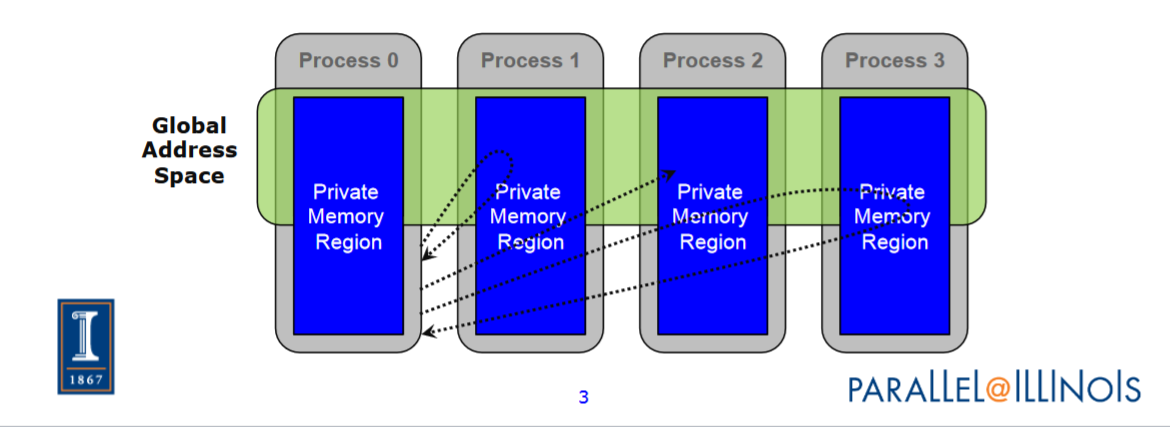
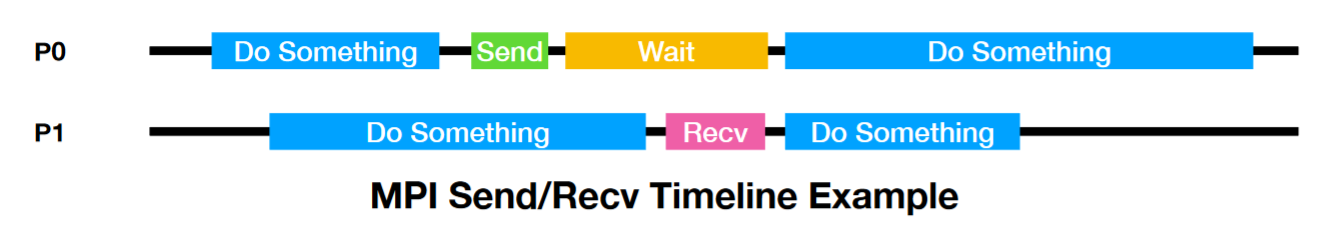
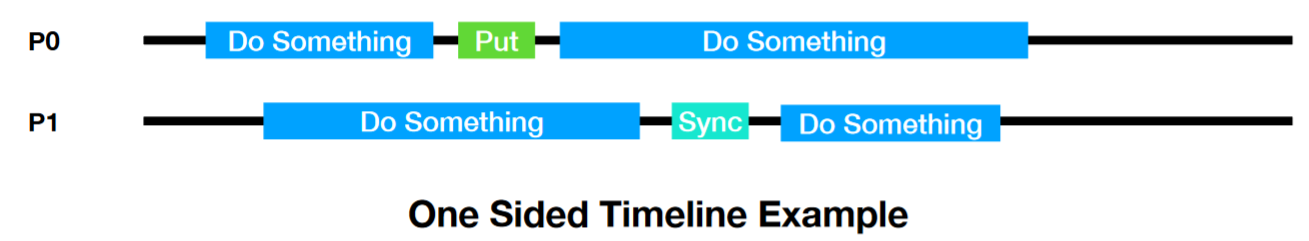
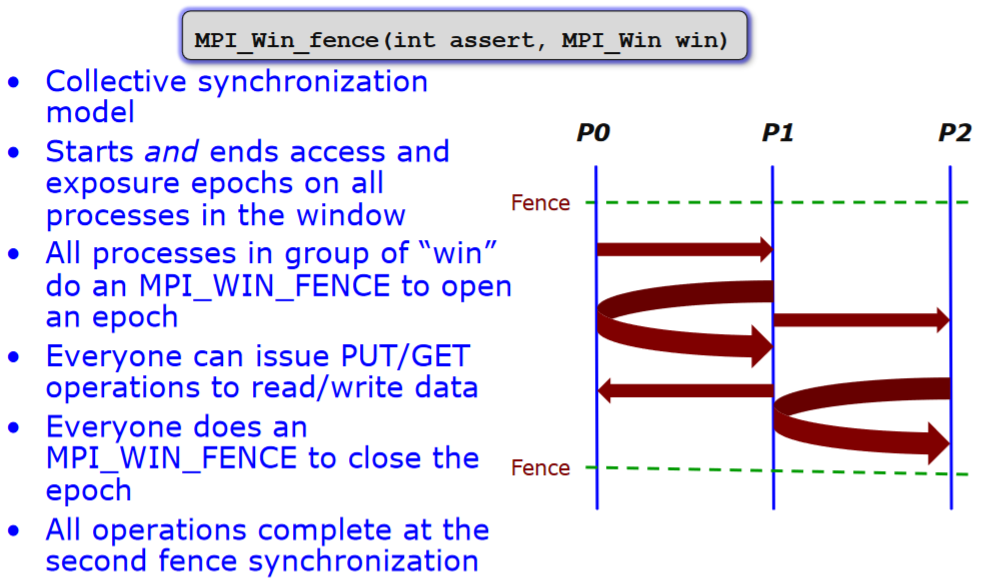
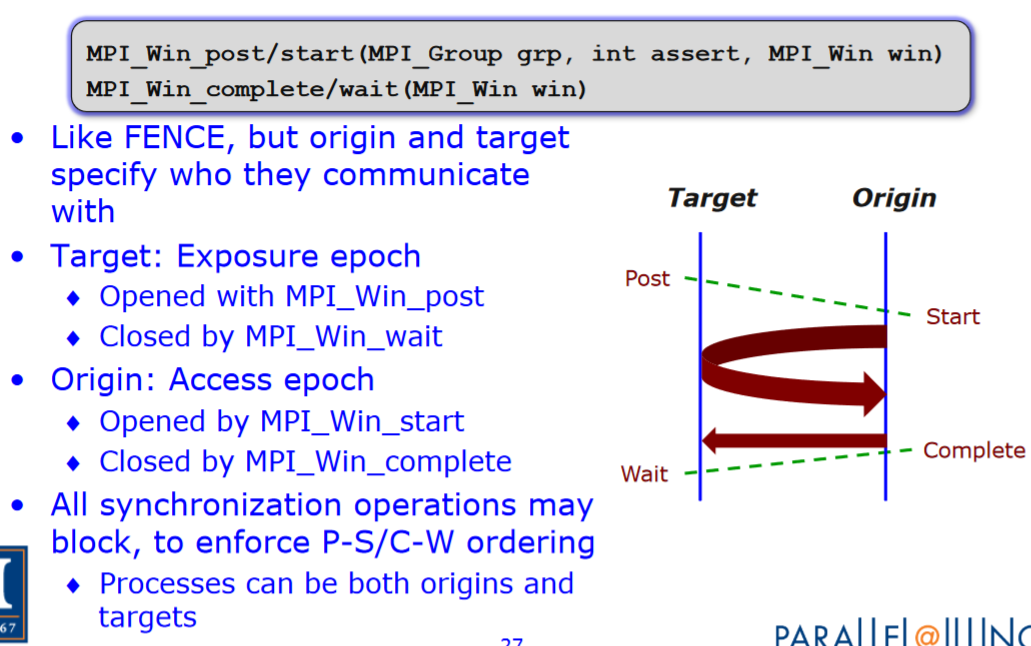
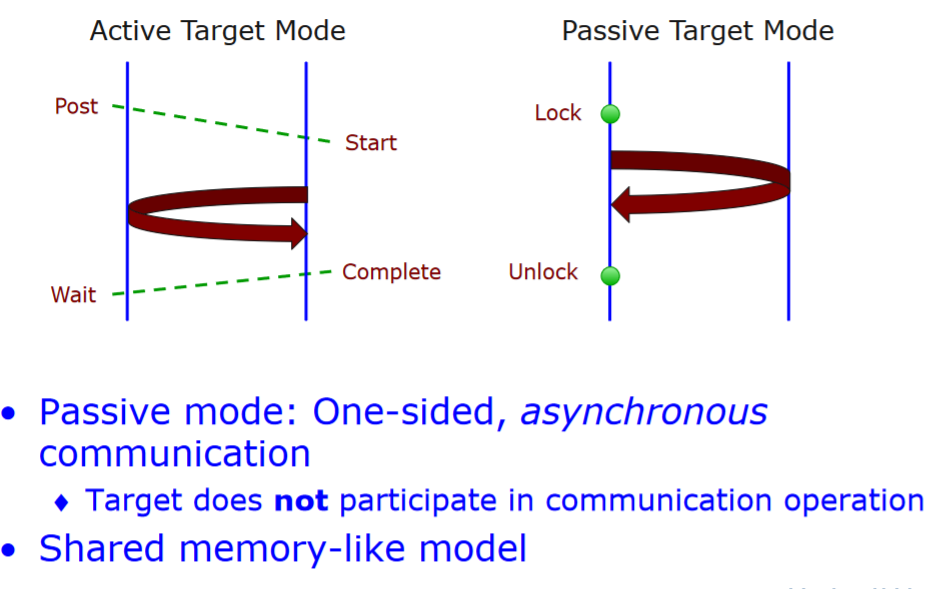
*Overview:*

One-sided communication tutorial using a matrix-matrix multiplication example.

*One-sided communication:*

* In standard communication, the process with the data sends it to the destination process, and the receiving process posts a receive
* One-sided communication decouples data movement with process synchronization
  + Each process exposes part of its memory to other processes
  + Other processes can read from or write to this exposed section of memory
* Compared to two-sided send and receive: 



* Can do multiple data transfers and then a single instance of synchronization
  + Example : Irregular communication where pattern is not known beforehand but data locations are known (i.e. SpMV) can just get data instead of telling sending process that I need it
* Additional speedup on shared memory systems (hardware support for remote memory accesses)
* RMA stands for remote accessible memory
* Data allocated by a process is, by default, only accessible by this process
* Once memory is allocated, user can declare memory as remotely accessible
  + MPI window: remotely accessible memory
  + MPI\_Win\_create : collective operation that creates window object (specifies which memory on each process is remotely accessible)
  + MPI\_Win\_free : deallocates window object
* After creating MPI window:
  + MPI\_Put moves data from local memory to a remote memory
    - Move data from origin, to target
  + MPI\_Get retrieves data from a remote memory to local memory
    - Move data to origin, from target
  + MPI\_Accumulate updates remote memory using local values
    - Element-wise atomic update operation, similar to a put
* All data movement operations are non-blocking
* We need synchronization on window object to tell when operation is complete
* Three MPI modes of synchronization
  + Fence
  + Post-start-complete-wait
  + Lock/Unlock

*Paper Sections*

1. Introduction
2. Explain One-Sided (Methods)
3. Results
4. Discussion and Conclusion

Here are the steps to implementing dense matrix-matrix multiplication with one-sided communication in MPI:

1. Initialize the MPI environment by calling the MPI\_Init function. This function initializes the MPI library and must be called by all processes before any other MPI functions are used.
2. Create a communicator that includes the processes that will participate in the matrix-matrix multiplication. This can be done using the MPI\_Comm\_dup function, which creates a new communicator from an existing one, or the MPI\_Comm\_split function, which creates a new communicator from a group of processes.
3. Divide the matrices among the processes by assigning each process a portion of the rows and columns of the matrices. This can be done by calculating the number of rows and columns each process should handle based on the number of processes and the dimensions of the matrices.
4. Allocate memory for the local sub-matrices of the input matrices and the output matrix using the MPI\_Alloc\_mem function. This function allocates memory in the shared address space of the processes, which allows the processes to access the matrices.
5. Initialize the local sub-matrices of the input matrices with the appropriate values using standard C/C++ techniques, such as using a loop to copy the data from the input arrays into the local sub-matrices.
6. Create a window object that represents the global output matrix using the MPI\_Win\_create function. This function creates a window object that can be used by the processes to access the global output matrix.
7. In each process, perform the local matrix-matrix multiplication by multiplying the local sub-matrices of the input matrices and storing the result in the local sub-matrix of the output matrix.
8. In each process, put the local sub-matrix of the output matrix into the global output matrix using the MPI\_Put function. This function allows the processes to put their local results into the global output matrix without the need for explicit communication or synchronization.
9. Once all processes have completed their local matrix-matrix multiplication and put their results into the global output matrix, the processes can wait for all operations to complete using the MPI\_Win\_fence function. This function ensures that all operations on the window object have completed before the processes proceed.
10. After the matrix-matrix multiplication is complete, the processes should free the allocated memory for the local sub-matrices and the window object using the MPI\_Free\_mem and MPI\_Win\_free functions, respectively.
11. Finally, the MPI environment should be finalized by calling the MPI\_Finalize function. This function cleans up the MPI library and must be called by all processes after they are finished using MPI.

One-sided communication in MPI, also known as asymmetric communication, refers to the ability of a process in a parallel program to access the memory of another process without the need for explicit coordination or synchronization. This type of communication can be useful for improving the performance of parallel programs by allowing processes to access data directly from each other's memory, rather than having to send messages back and forth. However, it also introduces potential challenges and limitations, such as the need for careful management of memory access to avoid conflicts and the inability to support certain types of communication patterns. In this paper, we will discuss the concept of one-sided communication in MPI and its potential advantages and disadvantages.

It is not clear how one-sided communication would be specifically useful for matrix-matrix multiplication in MPI.In general, matrix-matrix multiplication can be performed using a variety of communication patterns, including one-sided communication, depending on the specific implementation and the requirements of the application. It is ultimately up to the person implementing the parallel matrix-matrix multiplication algorithm to decide which communication pattern to use.

In the context of MPI (Message Passing Interface), "post-start-complete-wait" (PSCW) refers to a pattern of communication in which one process sends a message to another process, and then waits for a response before continuing. This is an example of one-sided communication, because only one of the processes is actively involved in sending and receiving messages.

In the PSCW pattern, the sending process first "posts" a message, which means that it prepares the message and indicates that it is ready to be sent. The receiving process then "starts" the communication by accepting the message and starting to receive it. Once the message has been completely received, the receiving process sends a confirmation ("complete") back to the sending process. Finally, the sending process "waits" for this confirmation before continuing with its work.

This pattern is useful because it allows for efficient communication between processes, while also ensuring that the sending process does not continue until it has received a response from the receiving process. It is typically used in situations where the sending process needs to know that the receiving process has successfully received the message before continuing.

In the context of MPI (Message Passing Interface), "fence synchronization" is a technique used to coordinate one-sided communication between processes. It is a type of synchronization that ensures that all processes have completed certain operations before any of them can continue.

In the fence synchronization pattern, a "fence" is inserted into the communication flow between the processes. This fence acts as a barrier that prevents any processes from continuing until all of the processes have reached the fence and signaled that they are ready to continue. This ensures that all processes are synchronized and that no process can continue until all of the processes have completed their operations.

Fence synchronization is useful in situations where multiple processes need to communicate with each other, but it is important that they do so in a coordinated and synchronized manner. By using the fence synchronization pattern, processes can ensure that they are all working together and that no process can continue until all of the processes are ready. This can help to prevent errors and improve the overall efficiency of the communication between processes.

In the context of MPI (Message Passing Interface), "lock/unlock synchronization" is a technique used to coordinate one-sided communication between processes. It is a type of synchronization that ensures that only one process can access a shared resource at a time.

In the lock/unlock synchronization pattern, each process must first "lock" the shared resource before it can access it. This prevents any other process from accessing the resource at the same time. Once the process has finished using the resource, it must then "unlock" it, which allows other processes to access the resource.

This pattern is useful in situations where multiple processes need to communicate with each other, but it is important that they do not interfere with each other's access to shared resources. By using the lock/unlock synchronization pattern, processes can ensure that only one process can access the shared resource at a time, which can help to prevent errors and improve the overall efficiency of the communication between processes.

*References:*

* Gropp, W., University of Illinois Urbana-Champaign, Lecture 34: One-sided Communication in MPI, 2016
  + [**https://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture34.pdf**](https://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture34.pdf)
* Ghosh, S. , Hammond J., Pena A., One-Sided Interface for Matrix Operations using MPI-3 RMA: A Case Study with Elemental. Washington State University School of Electrical Engineering and Computer Science, 2016.
  + [**https://eecs.wsu.edu/~assefaw/publications/icpp2016-elemental.pdf**](https://eecs.wsu.edu/~assefaw/publications/icpp2016-elemental.pdf)
* Gropp, W., Thakur, R. (2005). An Evaluation of Implementation Options for MPI One-Sided Communication. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2005. Lecture Notes in Computer Science, vol 3666. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11557265\_53
  + [**https://link.springer.com/chapter/10.1007/11557265\_53**](https://link.springer.com/chapter/10.1007/11557265_53)
* [**https://arxiv.org/pdf/1705.10218.pdf**](https://arxiv.org/pdf/1705.10218.pdf)
* Thakur, R., Gropp, W.D., Toonen, B. (2004). Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2004. Lecture Notes in Computer Science, vol 3241. Springer, Berlin, Heidelberg. <https://doi.org/10.1007/978-3-540-30218-6_15>
  + [**https://wgropp.cs.illinois.edu/bib/papers/pdata/2004/mpich2-rmasync-eurompi04.pdf**](https://wgropp.cs.illinois.edu/bib/papers/pdata/2004/mpich2-rmasync-eurompi04.pdf)
* Ziheng Wang, Heng Chen, Xiaoshe Dong, Weilin Cai, and Xingjun Zhang. 2022. LogSC: Model-based one-sided communication performance estimation. Future Gener. Comput. Syst. 132, C (Jul 2022), 25–39. <https://doi.org/10.1016/j.future.2022.02.004>
  + [**https://www.sciencedirect.com/science/article/abs/pii/S0167739X22000486**](https://www.sciencedirect.com/science/article/abs/pii/S0167739X22000486)
* [**https://dl.acm.org/doi/10.1145/3093172.3093228**](https://dl.acm.org/doi/10.1145/3093172.3093228)
* Karlbom, David. “A Performance Evaluation of MPI Shared Memory Programming.” 2016.
  + [**https://www.diva-portal.org/smash/get/diva2:938293/FULLTEXT01.pdf**](https://www.diva-portal.org/smash/get/diva2:938293/FULLTEXT01.pdf)
* [**https://docs.oracle.com/cd/E19061-01/hpc.cluster6/819-4134-10/1-sided.html**](https://docs.oracle.com/cd/E19061-01/hpc.cluster6/819-4134-10/1-sided.html)