Fork-join pattern

UROŠ LOTRIČ

Introduction

Fork-join pattern (slo. razcepi-združi) can generate high level od parallelism

Serial divide-and-conquer algorithms can be efficiently parallelised with fork-join pattern

- Limits on speedup
- Most of the work should go deep into the recursion

Recursive approach to parallelism

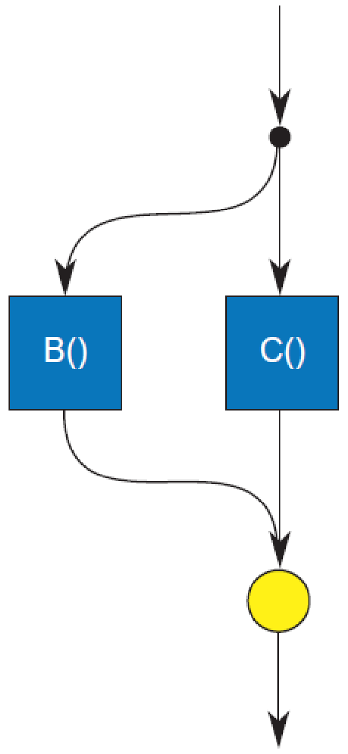
- Need for work schedulers

Fork-join parallelism

Control flow forks (divides) into multiple flows
- One flow turns into more separate flows
- Each flow is independent and not constrained to do similar computation

Multiple flows join (combine) latter
- After join only one flow continues

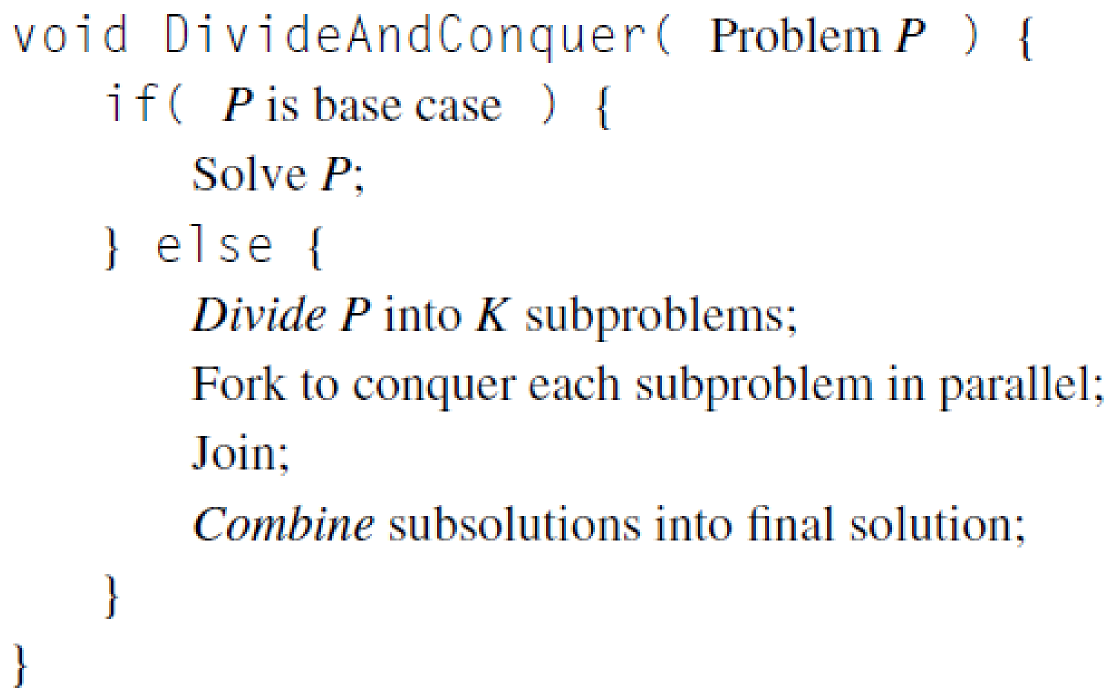
Fork-join as directed graph
- Example: two tasks B() and C() are executed in parallel and joined afterwards



Divide-and-conquer

Typical divide-and-conquer pattern

- Subproblems must be independent



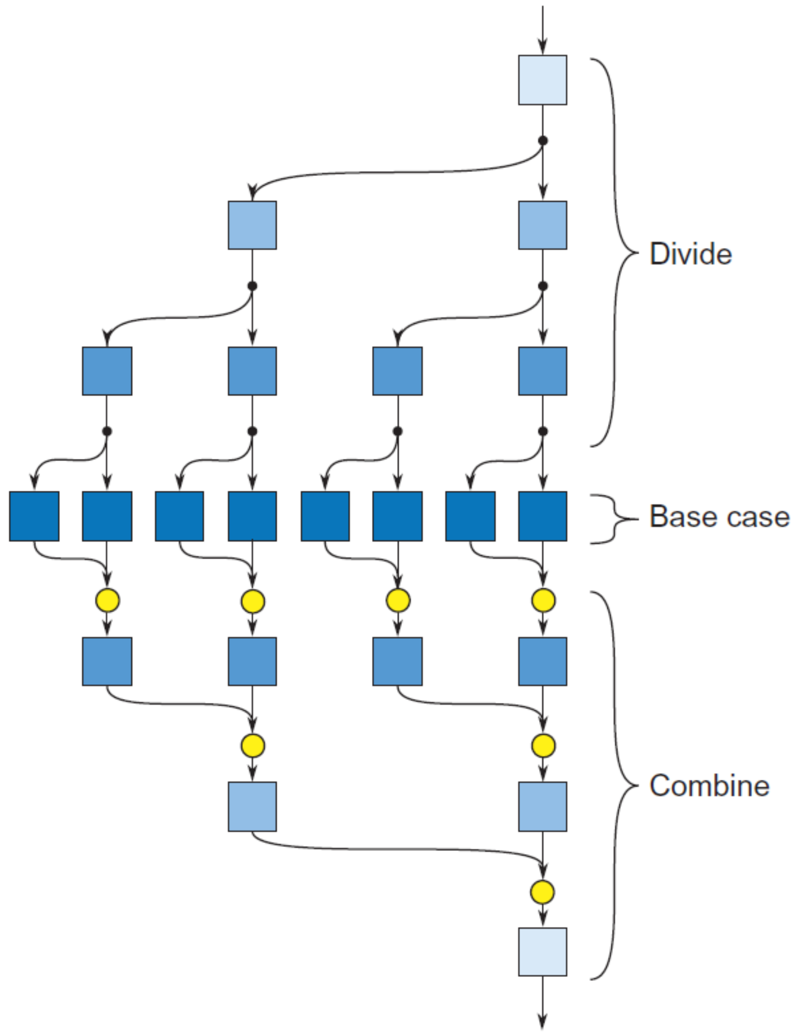
```
void DivideAndConquer( Problem P ) {
    if( P is base case ) {
        Solve P;
    } else {
        Divide P into K subproblems;
        Fork to conquer each subproblem in parallel;
        Join;
        Combine subsolutions into final solution;
    }
}
```

Divide-and-conquer

Typical divide-and-conquer pattern

- Subproblems must be independent
- K subproblems on N levels permits up to K^N parallel tasks
- Vast majority of work must be deep in the recursion where the parallelism is high

- For good performance it is important to select proper size of base case
  - The recursion should not go too deep as scheduling overheads will start to dominate
  - The operations before fork and after join should be fast so they do not strangle speedup



Programming model for fork-join

OpenCL

- Patters with a lot of control do not fit well to GPU concepts

MPI

- Dynamic changing of number of processors makes it possible
- Not used much as the number of processes on clusters must be determined at job submission stage

OpenMP

- From version 3.0 onwards has explicit support for tasks

Programming model for fork-join

OpenMP tasks and threads

- Tasks are independent units of work
  - Tasks present additional level of abstraction
- Threads are assigned to perform the work of each task
  - Tasks may be executed immediately
  - Tasks may be deferred (for example waiting for task dependencies to be fulfilled)
- OpenMP scheduler assigns tasks to threads
  - Schedulers are implementation specific, not determined by standard
- OpenMP 1.0, 2.,0
  - #pragma omp parallel creates tasks **implicitly**
- OpenMP 3.0
  - Adds a way to create tasks **explicitly**
  - Tasks can be **nested**

Programming model for fork-join

OpenMP

- #pragma omp task
  - Indicates that subsequent statements can be independently forked as tasks
  - By default variables are of type firstprivate
- Join with #pragma omp taskwait
  - Waits for all tasks to join
- Tasks can only be used inside a parallel region
  - Only one thread (master) starts the execution
  - Example:
    - function A and its call from the main routine

```
#pragma omp task
B();
C();
#pragma omp taskwait
```

```
function A(problem)
    subproblem1 = f(problem, 1);
    subproblem2 = f(problem, 2);
    #pragma omp task
    solution1 = A(subproblem1);
    solution2 = A(subproblem2);
    #pragma omp taskwait
    solution = g(solution1, solution2);

#pragma omp parallel
#pragma omp master
A(problem)
```

Programming model for fork-join

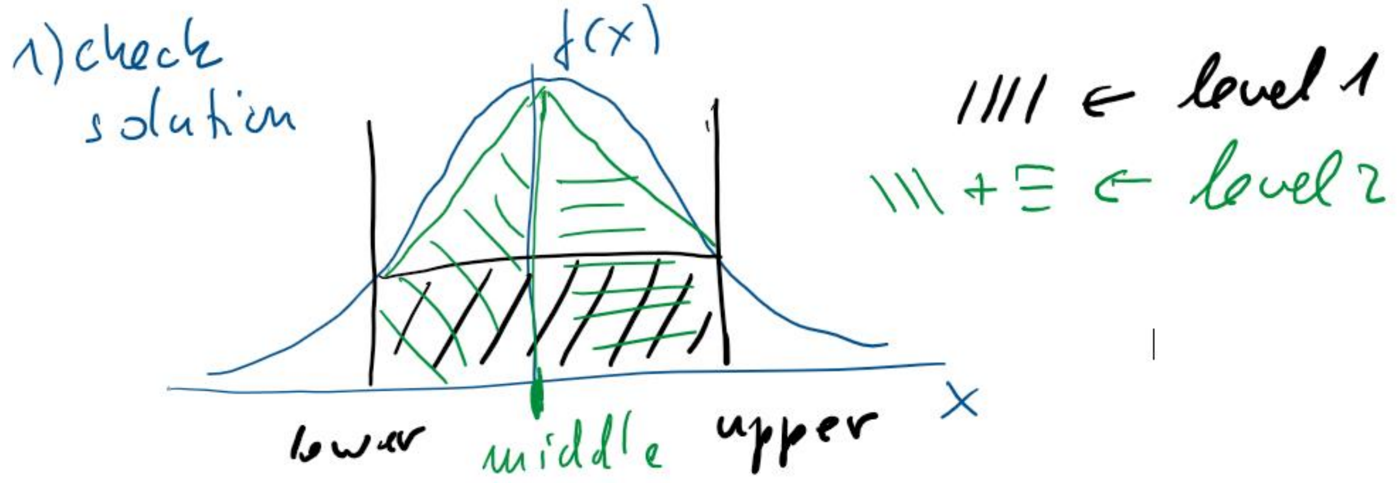
OpenMP

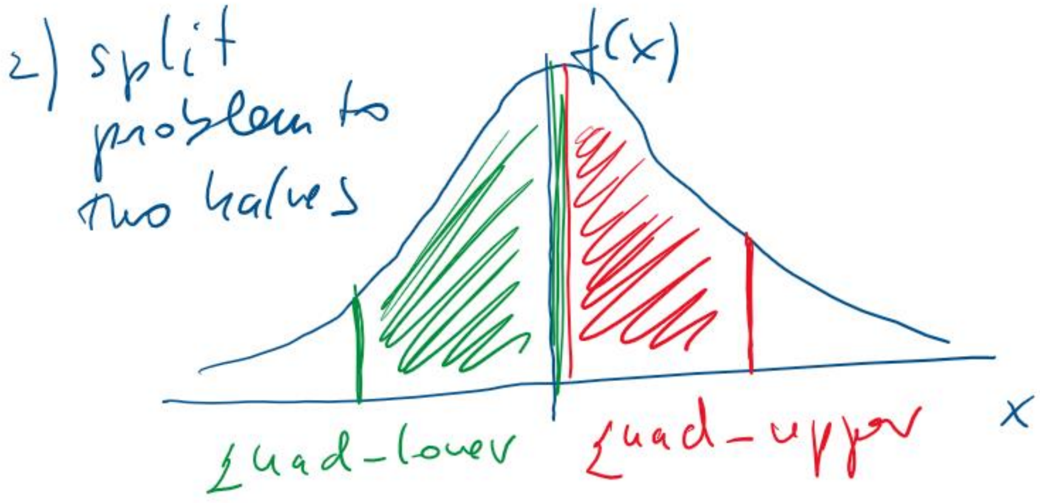
- Programmer has some control on forking
- #pragma omp task final(condition)
  - When condition executes to true, new tasks are not generated anymore
  - The computation is performed inside the calling task

Recursive implementation of map-reduce

Adaptive quadrature

- Trapezoidal rule
- Compare quadrature on two levels
- If difference is grater than allowed,
  split interval to two halves
  and repeat quadrature
  on each halve

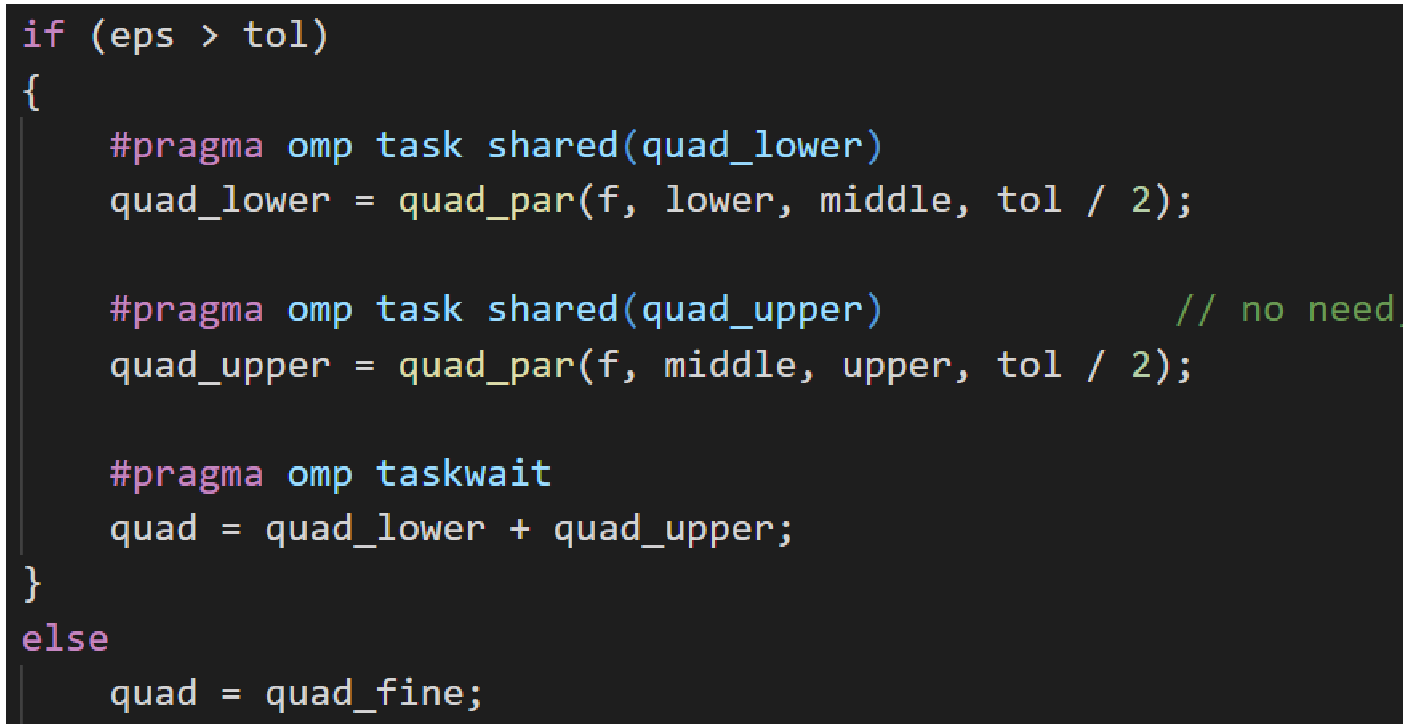




Recursive implementation of map-reduce

Adaptive quadrature

- Serial: adaptquad_ser.c
  - 1 core: 10.5 s
- Parallel: adaptquad_par.c
  - Forking of two tasks
    - 1: 27s, 2: 96s, 4: 211s, 8: 293s
  - Forking of one task
    - 1: 18s, 2: 48s, 4: 103s, 8: 176 s

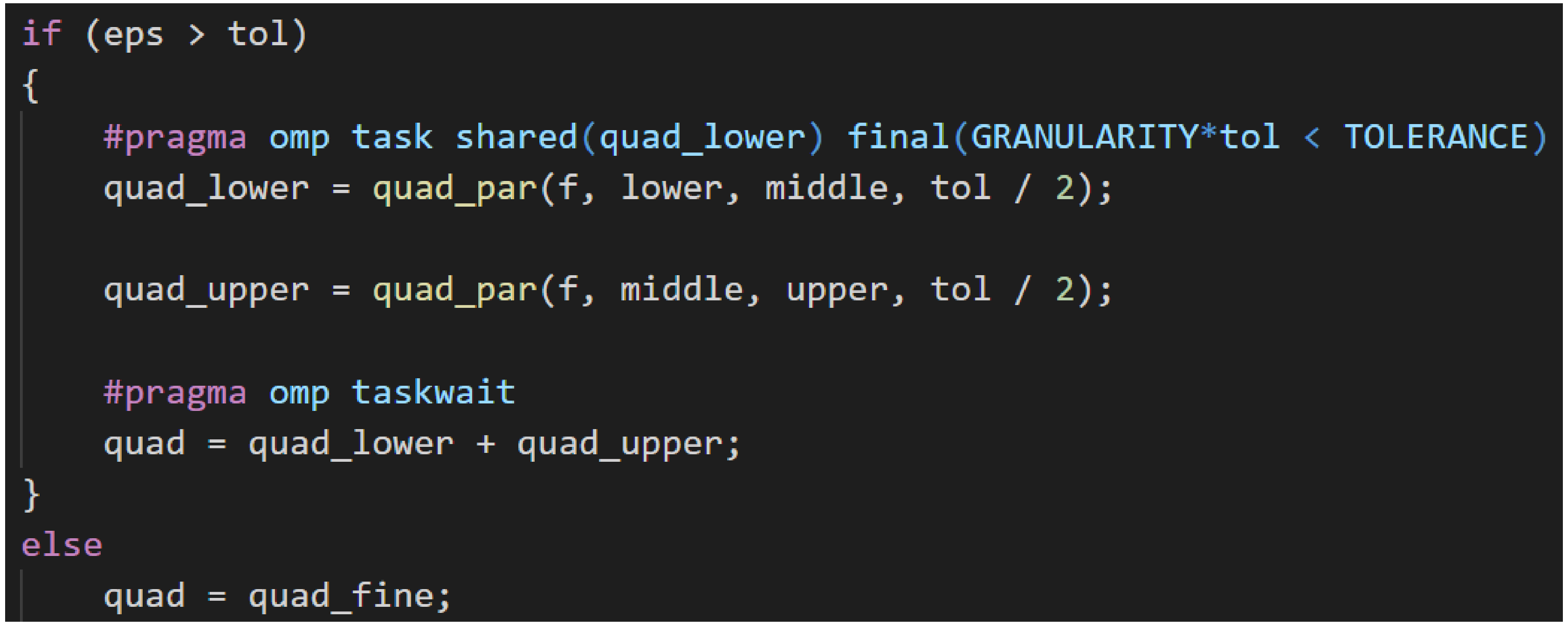


```
if (eps > tol)
{
    #pragma omp task shared(quad_lower)
    quad_lower = quad_par(f, lower, middle, tol / 2);

    #pragma omp task shared(quad_upper)              // no need
    quad_upper = quad_par(f, middle, upper, tol / 2);

    #pragma omp taskwait
    quad = quad_lower + quad_upper;
}
else
    quad = quad_fine;
```

Recursive implementation of map-reduce

Adaptive quadrature

- Improved parallel: adaptquad_par2.c
  - Creation of new tasks is limited
  - 1: 12.7s, 2: 6.7s, 4: 5.4s, 8: 3.1 s



```
if (eps > tol)
{
    #pragma omp task shared(quad_lower) final(GRANULARITY*tol < TOLERANCE)
    quad_lower = quad_par(f, lower, middle, tol / 2);

    quad_upper = quad_par(f, middle, upper, tol / 2);

    #pragma omp taskwait
    quad = quad_lower + quad_upper;
}
else
    quad = quad_fine;
```

Choosing base cases

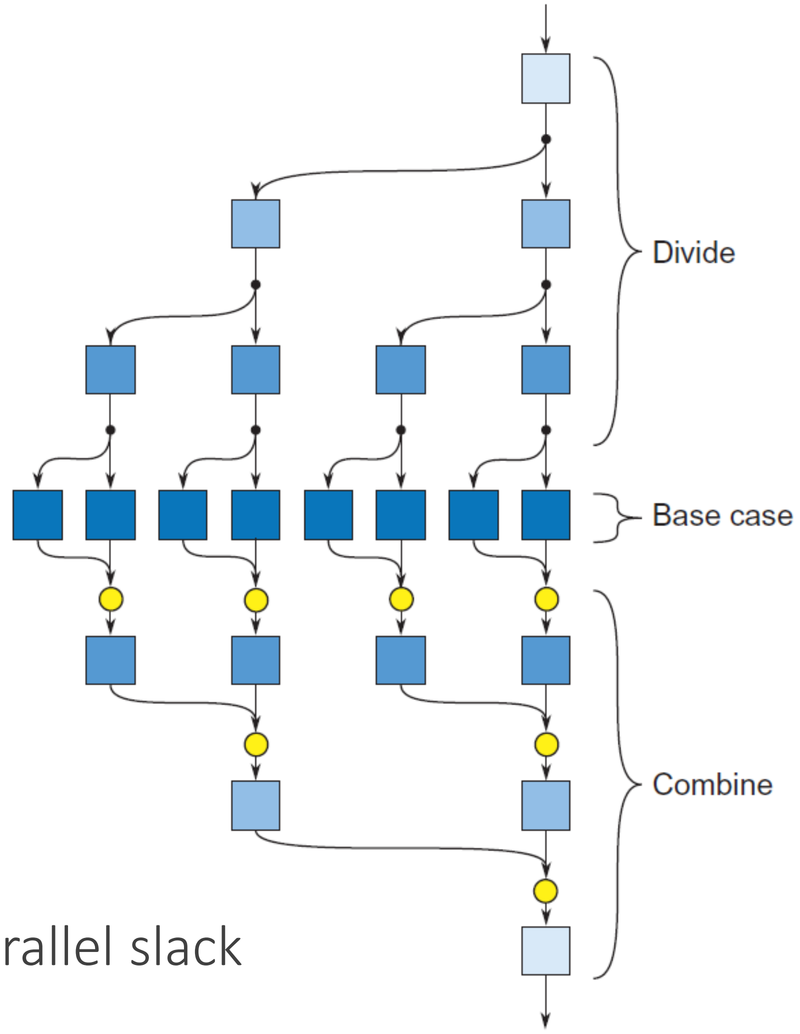
When recursion goes to deep scheduling overheads tend to swamp useful work

Two separate base cases

- A base case for stopping parallel recursion
  - Parallel scheduling overheads
- A base case for stopping serial recursion
  - Function call overheads
- Completely at different levels
    - Serial recursion stops at much smaller problem sizes

It is tempting to set the number of base cases equal to the number of parallel hardware threads

- Scheduler has no flexibility to balance load
  - Even if problem is well balanced, operating system can cause issues
- Better is to over-decompose the problem and create some parallel slack



Complexity of parallel divide-and-conquer

Work and span model

- Let B and C denote tasks
- Serial execution time: $t_1(B\&C) = t_1(B) + t_1(C)$
  - Sum of the work along each path
- Parallel execution time: $t_\infty(B\&C) = \max(t_\infty(B) + t_\infty(C))$
  - Maximum span of any path
- Speedup limit: $S(B\&C) = t_1(B\&C)/t_\infty(B\&C)$

- The model ignores overhead for forking and joining
- To compute $t_1(B\&C)$ and $t_\infty(B\&C)$ we need to define recurrence relations and solve them
  - Simpler to get asymptotic solutions than taking into account base cases
  - Equations are usually similar but differ in constant factors $\rightarrow$ asymptotic solutions can significantly differ

Complexity of parallel divide-and-conquer

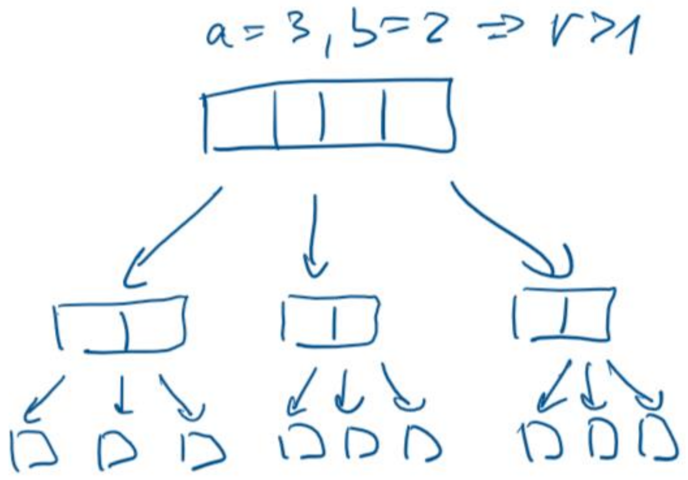
Majority of problems can be described with relation

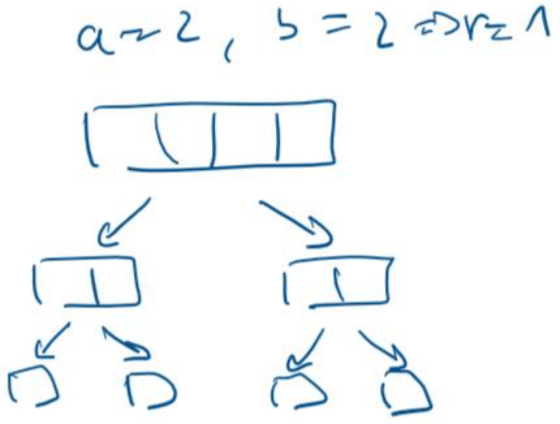
- $t(N) = at\left(\frac{N}{b}\right) + cN^d \quad , \quad t(1) = e$
- Task on a level has cN^d work itself
- Task on one level down has a$c(N/b)^d$
- Proportion: $r = \frac{t_{level+1}}{t_{level}} = \frac{a}{b^d}$

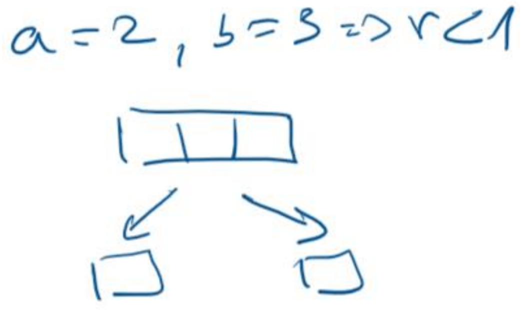
Complexity of parallel divide-and-conquer

Asymptotic solutions

- Case 1: $r > 1$: $t(N) = O\left(N^{\log_b a}\right)$
  - The work exponentially increases with depth, bottom levels dominate
- Case 2: $r = 1$: $t(N) = O\left(N^d \log_2 N\right)$
  - The work at each level is about the same
  - The work is proportional to the work at top level times the number of levels
- Case 3: $r < 1$: $t(N) = O\left(N^d\right)$
  - The work exponentially decreases with depth, top levels dominate
- Examples
  - $c = d = 1$







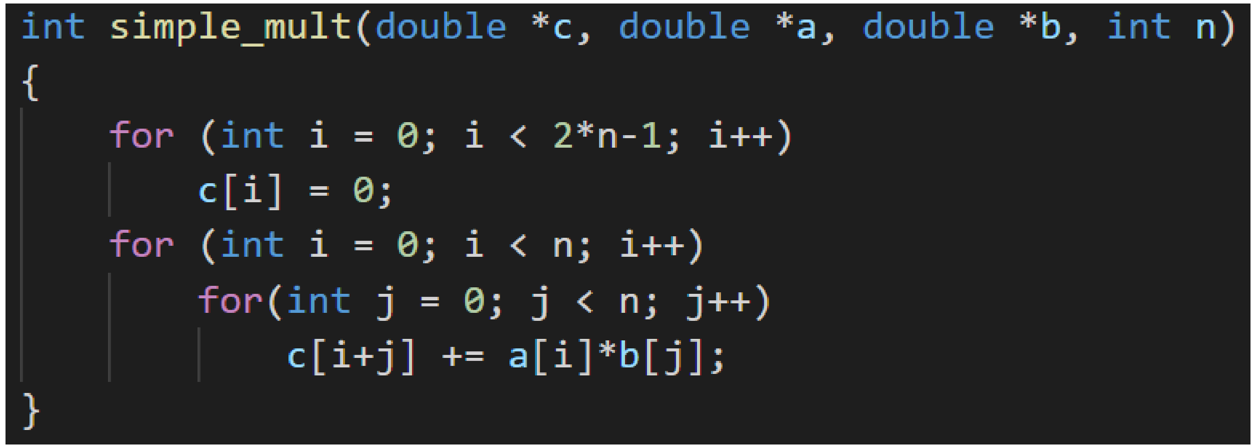
Karatsuba multiplication of polynomials

Input: polynomials a and b of degree $n-1$ (n coefficients)

Output: polynomial c of degree $2n-2$ ($2n-1$ coefficients)

The flat (high-school) method

- Concise and highly parallel for large n
- Creates $O(n^2)$ serial work



```c
int simple_mult(double *c, double *a, double *b, int n)
{
    for (int i = 0; i < 2*n-1; i++)
        c[i] = 0;
    for (int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            c[i+j] += a[i]*b[j];
}
```

Karatsuba multiplication of polynomials

Idea of Karatsuba method

- $c(x) = a(x)b(x)$
  $= (a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + (a_1b_0 + a_0b_1)x + a_0b_0$
- Only 3 multiplications
  - $t_0 = a_0b_0$
  - $t_2 = a_1b_1$
  - $t_1 = (a_0+a_1)(b_0 + b_1)$
  - $c(x) = t_2x^2 + (t_1 - t_0 - t_2)x + t_0$
- Each multiplication can be done by recursive application of Karatsuba method
- For small polynomials flat method becomes more efficient

Karatsuba multiplication of polynomials

Method is commonly used for exact integer multiplication

- Care must be taken of correct propagation of carries
- Example:
  - 1234×5678
  - 12×56

$$a = 1234 = 12 \cdot 100 + 34$$

$$b = 5678 = 56 \cdot 100 + 78$$

$$c = a \cdot b = (\underset{\overset{\uparrow}{a_1}}{12} \cdot \underset{\overset{\uparrow}{x}}{100} + \underset{\overset{\uparrow}{a_0}}{34}) \times (\underset{\overset{\uparrow}{b_1}}{56} \cdot \underset{\overset{\uparrow}{x}}{100} + \underset{\overset{\uparrow}{b_0}}{78})$$

$$t_0 = a_0 \cdot b_0 = 34 \cdot 78 = 2652$$

$$t_1 = (a_0 + a_1)(b_0 + b_1) = (12+34) \cdot (56+78) = 46 \cdot 134 = 6164$$

$$t_2 = a_1 \cdot b_1 = 12 \cdot 56 = 672 \longrightarrow (\underset{\overset{\uparrow}{a_1}}{1} \cdot \underset{\overset{\uparrow}{x}}{10} + \underset{\overset{\uparrow}{a_0}}{2}) \times (\underset{\overset{\uparrow}{b_1}}{5} \cdot \underset{\overset{\uparrow}{x}}{10} + \underset{\overset{\uparrow}{b_0}}{6})$$

$$t_1 - t_0 - t_2 = 2840$$

$$c = 672 \cdot 100^2 + 2840 \cdot 100 + 2652$$

$$\begin{array}{rr} = & 6720000 \\ & 284000 \\ & 2652 \\ \hline c = & 7006652 \end{array}$$

$$t_0 = a_0 \cdot b_0 = 2 \cdot 6 = 12$$

$$t_1 = (a_0 + a_1)(b_0 + b_1) = 3 \cdot 11 = 33$$

$$t_2 = a_1 \cdot b_1 = 5$$

$$t_1 - t_0 - t_2 = 33 - 12 - 5 = 16$$

$$c = 5 \cdot 10^2 + 16 \cdot 10 + 12$$

$$= 500 + 160 + 12 = \underline{\underline{672}}$$

Karatsuba multiplication of polynomials

Implementations

- Sequential: pmult_seq.c
- Parallel: pmult_par.c
  - Two new tasks are created for the first two products
  - Last product computes the calling task
  - When the base cases are smaller than CUTOFF, switch to flat algorithm

Karatsuba multiplication of polynomials

Time complexity

- Unlimited number of threads
- No early stopping of recursion
- Assume $N = 2^k$
- Additions and subtractions take time linear in N
- $t_1(N) = 3t_1\left(\frac{N}{2}\right) + O(N)$, $t_1(1) = O(1)$
  - 3 partial products
  - $r = \frac{3}{2}$, $t_1(N) = O(N^{\log_2 3})$
- $t_\infty(N) = t_\infty\left(\frac{N}{2}\right) + O(N)$, $t_\infty(1) = O(1)$
  - 3 partial products are computed in parallel
  - $r = \frac{1}{2}$, $t_\infty(N) = O(N)$
- $S(N) = \frac{O(N^{1.58})}{O(N)} = O(N^{0.58}) > O(\sqrt{N})$

$$T(n) = 3T\left(\frac{n}{2}\right) + n$$

$$n = 2^k$$

$$T(2^k) = 2^k + 3T(2^{k-1})$$

$$T(2^k) = 2^k + 3^1 \cdot 2^{k-1} + 3^2 \cdot 2^{k-2} + \dots + 3^k \cdot 2^0$$

$$= \sum_{i=0}^{k} 2^i \cdot 3^{k-i} = 3^k \cdot \sum_{i=0}^{k} \left(\frac{2}{3}\right)^i = 3^k \cdot \frac{1-\left(\frac{2}{3}\right)^{k+1}}{1-\frac{2}{3}} = 3^{k+1} - 2^{k+1}$$

$$T(2^k) = 3 \cdot 3^k - 2 \cdot 2^k = 3 \cdot \left(2^{\log_2 3}\right)^k - 2 \cdot 2^k = 3 \cdot (2^k)^{\log_2 3} - 2 \cdot 2^k$$

$$T(n) = 3 \cdot n^{\log_2 3} - 2 \cdot n$$

$$T(n) = T\left(\frac{n}{2}\right) + n$$

$$T(2^k) = 2^k + T(2^{k-1}) = 2^k + 2^{k-1} + \dots + 1$$

$$= \sum_{i=0}^{k} 2^{k-i} = 2^k \sum_{i=0}^{k} \left(\frac{1}{2}\right)^i = 2^k \cdot \frac{1-\left(\frac{1}{2}\right)^{k+1}}{1-\frac{1}{2}}$$

$$= 2^{k+1} - 1$$

$$T(n) = 2n - 1$$

Karatsuba multiplication of polynomials

Space complexity

- Sequential solution needs less memory as it can reuse structures
  - $M_1(N) = M_1\left(\frac{N}{2}\right) + O(N)$, $M_1(1) = O(1)$
- Parallel solution needs to store some temporary values
  - Coefficients of $a(x)$, b(x) and c(x) can be used on all levels of recursion
  - Space is needed to store sum of coefficients $a_0 + a_1, b_0 + b_1$ and their product t_1
  - $M_\infty(N) = 3M_\infty\left(\frac{N}{2}\right) + O(N)$, $M_\infty\,(1) = O(1)$

Cache-oblivious programming

Memory bandwidth constraints often limit speedup

In such cases it is important to reuse data in cache

Cache sizes vary among platforms, tailoring an algorithm to cache size becomes complicated

Suboptimal solution which works well is cache-oblivious programming

- Cache paranoid programming
- Code is written to work well regardless of the actual cache structure
- Divide-and-conquer approach results in good data locality at multiple scales
  - With division the problem first fits to outermost cache
  - With further divisions it sooner or later fits to the inner cache

Cache-oblivious programming

Matrix multiply and add

- Similar to Strassen algorithm
- If matrices are small, use serial multiplication
- If matrices are large, divide multiplication to two parts
- Take into account the following identities

$$\begin{bmatrix} A \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 \end{bmatrix} = \begin{bmatrix} A \times B_0 & A \times B_1 \end{bmatrix},$$

$$\begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \times \begin{bmatrix} B \end{bmatrix} = \begin{bmatrix} A_0 \times B \\ A_1 \times B \end{bmatrix},$$

$$\begin{bmatrix} A_0 & A_1 \end{bmatrix} \times \begin{bmatrix} B_0 \\ B_1 \end{bmatrix} = \begin{bmatrix} A_0 \times B_0 + A_1 \times B_1 \end{bmatrix}.$$

- The goal is to keep the matrices quadratic
  - Always split the longest axis
  - This way cache locality is maximized during multiplication
  - $C_{m\times n} = C_{m\times n} + A_{m\times k} \times B_{k\times n}$
  - $t_1 = O(mkn)$
  - $M(mn + mk + kn)$ is minimal when $m = k = n$

Cache-oblivious programming

Pseudo code

```
// C[m][n] += A[m][k] * B[k][n]
void MultiplyAdd(double **C, double **A, double **B)
{
    if (less then CUTOFF operations are required to compute C)
    {
        MultiplyAddNonRecursive(C, A, B);
    }
    else if (n >= max(m, k))
    {
        C = [C0 C1];
        B = [B0 B1];
        #pragma omp task
        MultiplyAdd(C0, A, B0);

        MultiplyAdd(C1, A, B1);
        #pragma omp taskwait
    }
    else if (m >= k)
    {
        C = [C0
             C1];
        A = [A0
             A1];
        #pragma omp task
        MultiplyAdd(C0, A0, B);

        MultiplyAdd(C1, A1, B);
        #pragma omp taskwait
    }
    else
    {
        A = [A0 A1];
        B = [B0
             B1];
        MultiplyAdd(C, A0, B0);
        MultiplyAdd(C, A1, B1);
    }
}
```

- last else statement prevents raise of memory bandwidth
  - consider A wide and B tall

Cache-oblivious programming

Time complexity

- $t_1 = O(mnk)$
- $t_\infty = (\log_2 m + \log_2 n + k)$
  - On the finest level each task is doing multiplication of one element in matrix C
  - First two terms give the number of partitions
  - Last terms gives time needed to get scalar product for one element in matrix C
- Parallel computation of the last case
  - Additional temporary memory structure
  - $t_\infty = (\log_2 m + \log_2 n + \log_2 k)$
    - Theoretically it is lower than above
    - In practice initialization of additional structures does not pay off
  - Multiply and add is used instead of multiply only to get rid of this additional data structure
- Asymptotic speedup:
  - $S = \frac{t_1}{t_\infty} \approx O(mn)$
  - $k \gg \log_2 m, \log_2 n$

Other applications

Sorting

Recurrences

Prefix scan