HardCoder19 final README.md 4ad2886 · 18 minutes ago

dosp-project-1

This project implements a parallelized algorithm for finding perfect squares that are sums of consecutive squares using the actor model.

Work Unit Size

we are dividing the problem by worker size which we have predefined and kept it as 8. We chose this work unit size because it balances the workload across the available workers and minimizes idle time. Larger work unit sizes lead to fewer task switches, reducing the overhead, while smaller sizes allow for better load balancing across cores.

Performance Consideration:

In our tests, this size allowed all cores to be used efficiently without overloading any single worker or leaving too many idle. Adjustments were made based on CPU usage analysis and trial runs with different values of `n` and `k`.

Results for `n=1000000 k=4`

- **Number of workers created**: 8 workers
- **Total results found**: 0
- **Time Taken**:

- **User time**: 0.29 seconds

- **System time**: 0.01 seconds

- **CPU usage**: 433%

- **Total time**: 0.069 seconds (real time)

Explanation:

- The high CPU usage percentage (433%) indicates that multiple cores were effectively utilized in parallel, suggesting good parallelism.
- The `user` time reflects the time spent by the CPU executing the user-level code, while the `system` time represents the time spent on system-level operations.
- Despite good CPU utilization, no results were found for the problem size `n = 1000000` and `k = 4`.

Largest Problem Solved

The largest problem we managed to solve was:

```
<filename> 1000000 4
```

This was done in **0.069 seconds**, demonstrating efficient parallelization. Larger problem sizes could also be solved, but this configuration provided a good balance between complexity and performance.

Running Instructions

To run the program, use the following format:

for single machines:

```
<filename> <n> <k>
```

following are the screenshots for different number of workers:

4 worker

```
(base) ➜  Shubham time ./Shubham 1000000 4
Program started
Arguments count: 3
Parsing arguments
Creating coordinator
Starting calculation
Coordinator created
Starting calculation for n=1000000, k=4
Total workers created: 4
Worker created: 1 to 250000
Worker created: 250001 to 500000
Worker finished: 1
Total finished: 1/4
Worker created: 500001 to 750000
Worker finished: 250001
Total finished: 2/4
Worker created: 750001 to 1000000
Worker finished: 500001
Total finished: 3/4
Worker finished: 750001
Total finished: 4/4
All workers finished
Total results found: 0
Completed
./Shubham 1000000 4  0.31s user 0.01s system 182% cpu 0.174 total
(base) ➜  Shubham
```

8 worker

```
(base) ➜  Shubham time ./Shubham 1000000 4
Program started
Arguments count: 3
Parsing arguments
Creating coordinator
Starting calculation
Coordinator created
Starting calculation for n=1000000, k=4
Total workers created: 8
Worker created: 1 to 125000
Worker created: 125001 to 250000
Worker finished: 1
Total finished: 1/8
Worker created: 250001 to 375000
Worker finished: 125001
Total finished: 2/8
Worker created: 375001 to 500000
Worker finished: 250001
Total finished: 3/8
Worker created: 500001 to 625000
Worker finished: 375001
Total finished: 4/8
Worker created: 625001 to 750000
Worker finished: 500001
Total finished: 5/8
Worker created: 750001 to 875000
Worker finished: 625001
Total finished: 6/8
Worker created: 875001 to 1000000
Worker finished: 750001
Total finished: 7/8
Worker finished: 875001
Total finished: 8/8
All workers finished
Total results found: 0
Completed
./Shubham 1000000 4  0.32s user 0.01s system 186% cpu 0.173 total
(base) ➜  Shubham
```

16 worker

```
(base) ➜  Shubham time ./Shubham 1000000 4
Program started
Arguments count: 3
Parsing arguments
Creating coordinator
Starting calculation
Coordinator created
Starting calculation for n=1000000, k=4
Total workers created: 16
Worker created: 1 to 62500
Worker created: 62501 to 125000
Worker finished: 1
Total finished: 1/16
Worker created: 125001 to 187500
Worker finished: 62501
Total finished: 2/16
Worker created: 187501 to 250000
Worker finished: 125001
Total finished: 3/16
Worker created: 250001 to 312500
Worker finished: 187501
Total finished: 4/16
Worker created: 312501 to 375000
Worker finished: 250001
Total finished: 5/16
Worker created: 375001 to 437500
Worker finished: 312501
Total finished: 6/16
Worker created: 437501 to 500000
Worker finished: 375001
Total finished: 7/16
Worker created: 500001 to 562500
Worker finished: 437501
Total finished: 8/16
Worker created: 562501 to 625000
Worker finished: 500001
Total finished: 9/16
Worker created: 625001 to 687500
Worker finished: 562501
Total finished: 10/16
Worker created: 687501 to 750000
Worker finished: 625001
Total finished: 11/16
Worker created: 750001 to 812500
Worker finished: 687501
Total finished: 12/16
Worker created: 812501 to 875000
Worker finished: 750001
Total finished: 13/16
Worker created: 875001 to 937500
Worker finished: 812501
Total finished: 14/16
Worker created: 937501 to 1000000
Worker finished: 875001
Total finished: 15/16
Worker finished: 937501
Total finished: 16/16
All workers finished
Total results found: 0
Completed
./Shubham 1000000 4  0.31s user 0.01s system 179% cpu 0.175 total
(base) ➜  Shubham
```

single machine on a different computer

```
 ~/projects/dosp/shubham on  main !1 ?6 ❯ time ./shubham 1000000 4
Program started
Arguments count: 3
Parsing arguments
Starting calculation
Starting calculation for n=1000000, k=4
All workers finished
Total results found: 0
Completed
./shubham 1000000 4  0.29s user 0.01s system 433% cpu 0.069 total
```

for multiple machines:

```
<filename> <n> <k> <ip> <port> <max_client_num>
```

where:

- `n` is the upper limit of the range to check.
- `k` is the number of consecutive squares to sum.
- `ip` is the ip of the system on which the server runs
- `port` is the port to which we want to connect
- `max_client_num` is the max number of clients

For example:

```
<filename> 1000000 4
```

Where:

- `n` is the upper limit of the range to check.
- `k` is the number of consecutive squares to sum.

Conclusion

This implementation efficiently divides the problem into work units and leverages parallelism using actors. The results demonstrate good utilization of CPU cores, and the system scales well with the problem size. Further optimization can focus on reducing overhead for small workloads and improving parallel efficiency for larger problem sets.

Multi-Machine TCP Client-Server System

Overview

This project implements a multi-machine TCP client-server system, where the server distributes computational tasks to multiple clients, and the clients process these tasks and send the results back to the server. The system is designed for performing calculations related to perfect squares that are sums of consecutive squares.

Components

The system consists of the following main components:

1. **Client (`client.pony`)**

2. **Server (`server.pony`, `listener.pony`, `net.pony`, `client_handler.pony`)**

Server Architecture

The server is responsible for managing incoming client connections and distributing computational tasks across multiple clients. It is divided into several classes:

1. **`Listener`**: A TCP listener that accepts incoming client connections. It assigns each client to a `ClientHandler`.

2. **`ClientHandler`**: Handles communication with a specific client. It manages receiving messages from the client and closing the connection when the client disconnects.

3. **`Server`**: The core of the system. It manages all connected clients, distributes tasks once all clients have connected, and handles incoming messages from the clients.

Client Architecture

The client connects to the server, receives a computational task, performs the calculation, and sends the result back to the server. The client is structured into the following parts:

1. `ClientSide` : Implements the client logic. It manages connecting to the server, sending data to the server, receiving tasks, and processing the tasks.

2. `TaskMaster` : The actor responsible for dividing the computational task into subtasks, distributing them to workers, and aggregating the results.

3. `TaskSlave` : Performs the actual calculation for a subtask and reports the result back to the `TaskMaster` .

System Workflow

1. **Server Start-Up**

   - The server is started by invoking the `Main` actor from the `net.pony` file. The server begins listening for incoming TCP connections on the specified IP and port.

   - The server distributes computational tasks after all clients have connected.

2. **Client Start-Up**

   - Clients are started by invoking the `ClientSide` class, which attempts to connect to the server.

   - Once a connection is established, the client sends a greeting and waits for a task to be assigned by the server.

3. **Task Distribution**

   - When all expected clients have connected, the server splits the main task into smaller chunks. Each chunk is a range of numbers that the clients will process to find perfect squares that are sums of consecutive squares.

   - The server sends each client a message formatted as `TASK:<start>:<end>:<window_size>` , where:

     - `start` is the starting number for the computation.

     - `end` is the ending number for the computation.

     - `window_size` is the number of consecutive squares to sum.

4. **Task Execution**

   - Upon receiving the task, the client processes the numbers in the given range using the `TaskMaster` and `TaskSlave` actors. Each `TaskSlave` checks if the sum of consecutive squares is a perfect square and reports any valid results back to the `TaskMaster` .

- Once all workers have completed their tasks, the client sends the final result back to the server.

5. **Result Handling**

- The server receives the results from all clients and may further aggregate or display the results.

How to Run the System

1. **Server Setup**

- Compile and run the server by passing the required arguments to the `Main` actor in `net.pony`:

```
./server <n> <k> <ip> <port> <max_client_num>
```

- `<n>`: The upper limit of the range to search for perfect squares.
- `<k>`: The number of consecutive squares to sum.
- `<ip>`: The IP address on which the server listens.
- `<port>`: The port on which the server listens.
- `<max_client_num>`: The number of clients expected to connect.

2. **Client Setup**

- Compile and run the client by invoking the `ClientSide` class:

```
./client <server_ip> <server_port>
```

- `<server_ip>`: The IP address of the server to connect to.
- `<server_port>`: The port of the server to connect to.

Example

1. **Server**:

```
./server 100 4 127.0.0.1 8080 2
```

This starts the server to distribute a task to find perfect squares that are sums of four consecutive squares in the range from 1 to 100, listening on `127.0.0.1:8080`, and expecting two clients to connect.

2. **Client 1**:

```
./client 127.0.0.1 8080
```

3. **Client 2**:

```
./client 127.0.0.1 8080
```

Once both clients are connected, the server will distribute the tasks, and the clients will process them and send back their results.

Key Features

- **Concurrency**: The system leverages Pony's actor model to handle multiple clients concurrently. Each task is split across multiple `TaskSlave` actors for parallel processing.
- **Resilience**: The server and clients handle connection errors and client disconnections gracefully.
- **Task Distribution**: The server dynamically divides the computational load among connected clients, ensuring balanced task allocation.

Dependencies

- **Pony**: This project uses the Pony programming language, which offers powerful concurrency and error-handling features.