Parallel Recursive Programs

Abhishek Somani, Debdeep Mukhopadhyay

Mentor Graphics, IIT Kharagpur

October 23, 2016

Overview

Outline

single directive

```
#pragma omp parallel
{
    ...
    #pragma omp single
    {
        //Executed by a single thread
        //Implicit barrier for other threads
    }
    ...
}
```

task directive

```
#pragma omp parallel
{
    ...
    #pragma omp single
    {
        ...
        #pragam omp task
        {
            ...
        }
        ...
        #pragma omp task
        {
            ...
        }
        ...
    }
    ...
}
```

Parallel Fibonacci

```
int fib(int n)
{
    if(n == 0 || n == 1)
        return n;
    int result, F_1, F_2;
#pragma omp parallel
    {
#pragma omp single
        {
#pragma omp task shared(F_1)
            F_1 = fib(n-1);
#pragma omp task shared(F_2)
            F_2 = fib(n-2);
#pragma omp taskwait
            res = F_1 + F_2;
        }
    }
    return res;
}
```

Outline

Outline

Comparison based sorting

- Theoretical lower bound : $O(nlog(n))$
- Recursive Doubling Formulation
- Two strategies for combining results at every level of recursion :
  - Ordered partition before sorting subarrays : quick sort
  - Merging results of sorted subarrays : merge sort

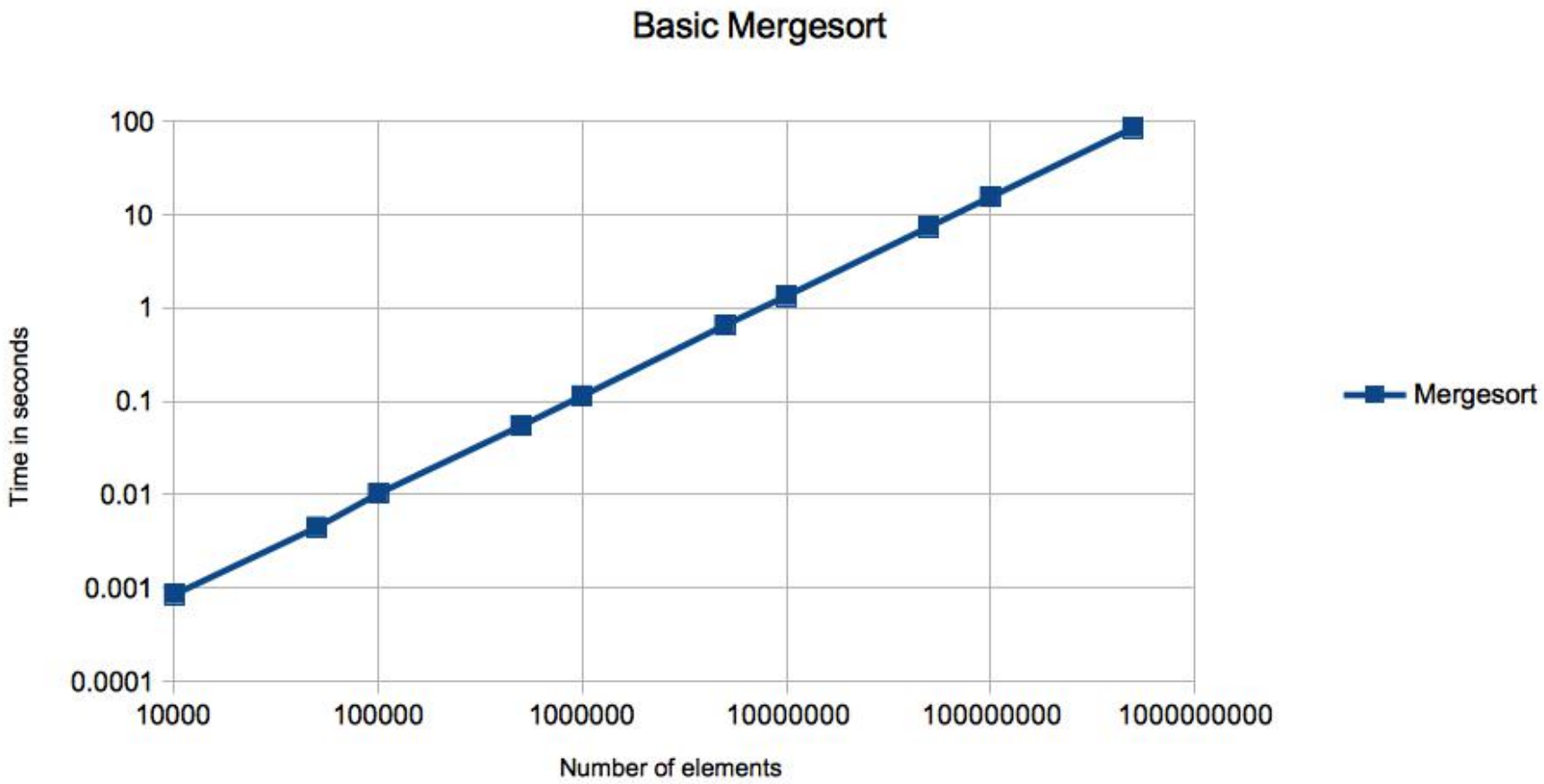
Merge Sort

```
void mergesort::sort1(int * a, int * b, const int lo, const int
    hi)
{
    if(hi - lo <= 1)
        return;
    const int mid = (hi + lo)/2;
    sort1(a, b, lo, mid);
    sort1(a, b, mid, hi);
    merge1(a, b, lo, mid, hi);
}
```

Merging in Merge Sort

```
void merge1(int * a, int * b, const int lo,
        const int mid, const int hi)
{
    memcpy(&b[lo], &a[lo], (hi - lo)*sizeof(int));
    int i = lo;
    int j = mid;
    for(int k = lo; k < hi; ++k)
    {
        if(i >= mid)
            a[k] = b[j++];
        else if(j >= hi)
            a[k] = b[i++];
        else if(b[j] < b[i])
            a[k] = b[j++];
        else
            a[k] = b[i++];
    }
}
```

Merge Sort Performance



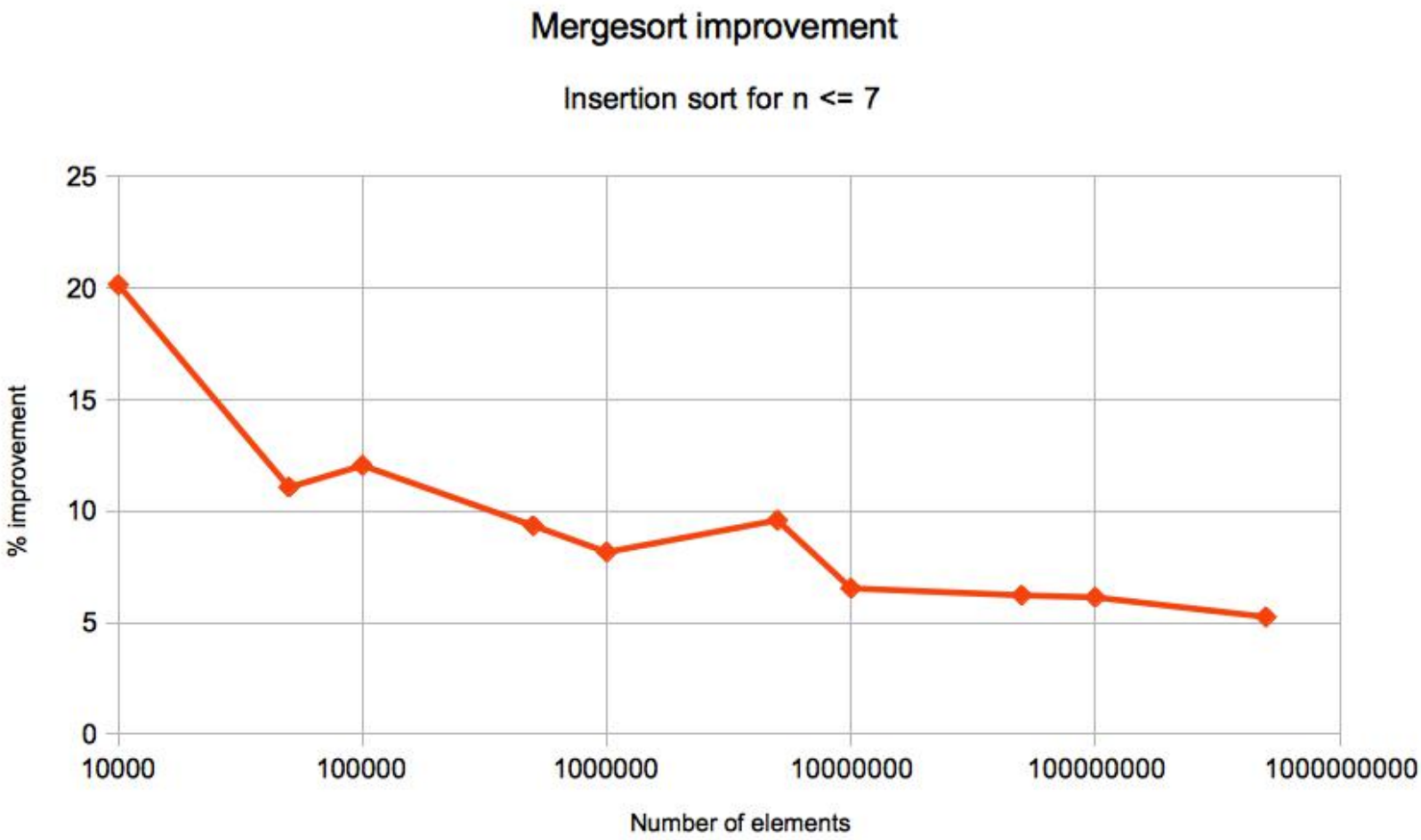
Improved Merge Sort

```
void mergesort::sort3(int * a, int * b, const int lo, const int
    hi)
{
    const int n = hi - lo;
    if(n <= 1)
        return;
    if(n <= 7)
    {
        insertionsort::sort(a, lo, hi);
        return;
    }
    const int mid = lo + n/2;
    sort3(a, b, lo, mid);
    sort3(a, b, mid, hi);
    merge1(a, b, lo, mid, hi);
    return;
}
```

Insertion Sort

```
void insertionsort::sort(int * a, const int lo, const int hi)
{
    for(int i = lo; i < hi; ++i)
    {
        for(int j = i; j > lo; --j)
        {
            if(a[j] < a[j-1])
                swap(a, j-1, j);
            else break;
        }
    }
}
```

Improved Merge Sort Performance



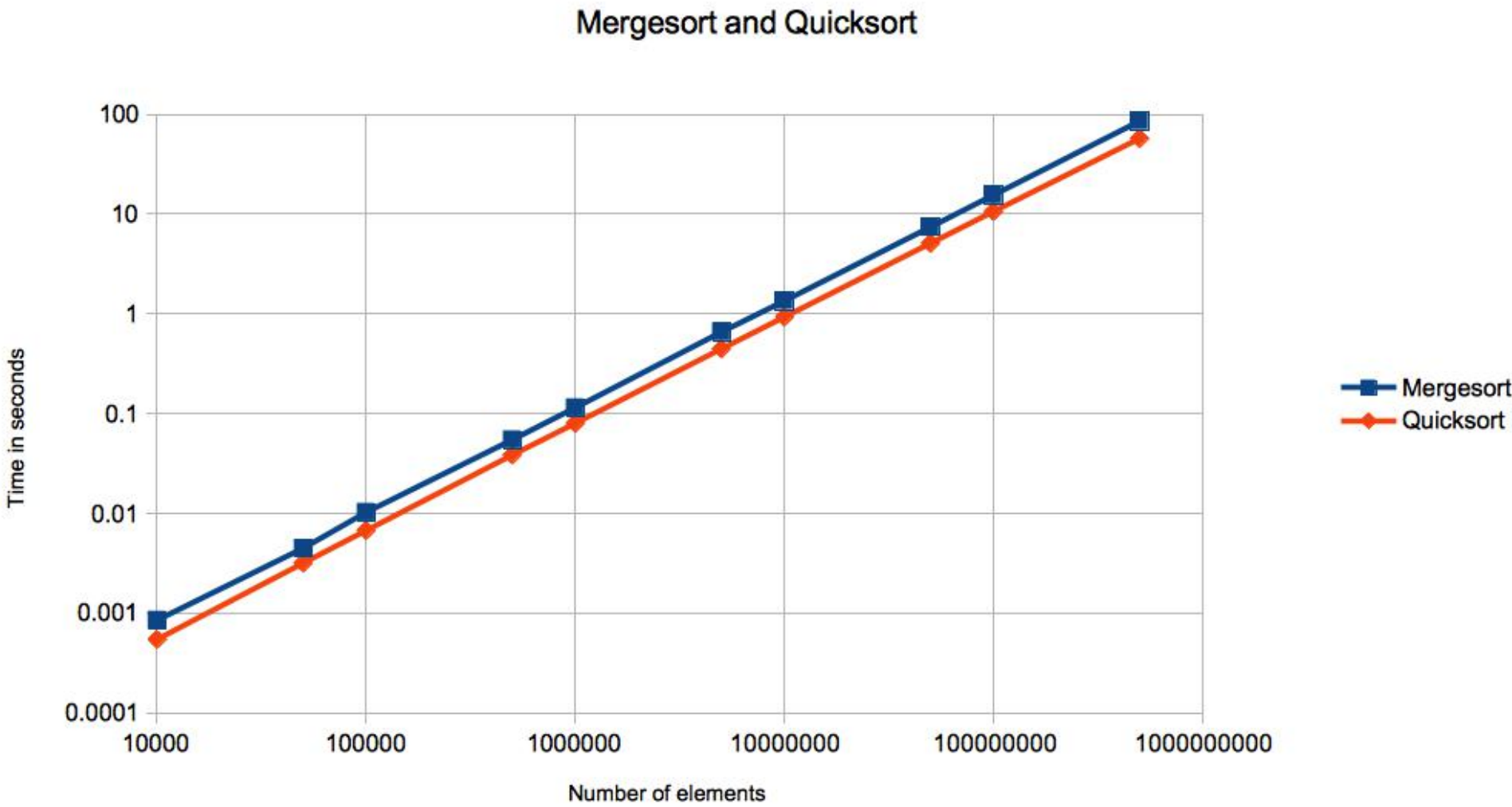
Quick Sort

```
void quicksort::sort(int * a, const int lo, const int hi)
{
    const int length = hi - lo;
    if(length <= 1)
        return;
    if(length <= 10)
    {
        insertionsort::sort(a, lo, hi);
        return;
    }
    const int divider = partition(a, lo, hi);
    sort(a, lo, divider);
    sort(a, divider+1, hi);
}
```

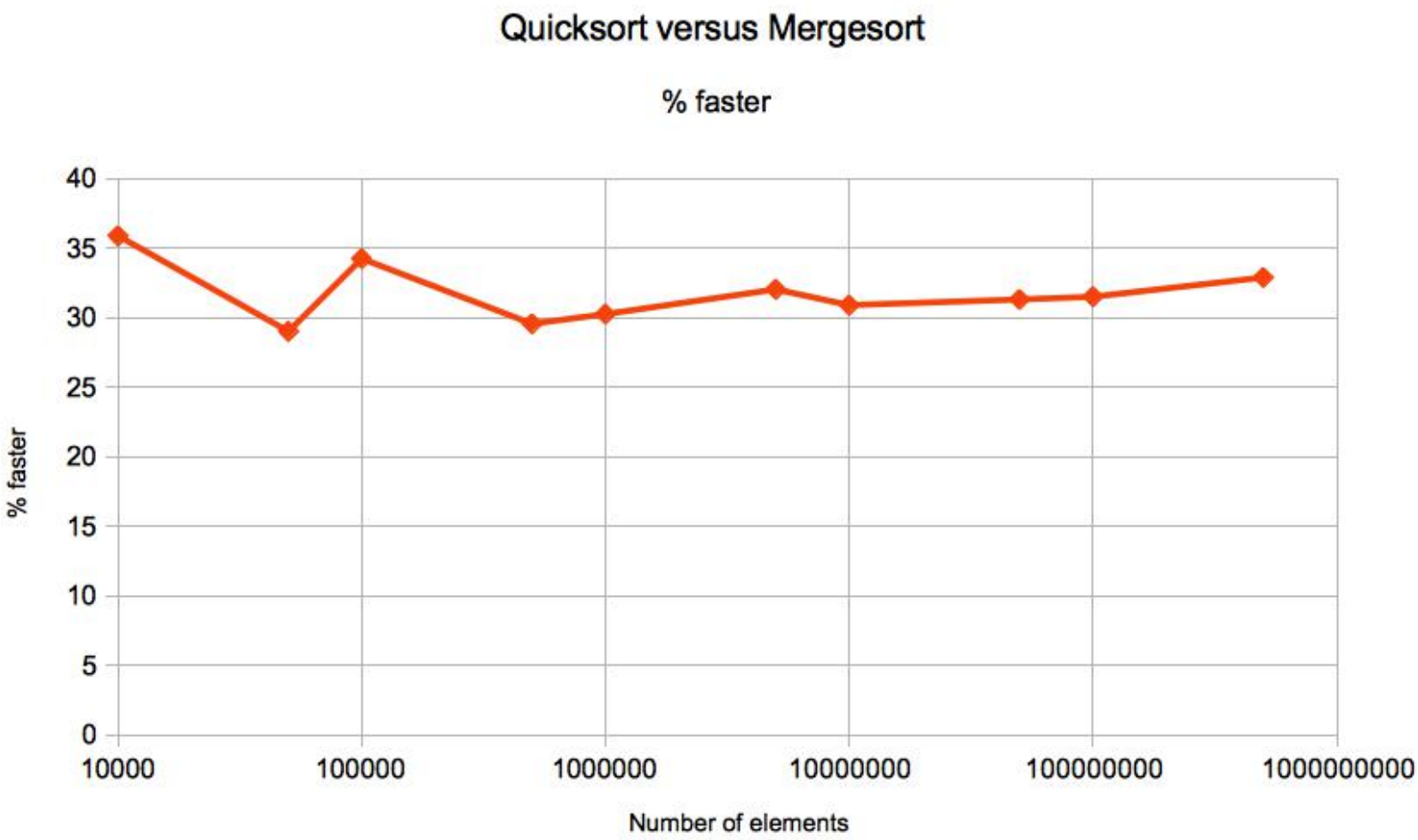
Partitioning in Quick Sort

```
int quicksort::partition(int * a, const int lo, const int hi)
{
    const int piv = quicksort::findPivot(a, lo, hi);
    swap(a, lo, piv);
    int i = lo, j = hi;
    while(true)
    {
        while(a[++i] < a[lo] && i < hi-1);
        while(a[--j] > a[lo] && j > lo);
        if(j <= i)
            break;
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
```

Quick Sort Performance



Quick Sort Performance



Outline

Parallel Quick Sort : Central Idea

```
    const int divider = partition(a, b, lo, hi, numThreads);
    int numThreadsLeft = floor(double(((divider - lo) * numThreads) / length) +
         0.5);
    if(numThreadsLeft == 0) ++numThreadsLeft;
    else if(numThreadsLeft == numThreads) --numThreadsLeft;
    const int numThreadsRight = numThreads - numThreadsLeft;
#pragma omp parallel
    {
#pragma omp single
        {
#pragma omp task
            {
                sort(a, b, lo, divider, numThreadsLeft, serialMethod);
            }
#pragma omp task
            {
                sort(a, b, divider+1, hi, numThreadsRight, serialMethod);
            }
        }
    }
```

Parallel Quick Sort : Termination

```
const int length = hi - lo;
if(length <= 1)
    return;
if(length <= 1000 || numThreads == 1)
{
    serial_sort(a, b, lo, hi, serialMethod);
    return;
}
```

Parallel Quick Sort : Analysis

- Number of elements : **n**
- Number of parallel threads : **p**
- Assume best pivot selection for this analysis
- Depth of recursion tree : $log(p)$
- Work for each leaf-level node : $\Theta(\frac{n}{p} log(\frac{n}{p}))$
- Work (partitioning) at any other level **l** : $\Theta(\frac{n}{2^l})$
- Overall complexity, $T_p = \Theta(\frac{n}{p} log(\frac{n}{p}) + \Theta(n) + \Theta(\frac{n}{p}))$
- Not optimal unless partitioning is done by a parallel algorithm

Parallel Partitioning

- Thread 1 chooses a pivot : $\Theta(1)$
- Each thread partitions a contiguous block of $\frac{n-1}{p}$ elements with the pivot : $\Theta(\frac{n}{p})$
- Globally rearranged array indices calculated - Can be done by prefix summation : $\Theta(log(p))$
- Each thread copies it's partitioned array to correct locations in a new array : $\Theta(\frac{n}{p})$
- Notice that unlike serial partitioning, parallel partitioning requires an auxiliary array
- Overall complexity of partitioning : $\Theta(\frac{n}{p}) + \Theta(log(p))$
- Overall complexity of parallel quicksort, $T_p = \Theta(\frac{n}{p}log(\frac{n}{p})) + \Theta(\frac{n}{p}log(p)) + \Theta(log^2(p))$
- Practically, $n >> p$, hence $T_p \approx \Theta(\frac{n}{p}log(n))$

Nested Threading

- The number of operating threads are different for different calls to the core functions
- Use nested threading in OpenMP; Set env variable **OMP_NESTED** to **TRUE**

```
omp_set_num_threads(numThreads);
omp_set_nested(1);
```

Parallel Partitioning : Initial setup

```
    const int elemsPerThread = (hi - lo)/numThreads;
    //Copy from a to b
#pragma omp parallel for num_threads(numThreads)
    for(int i = 0; i < numThreads; ++i)
    {
        const int start = lo + elemsPerThread * i;
        const int numElems = (i == numThreads-1) ? (hi -
            (numThreads-1)*elemsPerThread - lo) : elemsPerThread;
        memcpy(&b[start], &a[start], numElems * sizeof(int));
    }
    //Find pivot for b
    const int piv = quicksort::findPivot(b, lo, hi);
    //Make lo the pivot
    swap(b, lo, piv);
```

Parallel Partitioning : Partitioning of blocks

```
    //Find dividers for each thread
    std::vector<int> dividers(numThreads);
#pragma omp parallel for num_threads(numThreads)
    for(int i = 0; i < numThreads; ++i)
    {
        const int start = lo + 1 + elemsPerThread * i;
        const int stop = (i == numThreads-1) ? hi : (start +
            elemsPerThread);
        const int TID = omp_get_thread_num();
        dividers[i] = serialPartition(b, start, stop, b[lo]);
    }
```

Parallel Partitioning : Global indices computation

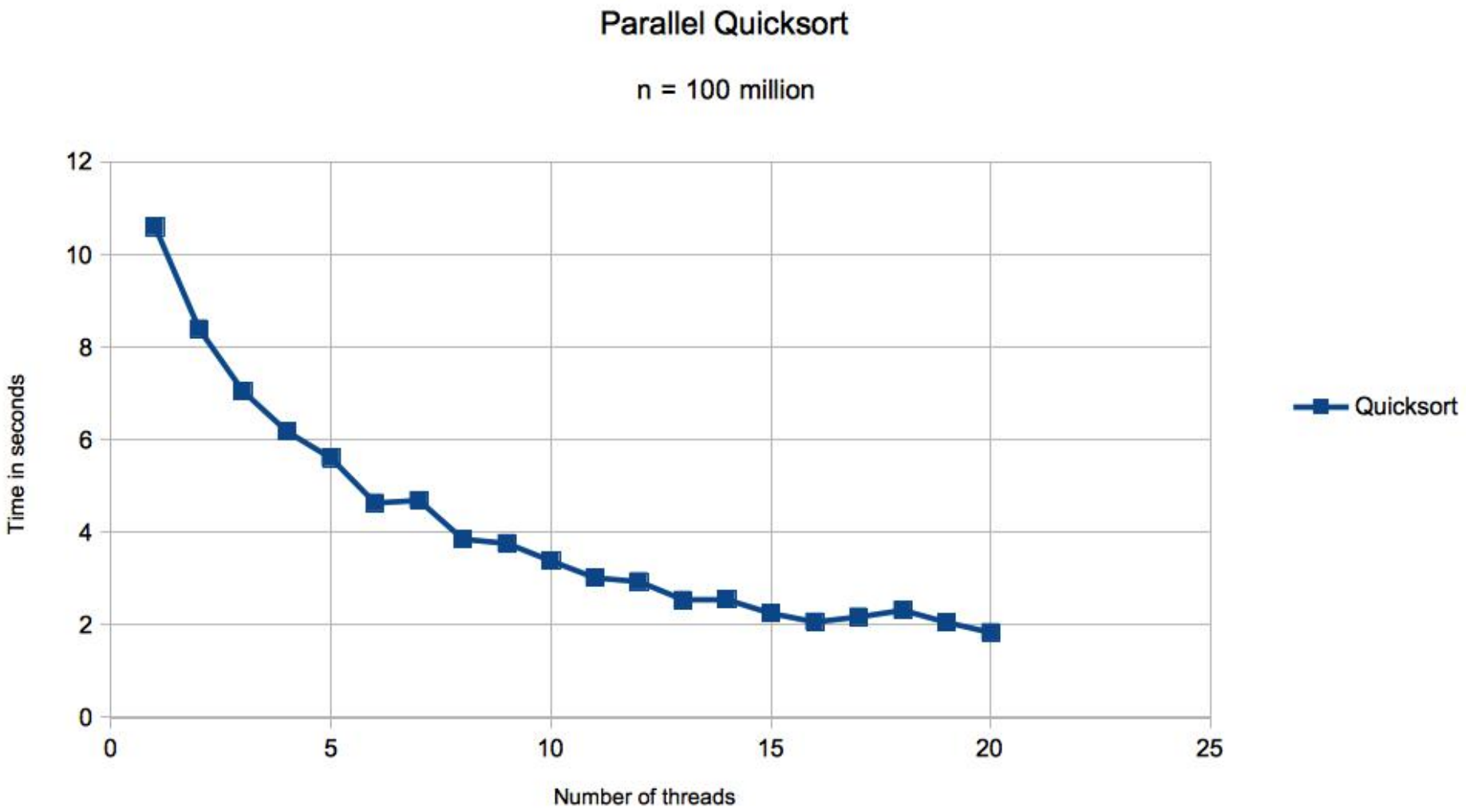
```
std::vector<int> dividerPrefixSum(numThreads);
dividerPrefixSum[0] = 0;
for(int i = 1; i < numThreads; ++i)
    dividerPrefixSum[i] = dividerPrefixSum[i-1] +
        dividers[i-1];
int globalDivider = dividerPrefixSum[numThreads-1] +
    dividers[numThreads-1];
```

- Notice that the implementation is $\Theta(p)$ and not $\Theta(log(p))$

Parallel Partitioning : Parallel Copying

```
    //Copy divided array b to a
#pragma omp parallel for num_threads(numThreads)
    for(int i = 0; i < numThreads; ++i)
    {
        const int elems = (i == numThreads-1) ? (hi - lo - 1 -
            i*elemsPerThread) : elemsPerThread;
        const int bStart = lo + 1 + i*elemsPerThread;
        const int bMid = bStart + dividers[i];
        const int bStop = bStart + elems;
        const int aLeftStart = lo + 1 + dividerPrefixSum[i];
        const int aRightStart = lo + 1 + globalDivider +
            i*elemsPerThread - dividerPrefixSum[i];
        memcpy(&a[aLeftStart], &b[bStart], (bMid - bStart) *
            sizeof(int));
        memcpy(&a[aRightStart], &b[bMid], (bStop - bMid) *
            sizeof(int));
    }
```

Quick Sort Scaling



Merge Sort

```
    const int mid = lo + length/2;
    const int numThreadsLeft = numThreads/2;
    const int numThreadsRight = numThreads - numThreadsLeft;
#pragma omp parallel
    {
#pragma omp single
        {
#pragma omp task
            {
                sort(a, b, lo, mid, numThreadsLeft, serialMethod);
            }
#pragma omp task
            {
                sort(a, b, mid, hi, numThreadsRight, serialMethod);
            }
        }
    }
```

Parallel Merging

- Parallel copying of array contents to an auxiliary array : $\Theta(\frac{n}{p})$
- Let **X** : first half of the array and **Y** : second half
- Divide **Y** into **p** equal sections; The last elements of each section is a key
- Do parallel search for these **p** keys in **X** : $\Theta(log(n))$
- Globally rearranged array indices calculated - Can be done by prefix summation : $\Theta(log(p))$
- Each thread merges corresponding sections in **X** and **Y** into correct locations in the original array : $\Theta(\frac{n}{p})$
- Overall complexity of merging : $\Theta(\frac{n}{p}) + \Theta(log(n)) + \Theta(log(p))$
- Overall complexity of parallel mergesort, $T_p = \Theta(\frac{n}{p}log(\frac{n}{p})) + \Theta(\frac{n}{p}log(p)) + \Theta(log(n)log(p)) + \Theta(log^2(p))$
- Practically, $n >> p$, hence $T_p \approx \Theta(\frac{n}{p}log(n))$

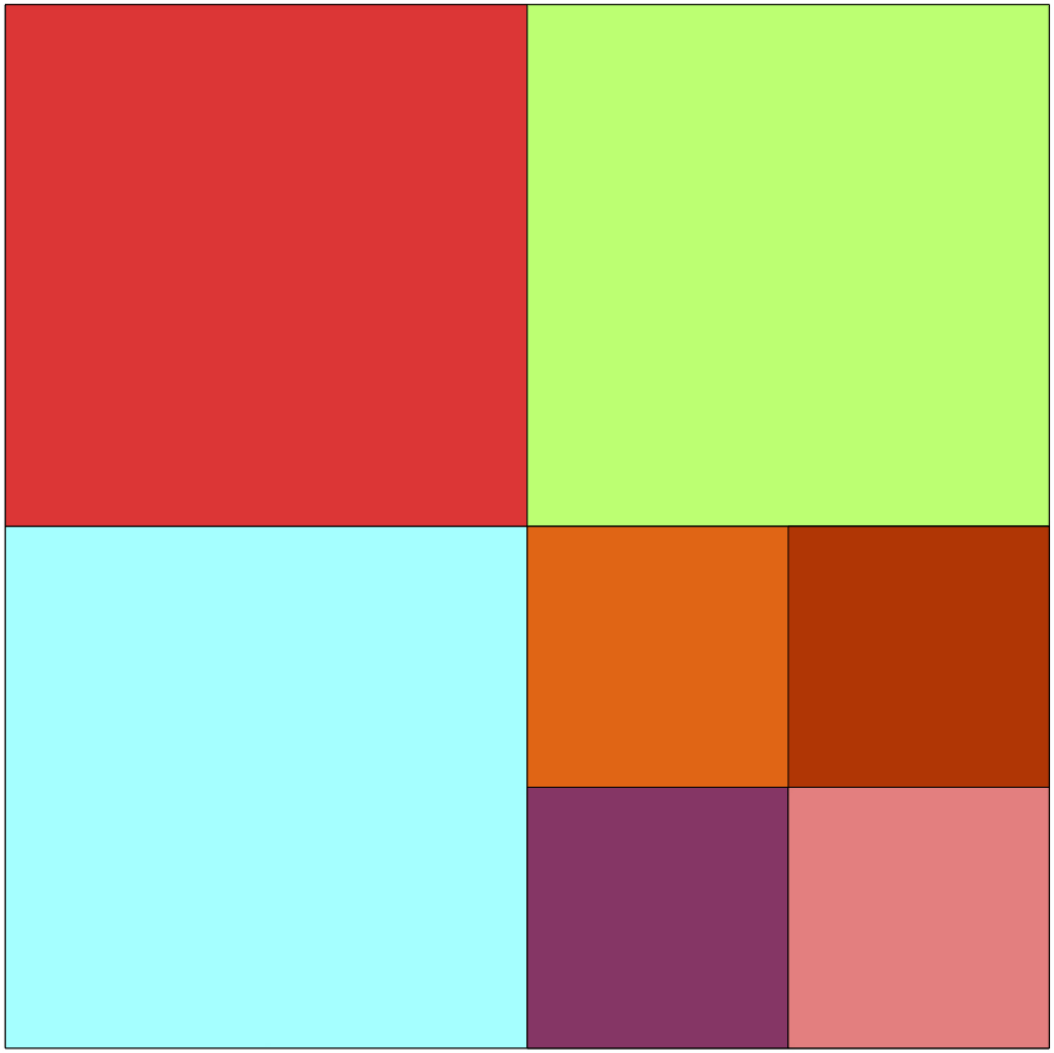
Outline

A different formulation

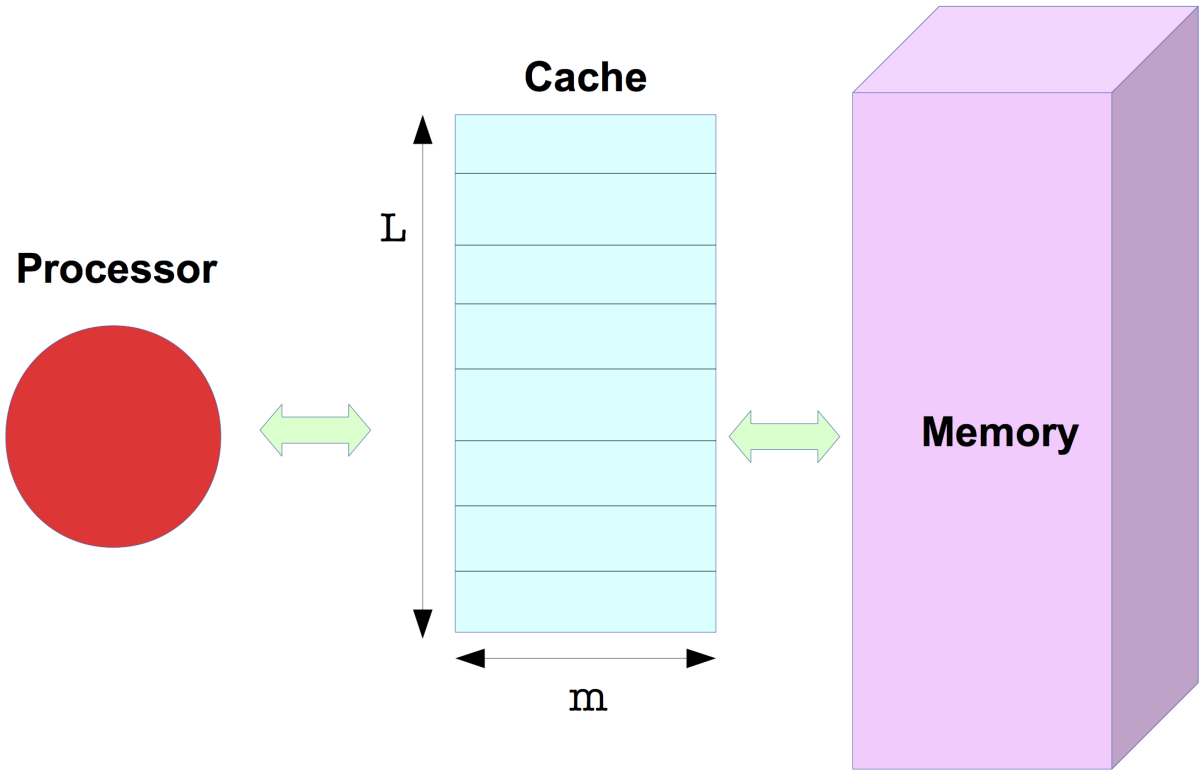
$$
\begin{aligned}
C &= A \times B \\
\Rightarrow \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\
\Rightarrow C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{aligned}
$$

- Solve for C_{11}, C_{12}, C_{21}, C_{22} recursively
- Recursion formula : $W(n) = 8W(\frac{n}{2}) + \Theta(1)$
- $W(n) = \Theta(n^3)$

This method is Cache Oblivious



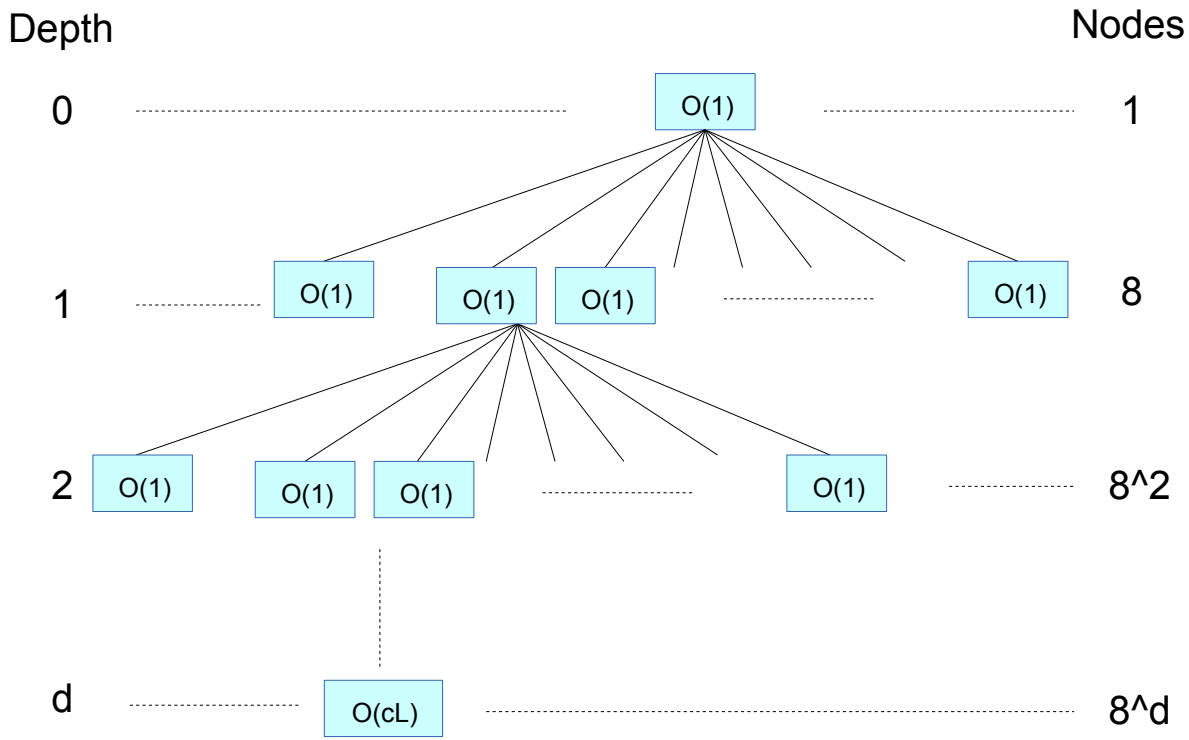
Recall : Cache model for analysis



- **L** lines of capacity **m** double precision numbers each
- Tall Cache assumption : $L > m$
- Replacement Policy : Least Recently Used
- No Hardware Prefetching

Cache Miss Analysis

$$Q(n) = \begin{cases} \Theta(\frac{n^2}{m}) & \text{for } n^2 < cmL \text{ where } 0 < c \leq 1, \\ 8Q(\frac{n}{2}) + \Theta(1) & \text{otherwise.} \end{cases}$$



Cache Miss Analysis...

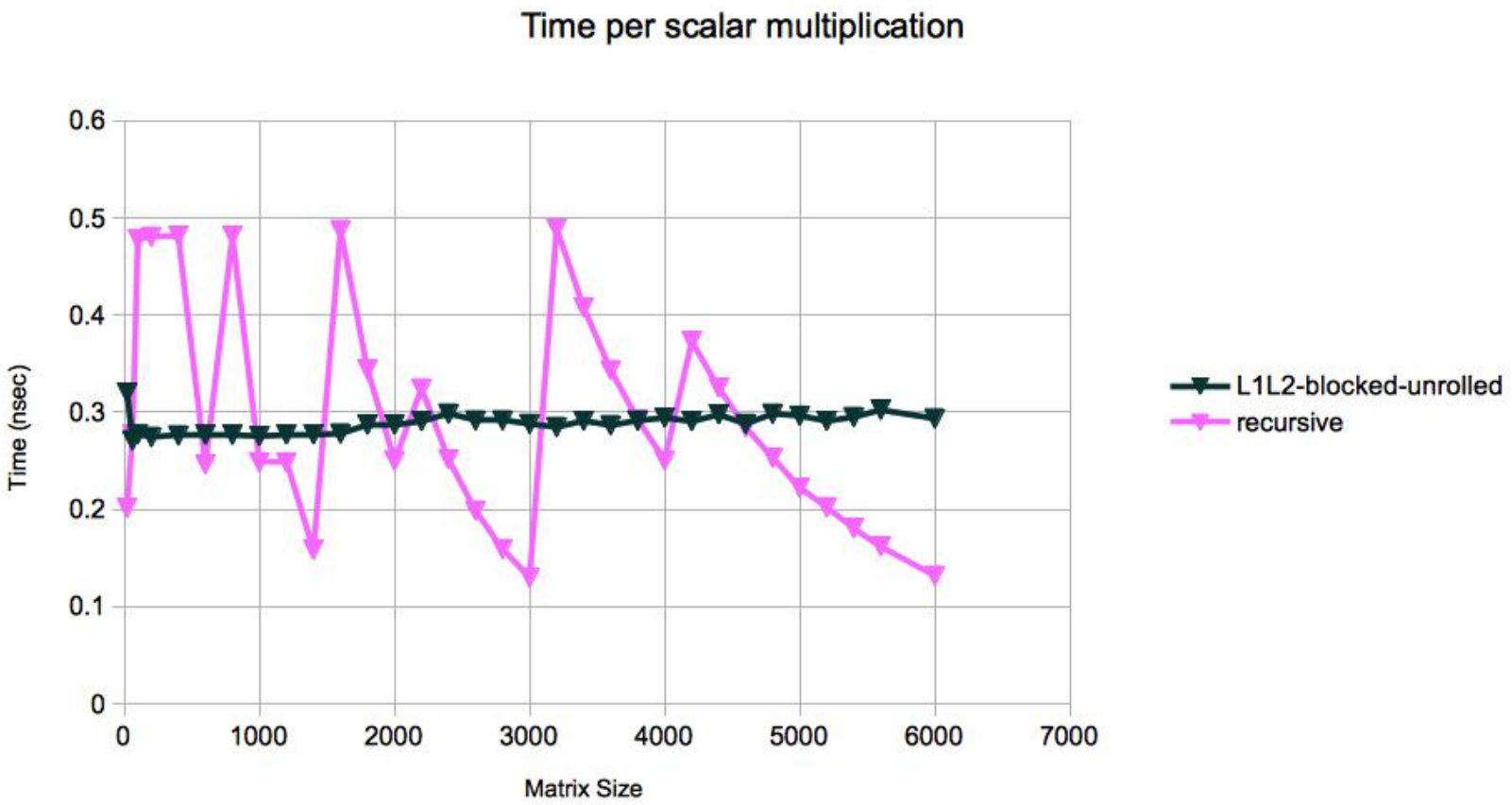
$$Q(n) = \begin{cases} \Theta(\frac{n^2}{m}) & \text{for } n^2 < cmL \text{ where } 0 < c \leq 1, \\ 8Q(\frac{n}{2}) + \Theta(1) & \text{otherwise.} \end{cases}$$

- Let n_d be the value of n at depth d. Then $n_d = \sqrt{cmL}$
- $n_0 = n$ and $n_i = \frac{n_{i-1}}{2} = \frac{n}{2^i}$. Therefore $n_d = \frac{n}{2^d}$ and $2^d = \frac{n}{\sqrt{cmL}}$
- $d = lg(\frac{n}{\sqrt{cmL}})$ and number of leaves $l = 8^d = \frac{n^3}{(cmL)^{\frac{3}{2}}}$
- Number of cache misses at leaf level :
  $l * \Theta(cL) = \Theta(\frac{n^3}{m\sqrt{cmL}}) = \Theta(\frac{n^3}{m\sqrt{mL}})$
- Total number of cache misses : $\Theta(\frac{n^3}{m\sqrt{mL}}) = \Theta(\frac{n^3}{m\sqrt{\text{Cache Size}}})$
- Recall that the total number of cache misses for tiled multiplication was found to be $\Theta(\frac{n^3}{m\sqrt{\text{Cache Size}}})$

Recursive Multiply

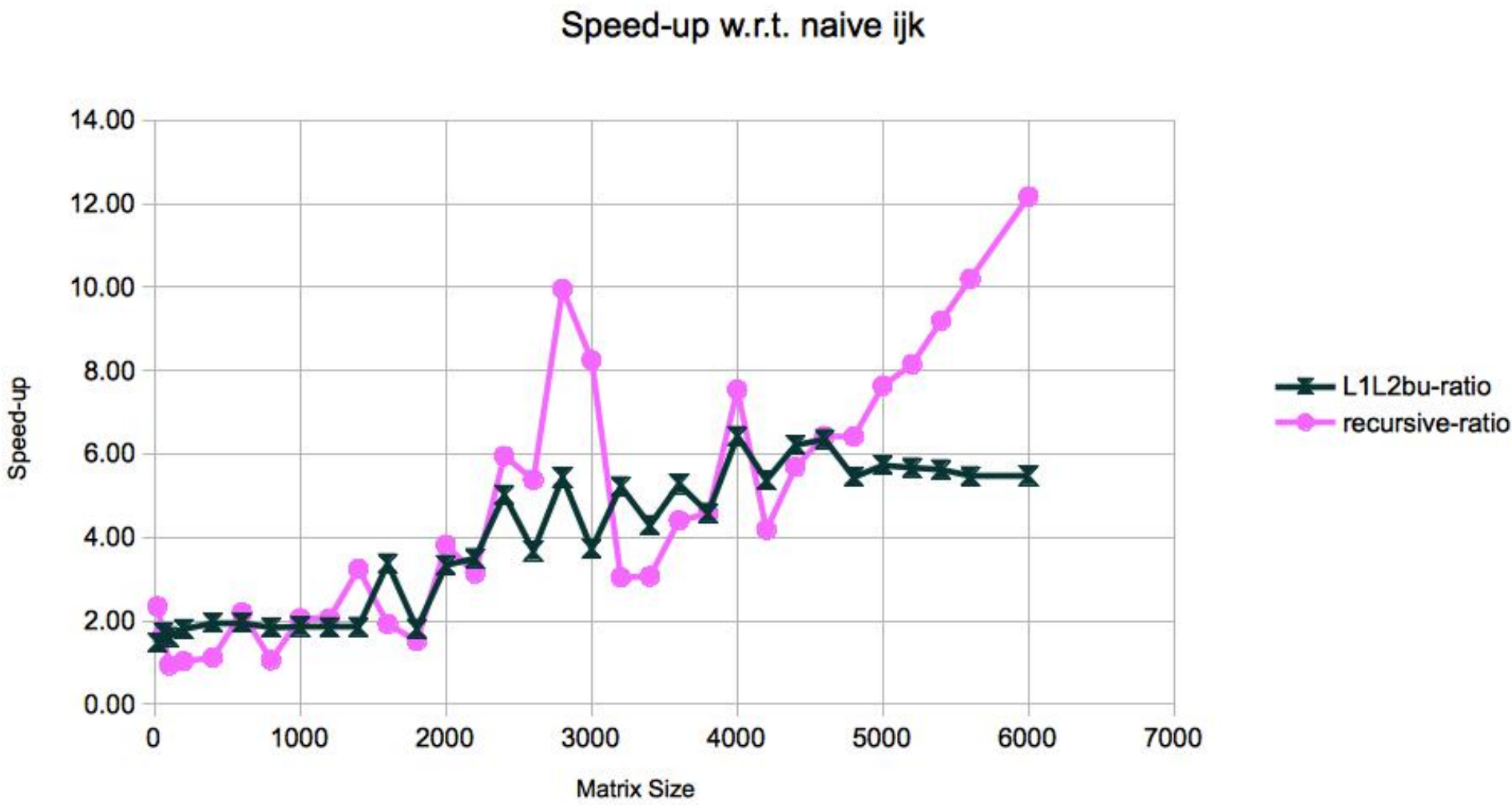
```
void multiply(const int n, const int m, double * A, double * B, double * C)
{
    //Base cases
    if(m == 2)
    {
        baseCase2(n, m, A, B, C);
        return;
    }
    else if(m == 1)
    {
        C[0] += A[0] * B[0];
        return;
    }
    //Recursive multiply
    const int offset12 = m/2;
    const int offset21 = n*m/2;
    const int offset22 = offset21 + offset12;
    multiply(n, m/2, A, B, C);
    multiply(n, m/2, A+offset12, B+offset21, C);
    multiply(n, m/2, A, B+offset12, C+offset12);
    multiply(n, m/2, A+offset12, B+offset22, C+offset12);
    multiply(n, m/2, A+offset21, B, C+offset21);
    multiply(n, m/2, A+offset22, B+offset21, C+offset21);
    multiply(n, m/2, A+offset21, B+offset12, C+offset22);
    multiply(n, m/2, A+offset22, B+offset22, C+offset22);
    return;
}
```

Recursive multiply serial performance

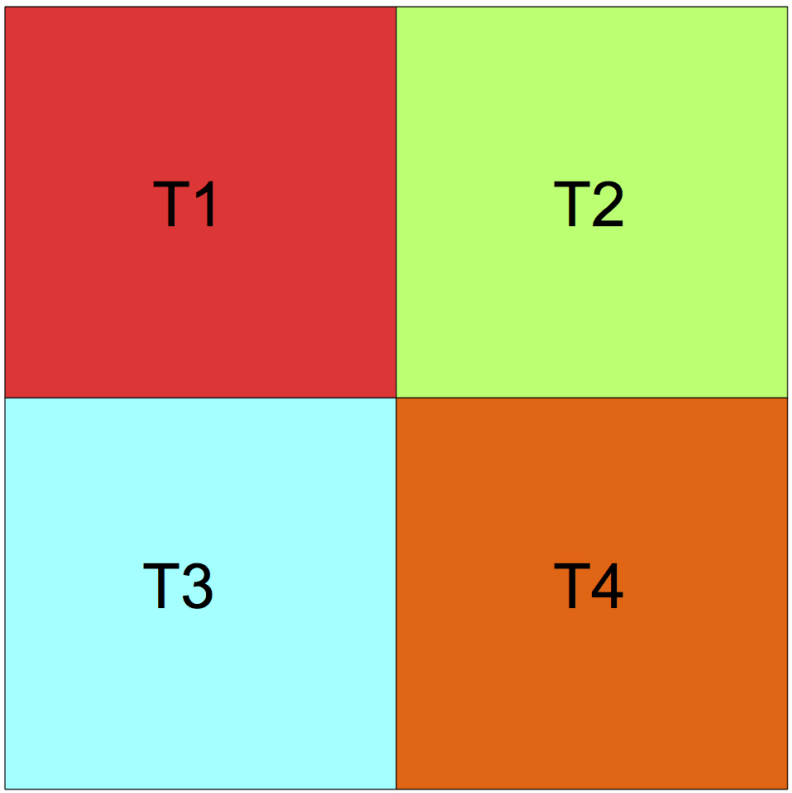


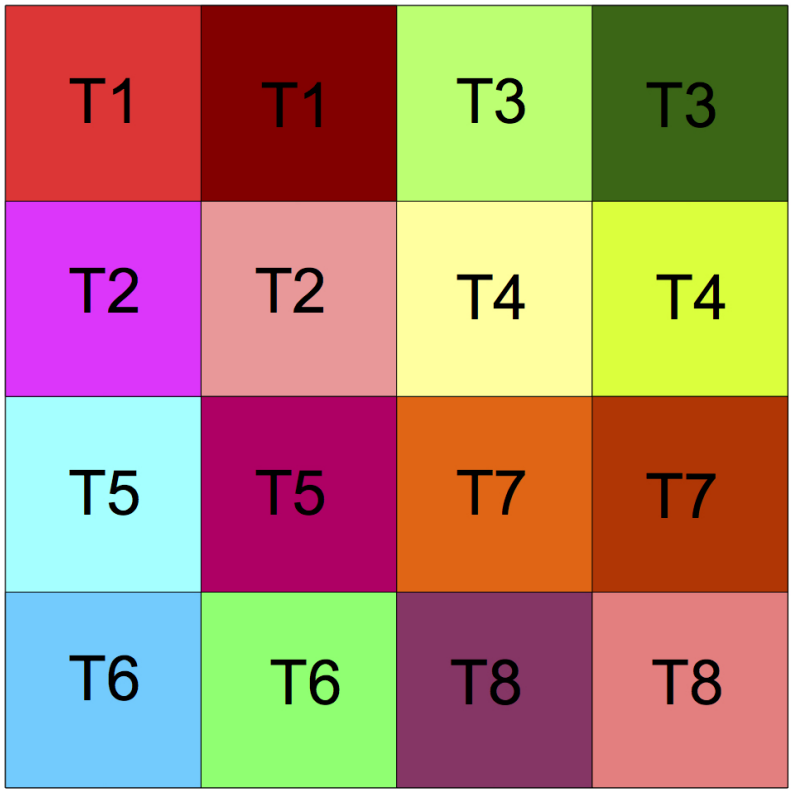
- Can be made more consistent by writing base case code for matrices of sizes > 2

Comparison with naive(first) implementation



Parallelization



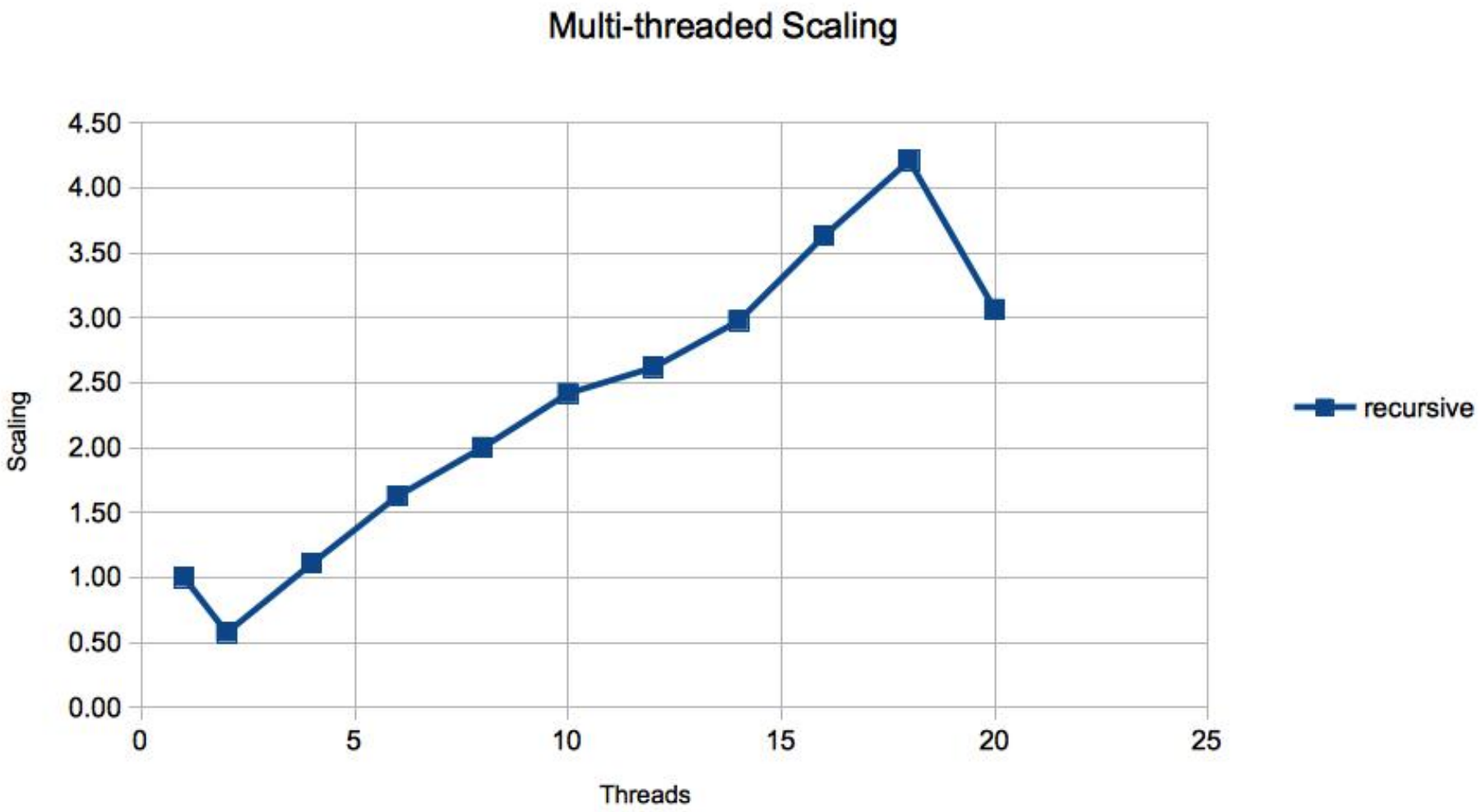


- How about 6, 10, 12, 14, 18 or 20 threads ?

task based recursive multiplication

```
    const int offset12 = m/2;
    const int offset21 = n*m/2;
    const int offset22 = offset21 + offset12;
#pragma omp task
    {
        multiply(n, m/2, A, B, C);
        multiply(n, m/2, A+offset12, B+offset21, C);
    }
#pragma omp task
    {
        multiply(n, m/2, A, B+offset12, C+offset12);
        multiply(n, m/2, A+offset12, B+offset22, C+offset12);
    }
#pragma omp task
    {
        multiply(n, m/2, A+offset21, B, C+offset21);
        multiply(n, m/2, A+offset22, B+offset21, C+offset21);
    }
#pragma omp task
    {
        multiply(n, m/2, A+offset21, B+offset12, C+offset22);
        multiply(n, m/2, A+offset22, B+offset22, C+offset22);
    }
```

Recursive Multiplication Scaling



Improved Parallel Recursive Multiplication

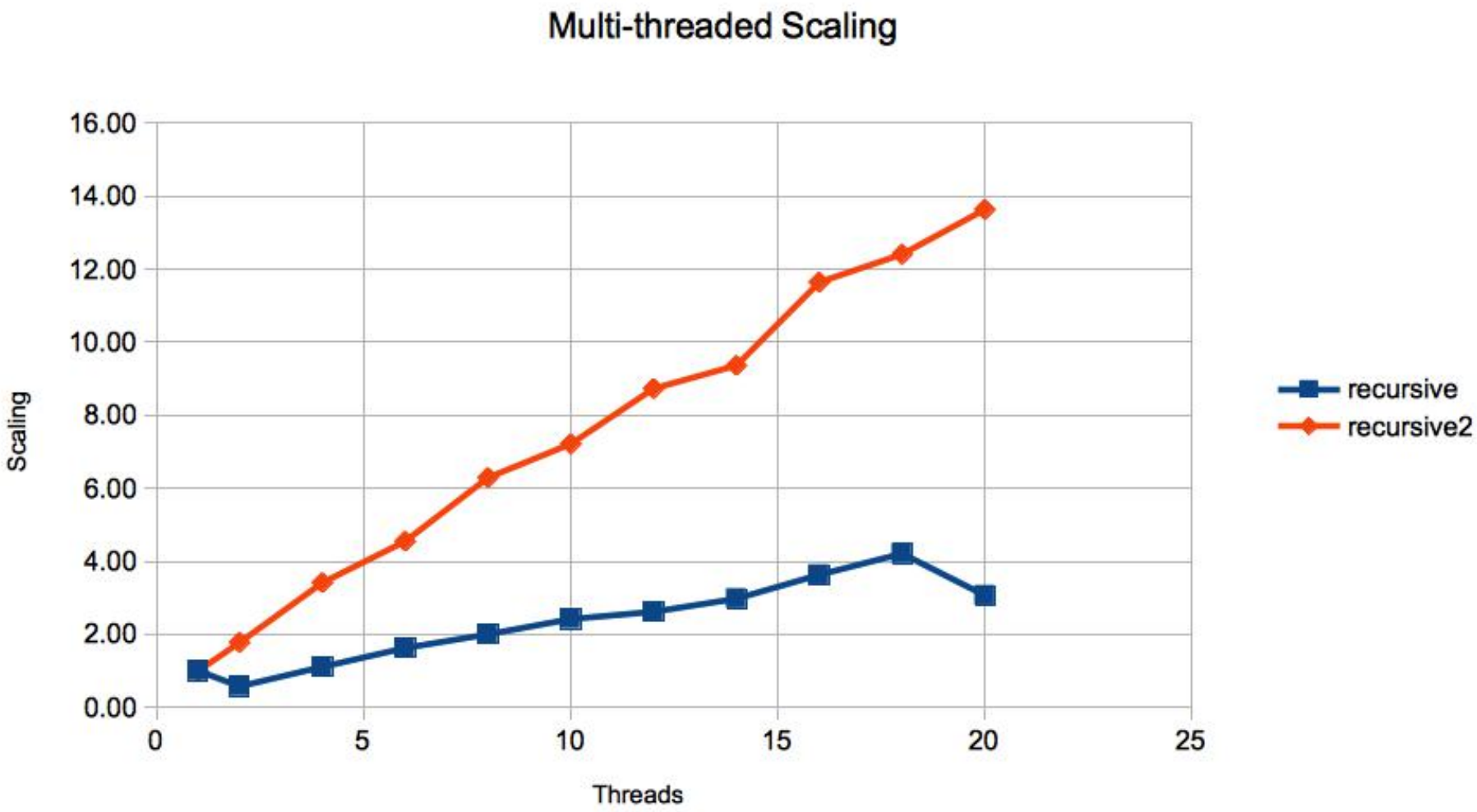
- Too many tasks of varying sizes
- Assignment of tasks to threads is arbitrary; does not consider locality
- At present, **gcc** support for OpenMP tasks is not that great
- Explicit control of task creation
- Create tasks only at a pre-computed depth in the recursion tree

```
int criticalDepth = 0;
int leaves = 1;
int branches = 4;
while(2*numThreads > leaves)
{
    criticalDepth++;
    leaves *= branches;
}
```

Improved Parallel Recursive Multiplication...

```
    if(depth == criticalDepth)
    {
#pragma omp task
        {
            multiply(n, m/2, A, B, C, depth+1, criticalDepth);
            multiply(n, m/2, A+offset12, B+offset21, C, depth+1, criticalDepth);
        }
#pragma omp task
        {
            multiply(n, m/2, A, B+offset12, C+offset12, depth+1, criticalDepth);
            multiply(n, m/2, A+offset12, B+offset22, C+offset12, depth+1, criticalDepth);
        }
#pragma omp task
        {
            multiply(n, m/2, A+offset21, B, C+offset21, depth+1, criticalDepth);
            multiply(n, m/2, A+offset22, B+offset21, C+offset21, depth+1, criticalDepth);
        }
#pragma omp task
        {
            multiply(n, m/2, A+offset21, B+offset12, C+offset22, depth+1, criticalDepth);
            multiply(n, m/2, A+offset22, B+offset22, C+offset22, depth+1, criticalDepth);
        }
    }
    else
    {
        multiply(n, m/2, A, B, C, depth+1, criticalDepth);
        multiply(n, m/2, A+offset12, B+offset21, C, depth+1, criticalDepth);
        multiply(n, m/2, A, B+offset12, C+offset12, depth+1, criticalDepth);
        multiply(n, m/2, A+offset12, B+offset22, C+offset12, depth+1, criticalDepth);
        multiply(n, m/2, A+offset21, B, C+offset21, depth+1, criticalDepth);
        multiply(n, m/2, A+offset22, B+offset21, C+offset21, depth+1, criticalDepth);
        multiply(n, m/2, A+offset21, B+offset12, C+offset22, depth+1, criticalDepth);
```

Improved Parallel Recursive Multiplication...



References

- An Introduction to Parallel Algorithms - Joseph Jaja : Chapters 2.4, 4