Notebook

September 17, 2024

1 Homework-1 (COP-6526) | Alex Sciuto

1.1 Problem-1. Compare the following two Python codes, which sum all elments in a big vector.

I added some additional code lines to be able to calculate the Real Time (Wall-clock time), User Time, and System Time as discussed in our last lecture. ***NOTE: It is the exact same code with extra logs, the operations were not modified***

[11]:
```
import array, time, resource

# Create a large array of 100 million elements
arr = array.array('l', range(100000000))
total_sum = 0

# Start timing
begin_real = time.time()  # Wall-clock time (real time)
begin_user, begin_sys = resource.getrusage(resource.RUSAGE_SELF)[:2]  # User␣
 ↪and system time

# Perform the summation
for i in range(10000):
    for j in range(10000):
        total_sum += arr[i * 10000 + j]

# Stop timing
end_real = time.time()
end_user, end_sys = resource.getrusage(resource.RUSAGE_SELF)[:2]

# Print the computed sum
print("Computation time for Version 1 Code:")
print(f"Total Sum: {total_sum}")

# Print the time measurements
print(f"Real Time (Wall-clock time): {end_real - begin_real:.6f} seconds")
print(f"User Time: {end_user - begin_user:.6f} seconds")
print(f"System Time: {end_sys - begin_sys:.6f} seconds")
```

```
Computation time for Version 1 Code:
Total Sum: 4999999950000000
Real Time (Wall-clock time): 12.366380 seconds
User Time: 12.083830 seconds
System Time: 0.085803 seconds
```

[12]:
```python
import array, time, resource

# Create a large array of 100 million elements
arr = array.array('l', range(100000000))
total_sum = 0

# Start timing
begin_real = time.time()  # Start wall-clock time (real time)
begin_user, begin_sys = resource.getrusage(resource.RUSAGE_SELF)[:2]  # Start␣
 ↪user and system time

# Perform the summation
for i in range(10000):
    for j in range(10000):
        total_sum += arr[j * 10000 + i]

# Stop timing
end_real = time.time()  # End wall-clock time
end_user, end_sys = resource.getrusage(resource.RUSAGE_SELF)[:2]  # End user␣
 ↪and system time

# Print the computed sum
print("Computation time for Version 2 Code:")
print(f"Total Sum: {total_sum}")

# Print the time measurements
print(f"Real Time (Wall-clock time): {end_real - begin_real:.6f} seconds")
print(f"User Time: {end_user - begin_user:.6f} seconds")
print(f"System Time: {end_sys - begin_sys:.6f} seconds")



#
```

```
Computation time for Version 2 Code:
Total Sum: 4999999950000000
Real Time (Wall-clock time): 13.777952 seconds
User Time: 13.306791 seconds
System Time: 0.173770 seconds
```

1.1.1 Question-1 (10 points): Which code use cache better? Why? Please test the execution time of the two code versions, which code runs faster? Does this result match your original thought?

After running the two versions of the code, we find that `version 1` utilizes cache more efficiently from a lower completion time from `version 1` (11.70s) compared to `version 2` (13.49). The reason this is the case has to do with how the loop is formed.

`Version 1` is accessing the array in a sequential order `arr[i * 1000 + j]` which means that the data is being stored contiguously in memory allowing for CPU cache to load a block of data once and reuse it multiple times. This is because this code leverages the phenomenon known as **Spatial Locality**, where consectuvie memory access are likely to be within the same cacheline, minimizing cache misses.

Conversly, `Version 2` is accessing the array in a strided pattern `arr[j * 1000 + i]` which means that the code needs to jump around in memory while doing the computation. This suggests that the code is not leveraging Spatial Locaility well, leading to more frequent cache misses and therefore lower performance.

In terms of my original thought, I did not have a preconception for which one would run faster. I did not consider how switching the loop from `arr[i * 1000 + j]` to `arr[j * 1000 + i]` would have an impact on how cache is utilized, for I thought the python intepreter has its own mechanisims for increasing cache efficiency.

1.1.2 Question-2 (10 points): Does Python utilize CPU cache well? You can investigate this topic by reading online discussion using google.

What I read is that Python is implemented through `CPython`, which actively translates python code into C and compiles it into bytecode. This is different from lower level languages like `C` which directly do the operations, which make it more efficient. Here are some specific details that make Python inefficient at utilizing CPU Cache.

Object Allocation and Deallocation: Allocation refers to the process of reserving a block of memory for a program to use, where if a new variable is made, it is where it is stored. Whereas Deallocation refers to freeing up that memory for it to be used by another program. Python frequently allocates and deallocates objects in an inefficient manner, where every time a new object is made or removed, it requires several memory operations and disrupts cache locality.

Variable-Sized Integers: lower level languages like `C` have fixed-size integers which help with cache efficiency. Python defaults to variable-sized integers which uses more memory, and decrease cache efficiency. Whenever python does operations on these variable-sized integers, additional memory access is required beyond the cache to manage the objects.

Global Interpreter Lock (GIL): Python uses GIL so only one thread can be used at a time to execute python bytecode. We have computers that have multiple cores (e.g., my Mac has access to 10 cores), and Python is unable to maximize CPU efficiency across the cores when performing operations. Cache operations specifically benefit from multi-threading, which can't occur when running python code due to the GIL.

1.1.3 Question-3 (10 points): A much more efficient way is to use "NumPy". Please use NumPy to do the same computation. What's the execution time? Why does it run much faster?

[18]:
```
import numpy as np
import time

# Create a large NumPy array of 100 million elements
arr = np.arange(100000000, dtype=np.int64)

# Start timing
begin = time.time()

# Perform the summation using a vectorized operation
total_sum = np.sum(arr.reshape(10000, 10000))

# Print the computed sum
print("For Version 1 Code with NumPy:")
print(f"Total Sum: {total_sum}")

# Print the total time taken
print(f"Time taken: {time.time() - begin:.6f} seconds")
```

```
For Version 1 Code with NumPy:
Total Sum: 4999999950000000
Time taken: 0.060395 seconds
```

[19]:
```
import numpy as np
import time

# Create a large NumPy array of 100 million elements
arr = np.arange(100000000, dtype=np.int64)

# Start timing
begin = time.time()

# Perform the summation using a different access pattern
total_sum = np.sum(arr.reshape(10000, 10000).T)

# Print the computed sum
print("For Version 2 Code with NumPy:")
print(f"Total Sum: {total_sum}")

# Print the total time taken
print(f"Time taken: {time.time() - begin:.6f} seconds")
```

```
For Version 2 Code with NumPy:
Total Sum: 4999999950000000
Time taken: 0.069948 seconds
```

1.2 Question-3 (10 points): A much more efficient way is to use "NumPy". Please use NumPy to do the same computation. What's the execution time? Why does it run much faster?

When we are comparing the execution times, we find that for both code blocks it is significantly faster using numpy. For the `version 1` code, without numpy it compiled in 11.70s and with it 0.060 (i.e., 195x faster); For the `version 2` code, without numpy is compiled in 13.49s and with it 0.069s (i.e., 195x faster).

In terms of why it is much faster, here are a few explanations:

numpy's use of vectorization: numPy leverages vectorized operations where it applied a function to an entire array at once, reather than iterating through each element one-by-one. This reduces the overhead associated with python loops, allowing for much faster computation time.

Memory Contiguity: Since numPy uses arrays, they are stored in contiguous blocks of memory which is cache-friendly. This reduces the number of cache misses compared to using the default python arrays,

Underlying C Implementation: NumPy is direectly implemented in C, which allows use to leverage specific CPU-level instructions. For instance, NumPy specifies that variables should be fixed-size data types (e.g., int64) which minimizes memory allocation and improves cache utilization.

1.3 Problem-2. (40 points) Implement a parallel matrix-vector multiplication in Python using MPI. You may generate a randon matrix and a corresponding vector, then broadcast the vector and scatter the matrix to all processes, and finally gather the result back.

In order to complete this operation I am going to use the `mpi4py` python library. When using this package, you need to directly run the code in terminal using the `mpiexec` command. I created a python file called `matrix-multiplication.py` and I am going to run it using two cores. I have also added detailed logging to understand the sequential process of how the code completed the vector multiplication.

Below are the contents of `matrix-multiplication.py`

[ ]:
```
from mpi4py import MPI
import numpy as np
import logging
import time  # Import the time module for tracking computation time

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Set up logging
logging.basicConfig(level=logging.DEBUG, format="%(asctime)s [%(levelname)s]␣
 ↪%(message)s", datefmt="%Y-%m-%d %H:%M:%S")
logger = logging.getLogger()
```

```python
# Matrix and vector sizes
n = 1000  # Change this size for larger matrices

# Start tracking total computation time
total_start_time = time.time()

if rank == 0:
    # Master process: generate random matrix and vector
    matrix = np.random.rand(n, n)
    vector = np.random.rand(n)
    logger.info(f"Rank {rank}: Generated random matrix and vector.")
else:
    # Other processes: initialize empty matrix and vector
    matrix = None
    vector = np.empty(n, dtype='d')
    logger.info(f"Rank {rank}: Initialized empty matrix and vector placeholders.
 ↪")

# Broadcast the vector to all processes
logger.info(f"Rank {rank}: Broadcasting vector from rank 0 to all processes.")
comm.Bcast(vector, root=0)
logger.info(f"Rank {rank}: Broadcast completed.")

# Scatter the matrix to all processes
rows_per_process = n // size
local_matrix = np.empty((rows_per_process, n), dtype='d')
logger.info(f"Rank {rank}: Scattering matrix: each process will receive␣
 ↪{rows_per_process} rows.")
comm.Scatter(matrix, local_matrix, root=0)
logger.info(f"Rank {rank}: Matrix scatter completed.")

# Start timing the local computation
local_start_time = time.time()

# Perform local matrix-vector multiplication
logger.info(f"Rank {rank}: Performing local matrix-vector multiplication.")
local_result = np.dot(local_matrix, vector)

# End timing the local computation
local_end_time = time.time()
local_computation_time = local_end_time - local_start_time

logger.info(f"Rank {rank}: Local result computed: {local_result[:5]}...")  #␣
 ↪Print only the first 5 elements for brevity
logger.info(f"Rank {rank}: Local computation time: {local_computation_time:.6f}␣
 ↪seconds")
```

```
# Gather the local results back to the master process
if rank == 0:
    result = np.empty(n, dtype='d')
else:
    result = None

logger.info(f"Rank {rank}: Gathering results from all processes.")
comm.Gather(local_result, result, root=0)

# End tracking total computation time
total_end_time = time.time()
total_computation_time = total_end_time - total_start_time

if rank == 0:
    logger.info(f"Rank {rank}: Result of matrix-vector multiplication gathered:␣
 ↪{result[:5]}...")  # Print only the first 5 elements for brevity
    logger.info(f"Rank {rank}: Total computation time: {total_computation_time:.
 ↪6f} seconds")
    print("Result of matrix-vector multiplication is computed.")
```

Now that we have `matrix-multiplication.py`, we are going to run it in the terminal using the command below:

[8]:
```
!mpiexec -n 2 python maxtrix-multiplication.py
```

```
2024-09-17 12:36:28 [INFO] Rank 1: Initialized empty matrix and vector
placeholders.
2024-09-17 12:36:28 [INFO] Rank 1: Broadcasting vector from rank 0 to all
processes.
2024-09-17 12:36:28 [INFO] Rank 0: Generated random matrix and vector.
2024-09-17 12:36:28 [INFO] Rank 0: Broadcasting vector from rank 0 to all
processes.
2024-09-17 12:36:28 [INFO] Rank 0: Broadcast completed.
2024-09-17 12:36:28 [INFO] Rank 0: Scattering matrix: each process will receive
500 rows.
2024-09-17 12:36:28 [INFO] Rank 1: Broadcast completed.
2024-09-17 12:36:28 [INFO] Rank 1: Scattering matrix: each process will receive
500 rows.
2024-09-17 12:36:28 [INFO] Rank 1: Matrix scatter completed.
2024-09-17 12:36:28 [INFO] Rank 1: Performing local matrix-vector
multiplication.
2024-09-17 12:36:28 [INFO] Rank 0: Matrix scatter completed.
2024-09-17 12:36:28 [INFO] Rank 0: Performing local matrix-vector
multiplication.
2024-09-17 12:36:28 [INFO] Rank 1: Local result computed: [248.56277831
243.65929634 256.8254286  253.48377645 245.10409473]…
2024-09-17 12:36:28 [INFO] Rank 1: Local computation time: 0.000167 seconds
2024-09-17 12:36:28 [INFO] Rank 1: Gathering results from all processes.
```

```
2024-09-17 12:36:28 [INFO] Rank 0: Local result computed: [248.53301436
247.77791774 241.99700993 254.81127472 240.81912635]…
2024-09-17 12:36:28 [INFO] Rank 0: Local computation time: 0.000339 seconds
2024-09-17 12:36:28 [INFO] Rank 0: Gathering results from all processes.
Result of matrix-vector multiplication is computed.
2024-09-17 12:36:28 [INFO] Rank 0: Result of matrix-vector multiplication
gathered: [248.53301436 247.77791774 241.99700993 254.81127472 240.81912635]…
2024-09-17 12:36:28 [INFO] Rank 0: Total computation time: 0.016090 seconds
```

The output details a parallel matrix-vector multiplication using MPI, where Rank 0 (the master process) generates a random matrix and vector, broadcasts the vector, and scatters parts of the matrix to all processes, including itself. Both Rank 0 and Rank 1 then perform their local matrix-vector multiplications and log the partial results, which are later gathered back to Rank 0.

Timing information shows that Rank 1 completes its local computation in 0.000167 seconds, while Rank 0 takes 0.000339 seconds. The total time for the entire parallel operation, including all communication steps, is 0.016090 seconds, demonstrating efficient parallel processing and minimal overall execution time.