Interprocess Communication in Distributed Wireless Sensor Network

Luhan Cheng
Faculty of Information Technology
Monash University
lche0021@student.monash.edu

***Abstract*—Wireless Sensor Network (WSN) has been deployed in a wide range of mission critical task from bushfire detection to water quality monitoring. Such system can be abstracted by representing each sensor with one process in computer system. In this report, I conducted an experiment which utilize MPI to simulate the communication pattern in a 4 by 5 grid network whereas all communications in network are encrypted. Threads level parallelization is applied to cryptographic operations with OpenMP. The experiment shows that (1) minimized communication in WSN can be achieved by MPI (2) the speedup of encryption/decryption algorithm may not be observed by simply adding more shared-memory threads.**

Keywords-component; Inter-process Communication, shared memory parallel, Message Passing Interface, Wireless Sensor Network, OpenMP

I. INTRODUCTION

Inter-process Communication (IPC) defines a set of mechanism that support data sharing and communication among processes[1]. It is commonly used in wireless sensor network (WSN) as communication management method. There are various methods have been developed to satisfy the requirements for different applications. For example, TCP/UDP protocol is developed to facilitate web browsing service, message passing serves concurrency model and shared memory scheme is built in all POSIX systems.

This report aims to simulate the communication in a wireless sensor network to discover most efficient communication pattern. In addition to the simulation in distributed environment, shared memory parallelization is to be discussed to explore the potentiality of accelerating cryptographic operation. The potential of speedup is based on the hypothesis that OpenMP will improve the performance of both encryption and decryption.

Target network is assumed to be a 2-dimensional cartesian grid, where each coordinate represents a sensor (process). An extra process is introduced to simulate base station in the network. The simulation consists of multiple iterations, and for each iteration each sensor sends an encrypted random number to its neighbor. An encrypted event is to be reported to base station if any sensor receives at least 3 identical numbers from its neighbors.

There are three objectives being identified in this scenario. (1) Minimize the message passing between sensor and base station. (2) Minimize the communication among sensors. (3) Speedup the encryption of message with thread level parallelization.

In term of event number being generated. Random seed on each process is computed as the addition of current timestamp and rank number. This setup ensures different random seeds on different processes. Due to limited amount of resources, high probability of event is necessary to generate meaningful outcomes of the simulation. It is achieved by taking first n bits of random number, which means the random number has 2^n possible outcomes. The probability of one particular event is uniformly distributed across all events.

II. DESIGN SCHEME FOR IPC

A. Justify Chosen IPC

Message Passing Interface (MPI) is a library specification for message-passing in distributed system [2], which is one of the most popular message passing standard in industry[3]. MPI provides rich features of both point-to-point and collective communication. Compare to other standards, MPI provides several advantages. (1) MPI offers more portable libraries compare to older message passing standards (e.g. parallel virtual machine) [4]. (2) MPI is capable of delivering high performance on HPC system, and it is optimized on the hardware [4][5]. The implementation of MPI standard is vendor specific. One of the most popular open source implementations is OpenMPI [6]. Therefore OpenMPI is applied in experiment to simulate sensor-sensor and sensor-base communication.

OpenMP stands for Open Multi-Processing. It is an application programming interface that support shared memory parallelization. Its behaviors are defined by a set compiler directives and runtime environment variables [7]. OpenMP follows fork-join model whereas at the start of program, there is only one master thread. A set of slave threads can be dynamically forked, and workload is distributed across slave processes. In the experiment, OpenMP is applied in order to speedup encryption and decryption.

B. Overall Topology

The network is designed as following. 20 Sensors is repartitioned into 4 by 5 cartesian grid, which forms the communicator local to sensors whereas the sensor-base communications will go through global communicator. The event detection criteria specify that an event is to be reported to base station if any sensor has at least 3 of its neighbors generate the same random number. Such criteria indicate strong local connectivity as for each sensor, it need to exchange a message from its neighbors. For the grid with size

(|X|, |Y|), the number of message passing is $(|X| * (|Y| - 1) + |Y| * (|X| - 1)) * 2$ for each iteration.

MPI provides build-in cartesian constructor in any dimension. The communicator among sensors is constructed from excluding the rank of base station from global communicator.

C. local message exchange

In order to construct a single message, paddings need to be appended to random number. The messages are to be exchanged by each pair of neighbors. Nearest neighbor communication can be observed in this case. This type of communication pattern has been discussed in multiple literatures. In particular, Torsten and Jesper suggested that the scheduled sparse neighbor all-to-all can outperform naïve neighbor all-to-all operation by replace non-bloking message passing with bidirectional blocking MPI_Sendrecv[8]. It has been shown that 10% speedup can be observed by applying scheduled all-to-all operation[8]. This optimization is achieved by reducing communication contention on cartesian mesh grid and utilize bidirectional communication link [8]. The application of such operation is limited to case where global knowledge of problem is known, otherwise deadlock could be generated. In the case of WSN, scheduled all-to-all operation is applied in the local message exchange as line 13 in figure 1.

D. global event report

After local message passing stage, each node will need to iterate over all possible event values to determine if an event is triggered. If event is activated, the node will report the event to base station by blocking send in global communicator as line 20 in figure 1. Other event related information is also reported along with event number, such as iteration number, timestamp and aggregated encryption/decryption time. When simulation completes, each sensor will send a summary (line 22 figure 1), which includes its basic information, to base station. This message also signals the completion of simulation.

At base station side, at the start of simulation, assume there are N-1 sensors. Base station spawn $2N$ MPI_Irecv simultaneously along with a receiver array filled with corresponding request. The first half of the array is used for buffering event message, the second half of the array is used for receiving completion signal. The position in request array corresponds to the rank of base station is filled with null handler. During the simulation, for event received, the message is decrypted and stored in memory, and the same MPI_Irecv will be respawned as line 32 figure 1. If any completion signal is received, then both events receiver and completion signal receiver will be set to null. Those two types of message are distinguished by using different tags.

Local message exchange and global event report together forms an iteration. A single simulation consists of multiple iterations. All event reports are stored in local memory of base station until being written to storage at the end of simulation. This implementation could potential cause high memory consumption on base station in large scale simulation, but it also comes with the advantage of more available analysis on event reports. The pseudo-code for algorithm is presented in figure 1.

III. ENCRYPTION AND DECRYPTION

Advanced Encryption Standard (AES) is one of the most widely adopted symmetric encryption algorithms. It offers great security with little computational resource required [9]. In order to parallelize both encryption and decryption operation, counter mode AES is selected as encryption/decryption algorithm. Note that the most computational costly operation in counter mode is the cipher block initialization instead of the XOR operation between plaintext and cipher block [9]. Therefore, the speedup of parallelization may not be observed when message length is insufficient.

In realistic, the keys need to be distributed at initialization stage for security. IV need to be randomly generated in order to preserve the confidentially of message. In the experiment, a few assumptions are made for simplicity, I assume that all the parties involved in communication share the same 16 bytes key and the same 8 bytes initialization vector (IV, or nonce). The implementation of algorithm is taken from WjCryptLib [10]. Its counter mode implementation offers build-in OpenMP for parallelizing operations over cipher blocks. A sample message before and after encryption is presented in figure 2.

IV. RESEULT

The experiment is conducted on high performance cluster MonARCH[11]. The simulation consists of 1000 iterations with 10 milliseconds interval between each iteration. Each message carries 1-byte random number at the start with tailing zeros. 2 CPUs are allocated to each task. The simulations consist of 4 runs with 1,2,4,8 maximum OpenMP threads. The configuration is written to logfile as shown in figure 3. A sample summary of events activation is shown in figure 4. First few events records are shown in figure 5. The experiment is profiled with MPIP[12], which stand for lightweight, scalable MPI profiler. The results are shown in figure 9. Figure 11 shows that the event number generated over period of time is uniformly distributed.

Figure 7 shows MPI time for each rank, we can conclude that good load balance is achieve with presented algorithm.

For each event, there is only one message being passed from reference node to base station. The total amount of message is event number plus number of nodes as shown in figure 9. This communication pattern demonstrated near optimized message passing in WSN.

However, increasing the amount of OpenMP threads does not result in performance improvement. Instead, it leads to severe performance decay as shown in figure 10. There may be several causes for the issue. The problem is mostly likely caused by the limited resources being allocated (2 CPUs per task) to the submission. In addition, the processors use shared resources with other jobs on the cluster, which may indicate interference between simulation and other tasks running on the same node.

Figure 8 presented the time spent on each MPI function. We can conclude that 81.76% (sum of two Sendrecv) aggregated time is spent on sensor-sensor communication while 16.77% (Waitany) aggregated time are spent on sensor-sensor communication. Except message passing, MPI_Cart_Create and MPI_Cart_shift cost more than any other MPI function. This is because both operations require communications involves all processes.

V. Conclusion

This report presented that IPC method, especially MPI, can provide optimized communication scheme for wireless sensor network. But shared memory parallelization does not demonstrate performance improvement in this experiment. The experiment provides some future guidance towards designing more optimized WSN. There are a few possible works left for future. The algorithm can potentially be extended to adapt multi-dimension grid of sensor network. More secure encryption algorithm may be applied. Additionally, we could develop more flexible event detection criteria or adding the feature that regularly save batch of events to storage for less memory overhead on base station. Overall, the experiment shows great potentiality of applying parallelization on wireless distributed network.

[1] mcleanbyron, "Interprocess Communications - Windows applications." [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications. [Accessed: 13-Oct-2019].

[2] "Message Passing Interface." [Online]. Available: https://www.mcs.anl.gov/research/projects/mpi/. [Accessed: 15-Oct-2019].

[3] S. Sur, M. J. Koop, and D. K. Panda, "High-performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-depth Performance Analysis," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2006.

[4] W. Gropp and E. Lusk, "Why are PVM and MPI so different?," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1997, pp. 1–10.

[5] "Message Passing Interface," *Wikipedia*. 08-Oct-2019.

[6] "Open MPI: Open Source High Performance Computing." [Online]. Available: https://www.open-mpi.org/. [Accessed: 15-Oct-2019].

[7] tim.lewis, "Specifications," *OpenMP*. .

[8] T. Hoefler and J. L. Traff, "Sparse Collective Operations for MPI," p. 8.

[9] "Wayback Machine," 06-Mar-2016. [Online]. Available: https://web.archive.org/web/20160306104007/http://research.microsoft.com/en-us/projects/cryptanalysis/aesbc.pdf. [Accessed: 17-Oct-2019].

[10] WaterJuice, *WaterJuice/WjCryptLib*. 2019.

[11] "Welcome to the MonARCH documentation! — MonARCH Documentation documentation." [Online]. Available: https://docs.monarch.erc.monash.edu.au/. [Accessed: 19-Oct-2019].

[12] "mpiP: Lightweight, Scalable MPI Profiling." [Online]. Available: https://software.llnl.gov/mpiP/. [Accessed: 19-Oct-2019].

```
Algorithm 1: SIMULATE WSN
Input: x_size, y_size, n_iter, base, randbit
Output:
if rank ≠ base then
    local_group ← MPI_Group_Excl (MPI_COMM_GROUP, base)
    local_comm ← MPI_Cart_Create (local_group, x_size, y_size)
    nneighbor ← number of neighbors
    upperbound ← 2^(randbit−1)
    event_report_tag ← 0
    completion_tag ← 1
    message ← add_padding (rand num)
    for k in n_iter do
        for all dimensions d;all neighbor n do
            source, destination ←
              MPI_Cart_Shift (local_comm, d, rank)
            encrypt (message)
            exchange messages using
              MPI_Sendrecv (message, neighbor_results [n]) with both
              source and destination
        for i in nneighbor do
            decrypt (result_neighbor [n])
            event_counter [neighbor_results [i]] + +
        for i in event_counter do
            if i ≥ 3 then
                encrypt (event i )
                MPI_Send (event i, event_report_tag) to base
                reinitalize event_counter
    MPI_Send (completion_tag, base)
else flag ← size − 1
initialize request_array with length of 2 × size
for all ranks r do
    MPI_Irecv (r, request_array [r])
    MPI_Irecv (r, request_array [r + size])
while flag do
    MPI_Wait_Any (request_array, recv_from)
    if received completion signal then
        flag ← flag − 1
    else decrypt (received event)
    save the events
    respwan same MPI_Irecv

generate logfile and save to disk;
return 0
```

Figure 1. WSN simulation algorithm


```
message on rank 1 before encryption:
5000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
message on rank 1 after encryption:
ce3f2032f698cb327c926a176a8a79d046939bf16fbd3ae2891a3192a892c225fb6663f6471844fb23de99b9e3edd8d3c77c03d821261f434fef57a8a8571a269a7f262996c7e5a4834dacdd39ba2f
c0f1ef8a64d12be58d0746aa4abbd092aede253ab89dd4c40236e158f0a1a9e72cfa65a786192179d73c7f71330d0c9e2c694a87bab1ad99b5f5339ceab087f57679fab9a59e9074ad83d90b5621fd
9d94ff525638d4b13793b3d656515c9c6a2cec71e59e1225928f5040d02e87d854e7655b02940961df911c6ac91e4e11f6e79444c9bc1447b157732d6aea52e16f0693c9650df6de758b2805abcb1e
0d37385a1dfd5d54ae6862aff34af9a318c82bcd665dd11add857d4c37fbe68b273e78981d2124785aa2f0a420b265b844baa5187a77eb5dcac168d6d54e63dfdf5e40e1971027d892d1f3408de688
4277975cad46dd3090ff8a21a684d771c2a847366cb89535a637cc1f16f9480f28664cf1b1c907c348a311b161e1430ed101edb8df43e0dd2a3a71918d7b0920af2d894cb160bd7386b9c0266256fe
1e7d653b3608ca99297a680d5a5f43c5e41a39448bd5b26dc0cb5bb6f45bdd63cc1ab65d4281e94994c856813c4fa8458f762e060c2f9da11e39b95bf36e6d25c3c602aadb1f7ceade98954df571c2
bf2cb611517d9ae414f07f85a9758e34c943651e28924aa2684a119336e541d86c60d7412b7ac6d81be9e0cd72c828f0ccb42d22e03276a40572faf90ea95dc6dd6293b4be579419c11afa7f68c4c0
cfb19d4b9039ef54ef11814cc392b5d862b4a245756dda8628ecb395542541fb1876bacce4cb5eebd77f4f9b185928e26ae834616368c1f346b793a15a8f78fd9da51443f962c34ee1184354619a28
7fe1ba8af74e5279658fe256fca7dbb8866d1392b89e6819080be5f88debcd0a911629c2a14d6d91715bfab333a8baa35cdfc0b871be69f688235255f9a1fb9af68b86fe775a9bfd5aa1d6d5098471
6723721369a5f3192ed9b93582b2d45b8975291e11bf17221a16a31a88471f11a1cccc2b92a9a7597e3e89c6ddbd1d3192c99eb2f49ca78ff1d58366b14c51b724d32e11b93f627c1b19123a21ddc2
b9c34a51a13479d13835acf71454331b01f8c4f1b9354ac399a1caea08a735a504e16f9511b71224632aeb233b6f704c99a5448ccb46c27880d76f27bb64a5ab76f75142f910eb30244e4161c2994f
f84068eb14e5b1c63fba17281f131e423593bbff3a5713699a174994d9ddd5f786c88a8a755b61c0cd8da5988660782d864789e9d8162277e2abbe97a780fd7132b4f1a07c7b96e379fd734640a122
2082732692532aa6162cef2bb13d7c7bf0b79d9b544539718f5797c9bf1593dc20ee23f779ecedcd335d74b4
```

Figure 2. Sample message before and after encryption

```
Event detection in a fully distributed wireless sensor network - WSN

network configuration overview:
Simulation will run 2000 iterations with 10 milliseconds time interval between each pair of consecutive iteration
Network have 4 nodes in X dimension and 5 nodes in Y dimension
The size of random number is 3 bits, which indicate event number is bound by range [0, 8)
process and topology summary:
    rank 1 in MPI_COMM_WORLD has local rank 0 in local communicatior;
        Coordinate is (0, 0)
        Maximum threads: 1
    rank 2 in MPI_COMM_WORLD has local rank 1 in local communicatior;
        Coordinate is (0, 1)
        Maximum threads: 1
    rank 3 in MPI_COMM_WORLD has local rank 2 in local communicatior;
        Coordinate is (0, 2)
        Maximum threads: 1
    rank 4 in MPI_COMM_WORLD has local rank 3 in local communicatior;
        Coordinate is (0, 3)
        Maximum threads: 1
    rank 5 in MPI_COMM_WORLD has local rank 4 in local communicatior;
        Coordinate is (0, 4)
        Maximum threads: 1
    rank 6 in MPI_COMM_WORLD has local rank 5 in local communicatior;
        Coordinate is (1, 0)
        Maximum threads: 1
    rank 7 in MPI_COMM_WORLD has local rank 6 in local communicatior;
        Coordinate is (1, 1)
        Maximum threads: 1
    rank 8 in MPI_COMM_WORLD has local rank 7 in local communicatior;
        Coordinate is (1, 2)
        Maximum threads: 1
    rank 9 in MPI_COMM_WORLD has local rank 8 in local communicatior;
        Coordinate is (1, 3)
        Maximum threads: 1
    rank 10 in MPI_COMM_WORLD has local rank 9 in local communicatior;
        Coordinate is (1, 4)
        Maximum threads: 1
    rank 11 in MPI_COMM_WORLD has local rank 10 in local communicatior;
        Coordinate is (2, 0)
        Maximum threads: 1
```

Figure 3. network configuration for single thread, process summary displayed for first 12 processes

```
details of communication:
event activation summary:
    event 0 is activated 128 times
    event 1 is activated 120 times
    event 2 is activated 134 times
    event 3 is activated 136 times
    event 4 is activated 116 times
    event 5 is activated 120 times
    event 6 is activated 130 times
    event 7 is activated 99 times
    total encryption time : 0.069863 seconds
    total decryption time : 0.075239 seconds
number of message pass between base station and nodes : 1003
number of message passing happened among nodes: 124000
total events detected: 983
```

Figure 4. event activation summary

```
Iteration 0

event number 1 detected on local rank 10 (rank 11 globally) with coordinate (2, 0)
Timestamp : Sat Oct 19 15:31:20 2019
adjacent nodes are (local): 5 15 11

event number 0 detected on local rank 17 (rank 18 globally) with coordinate (3, 2)
Timestamp : Sat Oct 19 15:31:20 2019
adjacent nodes are (local): 12 16 18

Iteration 2

event number 4 detected on local rank 8 (rank 9 globally) with coordinate (1, 3)
Timestamp : Sat Oct 19 15:31:20 2019
adjacent nodes are (local): 13 7 9

Iteration 3

event number 1 detected on local rank 9 (rank 10 globally) with coordinate (1, 4)
Timestamp : Sat Oct 19 15:31:20 2019
adjacent nodes are (local): 4 14 8

event number 6 detected on local rank 6 (rank 7 globally) with coordinate (1, 1)
Timestamp : Sat Oct 19 15:31:20 2019
adjacent nodes are (local): 11 5 7

Iteration 4

event number 6 detected on local rank 10 (rank 11 globally) with coordinate (2, 0)
Timestamp : Sat Oct 19 15:31:20 2019
adjacent nodes are (local): 5 15 11

Iteration 3

event number 1 detected on local rank 13 (rank 14 globally) with coordinate (2, 3)
Timestamp : Sat Oct 19 15:31:20 2019
adjacent nodes are (local): 8 12 14
```

Figure 5. Sample details for each event

```
@ mpiP
@ Command : ./wsn
@ Version                   : 3.4.1
@ MPIP Build date           : Sep 20 2019, 12:30:34
@ Start time                : 2019 10 19 15:31:19
@ Stop time                 : 2019 10 19 15:31:47
@ Timer Used                : gettimeofday
@ MPIP env var              : [null]
@ Collector Rank            : 0
@ Collector PID             : 16571
@ Final Output Dir          : .
@ Report generation         : Single collector task
@ MPI Task Assignment       : 0 hc03
@ MPI Task Assignment       : 1 hc03
@ MPI Task Assignment       : 2 hc04
@ MPI Task Assignment       : 3 hc05
@ MPI Task Assignment       : 4 hc05
@ MPI Task Assignment       : 5 hc05
@ MPI Task Assignment       : 6 hc06
@ MPI Task Assignment       : 7 hc06
@ MPI Task Assignment       : 8 hc06
@ MPI Task Assignment       : 9 hc10
@ MPI Task Assignment       : 10 hc10
@ MPI Task Assignment       : 11 hc10
@ MPI Task Assignment       : 12 hc10
@ MPI Task Assignment       : 13 hc10
@ MPI Task Assignment       : 14 mi06
@ MPI Task Assignment       : 15 mi07
@ MPI Task Assignment       : 16 mi08
@ MPI Task Assignment       : 17 mi08
@ MPI Task Assignment       : 18 mi08
@ MPI Task Assignment       : 19 mi08
@ MPI Task Assignment       : 20 mi08
```

Figure 6. host list for running process

```
---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0       27.6       27.6    99.97
   1       27.6        7.3    26.47
   2       27.6       7.11    25.80
   3       27.6       7.25    26.31
   4       27.6       7.26    26.35
   5       27.6       7.29    26.44
   6       27.6       7.16    25.96
   7       27.6        7.1    25.77
   8       27.6        7.1    25.76
   9       27.6       7.22    26.18
  10       27.6       7.18    26.06
  11       27.6       7.23    26.22
  12       27.6       7.23    26.22
  13       27.6       7.23    26.21
  14       27.6       7.06    25.60
  15       27.6       7.12    25.82
  16       27.5       7.13    25.89
  17       27.6       7.14    25.88
  18       27.6       3.61    13.09
  19       27.6       3.95    14.31
  20       27.6       7.08    25.69
   *        579        164    28.38
```

Figure 7. MPI time on each rank

```
--------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
--------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Sendrecv                8   1.12e+05   19.35   68.16    0.39
Waitany                 7   2.76e+04    4.76   16.77    0.00
Sendrecv               14   2.22e+04    3.83   13.51    1.97
Cart_create             9   1.92e+03    0.33    1.17    0.06
Cart_shift             10        624    0.11    0.38    0.55
Irecv                   2       14.2    0.00    0.01    0.00
Send                   13       7.87    0.00    0.00    0.63
Type_commit             6       2.51    0.00    0.00    0.12
Irecv                   5      0.828    0.00    0.00    0.00
Send                   11      0.495    0.00    0.00    1.17
Group_excl              3       0.42    0.00    0.00    0.32
Cart_coords            12      0.368    0.00    0.00    0.27
Comm_group              4      0.345    0.00    0.00    0.21
Irecv                   1       0.26    0.00    0.00    0.00
```

Figure 8.   Time spent on each function

threads num	events activation	node to base message number	node to node message number	agreagted encryption time	agregated decryption time
1	983	1003	124000	0.069863	0.075239
2	1018	1038	124000	0.972942	0.769046
4	1042	1062	124000	2.688	5.005932
8	1048	1068	124000	7.698206	16.056965

Figure 9.   Experiment result

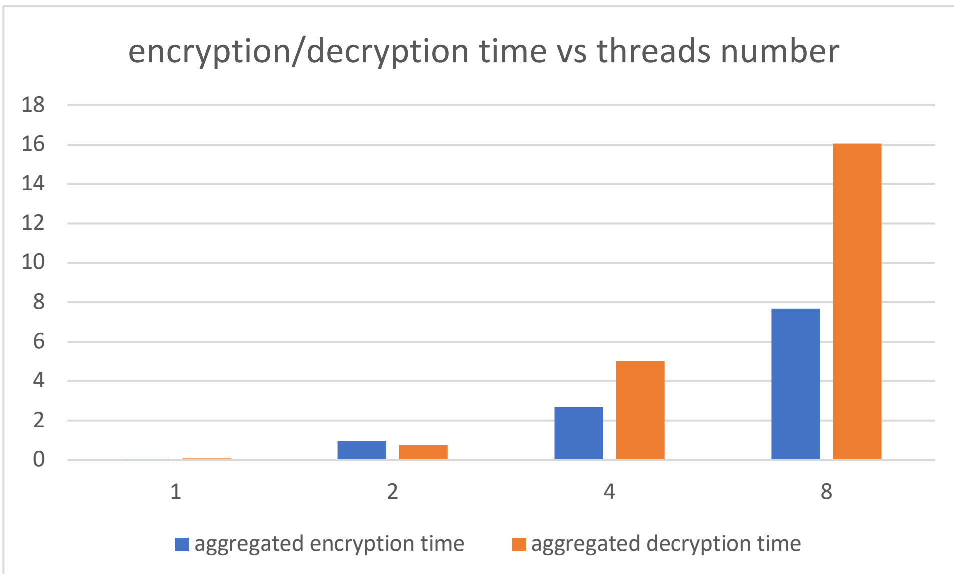

Figure 10. agregated encryption and decryption time with respect to number of threads

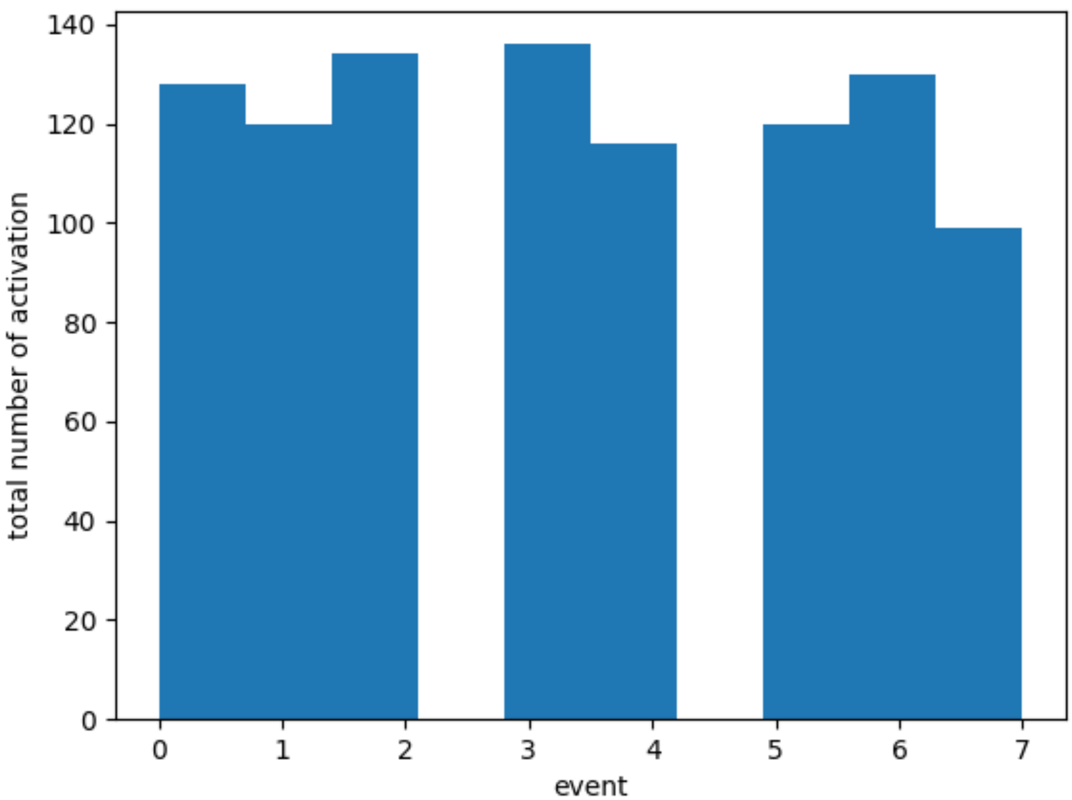

Figure 11. event activations