.07 | seconds |
| 32 | 5.633686 | 46.87 | seconds |
| 33 | 5.620245 | 46.35 | seconds |
| 34 | 5.369084 | 46.81 | seconds |
| 35 | 5.627324 | 46.29 | seconds |
| 36 | 5.283953 | 45.85 | seconds |
| 37 | 5.25876 | 46 | seconds |
| 38 | 5.219272 | 45.69 | seconds |
| 39 | 4.900815 | 45.47 | seconds |
| 40 | 4.887378 | 45.25 | seconds |
| 41 | 4.88867 | 44.89 | seconds |
| 42 | 4.680589 | 37.17 | seconds |
| 43 | 4.691093 | 36.95 | seconds |
| 44 | 4.624887 | 36.79 | seconds |
| 45 | 4.631523 | 36.71 | seconds |
| 46 | 4.368357 | 36.55 | seconds |
| 47 | 4.35927 | 36.6 | seconds |
| 48 | 4.347226 | 36.13 | seconds |
| 49 | 4.294488 | 35.97 | seconds |
| 50 | 4.31168 | 35.83 | seconds |
| 51 | 4.268058 | 35.49 | seconds |
| 52 | 4.126945 | 35.41 | seconds |
| 53 | 3.983642 | 35.5 | seconds |
| 54 | 3.982178 | 34.74 | seconds |
| 55 | 3.969886 | 34.75 | seconds |
| 56 | 3.952869 | 34.79 | seconds |
| 57 | 3.913393 | 34.58 | seconds |
| 58 | 3.937786 | 33.33 | seconds |
| 59 | 3.905426 | 33.96 | seconds |
| 60 | 3.855922 | 31.34 | seconds |
| 61 | 3.842177 | 29.86 | seconds |

import multiprocessing

import time

import math

# 定义要执行的函数

def slow\_square(x\_y):

    x, y = x\_y

    ans = 0

    y = float(y)

    for i in range(1, 10000 \* x + 1):

        ans += i\*\*1.2 + y\*\*6

    return ans

if \_\_name\_\_ == "\_\_main\_\_":

    for j in range(1, 62):

        # 清除之前的进程池

        with multiprocessing.Pool(j) as pool:

            # 记录开始时间

            start\_time = time.time()

            # 使用 map 将 slow\_square 应用到每个元素

            x\_values = range(1000, 1501)

            y\_values = range(1000, 1501)

            results = pool.map(slow\_square, zip(x\_values, y\_values))

            # 打印时间和进程数

            print(f"Number of processes: {j}")

            print(f"Elapsed time: {time.time() - start\_time:.2f} seconds")

    # 单独处理添加和删除进程池的部分（不太适用于 Python，清理进程池已经在上面实现）

    start\_time = time.time()

    with multiprocessing.Pool(61) as pool:

        pass  # 创建了61个进程，但不执行任何任务

    print(f"Elapsed time for cleanup: {time.time() - start\_time:.2f} seconds")