Prof. Dr. D. Kranzlmüller, Dr. K. Fürlinger

Parallel Computing

WS 2017/18

Session 11: OpenMP: Tasking, Scalability laws

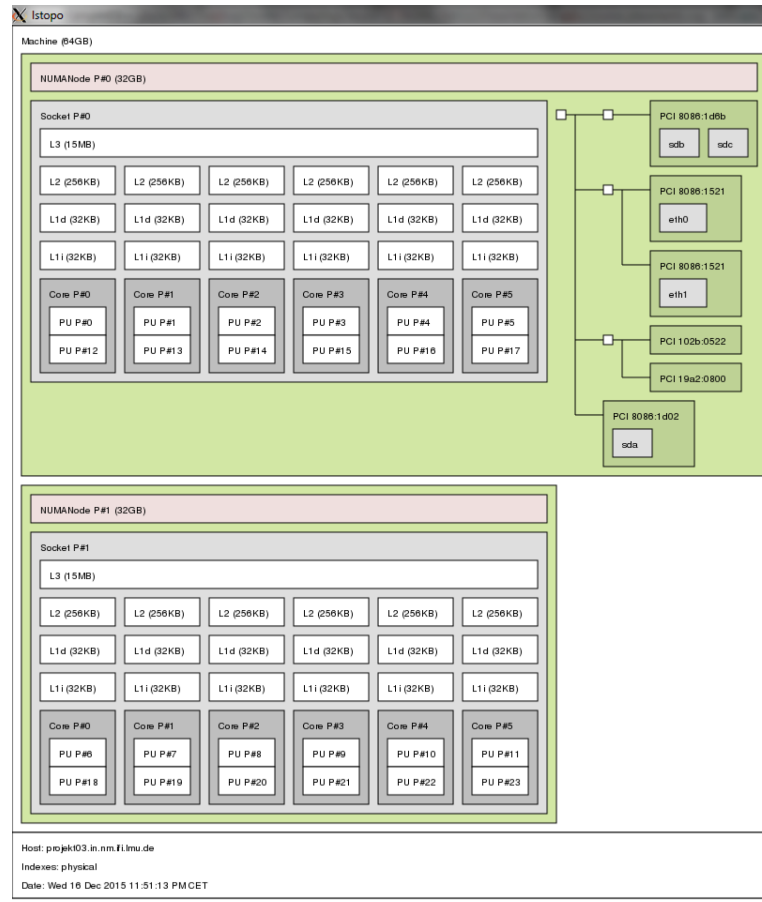
Tobias Fuchs, M.Sc.

tobias.fuchs@nm.ifi.lmu.de

NUMA Effects

NUMA Nodes

Let’s analyze this output from lstopo on a NUMA system …



Vector Triad


```
void vector_triad(double A[N],
                  double B[N],
                  double C[N],
                  double D[N]) {
    for (j=0; j < REPEAT; j++) {
      #pragma omp parallel for private(i)
      for (i=0; i < N; i++) {
        A[i] = B[i] + C[i] * D[i];
      }
    }
}
```

Vector Triad is a common benchmark, it used floating point addition and multiplication as otherwise FLOPS peak performance could not be reached.

NUMA Effects


```
A = (double*)(malloc(sizeof(double) * v_size));
B = (double*)(malloc(sizeof(double) * v_size));
C = (double*)(malloc(sizeof(double) * v_size));
D = (double*)(malloc(sizeof(double) * v_size));
for (i = 0; i < v_size; i++) {
      A[i] = ...;
      B[i] = ...;
      C[i] = ...;
      D[i] = ...;
}
#pragma omp parallel for
for (i = 0; i < v_size; i++) {
      A[i] = B[i] + C[i] * D[i];
}
```

Why is this a problem on NUMA systems?

NUMA Effects


```
A = (double*)(malloc(sizeof(double) * v_size));
B = (double*)(malloc(sizeof(double) * v_size));
C = (double*)(malloc(sizeof(double) * v_size));
D = (double*)(malloc(sizeof(double) * v_size));
for (i = 0; i < v_size; i++) {
      A[i] = ...;
      B[i] = ...;
      C[i] = ...;
      D[i] = ...;
}
#pragma omp parallel for
for (i = 0; i < v_size; i++) {
      A[i] = B[i] + C[i] * D[i];
}
```

➔ The notorious *First Touch Phenomenon*.

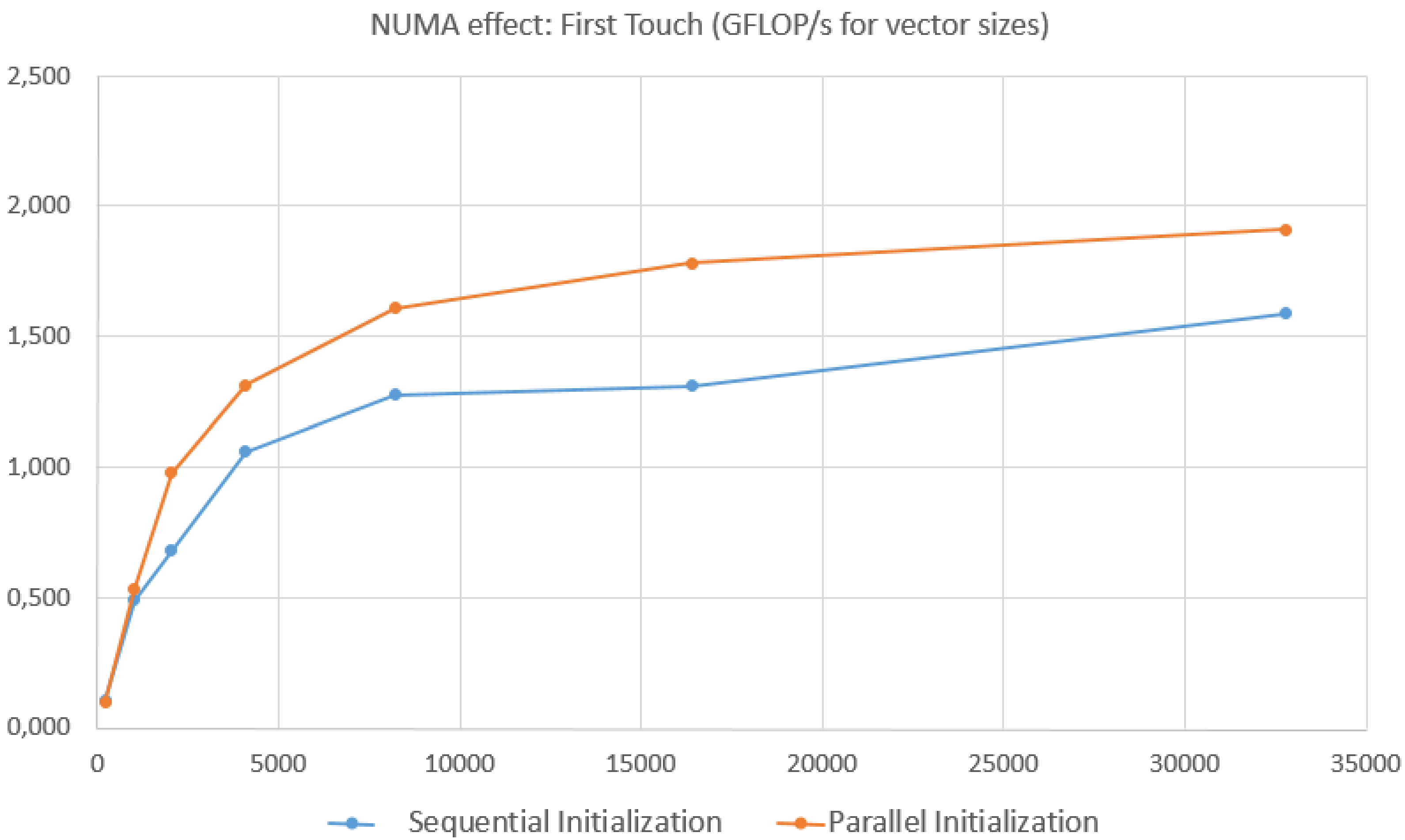
NUMA Effects


```
A = (double*)(malloc(sizeof(double) * v_size));
B = (double*)(malloc(sizeof(double) * v_size));
C = (double*)(malloc(sizeof(double) * v_size));
D = (double*)(malloc(sizeof(double) * v_size));
for (i = 0; i < v_size; i++) {
      A[i] = ...;  // first access by master, e.g. core 0
      B[i] = ...;  // ➔ all pages moved to local memory of core 0
      C[i] = ...;  // malloc does not matter, it does not actually
      D[i] = ...;  // place pages.
}
#pragma omp parallel for
for (i = 0; i < v_size; i++) {
      A[i] = B[i] + C[i] * D[i]; // data only local for core 0
}
```

NUMA Effects


```
A = (double*)(malloc(sizeof(double) * v_size));
B = (double*)(malloc(sizeof(double) * v_size));
C = (double*)(malloc(sizeof(double) * v_size));
D = (double*)(malloc(sizeof(double) * v_size));
#pragma omp parallel for
for (i = 0; i < v_size; i++) {
      A[i] = ...;  // first access on data by same core that will
      B[i] = ...;  // later perform calculation on the element.
      C[i] = ...;  // ➔ Data is placed where task is placed
      D[i] = ...;  // ➔ Good locality
}
#pragma omp parallel for
for (i = 0; i < v_size; i++) {
      A[i] = B[i] + C[i] * D[i];
}
```

NUMA Effects



Read this

Intel: *"Optimizing Applications for NUMA"*
https://software.intel.com/en-us/articles/optimizing-applications-for-numa

"What Every Programmer Should Know About Memory". Drepper, Ulrich. November 2007.

"Local and Remote Memory: Memory in a Linux/NUMA System". Lameter, Christoph. June 2006.

Scalability

Example:
OpenMP Matrix Multiplication

Matrix Product with OpenMP

```
void mmult_naive_par(double A[M][N], double B[N][K], double C[M][K]) {
    int    i, j, k;
    double sum;
    #pragma omp parallel for private(j,k,sum)
    for (i = 0; i < M; i++) {
        for (j = 0; j < K; j++) {
            sum = 0.0;
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

These are actual measurements submitted for a highly optimized implementation of mmult.

Spot the effects of

- Amdahl’s Law
- Gunther’s Law
- Gustafson’s Law

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Amdahl's Law

Performance increase (speedup) is **limited by sequential sections and degree of parallelism**.

Note that super-linear speedup is rare but possible!

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Gunther's Law

Performance may **worsen** if degree of parallelism is increased because contention rate increases.

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Gustafson's Law

Higher degree of parallelism can yield performance benefit **when problem size is increased**. For example:

N = 1000 → GLFOPS drop from 8 to 16 threads.

N = 2000 → GLFOPS saturated from 8 to 16 threads.

N = 4000 → GFLOPS increased from 8 to 16 threads.

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Gunther's Law + Gustafson's Law

Speedup saturation as predicted by Amdahl's and Gustafson's models **shifted** to higher degrees of parallelism **with increasing problem size**.

N = 1000 → Saturates with 4 threads

N = 2000 → Saturates with 8 threads

N = 4000 → Saturates with 16 threads

Vector Triad


```
void vector_blyad(double A[N],
                  double B[N],
                  double C[N],
                  double D[N]) {
    for (j=0; j < REPEAT; j++) {
      for (i=0; i < N; i++) {
        A[i] = B[i] + C[i] - D[i];
      }
    }
}
```

We fail to achieve peak performance because we have the wrong problem. Bad luck.

Vector Triad


```
void vector_triad(double A[N],
                  double B[N],
                  double C[N],
                  double D[N]) {
    for (j=0; j < REPEAT; j++) {
      for (i=0; i < N; i++) {
        A[i] = B[i] + C[i] * D[i];
      }
    }
}
```

Vector Triad is a common benchmark, uses floating point addition and -multiplication as otherwise FLOPS peak performance could not be reached ("fused multiply-add", I prefer the term "multiply-accumulate")

You and StackOverflow – a toxic relationship

Why you should ignore StackOverflow

The assignment on prefix sum implementations based on OpenMP tasking sure was hard.

As a precursor to possible solutions, this is what Google returned as the #1 match when searching for “openmp task prefix sum”:

https://stackoverflow.com/questions/18719257

WARNING: This is a **NEGATIVE EXAMPLE**

Also, do read the comments to the accepted answer (next slide). You could not make this up.

Why you should ignore StackOverflow

```
int recursiveSumBody(int *begin, int *end) {
  size_t length = end - begin; size_t mid = length / 2; int sum = 0;
  if (length < baseLength) {
    for (size_t ii = 1; ii < length; ii++) { begin[ii] += begin[ii - 1]; }
  } else {
#pragma omp task shared(sum)
    { sum = recursiveSumBody(begin, begin + mid); }
#pragma omp task
    { recursiveSumBody(begin + mid, end); }
#pragma omp taskwait
#pragma omp parallel for
    for (size_t ii = mid; ii < length; ii++) { begin[ii] += sum; }
  }
  return begin[length - 1];
}
void recursiveSum(int *begin, int *end) {
#pragma omp parallel
#pragma omp single
    { recursiveSumBody(begin, end); }
}
```

NEGATIVE EXAMPLE FOR DISCUSSION

Why you should ignore StackOverflow

stackoverflow Questions Developer Jobs Tags Users Log In

share improve this answer

edited Sep 10 '13 at 16:12

answered Sep 10 '13 at 15:22

4,670 ● 2 ● 23 ● 46

Thank you so much for posting a working example! I guess I was hoping for an answer that works with OpenMP 2.0 (so that it works in MSVC as well) but this is a good chance for me to use OpenMP tasks. I had to increase the `baseLength` to get the correct values for `n=10000`. Do you have any idea how efficient this method is? – Sep 10 '13 at 17:16

Well, I don't think that for this particular example tasks will be faster than the code you have written. What concerns me more is the fact you had to increase `baseLength` to get the correct value, which means there is a flaw somewhere. Anyhow I am not able to see a data race in the program. – Sep 10 '13 at 17:37

Well it appears that `baseLength` has to be equal to `n` to get the correct result. – Sep 10 '13 at 17:52

Why you should ignore StackOverflow

stackoverflow Questions Developer Jobs Tags Users Log In

share improve this answer

edited Sep 10 '13 at 16:12

answered Sep 10 '13 at 15:22

4,670 ● 2 ● 23 ● 46

> **The author is correct.**
> **It can indeed be safely stated that it is "not faster".**

Thank you so m… or an answer that works with OpenMP 2.0 (s… for me to use OpenMP tasks. I had to increase the `baseLength` to get the correct v… `10000`. Do you have any idea how efficient this method is? – Sep 10 '13 at 17:16

Well, I don't think that for this particular example tasks will be faster than the code you have written. What concerns me more is the fact you had to increase `baseLength` to get the correct value, which means there is a flaw somewhere. Anyhow I am not able to see a data race in the program. – Sep 10 '13 at 17:37

Well it appears that `baseLength` has to be equal to `n` to get the correct result. – Sep 10 '13 at 17:52

Performance tuning

Improve time to completion / throughput / latency / … achieved on a single computational unit

Evaluated system is arbitrary but its configuration does not change in benchmark scenarios

Hardware Scalability

Improve speedup achieved when adding computational components to the system

Scalability measures in general:

Extending computational capacities by a factor k ideally reduces time to completion by factor k (linear speedup).

Strong Scaling

Increase degree of hardware parallelism

Observe speedup for **fixed total problem size** (usually number of elements)

Weak Scaling

Increase degree of hardware parallelism

Observe speedup for **fixed problem size *per processor***

Scalability measures in general:

Extending computational capacities by a factor k ideally reduces time to completion by factor k (linear speedup).

Strong Scaling → expose Amdahl's Law

Increase degree of hardware parallelism

Observe speedup for **fixed total problem size** (usually number of elements)

Weak Scaling → expose Gustafson's Law

Increase degree of hardware parallelism

Observe speedup for **fixed problem size *per processor***

Scalability measures in general:

Extending computational capacities by a factor k ideally reduces time to completion by factor k (linear speedup).

Amdahl's Law

$$\boldsymbol{Speedup = \frac{1}{seq + (1 - seq)/p}}$$

Gustafson's Law

$$\boldsymbol{Speedup = \frac{(s + p \cdot N)}{s + p}}$$

Tobias Fuchs

tobias.fuchs@nm.ifi.lmu.de

www.mnm-team.org/~fuchst