**程序设计语言认知实验——实验报告**

1. **实验目的和内容**

本实验的主要内容：

在Linux虚拟机上搭建虚拟环境，通过使用不同的编程语言编写矩阵乘法程序，进行编译或解释执行，并对程序规模，程序运行性能等进行对比分析，从而对不同的编程语言有着相对全面的认知，并综合分析各个语言的优势和劣势，并对不同的语言找到合适的应用场合。

本实验涉及到的编程语言如下：

1. C
2. Java
3. Python
4. Haskell
5. 汇编

实验的主要目的：

1. 了解不同编程语言的环境搭建和配置；
2. 了解编译器语言和解释语言之间的差别；
3. 了解不同语言的语法特性和编码复杂程度；
4. 了解不同语言的性能差异；
5. 综合分析不同语言的优势和劣势，了解某一语言在某方面热门的原因；
6. **实验的具体过程和步骤**
7. **在电脑上搭建Linux虚拟机**

由于汇编、Haskell等编程语言环境在Linux系统上较好搭建，因此本实验搭建环境、编写语言以及编译程序均在Linux虚拟机进行。这里Linux虚拟机选用VMWare，使用的系统为Ubuntu 20.04，系统架构为x86（可以在<https://releases.ubuntu.com/focal/>里下载，选择ubuntu-20.04.5-desktop-amd64.iso）。

本实验所用虚拟机的硬件配置情况如下：

|  |  |
| --- | --- |
| **硬件类型** | **参数** |
| 内存 | 4096MB |
| CPU | 2核 |
| CPU架构 | Intel x86 |
| 硬盘 | 40GB |
| 网络适配器 | NAT |
| Cache大小 | 128KB |

1. **环境搭建准备工作**

由于一些依赖包需要通过国外网站去获取，再加上国内特殊的网络环境，因此需要使用VPN去访问国外资源。由于本实验使用的是Linux虚拟机，因此需要了解Linux虚拟机下连接本机代理的方法。

本实验所使用的VPN为Clash for Windows。Clash for Windows的代理端口为7890，并选择“允许局域网连接“，Linux的网络适配器需设成“NAT模式”。在本地使用ipconfig命令获取Linux虚拟机对应的虚拟网卡ip地址，在本例为192.168.238.1。然后在Linux的设置的Network界面，将代理模式设置为“手动“，然后各项如下填写：

|  |  |  |
| --- | --- | --- |
| **名称** | **IP地址** | **端口** |
| HTTP Proxy | 192.168.238.1 | 7890 |
| HTTPS Proxy | 192.168.238.1 | 7890 |
| FTP Proxy | 无 | 无 |
| Socks Host | 无 | 无 |

如果Linux虚拟机能够访问Github，Google等网页，说明代理配置成功。

同时，我们需要安装一些必备的常用工具，例如vim，curl等：

|  |
| --- |
| sudo apt install vim  sudo apt install curl |

1. **汇编语言/C语言运行环境搭建**

在Linux虚拟机终端运行以下命令：

|  |
| --- |
| sudo apt-get install build-essential gdb |

就可以安装汇编语言运行环境所需要的依赖包，与此同时，C语言的编译环境也随即搭建好。之后，可以通过编写任意一个c语言程序（例如hello\_world.c）的方式，然后运行以下代码进行验证：

|  |
| --- |
| gcc -S -m32 hello\_world.c  gcc -o hello\_world\_asm -m32 hello\_world.s  ./hello\_world\_asm |

如果hello\_world\_asm能够成功运行，说明汇编语言运行环境搭建成功。

1. **Haskell运行环境搭建**

搭建Haskell运行环境，只需要安装ghcup（https://www.haskell.org/ghcup/）即可。在此之前，需要对Haskell进行手动换源，否则下载步骤会卡在“Downloading the latest package list from hackage.haskell.org“这一步。

运行以下命令创建”~/.cabal/config”配置文件，并进行编辑：

|  |
| --- |
| touch ~/.cabal/config  vim ~/.cabal/config |

编辑的内容如下：

|  |
| --- |
| repository mirrors.ustc.edu.cn  url: https://mirrors.ustc.edu.cn/hackage/  secure: True |

编辑完成后，执行以下命令自动安装ghcup（此时需要关闭VPN代理）：

|  |
| --- |
| curl --proto '=https' --tlsv1.2 -sSf https://mirrors.ustc.edu.cn/ghcup/sh/bootstrap-haskell | BOOTSTRAP\_HASKELL\_YAML=https://mirrors.ustc.edu.cn/ghcup/ghcup-metadata/ghcup-0.0.7.yaml sh |

在安装过程中，对于软件的一些询问，遵循以下指示：

* 是否将 ghcup 目录加入 PATH？直接回车接受默认值，本实验由于要和Visual Studio进行适配，因此一定会用到。
* 是否安装 Haskell Language Server？。这里输入 Y 安装。
* 是否安装 Stack？输入 N，不安装。

安装完成后，我们需要手动配置 GHCup 源。我们上面用了环境变量临时修改了GHCup元数据地址，这里我们把镜像写到配置里让国内源永久生效。修改”~/.ghcup/config.yaml”（如不存在就创建），添加如下内容：

|  |
| --- |
| url-source:  OwnSource: https://mirrors.ustc.edu.cn/ghcup/ghcup-metadata/ghcup-0.0.7.yaml |

执行以上步骤后，Haskell的环境配置流程正式完成。之后，可以运行以下命令进行环境配置验证：

|  |
| --- |
| $ ghc --version  The Glorious Glasgow Haskell Compilation System, version 8.10.7  $ cabal --version  cabal-install version 3.6.2.0  compiled using version 3.6.2.0 of the Cabal library  $ ghci  GHCi, version 8.10.7: https://www.haskell.org/ghc/ :? for help  Prelude> putStrLn "Hello, Haskell!"  Hello, Haskell!  Prelude>  Leaving GHCi. |

出现以上语句，说明Haskell环境搭建成功。我们也可以编写任意一个Haskell代码”helloworld.hs”，并执行以下命令进行解释运行：

|  |
| --- |
| runghc helloworld.hs |

同时，我们也可以在某一文件夹下创建cabal项目，在app目录下会多出来Main.hs文件：

|  |
| --- |
| cabal init |

我们可以使用以下命令运行这个Main.hs，此时Haskell的运行模式为编译型：

|  |
| --- |
| cabal run |

在cabal的dist-newstyle/build目录下，我们也可以找到相应的已编译后的可执行文件（可执行文件所在目录较深，需要手动提取到其他文件夹）。

1. **Visual Studio运行环境搭建**

安装完Haskell之后，我们就可以搭建Visual Studio环境，以做到实时编辑和运行代码，提高程序编写效率。在<https://code.visualstudio.com/Download>安装完Visual Studio的deb包之后，运行以下命令进行自动安装：

|  |
| --- |
| sudo dpkg -i code\_1.75.1-1675893397\_amd64.deb |

运行完之后，在Linux的应用程序列表中就会出现Visual Studio工具，直接点击即可。如果要让我们的Visual Studio能够运行C语言、python、Haskell代码，直接安装coderunner这个扩展即可。同时，安装Haskell相关插件可以对Haskell语法进行高亮显示和编译报错分析。

1. **Java运行环境搭建**

接下来进行Java的环境搭建流程，在以下这个网站下载安装包（jdk-19\_linux-x64\_bin.deb）：

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

安装完之后，运行以下命令进行自动安装deb包：

|  |
| --- |
| sudo dpkg -i jdk-19\_linux-x64\_bin.deb |

安装完毕之后，为了能够使得系统能够直接调用java命令，并且和IDEA相适配，需要手动配置java的环境变量：

|  |
| --- |
| vim ~/.bashrc |

然后在最后一行加入以下内容：

|  |
| --- |
| export PATH=$PATH:/usr/lib/jvm/jdk-19/bin/ |

配置完成之后，重启终端，运行以下命令检查java是否成功安装：

|  |
| --- |
| java –version |

如果正确显示出Java版本，说明Java环境搭建成功。

1. **IDEA运行环境搭建**

在<https://www.jetbrains.com/idea/download/>这个网站安装IDEA的Linux版本（选择社区版Community），然后运行以下代码对下载的tar.gz包进行解压缩：

|  |
| --- |
| tar -zxvf ideaIC-2022.3.2.tar.gz |

之后IDEA会自动解压缩到和压缩包原来的目录，这时候直接移动到桌面上，然后进入到bin目录，之后直接执行以下代码运行IDEA即可：

|  |
| --- |
| ./idea.sh |

配置好IDEA，然后指定jdk路径之后，我们就可以实时修改、编译和运行Java代码。Java代码 编译后的文件在out里面，文件后缀名为.class，之后单独提取出来即可。

1. **Python运行环境搭建**

在Linux上，Python的环境安装相对容易，直接运行以下命令即可：

|  |
| --- |
| sudo apt-get install python3 |

同时，要使得python --version命令工作（默认运行命令是python3），可以安装以下的包：

|  |
| --- |
| sudo apt install python-is-python3 |

为了验证Python环境搭建是否成功，可以执行以下命令：

|  |
| --- |
| python --version |

如果正确显示出python的版本，说明Python环境搭建成功。

1. **测试程序编写**

配置完上述环境之后，我们就可以进行测试程序的编写。在本实验选择的测试程序为矩阵乘法，并且分别使用汇编、C、Python、Java、Haskell这五种语言进行编写，并比较上述五种程序的运行效率。

关于测试程序的一些细节：

1. 汇编语言使用架构为x86，表示方法为GAS（GNU assembly syntax），并且使用gcc自带的命令编译为可执行文件。
2. 汇编、C、Python、Java实现矩阵乘法时均使用传统的三重for循环方法进行，理论时间复杂度均为，其中Python存储数据使用的数据结构为二维数组，而不是List。
3. 由于Haskell的语法特性，在实现矩阵乘法时并未使用三重for循环，而是使用函数式编程和自带的map函数实现，虽然理论时间复杂度也为，但是实际执行时要考虑代码优化带来的影响。
4. 以上五种语言输入数据时均使用文件读入，并且不向标准流输出最终矩阵的结果，以消除不同语言的输出语句对程序性能分析的影响。
5. 特别地，对于Haskell这一函数式编程语言，为了消除懒惰运算对于程序性能分析的影响，采用输出矩阵第一个元素的方式，让矩阵乘法在Haskell语言中能够实际执行。
6. Haskell既支持解释型运行，也支持编译型运行，在测试程序效率时，需要同时针对这两种运行模式进行分析，便于比较解释型语言和编译型语言运行性能的差异。
7. **生成数据程序**

由于测试时所用的数据规模较大，因此我们需要编写自动化程序随机生成数据。在此之前，我们先要定义数据的格式，以方便测试程序进行读入。数据的格式如下：

|  |
| --- |
| n m  x x x x x … x  x x x x x … x  ……  x x x x x … x  m2 t  x x x x x … x  x x x x x … x  ……  x x x x x … x |

其中第一行为两个数n m，分别代表矩阵A的行数和列数。第二行到第(n+1)行代表着矩阵A的每一项，第(n+2)行为两个数m2 t，代表矩阵B的行数和列数，第(n+3)行到第(n+m2+3)行代表着矩阵B的每一项，其中每个数字用空格隔开，同时每行的末尾不允许多出多余的空格。同时，m和m2必须相等，否则不能进行矩阵乘法运算（我们假定生成的数据均满足这一条件）。

基于这个思路，用python编写的程序生成脚本如下：

|  |
| --- |
| import os  import random  import click  @click.command()  @click.option('--n', help='Ma rows', type=click.IntRange(min=0), default = 100)  @click.option('--m', help='Ma columns/Mb rows', type=click.IntRange(min=0), default = 100)  @click.option('--t', help='Mb columns', type=click.IntRange(min=0), default = 100)  def main(n, m, t):  with open('../data.txt', 'w') as f:  f.write(f"{n} {m}\n")  for i in range(n):  for j in range(m):  Aij = random.randint(100, 1000)  f.write(f"{Aij}")  if j < m-1:  f.write(" ")  f.write("\n")  f.write(f"{m} {t}\n")  for i in range(m):  for j in range(t):  Bij = random.randint(100, 1000)  f.write(f"{Bij}")  if j < t-1:  f.write(" ")  f.write("\n")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

其中使用了click库，意味着这个数据生成脚本可以在终端运行，并且可以采用以下的方式传递各个参数：

|  |
| --- |
| python data\_gen.py --n 300 --m 400 --t 500 |

例如在如上的命令中，我们将数据规模n设成300，m（也即m2）设成400，t设成500，借助以上的数据生成脚本和click库参数传递，我们就可以更高效地编写自动化的运行计时程序，并对比各个程序的运行效率。

1. **程序运行计时方法**

编写完测试程序和生成程序后，我们需要一个方便快捷的方式运行上述程序，并按照一个统一的标准进行计时。本实验的计时的方法是使用Linux自带的程序运行时间计时脚本time，同时，我们仅仅针对编译后的程序（如果语言为编译型）运行时间进行计时，忽略编译型语言的编译时间。编写的自动化计时程序program\_timing.sh如下：

|  |
| --- |
| echo -e "Time executed for ASM:"  cd Assembly/bin  time ./matrix\_mul  cd ../../  echo -e "\nTime executed for Haskell (interpreter):"  cd Haskell/src  time runghc matrix\_mul.hs  echo -e "\nTime executed for Haskell (compiled):"  cd ../bin  time ./matrix\_mul  cd ../../  echo -e "\nTime executed for C (compiled):"  cd C/bin  time ./matrix\_mul  cd ../../  echo -e "\nTime executed for Java (compiled):"  cd Java/bin  time java Main  cd ../../  echo -e "\nTime executed for python:"  cd Python/src  time python matrix\_mul.py  cd ../../ |

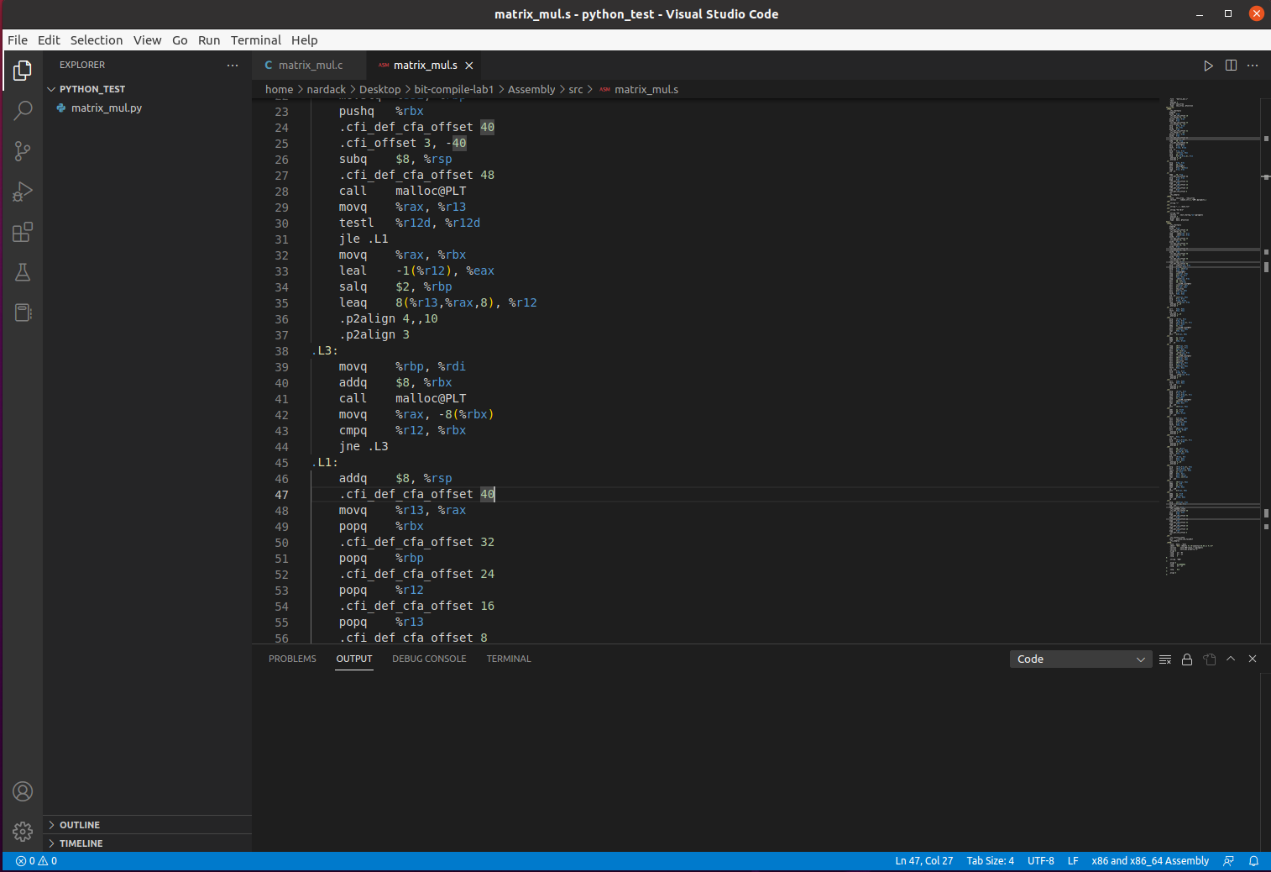
其中Haskell (interpreter)代表Haskell的解释型运行模式，Haskell (compiled)代表Haskell的编译型运行模式。同时，由于我们需要多组数据，因此我们还需要配合程序自动生成脚本data\_gen.sh实现自动化的运行时间测试：

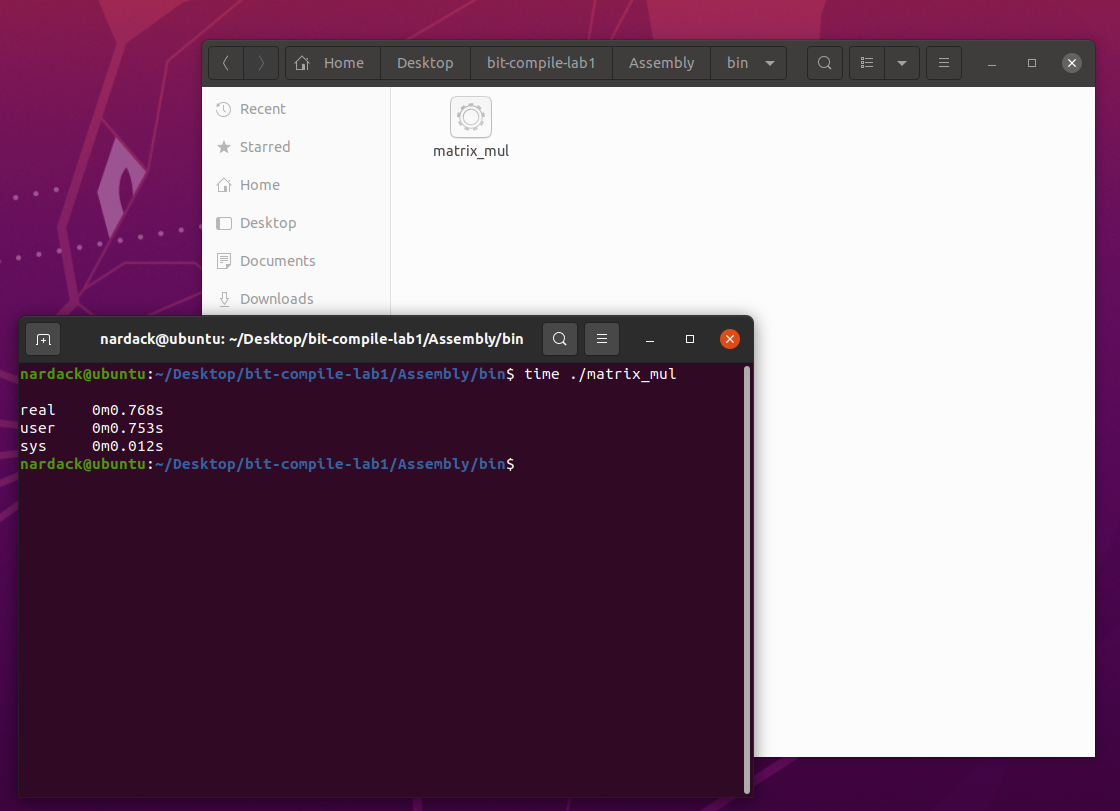
|  |
| --- |
| echo -e "Generating data for n=100, m=100, t=100"  cd data\_gen  python data\_gen.py --n 100 --m 100 --t 100  cd ../  ./program\_timing.sh  echo -e "----------------"  echo -e "Generating data for n=200, m=200, t=200"  cd data\_gen  python data\_gen.py --n 200 --m 200 --t 200  cd ../  ./program\_timing.sh  echo -e "----------------"  echo -e "Generating data for n=300, m=400, t=500"  cd data\_gen  python data\_gen.py --n 300 --m 400 --t 500  cd ../  ./program\_timing.sh  echo -e "----------------"  echo -e "Generating data for n=500, m=600, t=700"  cd data\_gen  python data\_gen.py --n 500 --m 600 --t 700  cd ../  ./program\_timing.sh  echo -e "----------------"  echo -e "Generating data for n=800, m=900, t=1000"  cd data\_gen  python data\_gen.py --n 800 --m 900 --t 1000  cd ../  ./program\_timing.sh  echo -e "----------------"  echo -e "Timing finished!" |

执行data\_gen.sh这个脚本之后，我们就可以一次性地对不同的数据规模下，不同语言的运行时间进行测试。选取的数据规模样本点如下：

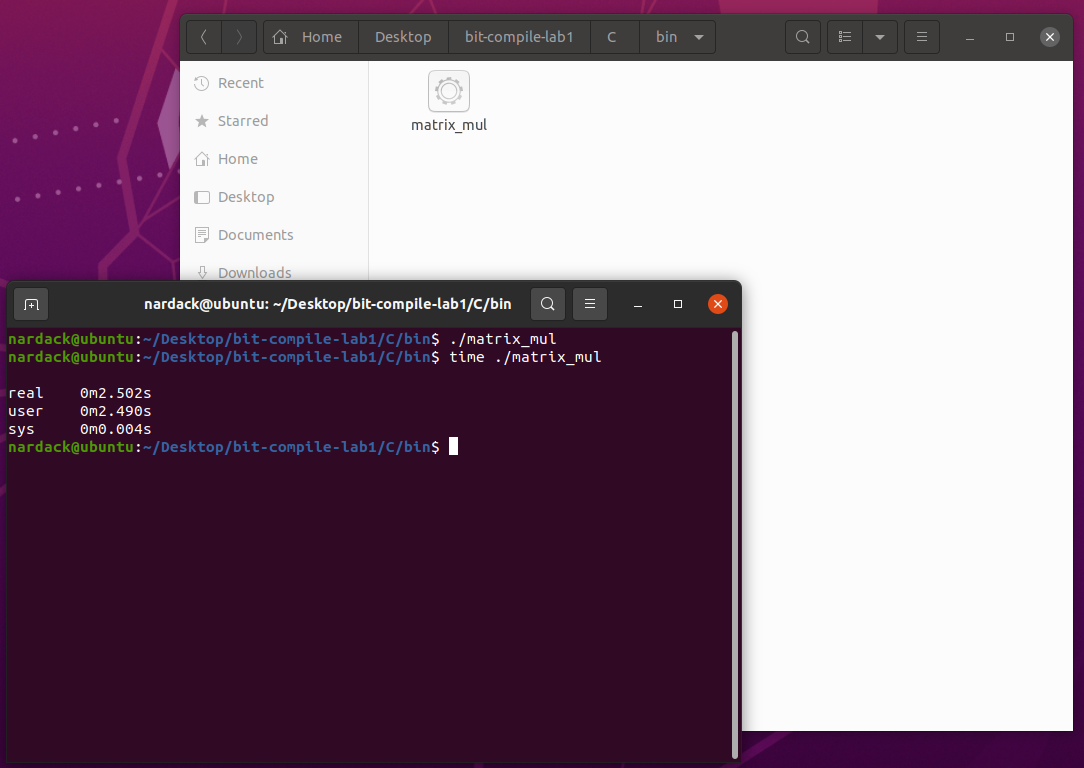
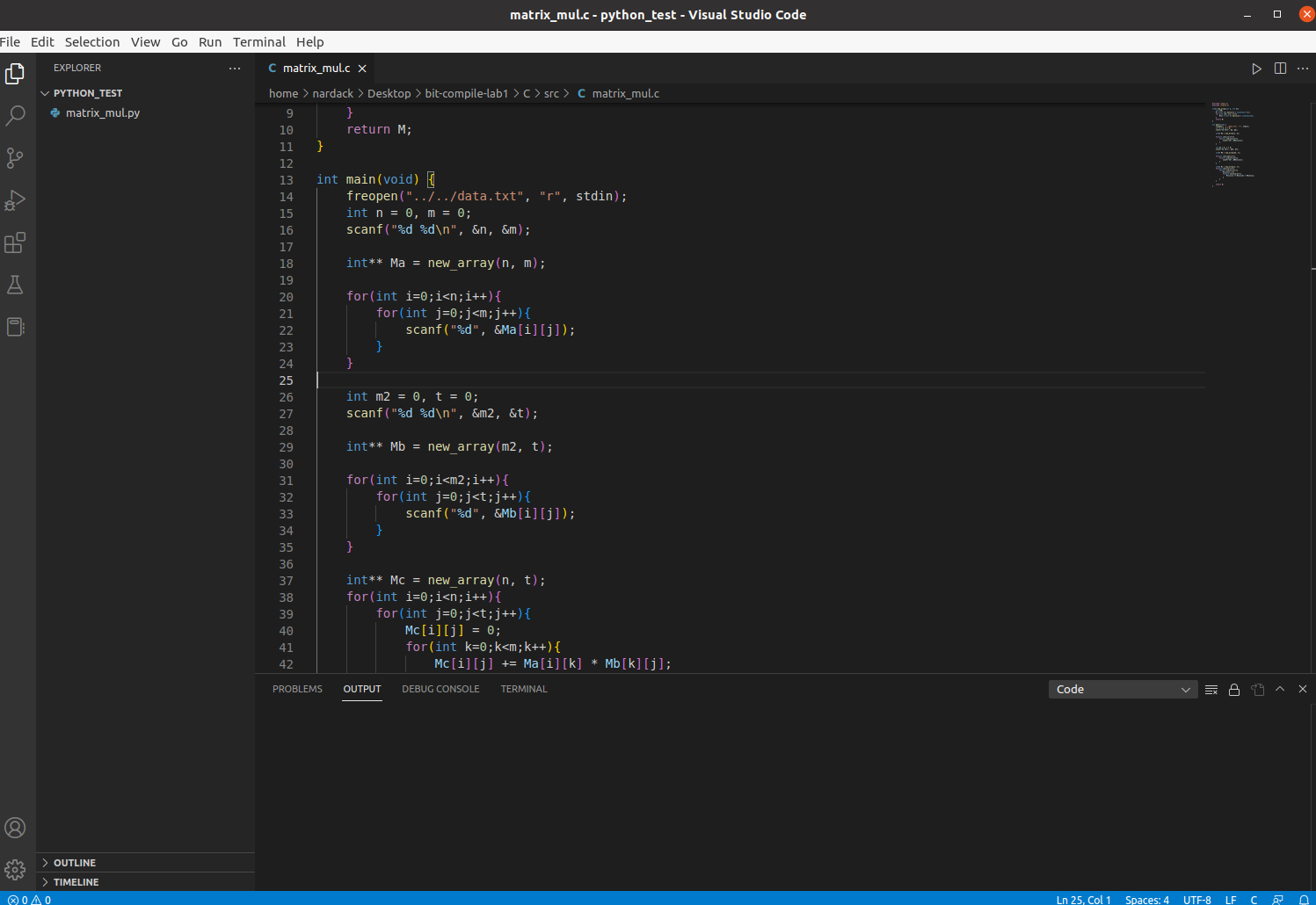
|  |  |  |
| --- | --- | --- |
| **n** | **m** | **t** |
| 100 | 100 | 100 |
| 200 | 200 | 200 |
| 300 | 400 | 500 |
| 500 | 600 | 700 |
| 800 | 900 | 1000 |

1. **运行效果截图**
2. **汇编语言运行截图**

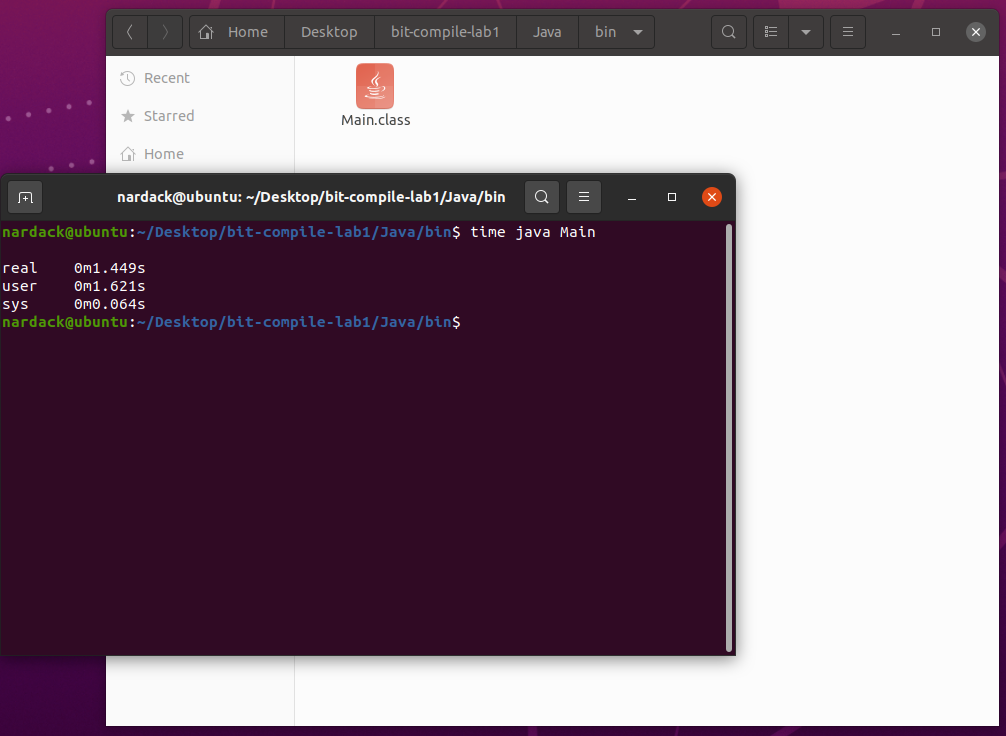
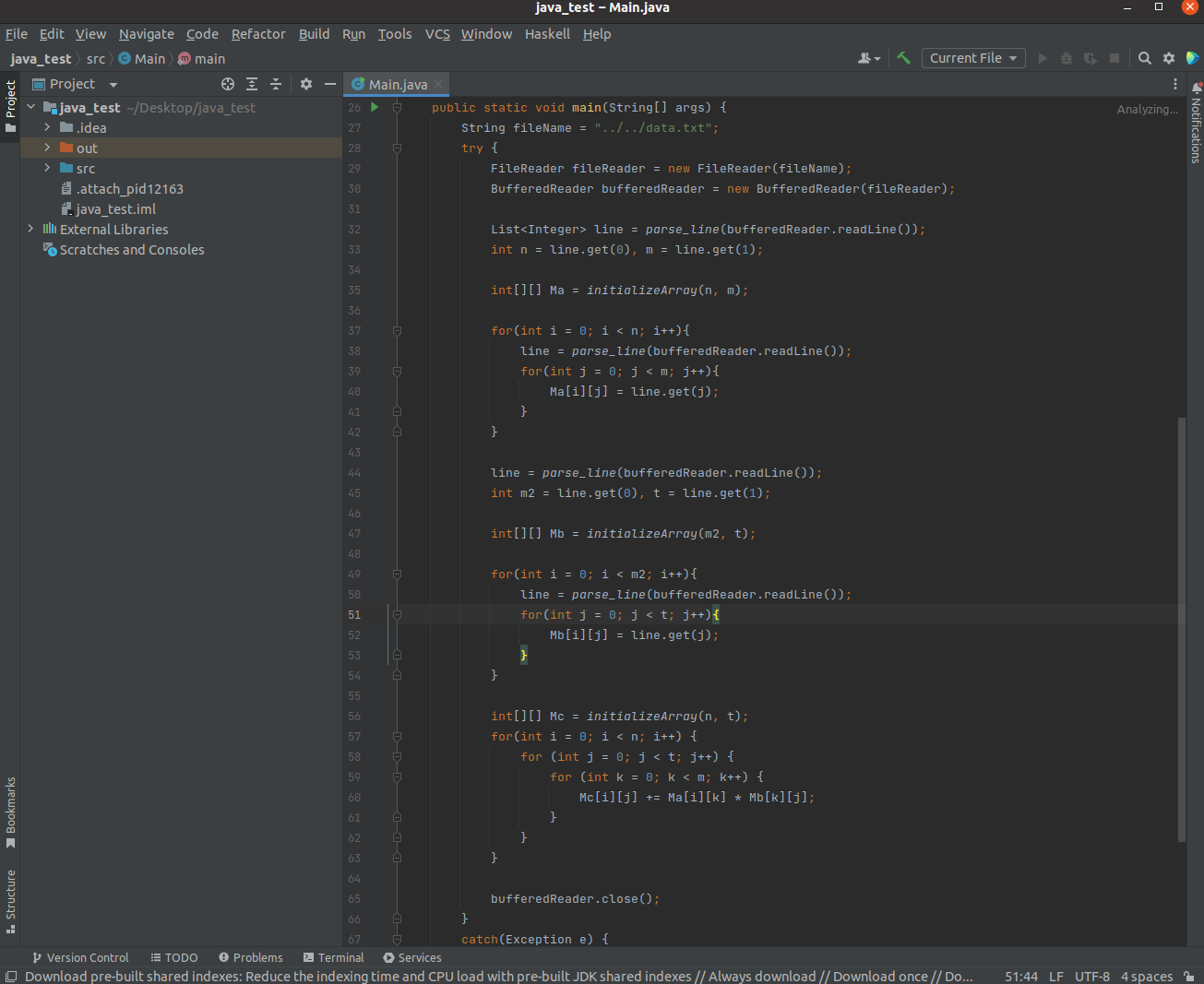




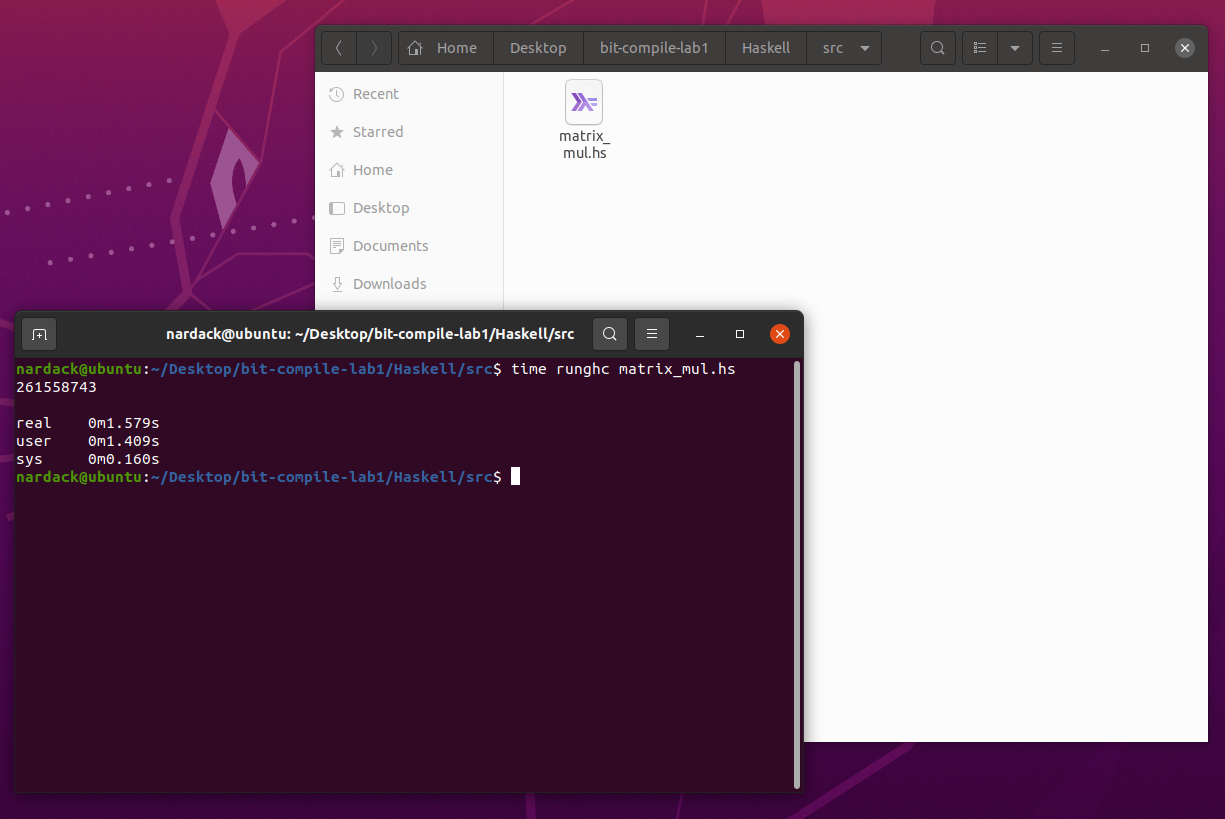
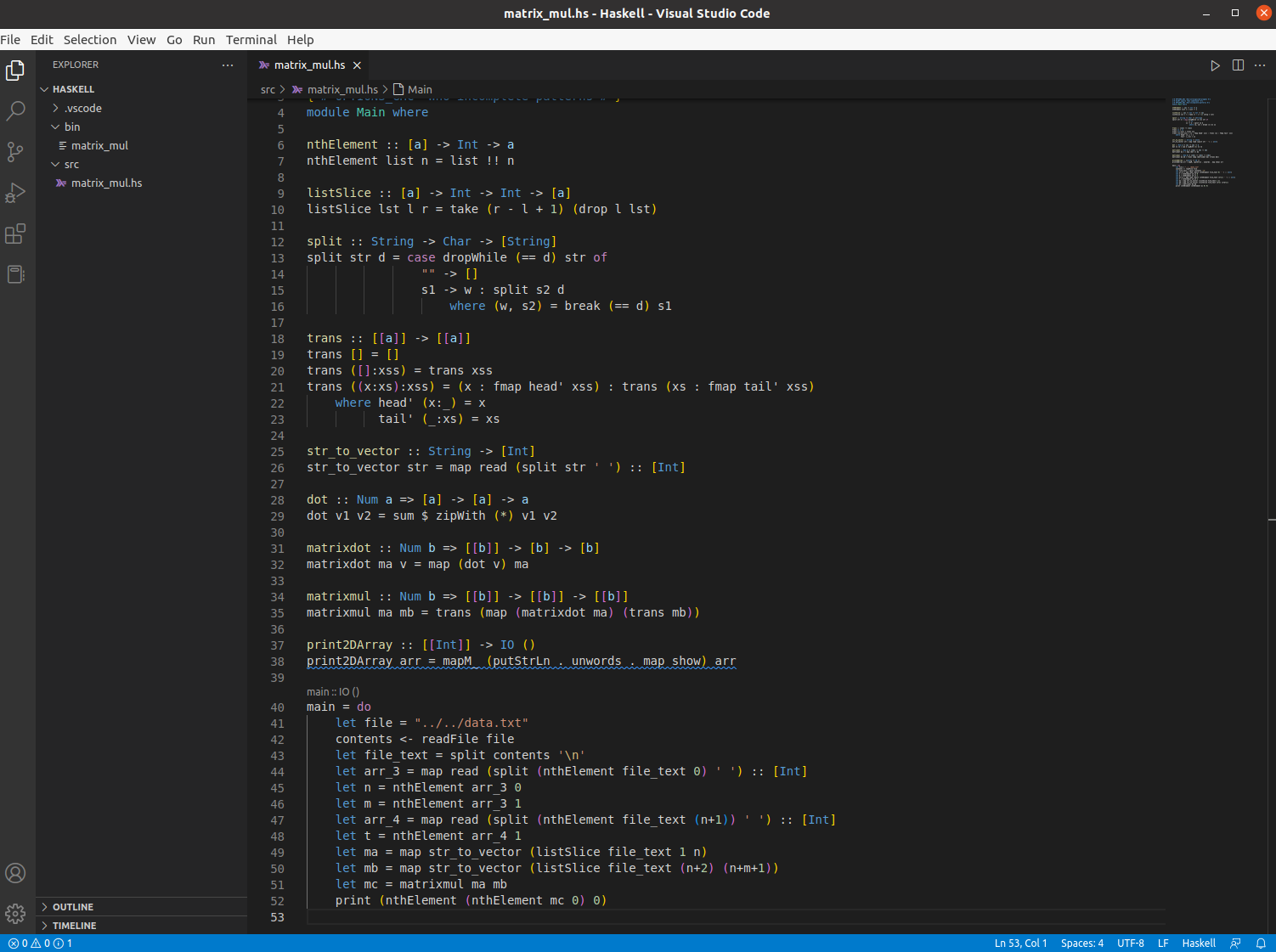
1. **C语言运行截图**



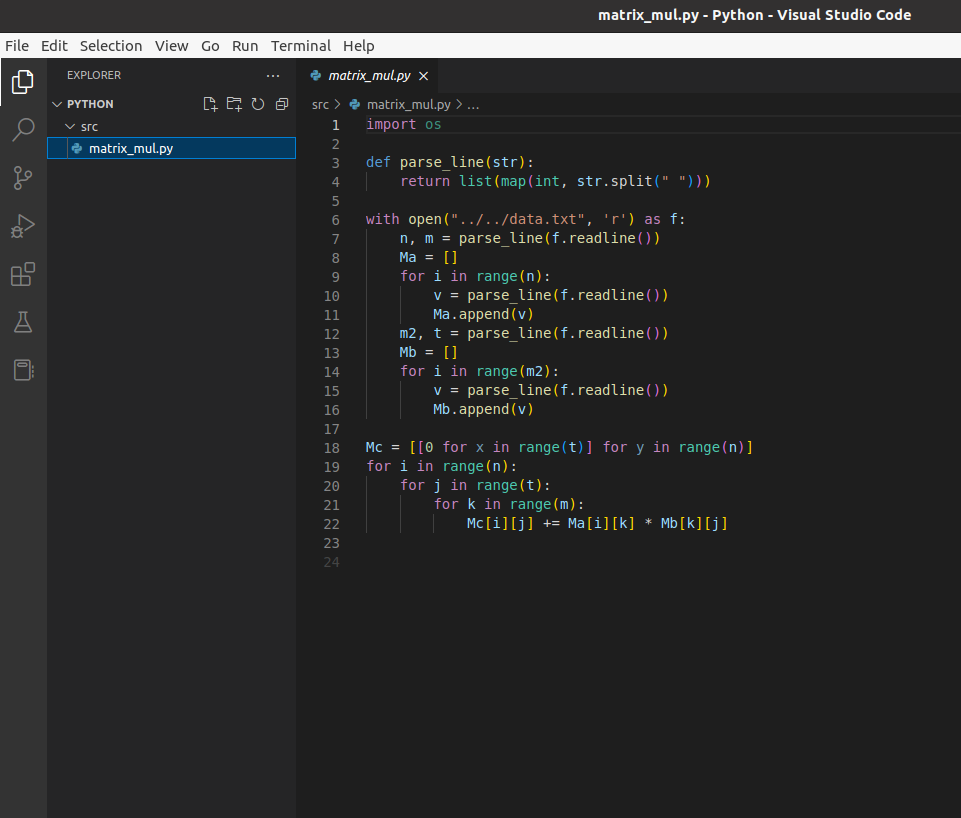
1. **Java语言运行截图**



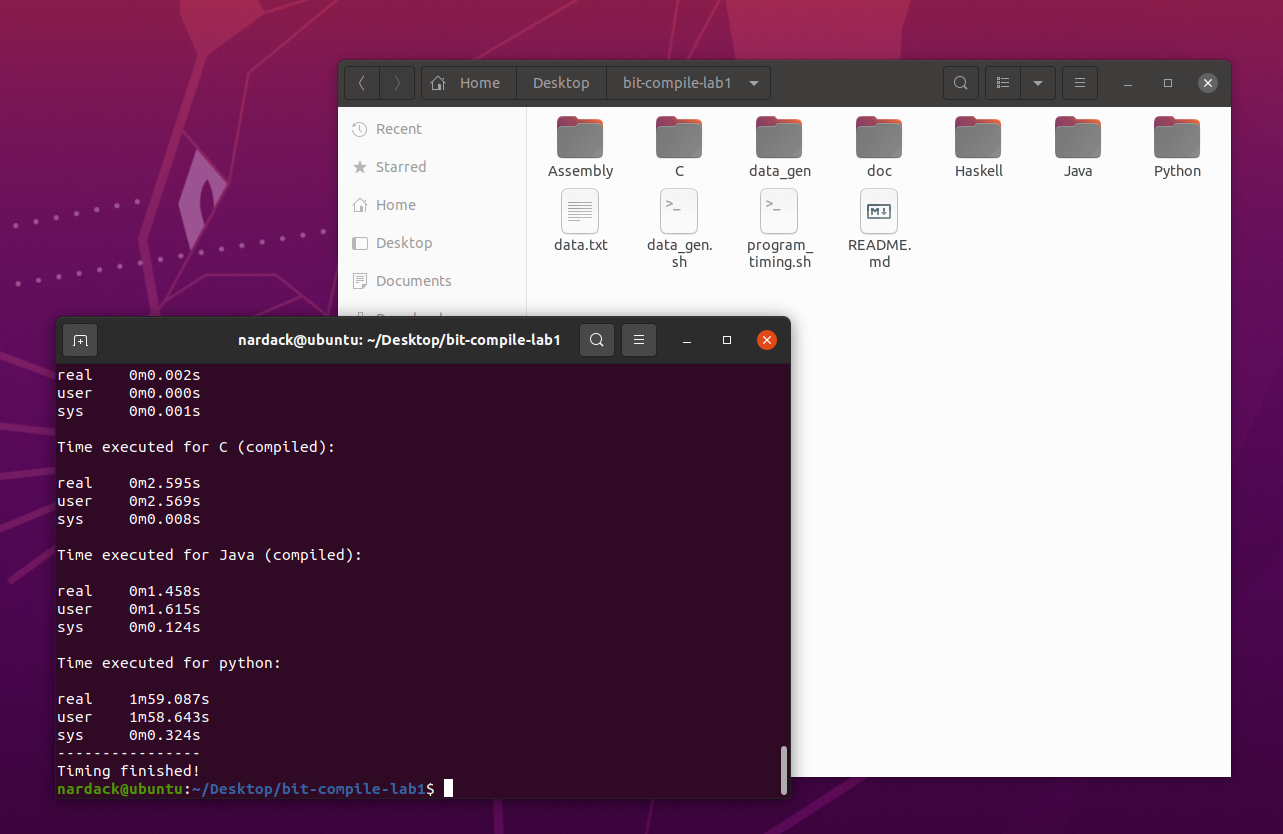
1. **Haskell语言运行截图**



1. **Python语言运行截图**

****

1. **程序计时运行截图**

****

1. **语言易用性和程序规模对比分析**
2. **易用性，程序规模汇总表格**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **编程语言** | **源码行数** | **语法复杂度** | **语言范式支持** | **集成功能数** | **环境配置难度** | **调试难度** | **易用性整体评价** |
| 汇编 | 284行 | 低 | 较为复杂的面向过程 | 无 | 难 | 非常高 | 非常不易用 |
| C | 48行 | 中 | 较为容易的面向过程 | 较少 | 中等 | 中等 | 中等 |
| Java | 72行 | 较高 | 面向过程，面向对象 | 中等 | 中等 | 较低 | 较为易用 |
| Haskell | 52行 | 高 | 仅函数式 | 较多 | 非常难 | 较高 | 不易用 |
| Python | 22行 | 非常高 | 面向过程，面向对象，函数式 | 非常多 | 容易 | 非常低 | 非常易用 |

1. 语言易用性对比分析

由以上表格再加上编写程序的感受，可以得出程序的易用性和各个指标的关系如下：

1. 调试难度越高，程序易用性越低。以汇编和Python为例：

* 汇编语言调试时，由于操作较为繁杂，接近底层，很难跟踪各个变量的值，输出各个变量的值都显得非常困难，因此学习汇编需要非常强的基本功作为支撑。
* Python作为解释型语言，对于变量类型的约束性较低，可以做到随调随改，可以很容易定义函数和提取功能，并且输出各个变量、整个列表也非常简单，即使没有较强的编程基本功，也可以很容易通过上述方法实时调整，直到输出结果符合期望为止，因此学习Python的门槛较低，这也是Python成为主流语言的主要原因之一。

1. 集成功能数越多，语言范式支持越多，程序易用性越高。

* 汇编仅支持硬件底层的操作，即使是文件输入输出也需要自己手写处理函数，光是这一步就需要消耗大量时间，面向过程、面向对象编程范式实现也非常麻烦；
* C语言支持标准输入、输出，以及较为简单的面向过程编程范式，让输入输出处理变得简单很多，但是手动分配内存仍然较为麻烦。此外，C原生不支持面向对象编程；
* Java作为面向对象语言，在支持C的面向过程编程范式以外，还提供了面向对象的自动分配内存机制，使得程序员不需要担心内存泄漏的问题，并且提供了多样化的字符串处理函数，提高了程序的易用性；
* Haskell作为函数式编程语言，提供了map、fold等函数式处理函数，但是由于Haskell支持的编程范式较少，仅仅擅长函数式编程范式，不能直接支持for循环等面向过程写法，以及java的面向对象写法，因此易用性仍然不太理想；
* 而Python则集合了C，Java，Haskell等语言的范式和特点，同时支持面向过程、面向对象、函数式编程范式，并且提供了非常多的相关库和功能，如网络访问相关库requests，数值计算库numpy等，因此Python的易用性非常强，也是非专业人员适合入门的编程语言之一；

1. 环境配置难度越高，程序易用性越低。用户接触编程语言时，总是希望环境的搭建越简单越好，最好能够达成像Python这样的一次性安装，如果超过三步以上，并且需要使用命令行，则说明编程语言的环境搭建就较为困难。Java和C编译程序时需要使用命令行，但是由于有较多的集成IDE支持，并且有自动化的安装系统，因此Java和C的环境配置相对容易。而汇编的环境搭建需要对处理器类型和架构有充分的了解，并且Windows系统和Linux系统下支持的汇编类型和语法格式均不相同，因此对于普通用户入手较为困难。而Haskell的环境配置则相比于上述语言要难很多，需要安装例如cabal等各式各样的依赖包，还要充分考虑各个依赖包之间的冲突问题，再加上国内特殊的网络环境，导致需要换源才能顺利完成安装，因此Haskell的环境配置本身就需要有经验的程序员进行，而这一点也是尤其是在国内，Haskell不如C、Java和Python流行的原因，光是Haskell较为麻烦的环境配置流程，就注定着Haskell成为较为冷门的编程语言。
2. 语法的复杂程度和程序的易用性并无直接关联。这里说的语法的复杂程度是指编译这一编程语言所需的词语数量以及语法规则的复杂程度，汇编的语法程度非常低，仅仅是数十条关键指令，但是由于集成功能非常少，再加上调试难度非常高，因此是表中语言中最不易用的。C的语法程度中等，但是由于集成功能不如Java、Python之类较多，因此易用性也不如后两者。而Python是弱类型语言，相应的语法糖非常多，语法规则也非常繁杂，以致于Python需要使用解释型的方式执行，但是由于Python的集成功能非常多，因此面对大部分问题就可以采取较为简单的，并且和C、Java等类似的写法，因此Python是表中语言中最为易用的。然而，Haskell虽然语法复杂度高，支持的语法规则也非常复杂，但是由于Haskell是纯函数式编程语言，原生不支持程序员较为习惯的for循环等面向过程写法，并且理解Haskell语法本身的难度也非常高，因此Haskell也成为仅次于汇编的不易用的语言，学习曲线也非常陡峭。从以上信息可以看出，编程语言的语法复杂程度不决定程序的易用性，程序的易用性需要结合多方面因素以及实际使用体验综合考量。
3. 程序规模对比分析

由上面表格可知，汇编语言所需的行数最多，为284行，而python所需的代码行数最少，仅22行。综合程序的易用性、集成功能数来看，集成功能越多的程序，完成同一个需求所需的程序规模（代码行数）就越小，也就意味着用户相对来说越容易实现某一功能。因此，完成同一需求，程序规模越小的编程语言，用户上手的难度也相对越低。但是，这一点也需要用辩证的观点来看，例如Haskell语言和C语言的代码行数差不多，但是由于Haskell不支持原生的面向过程写法，因此就完成矩阵乘法这一任务而言，Haskell语言的实现难度显然要比C语言要高很多。仅根据程序规模判断编程语言的好坏，也是较为片面的。

1. **程序运行性能对比分析**
2. 程序运行性能表格

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编程语言/数据规模** | 100,100,100 | 200,200,200 | 300,400,500 | 500,600,700 | 800,900,1000 |
| 汇编 | 0.002s | 0.008s | 0.063s | 0.243s | 0.882s |
| Haskell（解释运行） | 0.158s | 0.226s | 0.434s | 0.745s | 1.601s |
| Haskell（编译运行） | 0.009s | 0.018s | 0.061s | 0.139s | 0.349s |
| C | 0.005s | 0.026s | 0.182s | 0.682s | 2.489s |
| Java | 0.079s | 0.089s | 0.221s | 0.537s | 1.389s |
| Python | 0.151s | 1.148s | 9.999s | 33.821s | 120.629s |

1. 程序运行性能对比

由上述表格可知，总体而言汇编语言的运行性能最高，Haskell其次，C和Java性能较高，而Python很明显就能感受到性能方面的巨大差距。同时，光论执行矩阵乘法这一任务来说，我们可以发现以下比较有意思的事实：

1. 在数据规模较小的情况下，Java的运行时间比C长，数据规模较大的情况下，Java的运行时间要比C短。
2. 数据规模较大的情况下，Haskell编译型运行下的性能甚至比汇编语言要高，即使是解释运行模式下，运行性能也比纯编译型的C或者Java要高。
3. Haskell在解释型运行和编译型运行模式下能够明显感觉到性能方面的差异，编译型运行模式显然要比解释型运行快很多。

在完成需求和所使用算法相同的情况下，产生上述现象的原因，很可能在于各个语言在编译过程中做出了一定程度的编译优化。尤其是Haskell这一函数式编程语言，本实验在编写矩阵乘法时虽然遵循着三重循环的思路，但实际实现时并没有使用for循环这一面向过程方法，而是采用了Haskell的函数式编程写法。

由上述现象可以得知，Haskell在针对map等函数式编程语句时做出了一定程度的编译优化和算法优化。同理，Java在编译过程中也进行了一定程度的编译优化，从而相比于C语言在数据规模较大时有较为明显的性能提升。

1. 解释型语言和编译型语言的性能差别

我们特别关注Haskell这一门既支持解释型运行也支持编译型运行的编程语言。从以上表中数据可以看出，Haskell在编译型模式下运行效率明显就要比解释型模式下要快很多。这一点也不难理解，因为解释型模式运行需要额外的程序对源码进行解析，而编译型模式运行只需要执行机器代码，很明显，在同种类型的编程语言中，编译型运行耗费的代价就明显要比解释型运行要低很多。

然而，对于不同种类型的编程语言，即使完成同一任务所用的算法是一样的，解释型运行语言和编译型语言的运行效率没有绝对的关系。比如上表中，Haskell解释运行模式下运行性能比纯编译型的C或者Java要高，就是一个最好的例子。当然，由于Python的语法较为繁杂，解释运行模式下所耗费的工作量较大，因此在上表语言种性能最低。

解释型语言和编译型语言的运行性能，要综合语言的语法复杂度，以及编译器编译过程中做出的优化综合来看。只有像Haskell这种既支持解释型运行也支持编译型运行的语言，同等情况下比较运行效率的差异才是有意义的。

1. 程序性能优化分析

我们如果关注汇编、C、Java、Haskell（编译型）这四个语言的编译速度，我们会发现汇编最快、C其次、Java较慢，而Haskell则需要更长的时间。而产生编译速度差异的主要原因，一方面和语言的语法复杂程度有关，另一个最主要的原因，就是编译时做出的优化有多少。汇编语言的语法复杂度相对较低，并且不生成中间代码，因此编译速度自然最快。

从程序运行性能表格我们可以看出，Haskell相比于C和Java的执行速度要快很多，而达成这方面的代价是，Haskell的编译速度相比于C和Java就要慢一些，一方面和Haskell较高的语法复杂度相关，另一更关键的因素是Haskell编译期间做出的优化非常多，尤其是针对map等函数式编程函数做出了并行计算等优化和处理。

而对于Python这样的解释型语言，由于没有编译过程，因此我们不谈编译速度，相当于Python代码不需要等待即可立即执行，这给大型项目的调试带来了较大的方便，也是Python对于新人友好的一大原因之一。而和Python无需编译的优势相对应的劣势，就是较低的性能。

不管怎样，在编译速度、运行速度这两方面，需要找到较好的平衡，为了过分追求运行速度而加长编译时间，延误调试进度，以及完全不进行编译优化导致性能大幅度降低，都是不合适的。我们要根据项目的实际需求，针对不同的模块，对这两方面进行合理的调整。

1. **实验心得体会**

本次实验中，本人在Linux上对C、Java、Haskell、Python、汇编这几门语言进行了完整的环境搭建，既提升了本人的工程能力，也加深了对于各个语言的特性、易用性和运行性能的理解，同时认识到了各个语言擅长的部分，以及相应的优势和劣势。而认识这一点之后，我们就可以针对不同的应用场合，选择恰当合适的编程语言：

* 汇编：适用于直接接触硬件底层，需要处理的数据较少，并且极致追求运行效率的场合；
* C：适用于单片机、信号处理、硬件编程等较为底层，需要处理的数据较多，并且需要追求一定效率的场合；
* Java：适合面向对象以及大型项目的开发；
* Python：适合网络爬虫、数值运算、机器学习等较高层次的、需要大量算法和现有库使用的场合，但不适合开发大型项目；
* Haskell：适合极致追求效率的底层算法实现；

总而言之，对这几门语言进行代码编写和性能分析方面的实践，不仅对于编译原理的学习有好处，也极有助于拓宽自己的视野，更深层次了解各个语言的应用场合，提升自身的软件工程综合素养。在今后的工程实践中，这一意识也是非常重要的。