HPCSE II

Multithreading: memory considerations

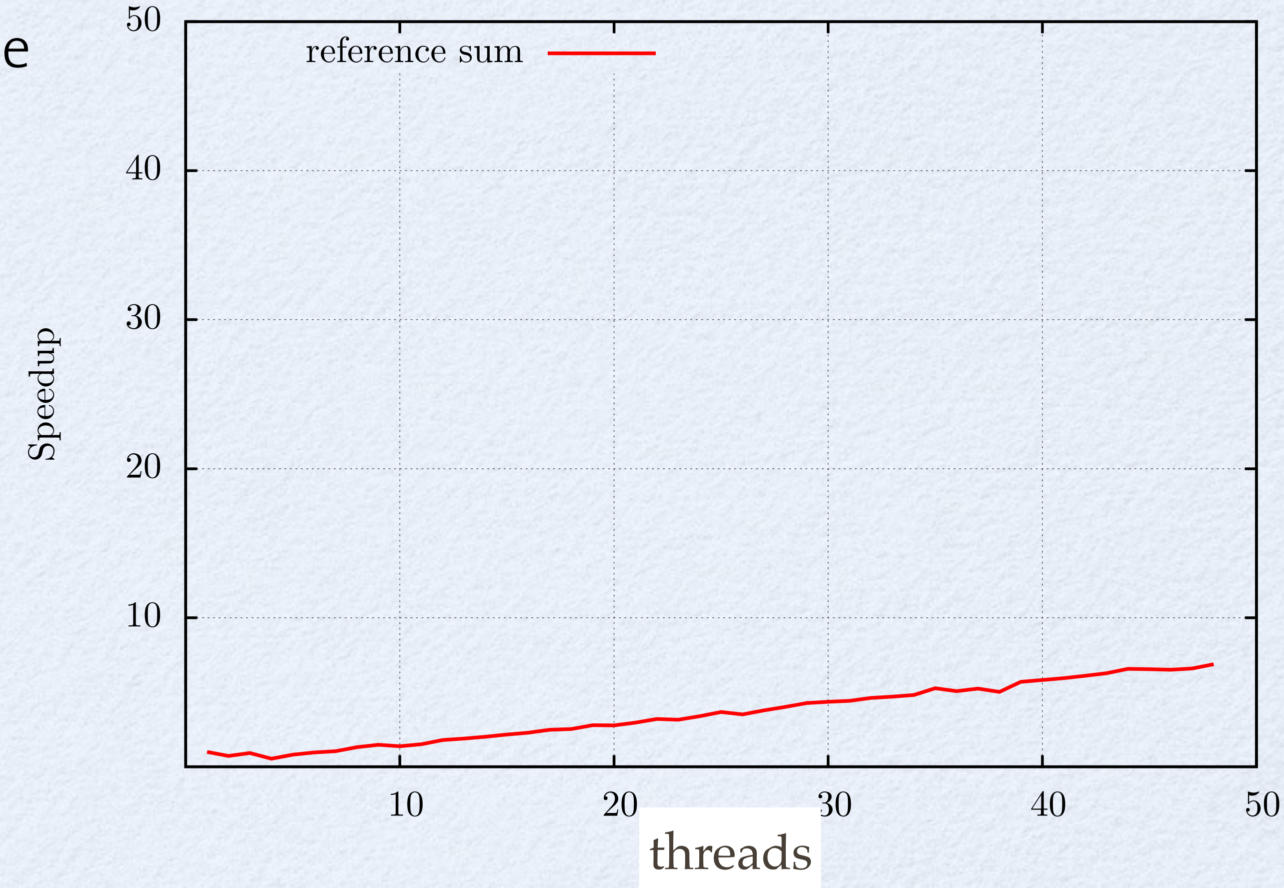
Review: calculating π through a series

```
#include <vector>
#include <iostream>
#include <thread>
#include <numeric>
#include <iomanip>


// sum terms [i-j) of the power series for
// pi/4
void sumterms(long double& sum,
              std::size_t i, std::size_t j)
{
  sum = 0.0;

  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
}
```

```
int main()
{
  // decide how many threads to use
  std::size_t const nthreads = std::max(1u,
          std::thread::hardware_concurrency());

  std::vector<std::thread> threads(nthreads);
  std::vector<long double> results(nthreads);

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  nthreads;

  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] =std::thread(
        sumterms, std::ref(results[i]),
        i * step, (i+1) * step
      );

  for (std::thread& t : threads)
    t.join();

  long double pi = 4 * std::accumulate(
             results.begin(), results.end(), 0.);

  std::cout << "pi=" << std::setprecision(18)
          << pi << std::endl;

  return 0;
}
```

Review: calculating π through a series

- We used a std::vector<long double> to store the result values from each thread.
  - Advantage: lock-free
  - Disadvantage?



- Why is the speedup so bad? **Cache thrashing!**

Cache thrashing

```
#include <vector>
#include <iostream>
#include <thread>
#include <numeric>
#include <iomanip>


// sum terms [i-j) of the power series for
// pi/4
void sumterms(long double& sum,
              std::size_t i, std::size_t j)
{
  sum = 0.0;

  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
}
```

One cache line contains the sum variables of multiple threads!

Cache thrashing: a thread invalidates the cache for other threads and sum has to be reloaded!

```
int main()
{
  // decide how many threads to use
  std::size_t const nthreads = std::max(1u,
          std::thread::hardware_concurrency());

  std::vector<std::thread> threads(nthreads);
  std::vector<long double> results(nthreads);

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  nthreads;

  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] =std::thread(
        sumterms, std::ref(results[i]),
        i * step, (i+1) * step
      );

  for (std::thread& t : threads)
    t.join();

  long double pi = 4 * std::accumulate(
             results.begin(), results.end(), 0.);

  std::cout << "pi=" << std::setprecision(18)
            << pi << std::endl;

  return 0;
}
```

Solving the cache-thrashing

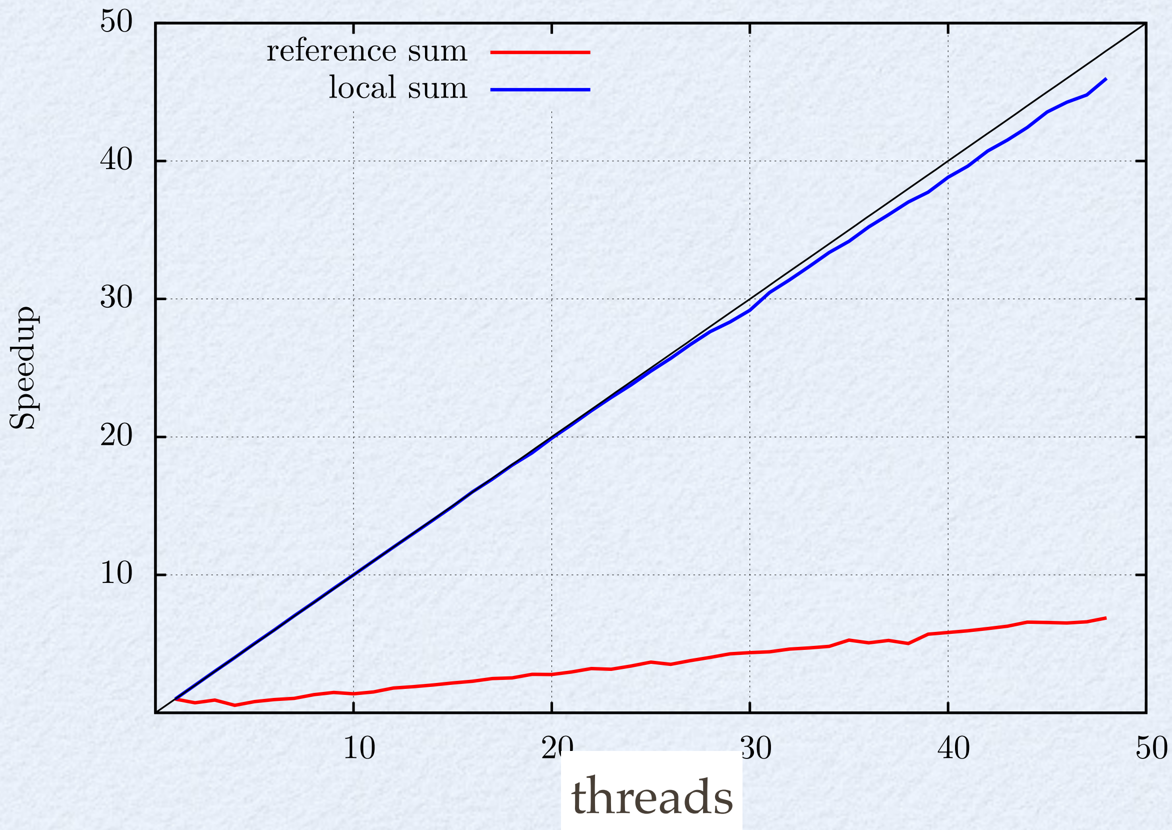
```
#include <vector>
#include <iostream>
#include <thread>
#include <numeric>
#include <iomanip>


// sum terms [i-j) of the power series for
// pi/4
void sumterms(long double& result,
              std::size_t i, std::size_t j)
{
  long double sum = 0.0;

  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);

  result = sum;
}
```

Solution: use a thread-local variable for the summation

```
int main()
{
  // decide how many threads to use
  std::size_t const nthreads = std::max(1u,
          std::thread::hardware_concurrency());

  std::vector<std::thread> threads(nthreads);
  std::vector<long double> results(nthreads);

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  nthreads;

  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] =std::thread(
        sumterms, std::ref(results[i]),
        i * step, (i+1) * step
      );

  for (std::thread& t : threads)
    t.join();

  long double pi = 4 * std::accumulate(
             results.begin(), results.end(), 0.);

  std::cout << "pi=" << std::setprecision(18)
          << pi << std::endl;

  return 0;
}
```

Now the scaling works

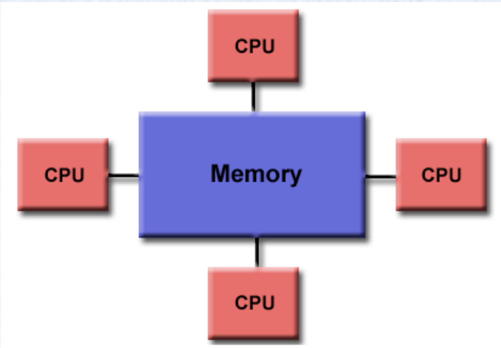
- Lessons learned:
  - always make scaling plots
  - avoid to pollute the cache of other threads
  - efficient multi-threading is non-trivial



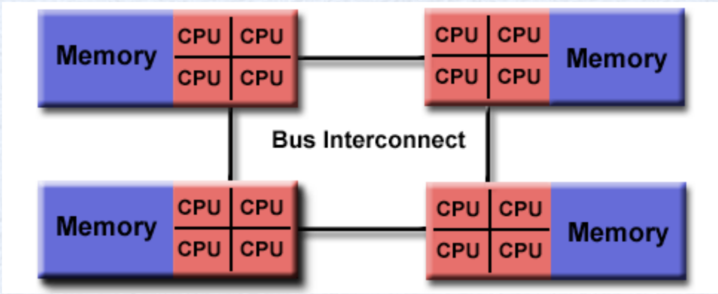
NUMA effects

- Recall NUMA (non-uniform memory access)

Uniform Memory Access (UMA):



Non-Uniform Memory Access (NUMA):

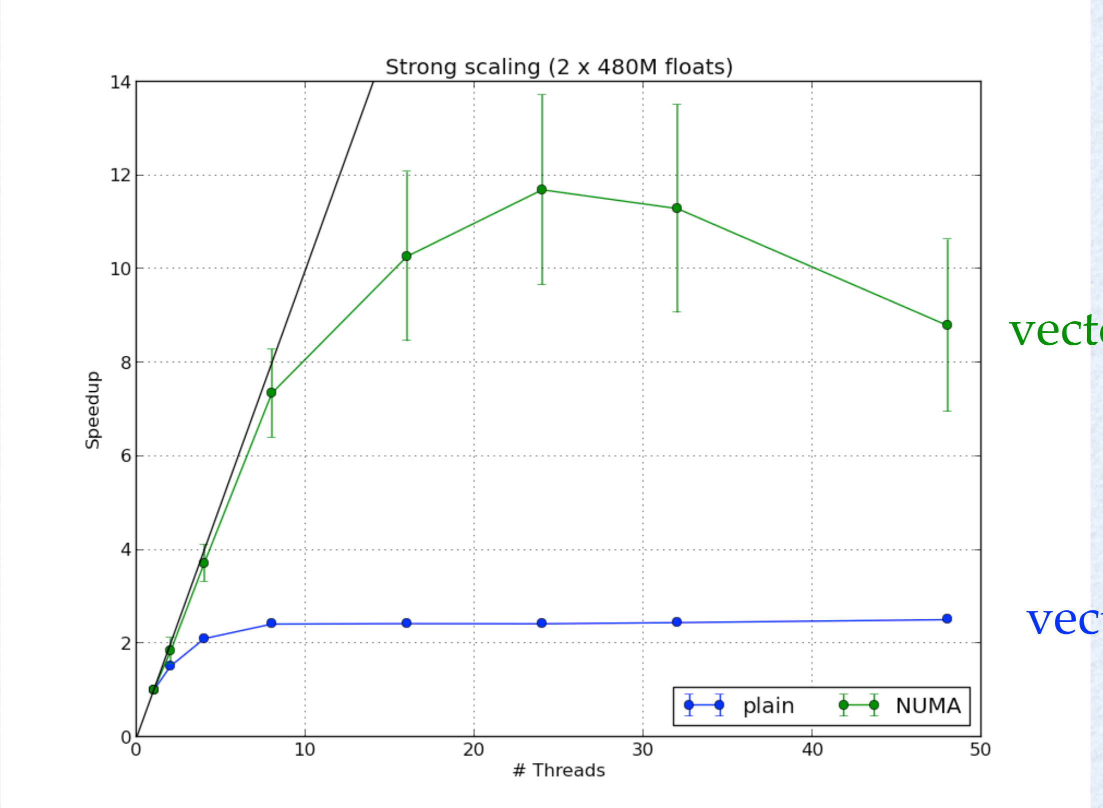


  - Advantage of NUMA: better scalability
  - Disadvantage of NUMA: memory latency depends on where data is allocated in memory

- **Important**: place the data on the memory of the CPU where the thread runs

First-touch policy

- The thread touching (not allocating) the memory first decides which part of the memory it gets placed in.
- One thread allocates the memory, and then
  - one thread initializes the memory (“plain”)
  - every thread initializes its part of the memory (“NUMA”)



vector_multiply_numa.cpp

vector_multiply.cpp