```
\documentclass{article}

\usepackage[utf8]{inputenc}

\usepackage{enumerate}

\usepackage{amsmath}

\usepackage{graphicx}


\title{Lab 7}

\author{Jack Jiang}

\date{April 16, 2021}


\begin{document}


\maketitle


\section*{Tasks}

\begin{enumerate}[(a)]

    \item It is better to separate the distribution and collection of the array from the computation of the
parallel operation because when the vector is parallelized, the vector is scattered beforehand.
Therefore, each vector possibly has different lengths and they do not share the total length of the
original vector. So when calling functions on these local scattered vectors, the computations will be
different because the norm calculates the local length and normalize then divides each vector's
component by the local length, and this leads to incorrect normalization of the vector. Performing the
distribution and collection of the array from the computation will allow us to do the computation
correctly because we can compute the correct length when doing it separately by summing the square
of the norm of the local vectors and using a sum reduction to get the total length of the vector, then
normalizing the local vectors with the correctly computed total length of the vector.


    If we have a vector of length $9$, the scattered vector will scatter it into $3$ local vectors of length
$3$. The parallelized operation will then compute the normalization with its local vector length of $3$,
when it should be using a length of $9$. In order to fix this, we must tell the processors the length is
$9$, not $3$. What we can do is calculate the norm of the local vector, square it, then use a sum
reduction, and then broadcast that to all the other ranks to get the total length of the vector, $9$, and
normalize it with the total length.
```

```latex
    \item

    Strong scalability study plot:

    \begin{center}

        \includegraphics[strong]{strong.png}

    \end{center}


    Weak scalability study plot:

    \begin{center}

        \includegraphics[weak]{weak.png}

    \end{center}

    Some of the changes in performance in the plots appear most likely due to other users of TinkerCliffs
being on the same node the scalability study was run on.

    The implementation that appears to perform the best is the mpi function. It has the lowest times
overall. I think this is the case because MPI is designed to use the computers resources properly and
efficiently, so the one MPI call in the function does so. The function with 2 MPI calls performs similarly,
until the $n_procs$ gets to $200$, then it sort of blows up. Theoretically, mpi should be the quickest due
to the design of this lab and MPI's capabilities and it holds true. I think MPI is a very powerful tool to use
as a parallel processing interface in high performance computing.


    \item I only received help from instructor and TAs.

\end{enumerate}


\end{document}
```