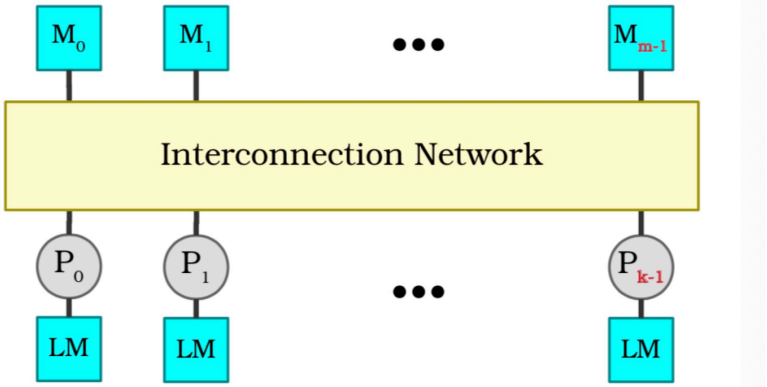
CompSci-230: Homework 1

Jitao Zhang

1. Introduction:

The main purpose is to simulate the process when there are a certain number of processors and a certain number of memories and to find the trends in **average waiting time(not include 1 cycle for processing)** under some assumptions.



For simplifying the problem, here are some key concepts:

1) The certain numbers of processors in {2, 4, 8, 16, 32, 64}
2) The certain numbers of memories ranging from 1 to 2048
3) Each request from processor is only for a certain memory and process it for only one cycle, the memory will be relinquished at the end of the current cycle.
4) Each request will wait in a queue until it is fulfilled, it generate a new request in the beginning of the next cycle.
5) The queue is a priority queue, the request which has the longest waiting time is first served, if there is a tie, use the lower index.
6) **Handle out of boundary in Gaussian distribution**. The memory id generated by Gaussian distribution might be out of boundary (not in the range of 1 to the total number of memories), in this case, loop to the reverse end, which means 1 is adjacent with the last memory.
7) We are looking for the **average waiting time(not include 1 cycle for processing)**

2. Thinking

The data structure we should use:

1) An array (size is the number of processors), represents each processor and its connecting memory
2) An array (size is the number of processors), represents the total waiting time for each processor
3) An array (size is the number of processors), represents the current waiting time for each processor
4) An array (size is the number of memories), represents whether the memory is holding
5) A while loop to simulate the process, and the termination condition is the current value differs from the previous one by less than 0.02%. So in the mean time we need to use two variables storing the previous cycle average waiting time and the current cycle average waiting time.

```
function diff(curr, prev) {
  return prev === null || (Math.abs(curr - prev) / prev) >= 2e-4;
}
```

6) When we break out the while loop, we get the final data we want!
7) The uniform distribution generator

```
function uniformDistribution(memories) {
  return Math.floor(Math.random() * memories);
}
```

8) The gaussian distribution generator

```
function getNumberIngaussianDistribution(mean, std_dev, total) {
  // using gaussianDistribution function generating the memory index
  let diff = Math.round(gaussianDistribution(mean, std_dev));
  // find the direction 1 or -1
  let direct = diff > 0 ? 1 : -1;
  for (let i = 0; i < Math.abs(diff); i++) {
    // add the direct diff times and guarantee the range in 1 - totoalNumber of memories
    mean += direct;
    if (mean >= total) mean = 0;
    if (mean < 0) mean = total - 1;
  }
  return mean;
}

function gaussianDistribution(mean, std_dev){
    return mean + (uniform2NormalDistribution() * std_dev);
}

function uniform2NormalDistribution(){
    let sum = 0.0;
    for(let i = 0; i < 12; i++){
        sum = sum + Math.random();
    }
    return sum - 6.0;
}
```

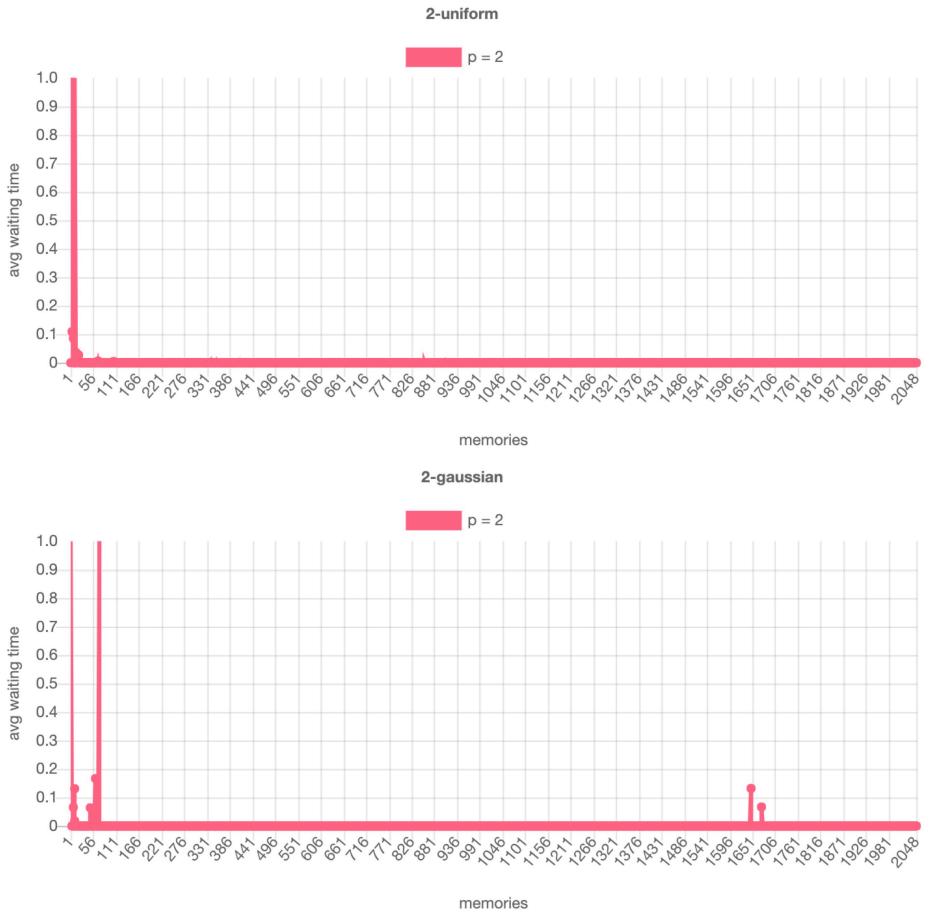
3. Main function

```
/**
 * calculate the average waiting time
 * @param {Number} M the number of the memories
 * @param {Number} P the number of the processors
 * @param {String} distribution the type of distribution(uniform or
gaussian)
 */
function calculate(M, P, distribution) {
  // size is P, requests[i] = j means processor i connects to memory
j
  const requests = [];
  // size is P, waitCounters[i] = j means the total waiting time for
processor i is j
  const waitCounters = [];
  // size is P, waitCounters[i] = j means the current waiting time
for processor i is j
  const curWaitCounters = [];
  // size is M, memories[i] = 1 means memory i connected to a
processor (0 means not)
  const memories = [];
  // size is P, means[i] = j means the mean of processor i is j in
Gaussian Distribution
  const means = [];

  // total number of successful requests
  let totalRequests = 0;
  // total number of cycles
  let cycles = 0;
  let prevAverageWaitingTime = null;
  let currAverageWaitingTime = null;

  // init
  for (let i = 0; i < P; i++) {
    requests.push(null);
    waitCounters.push(0);
    curWaitCounters.push(0);
  }

  for (let i = 0; i < M; i++) {
    memories.push(0);
  }

  if (distribution === "gaussian") {
    for (let i = 0; i < P; i++) {
      means.push(uniformDistribution(M));
    }
  }

/**
  * main while for calculate, the termination condition to count
average wait time is
  * when the current value differs from the previous one by less than
0.02%.
  */
  while (diff(currAverageWaitingTime, prevAverageWaitingTime)) {
    cycles++;
```

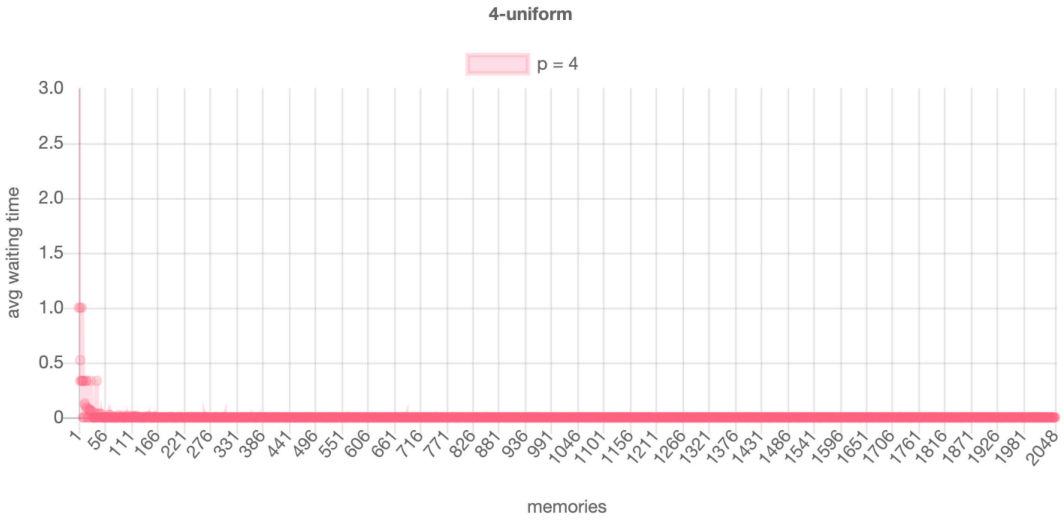
```
    prevAverageWaitingTime = currAverageWaitingTime;

    // make new requests for each processor if they have just
finished one
    // based on the current distribution
    for (let i = 0; i < requests.length; i++) {
      // if requests[i] == null
      if (!requests[i]) {
        if (distribution === "gaussian") {
          requests[i] = getNumberIngaussianDistribution(means[i],
std dev, M);
        } else if (distribution === "uniform") {
          requests[i] = uniformDistribution(M);
        } else {
          throw new Error("The distribution is invalid");
        }
      }
    }

    // To avoid processing element starvation, prioritize the one
waiting longestr.
    // for priorityMap, key is the memoryId, value is the processorId
and curWaitTime
    let priorityMap = new Map();

    for (let i = 0; i < requests.length; i++) {
      let memoryId = requests[i];
      let curWaitTime = curWaitCounters[i];

      if (!priorityMap.has(memoryId)) {
        // if memoryId is not in priorityMap, just assign the memory
to processor i
        priorityMap.set(memoryId, [i, curWaitTime]);
      } else {
        // if memoryId is in priorityMap, compare the value
        maxWaitTime = priorityMap.get(memoryId)[1];
        if (curWaitTime > maxWaitTime) {
          priorityMap.set(memoryId, [i, curWaitTime]);
        }
      }
    }

    // invert the key value pairs in priorityMap, key is processorId
now
    let priorityProcessor = new Map();

    for (let memoryId of priorityMap.keys()) {
      let processor = priorityMap.get(memoryId);
      priorityProcessor.set(processor[0], memoryId);
    }

    // use the results to update the requests
    for (let i = 0; i < requests.length; i++) {
      if (priorityProcessor.has(i)) {
        // current processor has been assigned a new memory
        let memoryId = priorityProcessor.get(i);
        memories[memoryId] = 1;
        requests[i] = null;
```

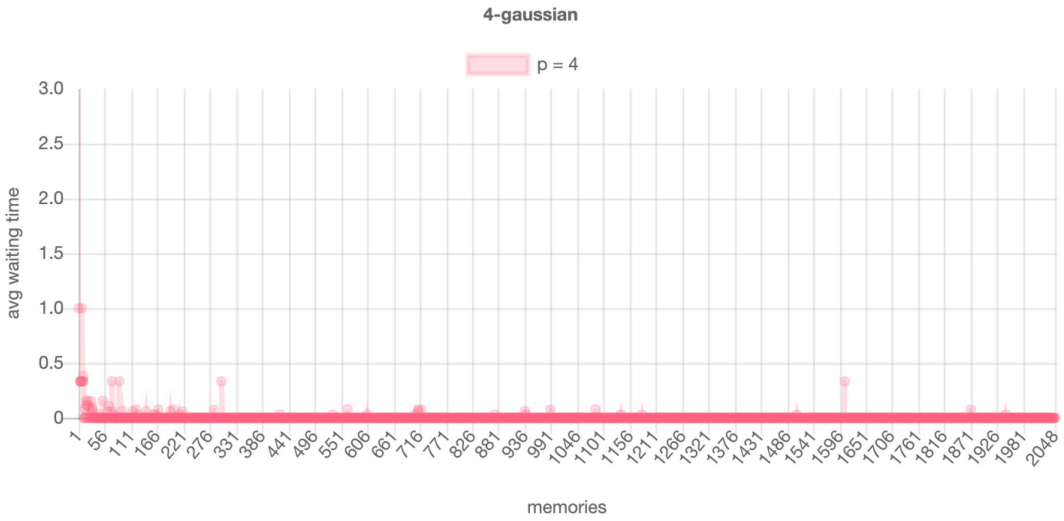
```
        curWaitCounters[i] = 0;
        totalRequests++;
      } else {
        // current processor has not been assigned a new memory, add
the waiting time
        waitCounters[i]++;
        curWaitCounters[i]++;
      }
    }

    // calculate the currAverageWaitingTime
    currAverageWaitingTime = waitCounters.reduce((acc, cur) => acc +
cur) / totalRequests;

    // free the memories
    for (let i = 0; i < memories.length; i++) {
      memories[i] = 0;
    }
  }

  return currAverageWaitingTime;
}
```

4. Results

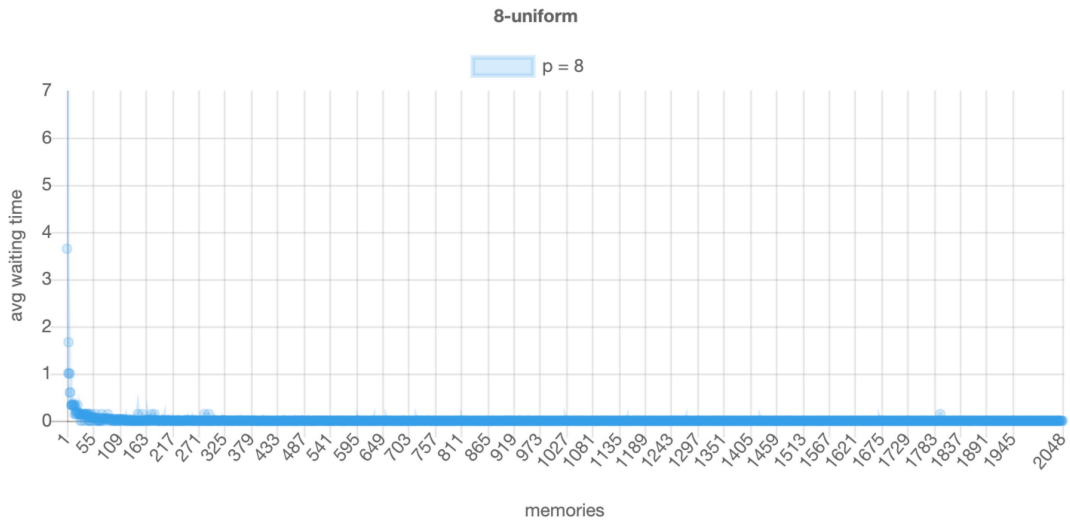


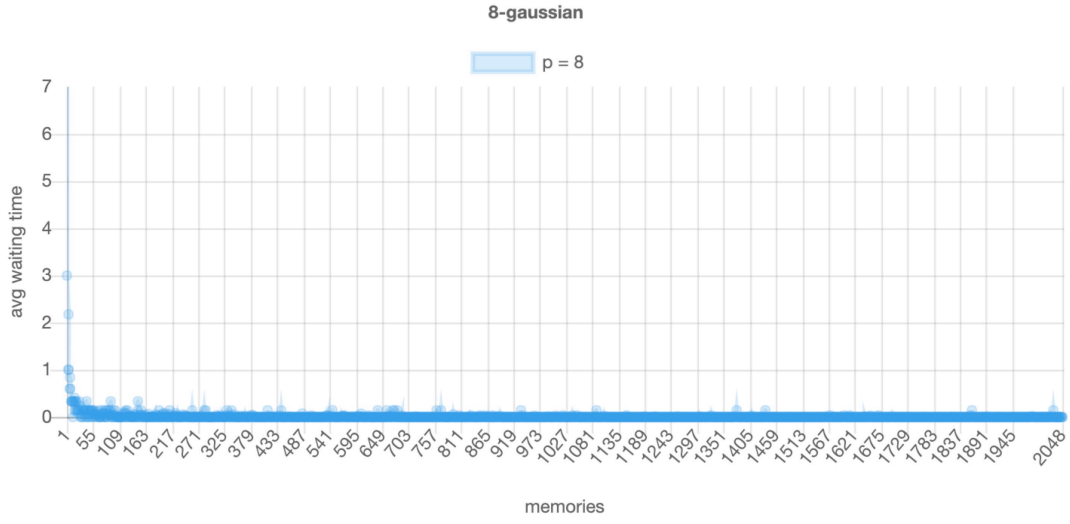
the number of processors is 2



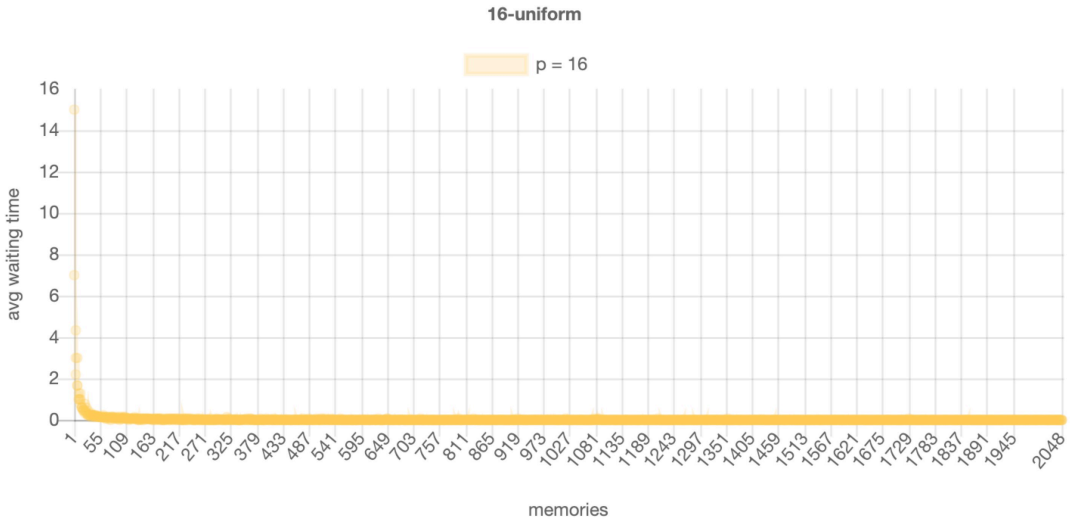


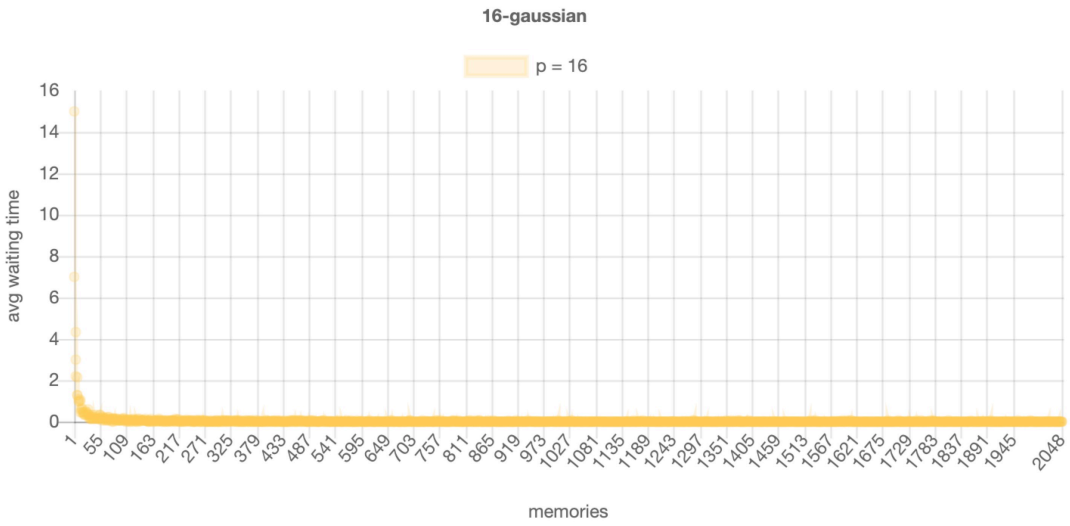
the number of processors is 4



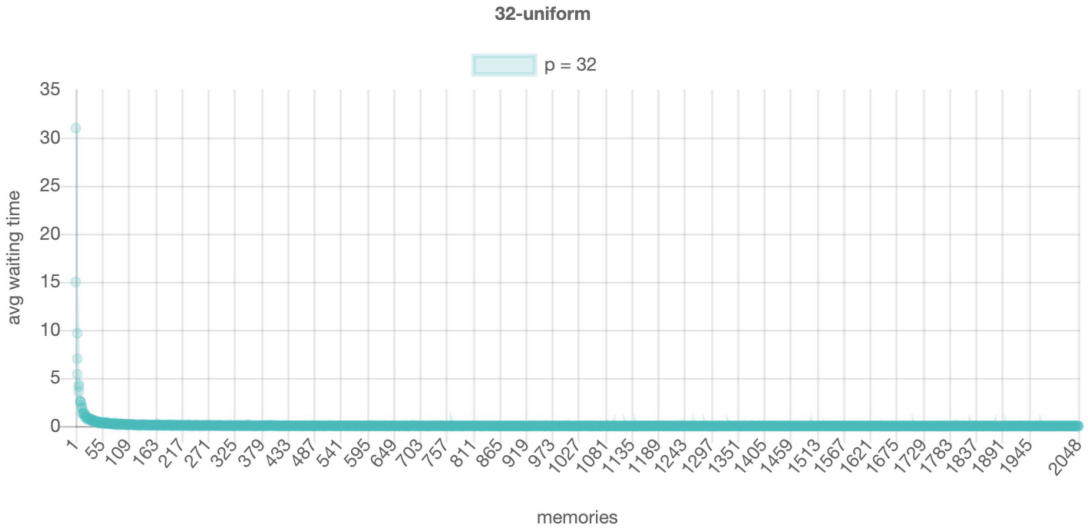


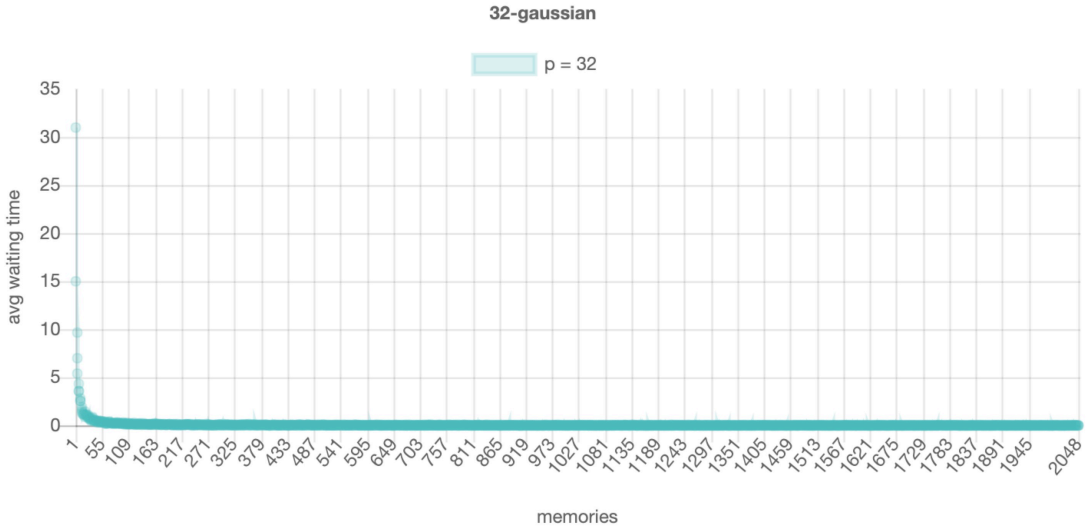
the number of processors is 8



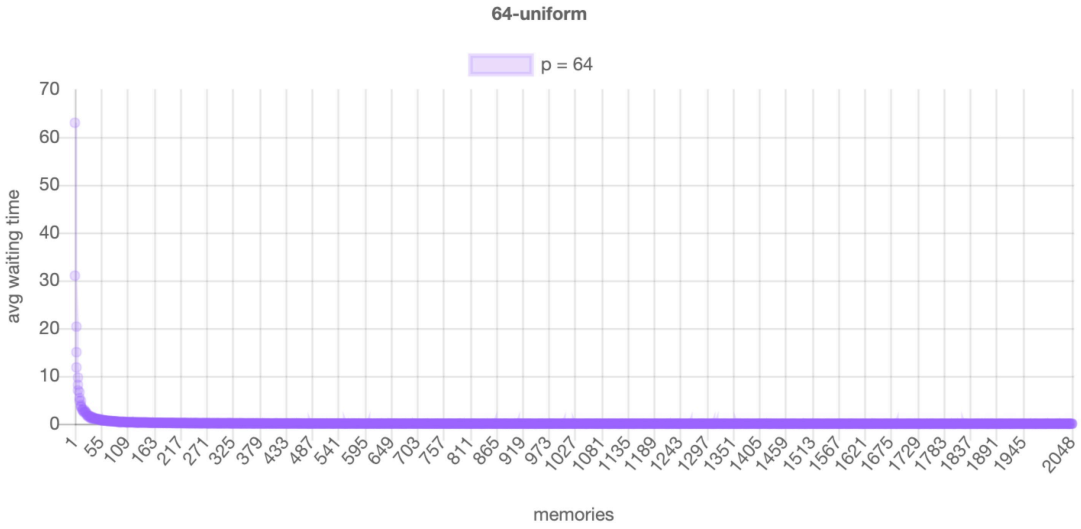


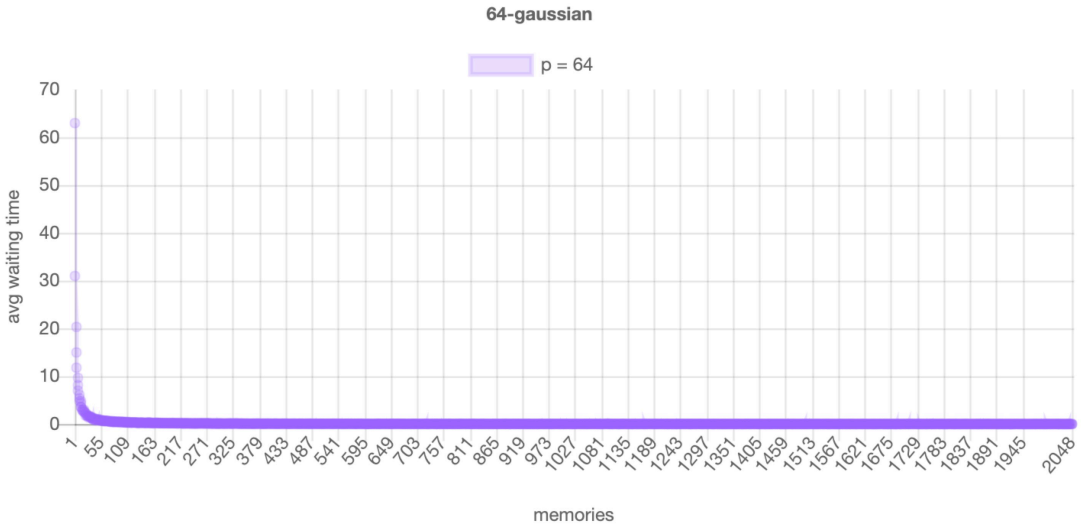
the number of processors is 16





the number of processors is 32



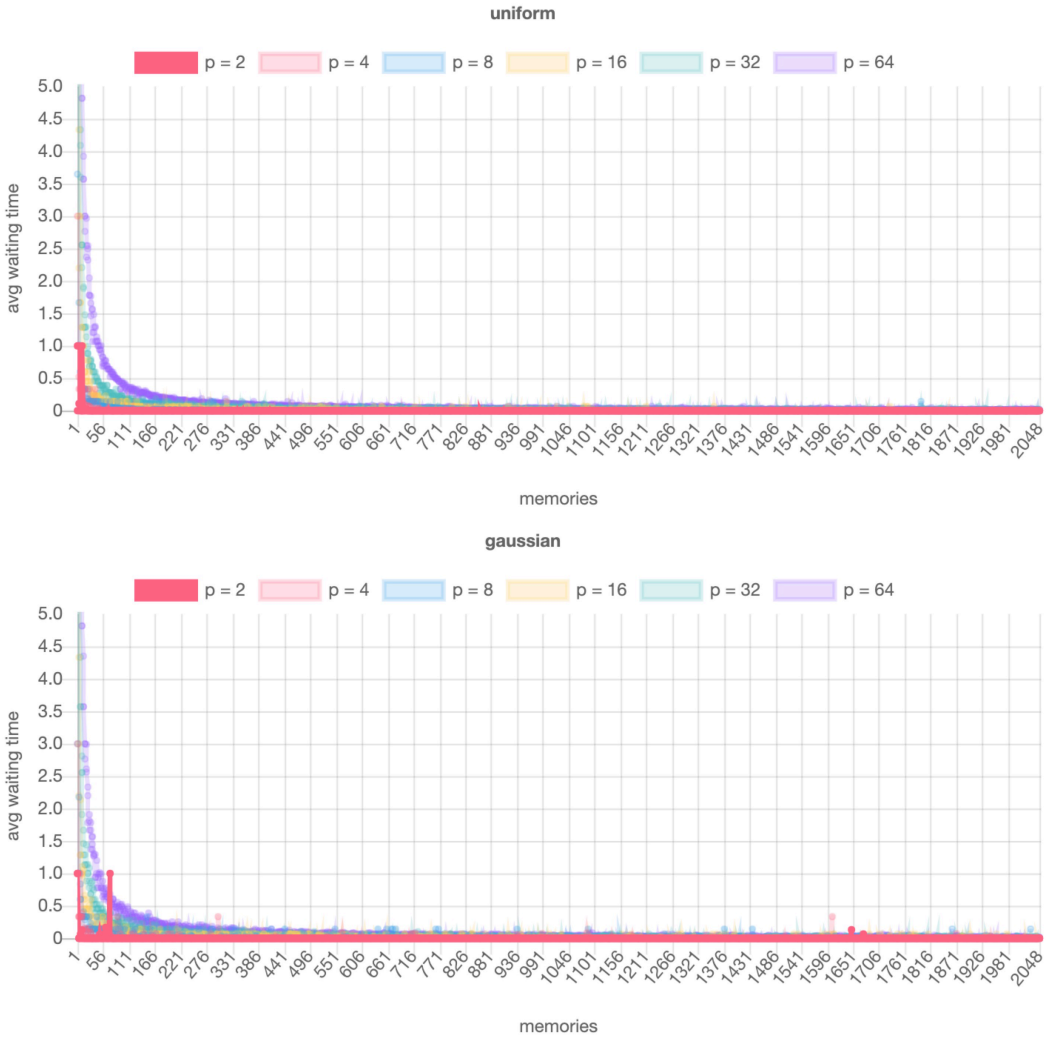


the number of processors is 64

Discussion:

1. It is so obvious the trends in different number of processors are all the same. It is so rapidly decreasing at the range from 1 – 100 and after that the average time is nearly 0 to the end. The so rapidly decreasing is because the range of processors are {2, 4, 8, 16, 32, 64}, but the max number of memory is 2048 which means the processor are easy to find a not holding memory when the memory number is much more than the number of processors.
2. The Y axis range of different processors are different. For {2, 4, 8, 16, 32, 64}, the average waiting time when the number of memory is 1 is {1, 3, 7, 15, 31, 63}. Here is the reason. In the first cycle, all the processors want to access the only 1 memory, but the #1 processor wins. So if the total number of processors is P, so the average

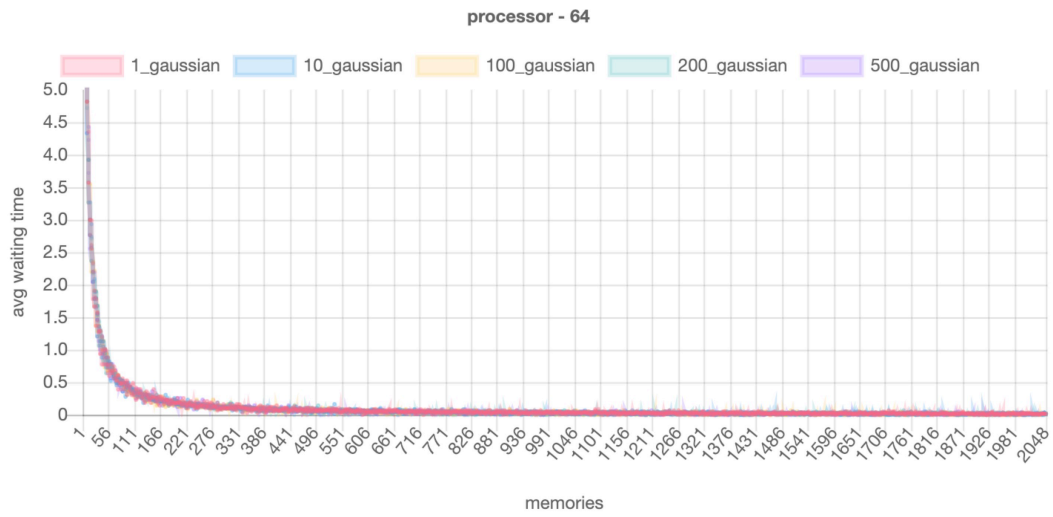
waiting time for cycle 1 is P – 1. In next cycle, the #2 wins, because the priority rule. But the #1, #3 - #P all add 1 waiting time, so average waiting time is still P – 1. The while loop break out.



Comparing all the data in each distribution in 5.0 – 0.0 in Y-axis

Discussion:

It obviously shows that the decreasing degree is different for the different processors. For the same number of memories, the larger number of processors results in higher average waiting time because there are more processors to compete. As you can see, the number of processors are more small, the curve is more easily to reach to 0.



Discussion:

I tried varying the standard deviation in {1, 10, 100, 200, 500}, but It is not easy to find some meaningful things. The idea is that when std is larger, the locality reference is widen than before, more like a uniform distribution. So I think it should have some patterns, but I did not see it might because that the number of processors is so small than the number of memories.