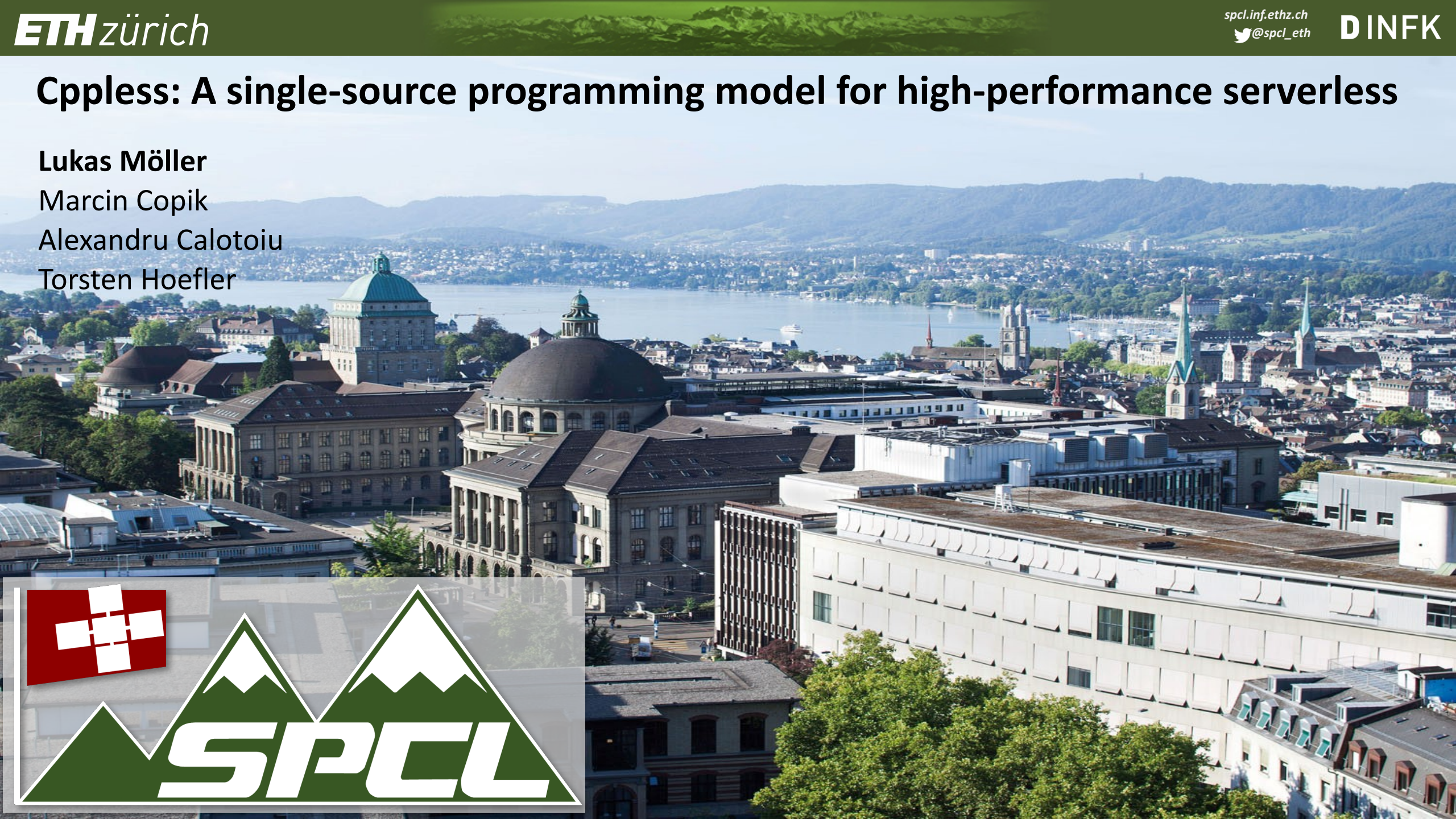
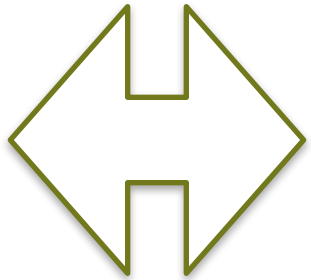
Cppless: A single-source programming model for high-performance serverless

Lukas Möller
Marcin Copik
Alexandru Calotoiu
Torsten Hoefler



Client Code

```
#include <iostream>

Aws::String functionName = "pi-estimate-worker";

void InvokeFunction(std::shared_ptr<Aws::Lambda::LambdaClient> client,
                    int invocations, int &result) {
  Aws::Lambda::Model::InvokeRequest invokeRequest;
  invokeRequest.SetFunctionName(functionName);
  invokeRequest.SetInvocationType(
      Aws::Lambda::Model::InvocationType::RequestResponse);
  std::shared_ptr<Aws::IOStream> payload = Aws::MakeShared<Aws::StringStream>();
  Aws::Utils::Json::JsonValue jsonPayload;
  jsonPayload.WithInt64("iterations", iterations);
  *payload << jsonPayload.View().WriteReadable();
  invokeRequest.SetBody(payload);
  invokeRequest.SetContentType("application/json");
  auto outcome = client->Invoke(invokeRequest);
  if (outcome.IsSuccess()) {
    auto &result = outcome.GetResult();
    Aws::IOStream &payload = result.GetPayload();
    Aws::String functionResult;
    std::getline(payload, functionResult);
    result std::stoi(functionResult);
  }
}

int main() {
  int n = 10000;
  int np = 10;

  Aws::SDKOptions options;
  Aws::InitAPI(options);
  {
    Aws::Client::ClientConfiguration clientConfig;
    std::shared_ptr < Aws::Lambda::LambdaClient >> m_client =
        Aws::MakeShared<Aws::Lambda::LambdaClient>(ALLOCATION_TAG,
                                                   clientConfig);

    std::vector<int> results(np);

    std::vector<std::thread> threads;
```



Lambda Code

```
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{

    using namespace Aws::Utils::Json;

    JsonValue json(request.payload);
    if (!json.WasParseSuccessful()) {
        return invocation_response::failure("Failed to parse input JSON",
"InvalidJSON");
    }

    auto iterations = json.GetInt64("iterations");
    auto result = pi_estimation(iterations);
    auto response = std::to_string(result);
    return invocation_response::success(response, "application/json");
}

int main()
{
    run_handler(my_handler);
    return 0;
}
```

```cpp
double pi_estimate(int n);

int main(int, char*[])
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std::vector<double> results(np);
  auto fn = [=] { return pi_estimate(n / np); };
  for (auto& result : results)
    cppless::dispatch(aws, fn, result);
  cppless::wait(aws, np);

  auto pi = std::reduce(results.begin(),
    results.end()) / np;
  std::cout << pi << std::endl;
}
```

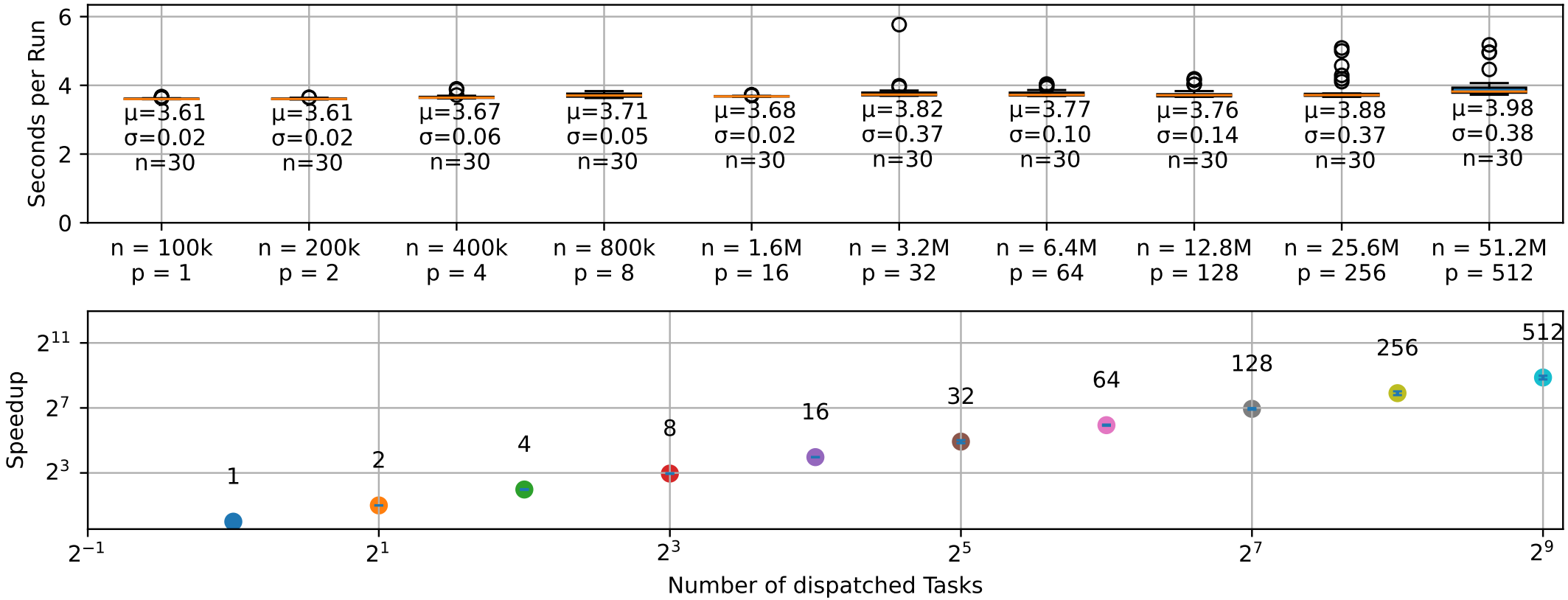
- **Create serverless function provider abstraction**
- **Define serverless function using lambda expressions**
- **Transparent invocation using dispatcher interface**
  - **Automatic Serialisation**
  - **Abstraction of provider-specific API**
- **Wait for results to be written into results**

Evaluation

- **Benchmarks**
  - **Fibonacci** - Calls serverless functions recursively
  - **Floorplan** - NP hard, laying out 2D boxes on a grid, minimizing bounding box
  - **Knapsack** - NP hard, branch and bound implementation
  - **N-Queens** - Determine number of ways queens can be placed on NxN chess board without interfering with each other
  - **CPU-Raytracer** - Monte-Carlo raytracer derived from RTW1
  - **Pi-Estimation**

Evaluation - PI Estimation, Speedup

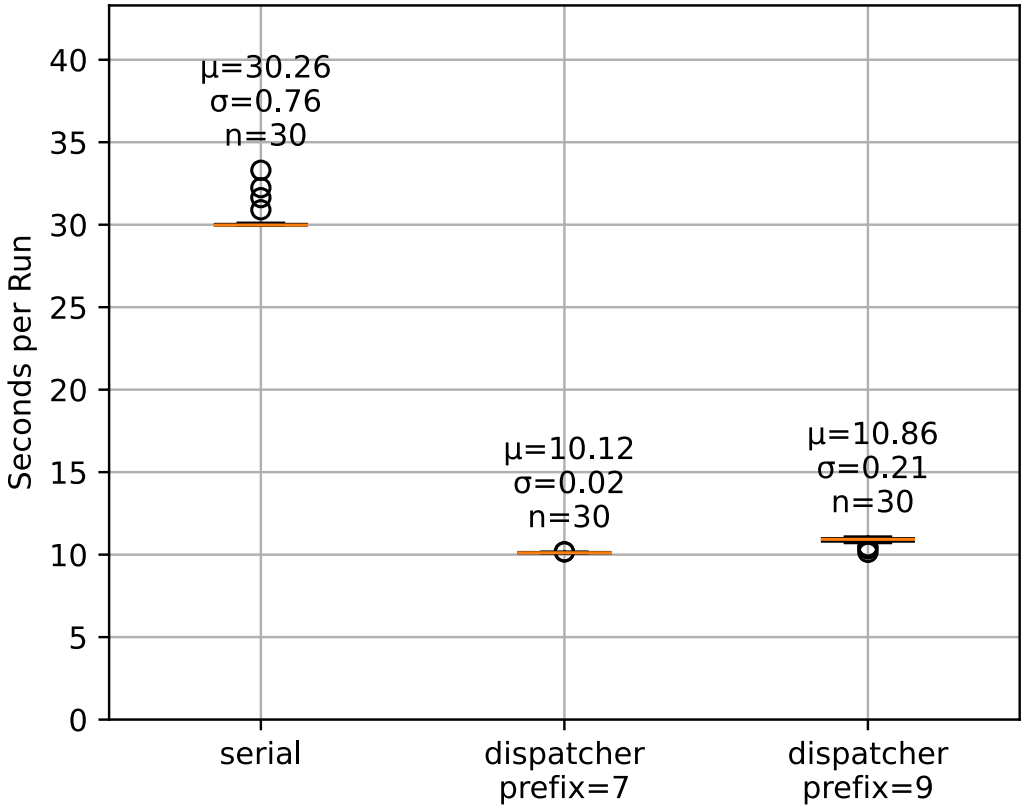
Workload scales with parallelism

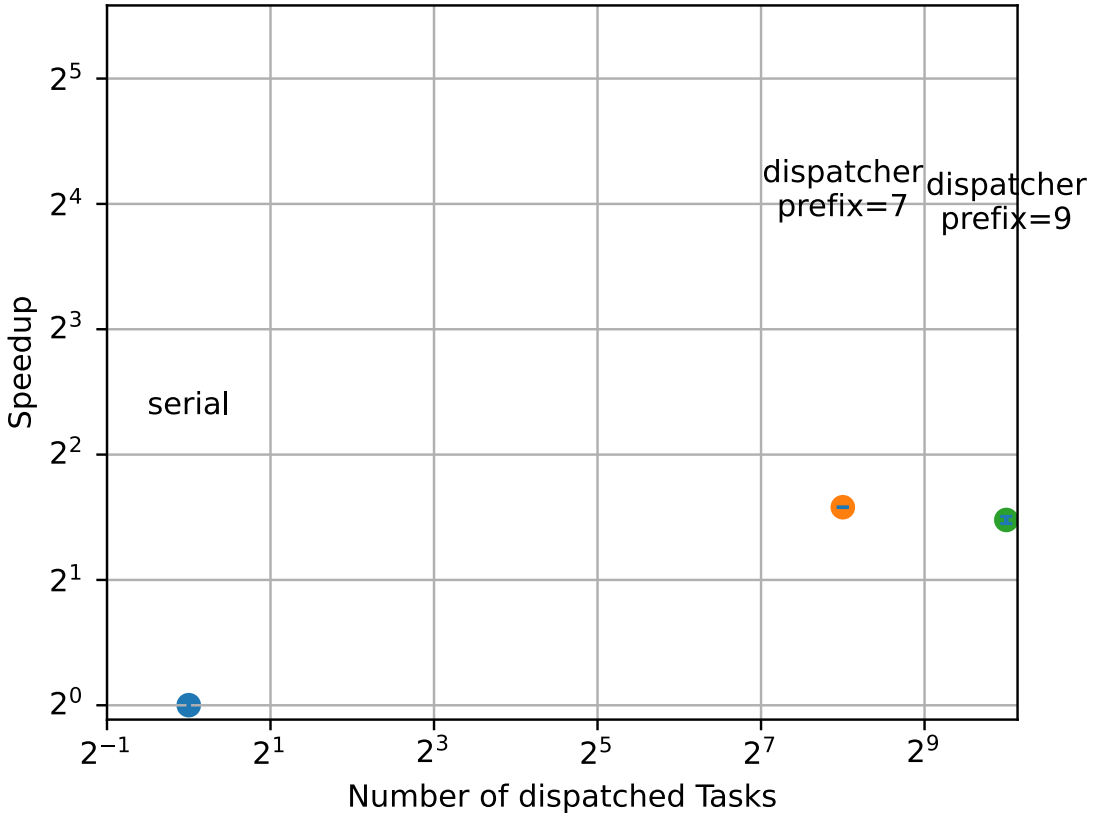


- **Near-perfect scaling due to uniform task length and embarrassingly parallel implementation.**

Evaluation - Knapsack(40), Speedup

Same Workload executed with different parallelism

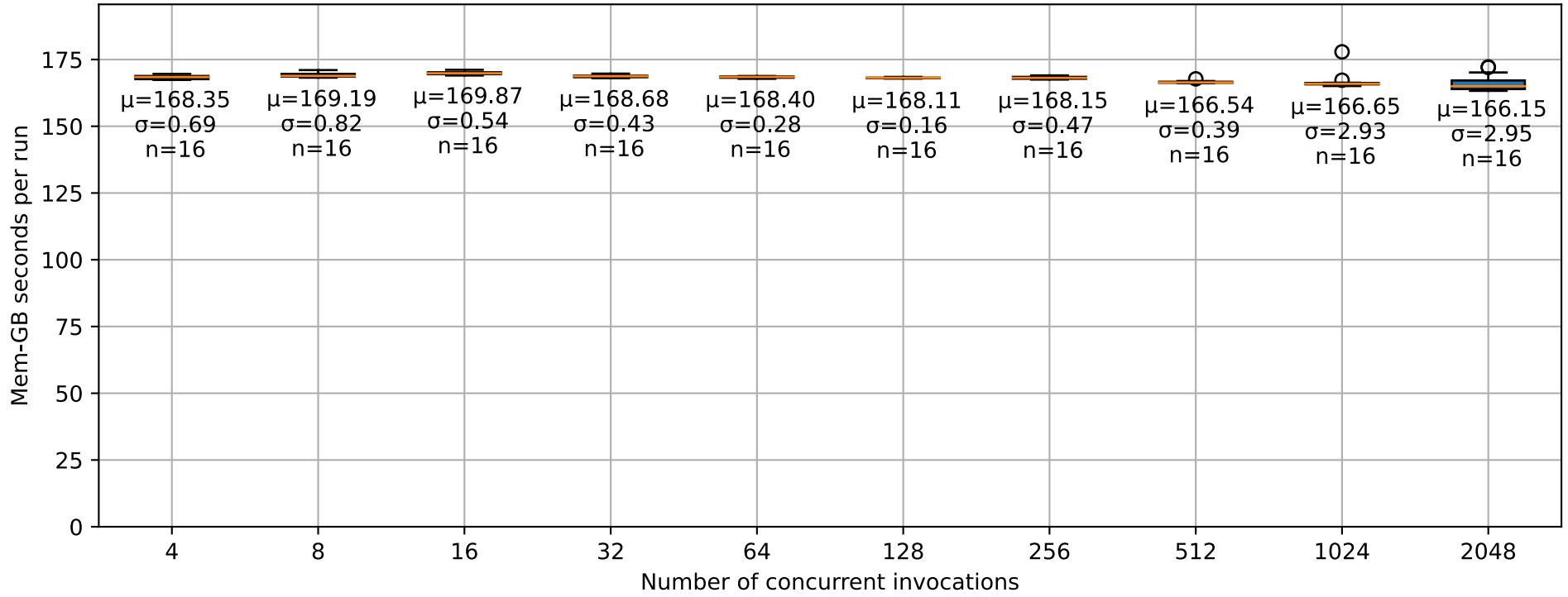




- **Rather mediocre speedup, even with a lot of parallelism.**

Evaluation - CPU Raycasting, Cost Analysis

Same Workload executed with different parallelism, Mem-GB seconds as billed by AWS



- Although speedup might be suboptimal, the overhead added at the cloud provider is statistically insignificant

Conclusion

- The cppless compilation architecture with its language extensions allows for **elegant compile-time definitions of serverless functions**
- The overhead incurred by the framework is only significant if many small tasks are dispatched or if each task requires a large amount of data
  - For trivial tasks the overhead is around <1ms, increases as more data needs to be serialised

---

- Proof-of-concept Implementation:
  - https://github.com/spcl/cppless-clang
  - https://github.com/spcl/cppless