Parallel n-body Problem in C#

SICSA MultiCore Challenge 2011

Hans-Wolfgang Loidl

<hwloidl@macs.hw.ac.uk>

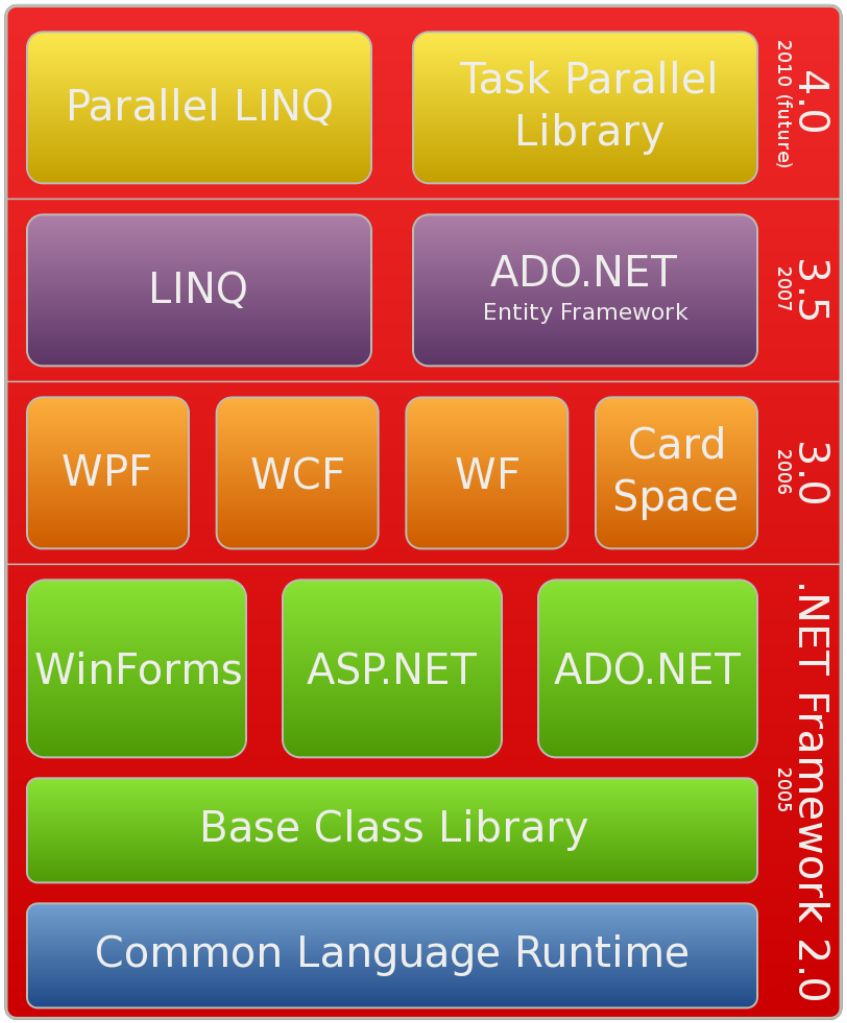
**School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh**

May, 2011

Why C#

- The *Parallel Pattern* approach for C# advocates a high-level parallel programming model.
- In essence, these are *skeletons* in disguise.
- From .Net 4.0 onwards this is supported through the Task Parallel Library (TPL).
- This acknowledges that more user-friendly approaches to parallel programming are desirable in the age of desktop parallelism on multi-cores.
- Based on the recent book: "*Parallel Programming with Microsoft .NET — Design Patterns for Decomposition and Coordination on Multicore Architectures*", by C. Campbell, R. Johnson, A. Miller, S. Toub. Microsoft Press. August 2010.
  `http://msdn.microsoft.com/en-us/library/ff963553.aspx`

DotNet Structure



The .NET Framework Stack

Focus of the implementation

- Explore the claim of easy parallelism.
- Test the sequential efficiency of the Mono implementation of C# and .NET under Linux.
- Some parallel performance tuning, but not much.

Data Parallelism with C#'s Patterns

```
var options = new ParallelOptions()
  MaxDegreeOfParallelism = k ;
Parallel.For(m, n, options, i =>
{
  ...
});
```

Improving Granularity by Partitioning

```
int size = bodies.Length / k;  // make a partition large enough

Parallel.ForEach(
  Partitioner.Create(0, bodies.Length-1, size),
  (range) => {
    double dx, dy, dz, distance, mag;
    for (int i = range.Item1; i < range.Item2; i++) {
      for (int j=i+1; j < bodies.Length; j++) {
        ...
```

Improving Locality by Aggregation

```
Parallel.ForEach(Partitioner.Create((int)m, (int)n, size),
 () => 0, // initialisation
 // The loop body
 (range, loopState, partialResult) => {
   for (int i = range.Item1; i < range.Item2; i++)  {
       partialResult += euler(i);
   }
   return partialResult; },
 // The final step of each local context
 (localPartialSum) => {
   // Enforce serial access to single, shared result
   lock (lockObject) {
     sum += localPartialSum;
   }  });
```

A naive parallel version

```
Parallel.For(0, bodies.Length-1, options, i => {
    double dx, dy, dz, distance, mag;
     for (int j=i+1; j < bodies.Length; j++) {
        dx = bodies[i].x - bodies[j].x; ...
        distance = Math.Sqrt(dx*dx + dy*dy + dz*dz + 0.01);
        mag = dt / (distance * distance * distance);
        lock (bodies[i]) {
          bodies[i].vx -= dx * bodies[j].mass * mag; ...
        }
        lock (bodies[j]) {
          bodies[j].vx += dx * bodies[i].mass * mag; ...
        }
    }} );

  foreach (Body body in bodies) {
     body.x += dt * body.vx; ...
  }
```

Top-level n-body Code

```
Parallel.ForEach(// The values to be aggregated
  Partitioner.Create(0, bodies.Length-1, size),
  // options, specify degree of parallelism
  // The local initial partial result
  () => { return InitDelta(bodies.Length); },
  // The loop body; delta is a local accumulator
  (range, loopState, delta) => {
      double dx, dy, dz, distance, mag;
      for (int i = range.Item1; i < range.Item2; i++) {
        for (int j=i+1; j < bodies.Length; j++) {
          dx = bodies[i].x - bodies[j].x; ...
          distance = Math.Sqrt(dx*dx + dy*dy + dz*dz + 0.01);
          mag = dt / (distance * distance * distance);
          bodies[i].vx -= dx * bodies[j].mass * mag; ...
          delta[j].vx += dx * bodies[i].mass * mag; ...
        }};
      return delta ;},
```

Top-level n-body Code

```
// The final step of each local context
(delta) =>   {
    // Enforce serial access to single, shared result
    for (int j=0; j < bodies.Length; j++) {
      lock (bodies[j]) {
         bodies[j].vx += delta[j].vx; bodies[j].x += dt * bodie
         bodies[j].vy += delta[j].vy; bodies[j].y += dt * bodie
         bodies[j].vz += delta[j].vz; bodies[j].z += dt * bodie
      }
    }
  });
```

Discussion

- Uses thread-local variable `delta` to accumulate changes
- Parallelism is unbalanced, since at position i, $n - i$ elements have to be processed
- The aggregation phase updates both velocities and positions

Hardware and Software Setup

Hardware:

- Eight-core Intel Xeon E5410,
- 2.33GHz,
- 8GB RAM,
- 6MB L2 cache

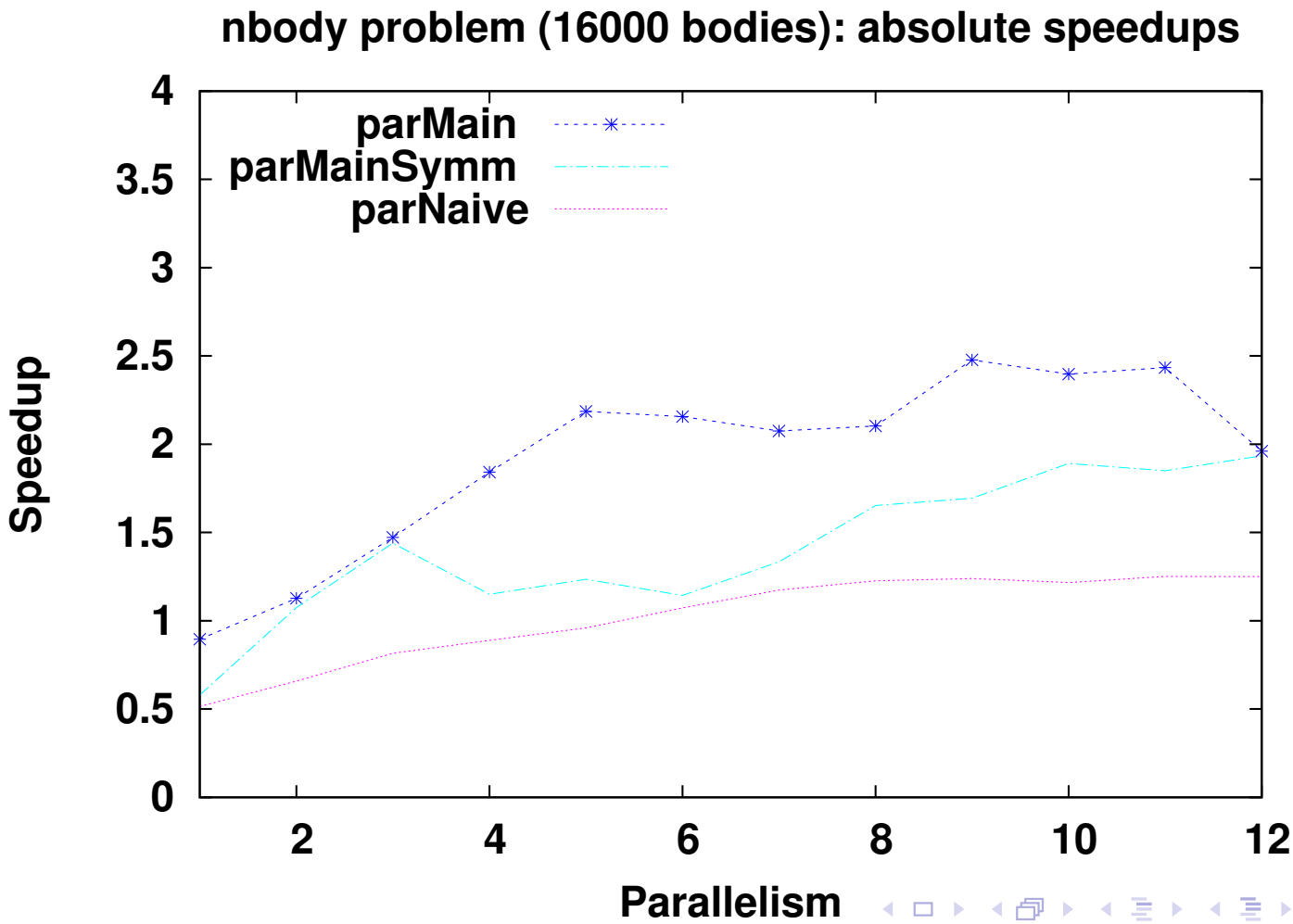
Software:

- CentOS 5.5
- Mono C# & JIT compiler version 2.10.2.0
- Mono RTE & JIT compiler (to amd64) version 2.8.0.0

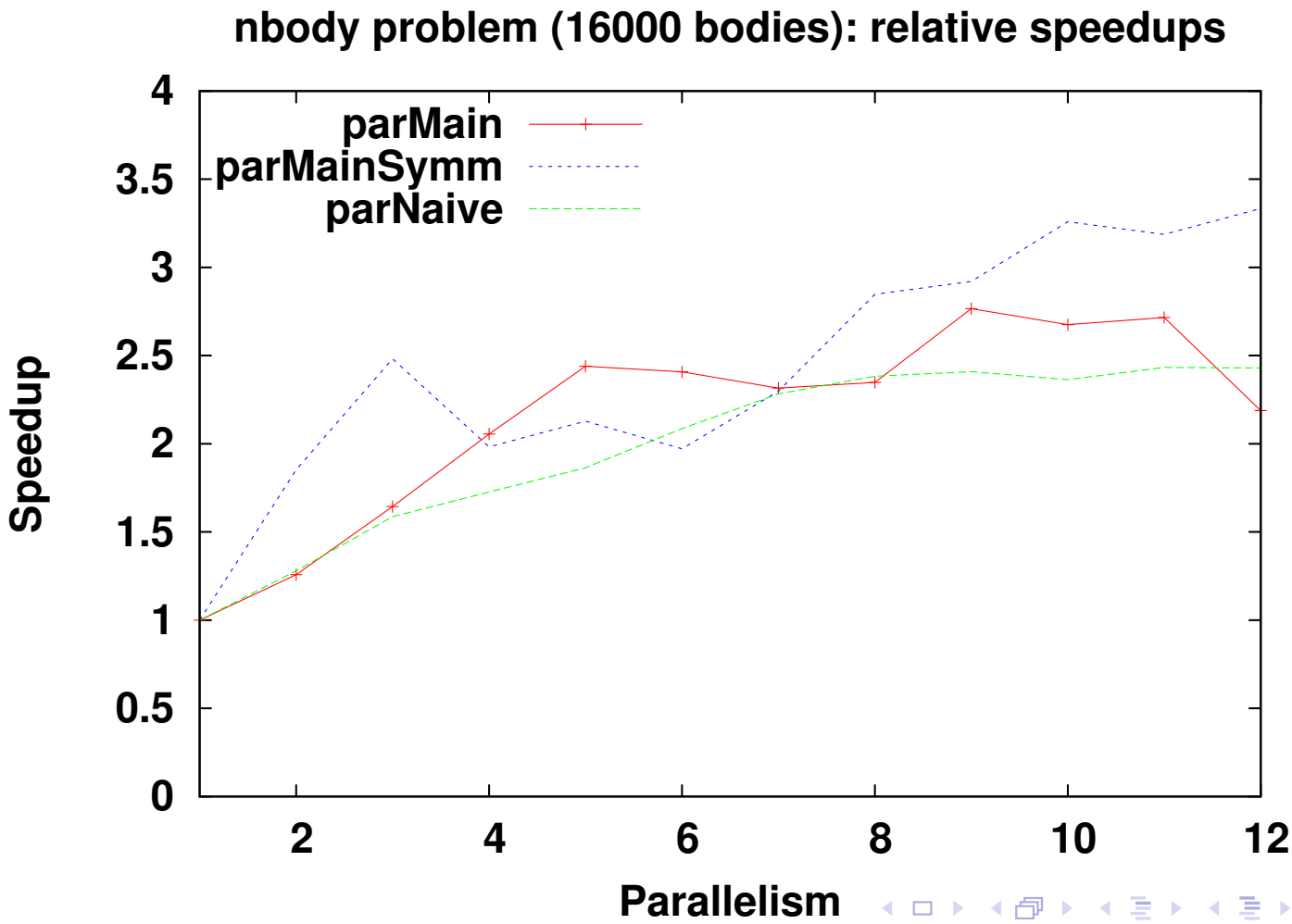
Sequential runtime:

- 1024 bodies: 0.062s
- 8000 bodies: 2.4s
- 16000 bodies: 10.12s

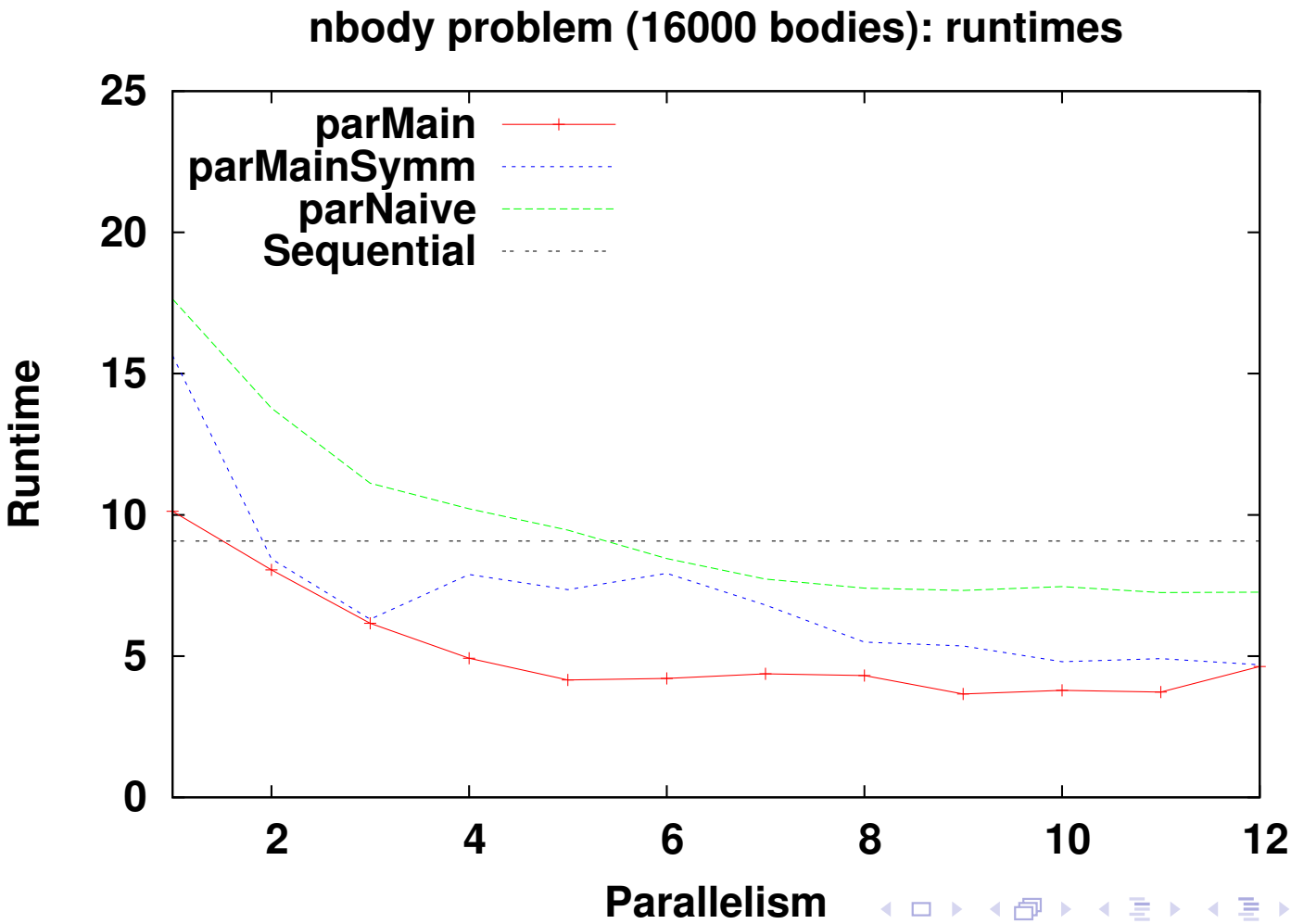
Performance Results: Speedups



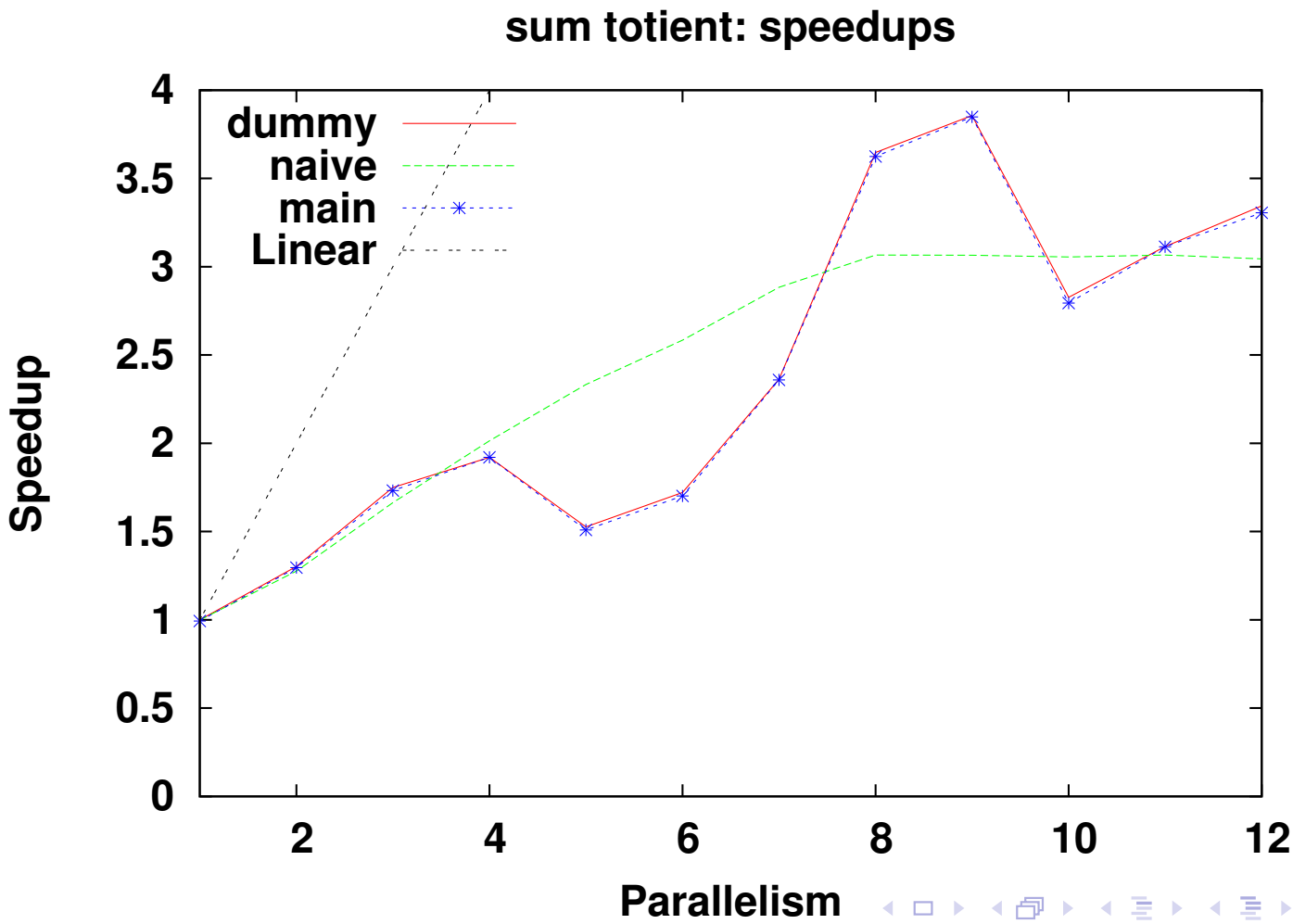
Performance Results: Relative Speedups



Performance Results: Runtimes



Performance Results: Runtimes



Conclusions

- Parallel patterns make heavy use of *delegates* in C# to realise skeletons, i.e. higher-order functions with parallel execution.
- Variants of a parallel for loop allow parallel performance tuning, increasing granularity and data locality.
- Tool support is fairly poor.
- Results are very sensitive to the version of Mono.
- Despite some parallel performance tuning absolute speedups are unimpressive: up to 2.5 on 8 cores
- Speedups on an embarrassingly parallel program with the same structure show speedups of up 4 on 8 cores.

An Example of Parallel Aggregates

```
var options = new ParallelOptions() {
                    MaxDegreeOfParallelism = k};
Parallel.ForEach(seq /* sequence */, options,
                    () => 0,  // The local initial partial result
                    // The loop body
                    (x, loopState, partialResult) =>  {
                       return Fib(x) + partialResult; },
                    // The final step of each local context
                    (localPartialSum) => {
                       // Protect access to shared result
                       lock (lockObject)
                         {
                           sum += localPartialSum;
                         }
                    });
```