Message Parsing Interface (MPI)

Differences between OpenMP and MPI

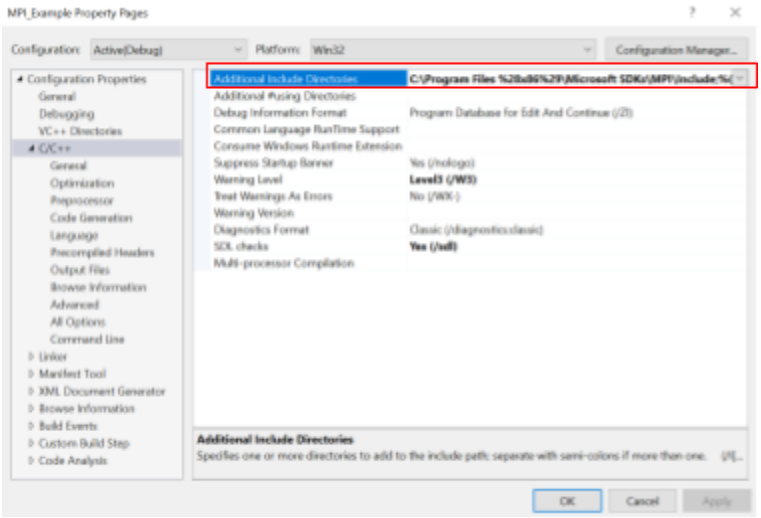
OpenMP	MPI
Add pragmas to existing program	Must rewrite program to describe how single process should operate on its data and communicate with other processes
Compiler + runtime system arrange for parallel execution	Explicit data movement: programmer must say exactly what data goes where and when
Rely on shared memory for communication	Advantage: Can operate on systems that don't have shared memory

Configuring MPI in Visual Studio

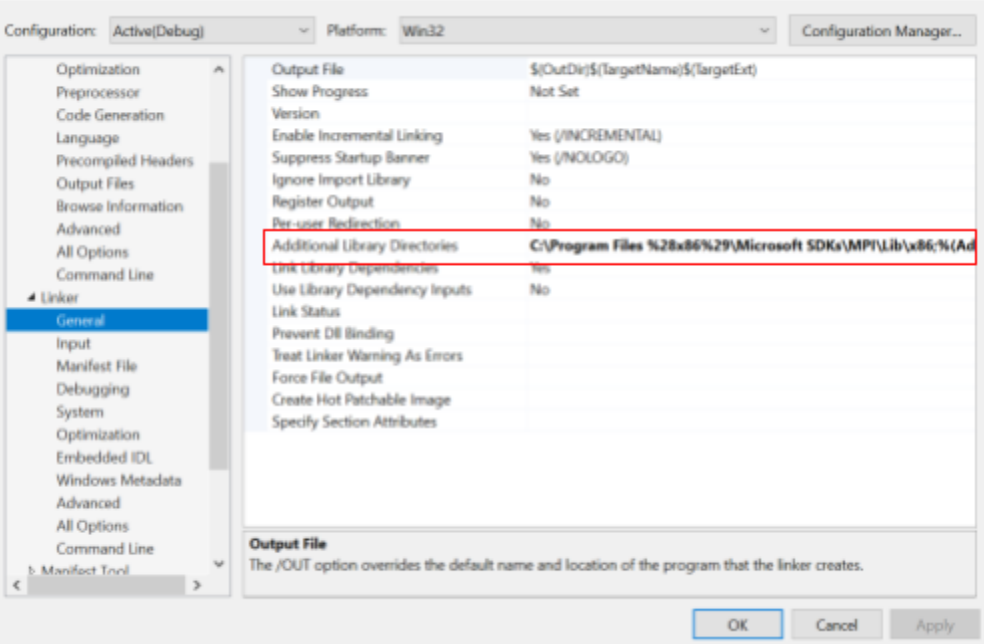
1. Download Microsoft MPI here, please install **both** files:

   Install ***msmpisetup.exe*** for MPI executable file
   Install ***msmpisdk.msi*** for MPI library

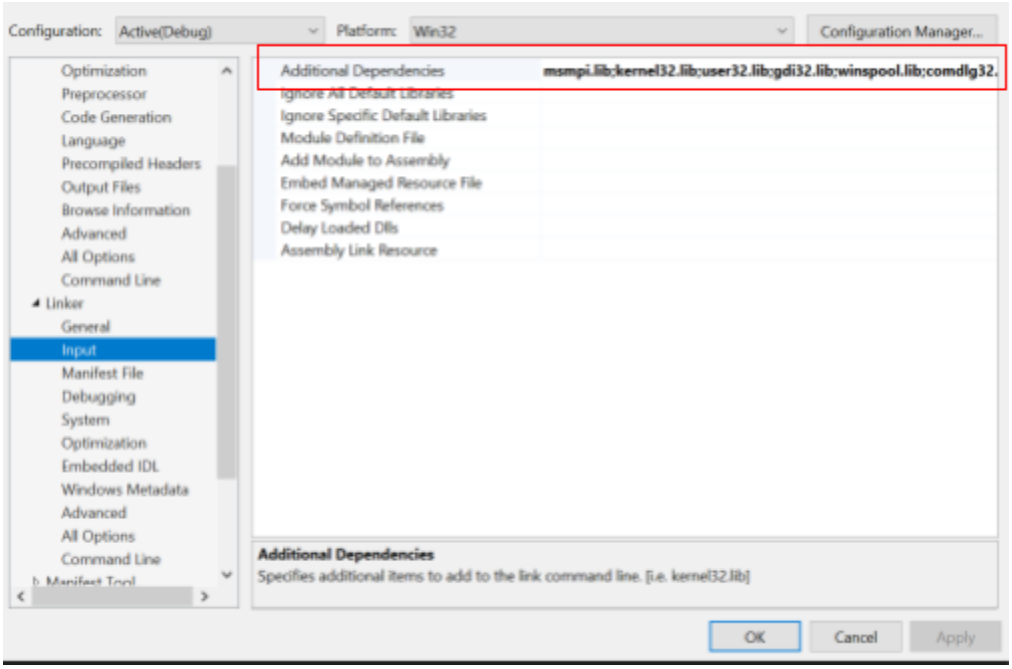
2. Configure project properties → C/C++ → Additional Include Directories (The path is where *mpi.h* is stored, normally is installed at "*C:\Program Files (x86)\Microsoft SDKs\MPI\Include"*)
   *** NOT "...\Include\x86…"**



3. Configure project properties Linker → Additional library Directories (The path is where msmpi.lib is stored, normally is installed at "*C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86*")



4. Configure project properties Linker → Input → Additional Dependencies, add in *msmpi.lib;*



5. Add a C++ file with code obtained from P6Q1. Then run the exe file in Powershell with the command below:

   ```
   mpiexec -n 4 filename.exe
   ```

   The command **-n <np>** specifies the number of processes to be used

   *Reference: **mpiexec command***

Example of output:

```
PS C:\Users\X> mpiexec -n 4 "D:\Dr Tew\BMCS3003\Project1\Debug\Project1.exe"
Hello World
Hello World
Hello World
Hello World
Hello World from process rank(number) 0 from 4
Hello World from process rank(number) 3 from 4
Hello World from process rank(number) 2 from 4
Hello World from process rank(number) 1 from 4
PS C:\Users\X>
```

Starting And Running MPI Processes

The **Single Program, Multiple Data (SPMD)** model is only a single source, compiled into a single executable, the parallel run comprises a number of independently started MPI processes.

The example above is designed to give you an intuition for this one-source-many-processes setup.

For more information about Microsoft MPI functionality, please refer to:

https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi

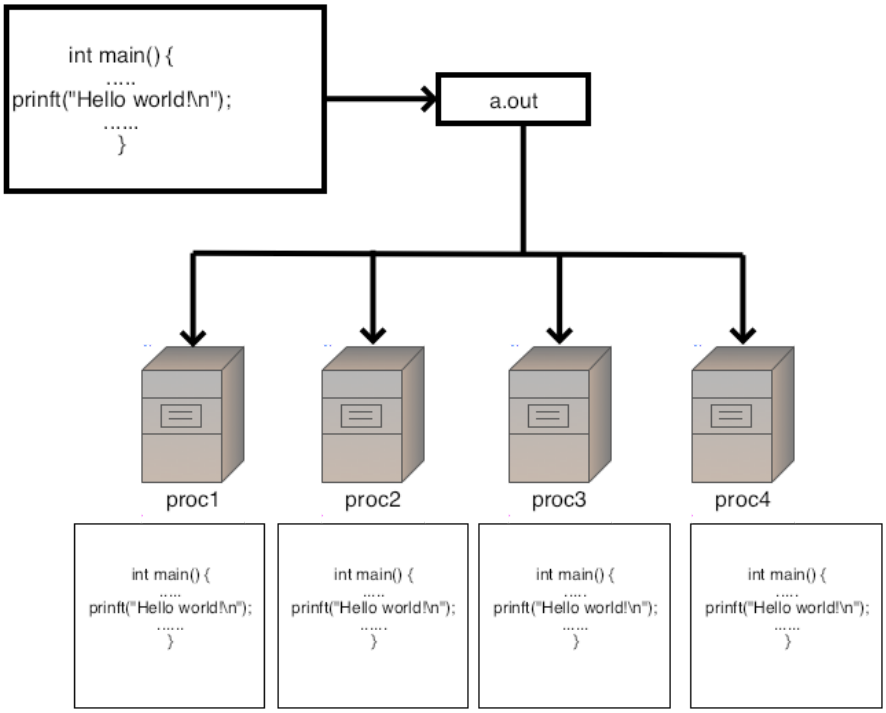


Figure: Running a hello world program in parallel

Question 1:

Now use the command **MPI_Get_processor_name** in between the init and finalize statement, and print out the name of the processor who runs your process. Confirm that you are able to run the program in parallel with the corresponding number of processes.

The name's character buffer needs to be allocated by you, it is not created by MPI, with size at least **MPI_MAX_PROCESSOR_NAME**.

Example of output:

```
Windows PowerShell
PS C:\Users\X> mpiexec -n 4 "D:\Dr Tew\BMCS3003\Project1\Debug\Project1.exe"
Hello World
Hello World
Hello World
Hello World
Hello World from process rank(number) 2 from 4 on DESKTOP-6IVO3I7
Hello World from process rank(number) 0 from 4 on DESKTOP-6IVO3I7
Hello World from process rank(number) 3 from 4 on DESKTOP-6IVO3I7
Hello World from process rank(number) 1 from 4 on DESKTOP-6IVO3I7
```

Process And Communicator Properties: Rank And Size

To distinguish between processes in a communicator, MPI provides two calls:

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
states the process ID in rank

MPI_Comm_size(MPI_COMM_WORLD, &size);
reports how many processes there are in the communication

If every process executes the MPI_Comm_size call, they all get the same result, namely the total number of processes in your run. On the other hand, if every process executes MPI_Comm_rank , they all get a different result, namely their own unique number, an integer in the range from zero to the number of processes minus 1. See Figure below:

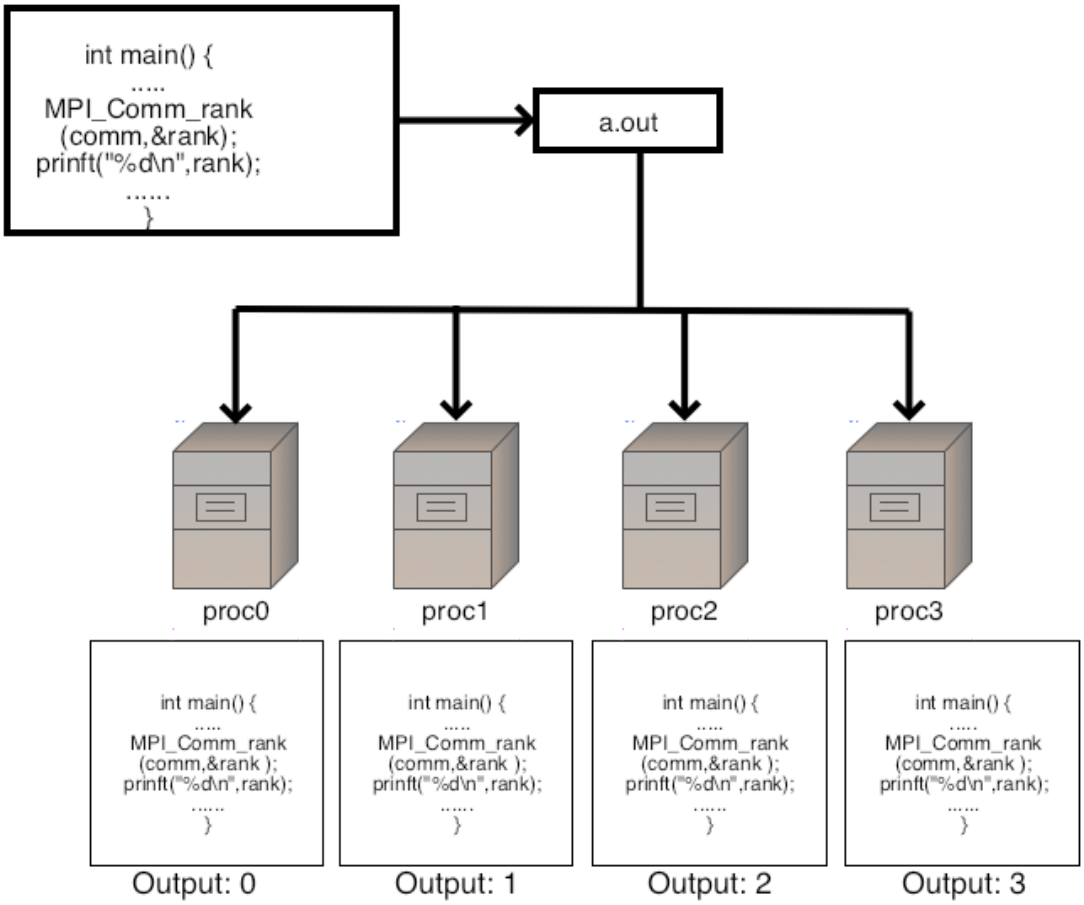


Figure: Parallel program that prints process rank

MPI_Reduce

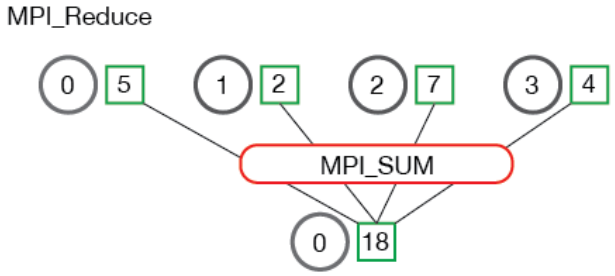
`MPI_Reduce` takes an array of input elements on each process and returns an array of output elements to the root process (similar to **`MPI_Gather`** in question 5). The output elements contain the reduced result. The prototype for `MPI_Reduce` looks like this:

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

The `send_data` parameter is an array of elements of type `datatype` that each process wants to reduce. The `recv_data` is only relevant to the process with a rank of `root`. The `recv_data` array contains the reduced result and has a size of `sizeof(datatype) * count`. The `op` parameter is the operation that you wish to apply to your data. MPI contains a set of common reduction operations that can be used. Although custom reduction operations can be defined, it is beyond the scope of this lesson. The reduction operations defined by MPI include:

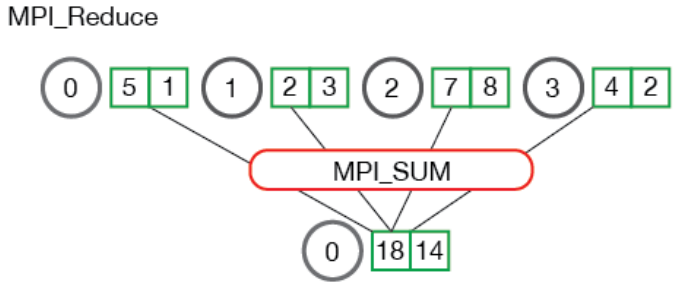
- `MPI_MAX` - Returns the maximum element.
- `MPI_MIN` - Returns the minimum element.
- `MPI_SUM` - Sums the elements.
- `MPI_PROD` - Multiplies all elements.
- `MPI_LAND` - Performs a logical *and* across the elements.
- `MPI_LOR` - Performs a logical *or* across the elements.
- `MPI_BAND` - Performs a bitwise *and* across the bits of the elements.
- `MPI_BOR` - Performs a bitwise *or* across the bits of the elements.
- `MPI_MAXLOC` - Returns the maximum value and the rank of the process that owns it.
- `MPI_MINLOC` - Returns the minimum value and the rank of the process that owns it.

Below is an illustration of the communication pattern of `MPI_Reduce`.



In the above, each process contains one integer. `MPI_Reduce` is called with a root process of 0 and using `MPI_SUM` as the reduction operation. The four numbers are summed to the result and stored on the root process.

It is also useful to see what happens when processes contain multiple elements. The illustration below shows reduction of multiple numbers per process.



The processes from the above illustration each have two elements. The resulting summation happens on a per-element basis. In other words, instead of summing all of the elements from all the arrays into one element, the i^{th} element from each array is summed into the i^{th} element in the result array of process 0.

Question 2:

$f(x)=4/(1+x^2)$, so PI is the integral of f(x) from 0 to 1. Then PI can be easily calculated by using the trapezoidal rule. Obtain the code P6Q2. Besides using MPI_Comm_rank and MPI_Comm_size above, initiate and terminate MPI execution environments as well.
Also, synchronize the parallel processes by adding the code below at the appropriate segment.

To synchronize all element, use
```
MPI_Barrier(MPI_COMM_WORLD);
```

Reference: MPI_Barrier

To run the MPI operation (i.e., MPI_SUM) use

```
MPI_Reduce(&result, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,
 MPI_COMM_WORLD);
```

```
np= 4;    Time=0.014481s;    PI=3.141593
```

Try on different number of processes (e.g., `mpiexec -n 16 filename.exe` **)**

Question 3:

A common need is for one process to get data from the user, either by reading from the terminal or command line arguments, and then to distribute this information to all other processors [*MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD*)].

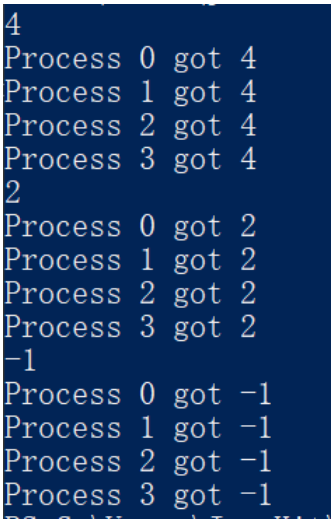
Write a program that reads an integer value from the terminal and distributes the value to all of the MPI processes. Each process should print out its rank and the value it received. Values should be read until a negative integer is given as input.

You may want to use these MPI routines in your solution:
MPI_Init, MPI_Comm_rank, MPI_Bcast, MPI_Finalize

Obtain the code P6Q3. You may also make use of fflush(stdout) to display the output after every input.

*** fflush(steam) definition :**
If the given *stream* was open for writing (or if it was open for updating and the last i/o operation was an output operation) any unwritten data in its output buffer is written to the file.

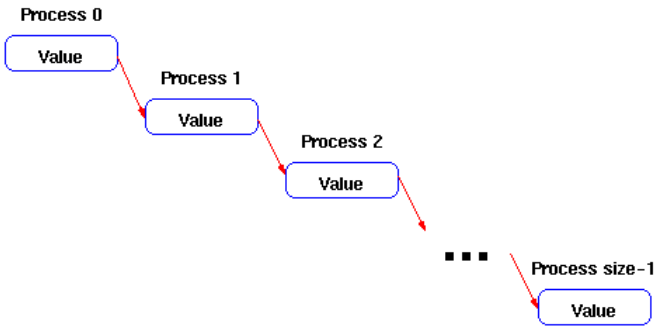
Output:



Try on different number of processes (e.g., `mpiexec -n 16 filename.exe` **)** and observe the sequence of each processing rank.

Question 4:

Write a program that takes data from process zero and sends it to all of the other processes by sending it in a sequence. That is, process *i* should receive the data and send it to process *i*+1, until the last process is reached.



Assume that the data consists of a single integer. Process zero reads the data from the user.

You may want to use these MPI routines in your solution:

i.MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);

ii.MPI_Recv(&value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);

Obtain the code P6Q4.

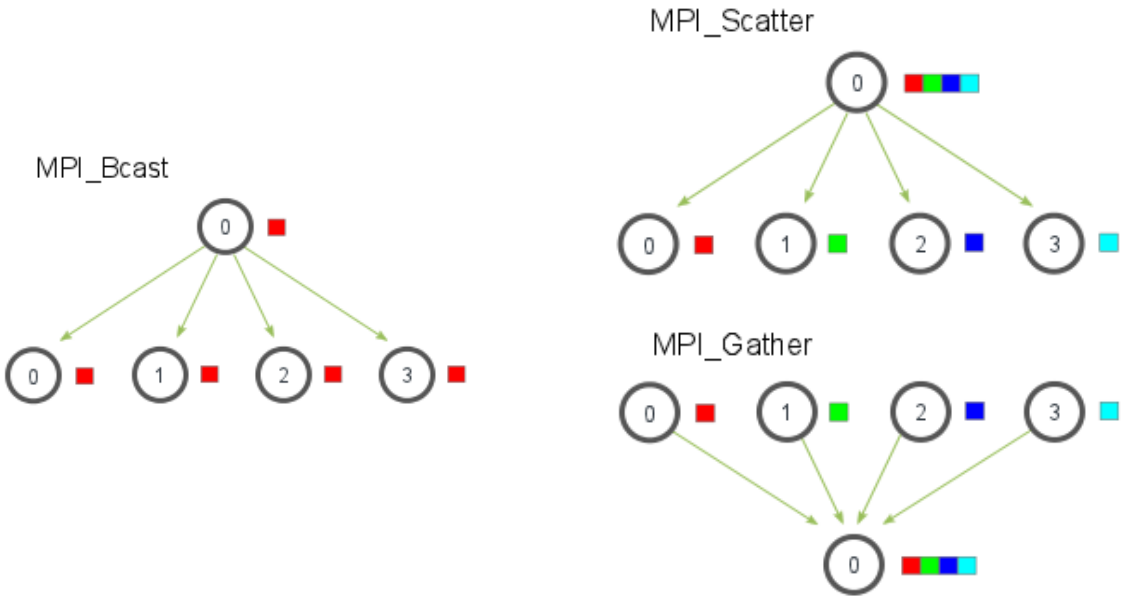
Sample output:

```
PS C:\Users\X> mpiexec -n 4 "D:\Dr Tew
\BMCS3003\Project1\Debug\Project1.exe"

88
Process 0 got 88
Process 1 got 88
Process 2 got 88
Process 3 got 88
-88
Process 0 got -88
Process 1 got -88
Process 2 got -88
Process 3 got -88
PS C:\Users\X>
```

Try on different number of processes (e.g., `mpiexec -n 16 filename.exe` **)** and observe the sequence of each processing rank. Why is the sequence in order ?

More info:
Graphical representation for MPI_Gather, MPI_Bcast and MPI_Scatter functions.



Question 5:

Computing average numbers with MPI_Scatter and MPI_Gather.

The program takes the following steps:

1. Generate a random array of numbers on the root process (process 0).
2. Scatter the numbers to all processes, giving each process an equal amount of numbers.
3. Each process computes the average of their subset of the numbers.
4. Gather all averages to the root process. The root process then computes the average of these numbers to get the final average.

You may want to use these MPI routines in your solution:

i. *MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums, num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);*
ii. *MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);*

Obtain the code P6Q5.

```
PS C:\Users\X> mpiexec -n 4 "D:\Dr Tew\BMCS3003\Project1\Deb
ug\Project1.exe" 5
Avg of all elements is 0.504950
Avg computed across original data is 0.504950
```