Lecture 7

Parallel Algorithmic Paradigms [PPP 7]

- Parallel loops [PPP 7.1]
  - dependence analysis, static / dynamic scheduling
- Data parallelism [PPP 7.2]
  - parallel matrix operations, Gaussian elimination
- Parallel divide-and-conquer [PPP 1, 7.3]
- Pipelining [PPP 7.7]
- Domain decomposition and irregular parallelism [PPP 7.8]
  - parallel N-body calculation
- Asynchronous parallel data structures [PPP 7.4]
  - parallel hashtable, FIFO queue, skip list
- Asynchronous parallel task queue [PPP 7.5]

Parallel Loops

Loops in sequential code
are a promising source of potential parallelism:

+ usually large number of iterations, dominate execution time
  → starting point for incremental parallelization
+ often similar execution times for iterations
  → partitioning, mapping fairly easy
+ simple control flow structure
  → theory of data dependence analysis and loop transformations
+ low syntactic effort to mark a loop as executable in parallel
  → derive a parallel implementation from the sequential code structure

Dependence Analysis

+ important for loop optimizations and parallelization,
  instruction scheduling, data cache optimizations
+ conservative approximations to disjointness of pairs of memory accesses
  (weaker than data-flow analysis)
+ loops, loop nests → iteration space
+ array subscripts in loops → index space
+ dependence testing methods
+ data dependence graph
+ data + control dependence graph → program dependence graph
+ further references: [Padua/Wolfe CACM'86], [Polychronopoulos'88], [Zima/Chapman'91], [Banerjee'88,'93,'94], [Wolfe'96] [Allen/Kennedy'02]

Control and data dependences, dependence graph

S_1 statically (textually) precedes S_2 S_1 pred S_2

S_1 dynamically precedes S_2 $S_1 \lhd S_2$

Dependence = constraint on execution order

control dependence by control flow: $S_1 \delta^c S_2$

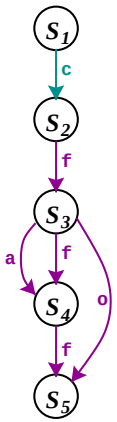
data dependence:

flow / true dependence: $S_3\ \delta^f\ S_4$
$S_3 \lhd S_4$ and $\exists b : S_3$ writes b, S_4 reads b

anti-dependence: $S_3\ \delta^a\ S_4$
$S_3 \lhd S_4$ and $\exists c : S_3$ reads c, S_4 writes c

output dependence: $S_3\ \delta^o\ S_5$
$S_3 \lhd S_5$ and $\exists b : S_3$ writes b, S_5 writes b

S_1: **if** (e) **goto** S_3
S_2: $a \leftarrow \ldots$
S_3: $b \leftarrow a * c$
S_4: $c \leftarrow b * f$
S_5: $b \leftarrow x + f$



Data dependence graph

Data dependence graph: directed graph

(statements, precedence constraints due to data dependences)

For basic blocks:

acyclic dependence graph (DAG)

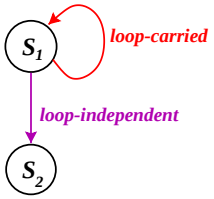
Within loops, loop nests: pred $\neq \lhd$

edge if dependence may exist for *some* pair of iterations

$\rightarrow$ cycles possible

loop-independent versus loop-carried dependences

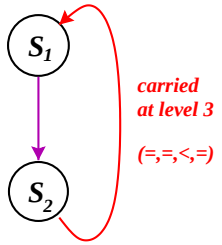
```
  for (i=1; i<n; i++) {
S1:  a[i] = b[i] + a[i-1];
S2:  b[i] = a[i];
  }
```



Dependences in loops: Iteration space

Canonical loop nest:

for i_1 **from** 1 **to** n_1 **do**
  for i_2 **from** 1 **to** n_2 **do**
    ...
      for i_k **from** 1 **to** n_k **do**
        $S_1(i_1,...,i_k):\quad A[i_1, 2*i_3] \leftarrow B[i_2, i_3] + 1$
        $S_2(i_1,...,i_k):\quad B[i_2, i_3+1] \leftarrow 2 * A[i_1, 2*i_3]$



Iteration space: $ItS = [1..n_1] \times [1..n_2] \times ... \times [1..n_k]$

(the simplest case: rectangular, static loop bounds)

Index vector $\vec{i} = \langle i_1, ..., i_k \rangle \in ItS$

Dependence distance and direction

Lexicographic order on index vectors $\rightarrow$ dynamic execution order:

$S_1(\langle i_1, ..., i_k \rangle) \lhd S_2(\langle j_1, ..., j_k \rangle)$ iff

either S_1 pred S_2 and $\langle i_1, ..., i_k \rangle \leq_{lex} \langle j_1, ..., j_k \rangle$

or $S_1 = S_2$ and $\langle i_1, ..., i_k \rangle <_{lex} \langle j_1, ..., j_k \rangle$

distance vector $\vec{d} = \vec{j} - \vec{i} = \langle j_1 - i_1, ..., j_k - i_k \rangle$

direction vector $dirv = \text{sgn}(\vec{j} - \vec{i}) = \langle \text{sgn}(j_1 - i_1), ..., \text{sgn}(j_k - i_k) \rangle$

in terms of symbols $= < > \leq \geq *$

Example: $S_1(\langle i_1, i_2, i_3, i_4 \rangle)\ \delta^f\ S_2(\langle i_1, i_2, i_3, i_4 \rangle)$

distance vector $\vec{d} = \langle 0, 0, 0, 0 \rangle$, direction vector $dirv = \langle =, =, =, = \rangle$,

loop-independent dependence

Example: $S_2(\langle i_1, i_2, i_3, i_4 \rangle)\ \delta^f\ S_1(\langle i_1, i_2, i_3 + 1, i_4 \rangle)$

distance vector $\vec{d} = \langle 0, 0, 1, 0 \rangle$, direction vector $dirv = \langle =, =, >, = \rangle$,

loop-carried dependence (carried by i_3-loop / at level 3)

Dependence testing

For accesses to n-dimensional array A in a loop nest: $S_1: \quad ...A[f(\vec{i})]...$

$S_2: \quad ...A[g(\vec{i})]...$

Is there a dependence between $S_1(\vec{i})$ and $S_2(\vec{j})$ for some $\vec{i}, \vec{j} \in ItS$?

typically f, g linear: $f(\vec{i}) = a_0 + \sum_{l=1}^{n} a_l i_l, \qquad g(\vec{i}) = b_0 + \sum_{l=1}^{n} b_l i_l,$

Exist $\vec{i}, \vec{j} \in \mathbb{Z}^k$ with $f(\vec{i}) = g(\vec{j})$, i.e., $a_0 + \sum_{l=1}^{n} a_l i_l = b_0 + \sum_{l=1}^{n} b_l j_l$, dep. equation

subject to $\vec{i}, \vec{j} \in ItS$, i.e.,

$1 \leq i_1 \leq n_1, \qquad 1 \leq j_1 \leq n_1,$
$\vdots \qquad\qquad \vdots$
$1 \leq i_k \leq n_k, \qquad 1 \leq j_k \leq n_k$

linear inequalities

$\Rightarrow$ constrained linear Diophantine equation system $\rightarrow$ ILP (NP-complete)

Linear diophantine equations

$$\sum_{j=1}^{n} a_j x_j = c$$

with $n \geq 1$, $c, a_j \in \mathbb{Z}$, $\exists j: a_j \neq 0$, $x_i \in \mathbb{Z}$

$x + 4y = 1$ has ∞ many solutions, e.g. $x = 5,\ y = -1$

$5x - 10y = 2$ has no solution in $\mathbb{Z}$

(the RHS must be a multiple of 5)

Theorem:

$$\sum_{j=1}^{n} a_j x_j = c$$

has a solution in $\mathbb{Z}$ iff $\gcd(a_1, ..., a_n) \mid c$.

Dependence tests (1)

Often, a simple test is sufficient to prove independence: e.g.,

gcd-test [Banerjee'76], [Towle'76]:

independence if

$$\gcd\left(\bigcup_{l=1}^{n}\{a_l, b_l\}\right) \not| \sum_{l=0}^{n}(a_l - b_l)$$

constraints on ItS not considered

Example: **for** i **from** 1 **to** 4 **do**

S_1: $b[i] \leftarrow a[3*i-5]+2$

S_2: $a[2*i+1] \leftarrow 1.0/i$

solution to $2i+1 = 3j-5$ exists in $\mathbb{Z}$ as $\gcd(3,2) \mid (-5-2+3-2)$

not checked whether such i, j exist in $\{1, ..., 4\}$

Survey of dependence tests

gcd test

separability test (gcd test for special case, exact)

Banerjee-Wolfe test [Banerjee'88] rational solution in ItS

Delta-test [Goff/Kennedy/Tseng'91]

Power test [Wolfe/Tseng'91]

Simple Loop Residue test [Maydan/Hennessy/Lam'91]

Fourier-Motzkin Elimination [Maydan/Hennessy/Lam'91]

Omega test [Pugh/Wonnacott'92]

subscript-wise or linearized or hierarchical [Burke/Cytron'86]

for i ...
S_1: ...$A[x[i], 2*i]$...
S_2: ...$A[y[i], 2*i+1]$...

for i ...
S_1: ...$A[i, i]$... $A[i*(s_1+1)]$
S_2: ...$A[i, i+1]$... $A[i*(s_1+1)+1]$

Loop parallelization

A transformation that reorders the iterations of a level-k-loop,
without making any other changes,
is valid if the loop carries no dependence.

```
  for (i=1; i<n; i++)
    for (j=1; j<m; j++)
      for (k=1; k<r; k++)
S:      a[i][j][k] = ... a[i][j-1][k] ...          (=,<,=)
```

It is valid to convert a sequential loop to a parallel loop
if it does not carry a dependence.

```
  for (i=1; i<n; i++)        -->        forall ( i, 0, n, p )
S:  b[i] = 2 * c[i];                      b[i] = 2 * c[i];
```

Loop transformations

Goal

+ modify execution order of loop iterations
+ preserve data dependence constraints

Motivation

+ data locality
  → increase reuse of registers, cache
+ parallelism
  → eliminate loop-carried dependences, increase granularity

Some important loop transformations

Loop interchange
Loop fusion vs. Loop distribution
Strip-mining, Tiling/Blocking vs. Loop linearization
Loop skewing

Loop interchange [Allen/Kennedy'84]

```
for (i=1; i<n; i++)
   for (j=0; j<m; j++)
      a[i][j] = a[i-1][j] + b[i][j];
```

↓ Loop interchange

```
for (j=0; j<m; j++)
   for (i=1; i<n; i++)
      a[i][j] = a[i-1][j] + b[i][j];
```

This loop nest is not interchangeable:

```
for (i=1; i<n; i++)
   for (j=0; j<m; j++)
      a[i][j] = a[i-1][j+1] + b[i][j];
```

Would violate (loop-carried) data dependences with dir. vector $(\ldots,<,>,\ldots)$

Loop interchange (cont.)

Goals for loop interchange:

for MIMD parallelization: move parallelizable loops to topmost levels

```
for (i=1; i<n; i++)
  farm                  // 1 barrier per i-iteration
    forall (j,0,m,p)
      a[i][j] = a[i-1][j] + b[i][j];
```

With loop interchange:

```
farm                    // 1 barrier at the end
  forall (j,0,m,p)
    for (i=1; i<n; i++)
      a[i][j] = a[i-1][j] + b[i][j];
```

for vectorization: move parallelizable loops to innermost levels

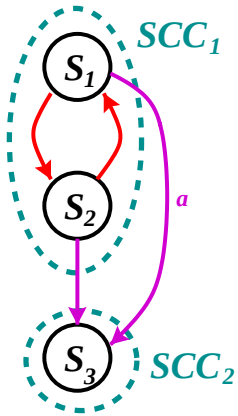
[Allen/Kennedy'87]

Loop distribution

```
   for (i=1; i<n; i++) {
S1:   a[i+1] = b[i-1] + c[i];
S2:   b[i]   = a[i] * k;
S3:   c[i]   = b[i] - 1;
   }
```

↓ Loop distribution

```
   for (i=1; i<n; i++) {
S1:   a[i+1] = b[i-1] + c[i];
S2:   b[i]   = a[i] * k;
   }
   for (i=1; i<n; i++)
S3:   c[i]   = b[i] - 1;
```



often enables vectorization.

Reverse transformation: Loop fusion

Strip mining / Loop blocking / Tiling

```
for (i=0; i<n; i++)
  a[i] = b[i] + c[i];
```

↓ loop blocking with block size s

```
for (i1=0; i1<n; i1+=s)            // loop over blocks
  for (i2=0; i2<min(n-i1,s); i2++) // loop within blocks
    a[i1+i2] = b[i1+i2] + c[i1+i2];
```

Tiling = blocking in multiple dimensions + loop interchange

Reverse transformation: Loop linearization

Loop linearization (aka. loop flattening / loop collapsing)

Flattens a multidimensional iteration space to a linear space:

```
int i, j;
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      iteration( i, j );
```

↓ Loop linearization

```
int i, j, ij;
for (ij=0; ij<n*m; ij++) {
   i = ij / m;
   j = ij % m;
   iteration( i, j );
}
```


Runtime dependence testing

Sometimes parallelizability cannot be decided statically.

```
if (is_parallelizable(...))
  farm
    forall( i, 0, n, p )   // parallel version of the loop
      iteration(i);
else
  seq
    for( i=0; i<n; i++ )   // sequential version of the loop
      iteration(i);
```

The runtime dependence test `is_parallelizable(...)`
itself may partially run in parallel.

Loop scheduling

Static scheduling — Programmer/compiler maps iterations to processors

- independent iterations
  - flat parallel loop
  - nestable parallel loop
- dependent iterations
  - → cycle shrinking, software pipelining

Dynamic scheduling — Iterations assigned to processors at runtime

- independent iterations
  - → iteration task queue, (guided) self-scheduling
- dependent iterations
  - → doacross-scheduling; inspector-executor technique

Static scheduling of independent loop iterations (1)

Case 1: Synchronous parallel loop, $p \leq n$

```
sync void foo()
{
  sh int p = #;
  int i;
  ...
  for ( i=$$; i<n; i+=p )
     execute_iteration ( i );
  ...
}
```

Cyclic distribution of iterations across processors:

P_0 executes iterations $0, p, 2p, \ldots,$

P_1 executes iterations $1, p+1, 2p+1,$ and so on.

Time for a parallel iteration dominated by longest-running iteration

in each interval $I_q = [qp : \min\{n, (q+1)p-1\}]$, for $q = 0, 1, \ldots$

Static scheduling of independent loop iterations (2)

Case 2: Asynchronous parallel loop, $p \leq n$

```
farm
  for ( i=$$; i<n; i+=p )
    execute_async_iteration ( i );
```

Shorthand macro, stride ≥ 1:

```
#define Forall(i,lb,ub,st,p) for(i=$$+(lb);i<(ub);i+=(st)*(p))
```

Shorthand macro, stride $= 1$:

```
#define forall(i,lb,ub,p) for(i=$$+(lb);i<(ub);i+=(p))
```

```
farm
  forall (i, 0, n, p)
    execute_async_iteration ( i );
```

$$t_{\texttt{forall}\ S}(n,p) = \sum_{q=0}^{\lceil n/p \rceil} \max_{i=qp}^{(q+1)p-1} \left(t_{S(i)} + t_1\right) + t_2 \leq \left\lceil \frac{n}{p} \right\rceil \cdot \max_{i=0}^{n-1} \left(t_S(i) + t_1\right) + t_2$$

Static scheduling of independent loop iterations (3)

Case 3: Synchronous, parallel loop, $p > n$

Assign $\lceil \frac{p}{n} \rceil$ resp. $\lfloor \frac{p}{n} \rfloor$ processors to each iteration

Example: two nested parallel loops:

```
 for (i=0; i<n; i++)
    for (j=0; j<m; j++)
       iteration(i,j);
```

```
int i, j;
if (# < n)      // exploit 1D parallelism for outer loop:
   forall( i, 0, n, # )
      for (j=0; j<m; j++)
         iteration(i,j);
else // exploit 2D parallelism, outer loop fully parallel:
   fork( n; @=$$%n; )
      forall( j, 0, m, # )
         iteration(@,j);
```


Static scheduling of independent loop iterations (4)

Alternative: apply loop linearization

Mapping $[0..nm-1] \rightarrow [0..n-1] \times [0..m-1]$:

```
i = ij / m;
j = ij % m;
```

With shorthand macro:

```
sh int p = #;
...
int i, j, ij;
Forall2( i, j, ij, 0, n, 1, 0, m, 1, p )
   iteration(i,j);
```

Dynamic scheduling of independent loop iterations

Self-scheduling, $p \ll n$, chunk size 1, only for asynchronous loops

Implicit iteration task queue: shared loop iteration counter
- integer semaphore `sh int iteration = 0;`
- atomic access by *fetch&incr* (in Fork: `i=mpadd(&iteration,1)`)
- no performance bottleneck on SB-PRAM
- but significant contention for large p on NUMA ($\rightarrow$ chunk scheduling)

```
for ( i=mpadd(&iteration,1); i<n; i=mpadd(&iteration,1) )
   execute_iteration ( i );
```

with shorthand macro:

```
FORALL( i, &iteration, 0, n, 1 )
   execute_iteration ( i );
```

$$t_{\texttt{FORALL}\ S}(n,p) = \max_{0 \le j \le p-1} \sum_{i \text{ by } P_j} \left(t_{S(i)} + t_1\right) + t_2$$

Dynamic scheduling of dependent loop iterations

```
for (i=0; i<n; i++)
   A[i] = B[i] * A[i-D[i]];
```

with $0 \le \texttt{D}[i] \le i$ for all i $\rightarrow$ doacross [Cytron'86]

Shared `iteration` counter + flag array `sh int done[n];`

delay execution of iteration i until iteration $i - D[i]$ finished:

```
farm
  forall ( i, 0, n, p )
    done[i] = 0;
iteration = 0;
farm
  FORALL ( i, &iteration, 0, n, 1 ) {
    if (D[i] > 0)
      while (! done[i-D[i]]) ;   // wait
    A[i] = B[i] * A[i-D[i]];
    done[i] = 1;
  }
```


Parallelization by idiom recognition

Traditional loop parallelization fails for loop-carried dep. with distance 1:

```
S0:  s = 0;
     for (i=1; i<n; i++)
S1:     s = s + a[i];

S2:  a[0] = c[0];
     for (i=1; i<n; i++)
S3:     a[i] = a[i-1] * b[i] + c[i];
```

↓ Idiom recognition (pattern matching) [K.'94],...

```
S1': s = VSUM( a[1:n-1], 0 );

S3': a[0:n-1] = FOLR( b[1:n-1], c[0:n-1], mul, add );
```

↓ Algorithm replacement

```
S1'': s = par_sum( a, 0, n, 0 );
```


Data parallelism

Elementwise apply a scalar operations to all elements of an array section

```
sh data x[N], d[N];
sh int p = #;
int i;
extern straight data f( data );
...
forall(i, 0, N, p)
   x[i] = f( d[i] );
```

In Fortran90 array syntax:

```
x = f( d );
```

In functional programming: apply higher-order function map to f:

x = map f d

Realized as a Fork routine `map`:

```
x = map( f, d );
```

Data parallelism (cont.)

More on data parallelism [PPP 7.2]

Nestable variants of `map`

Generic reductions and prefix computations

Examples: Matrix operations

- Matrix addition / subtraction
- Matrix-vector product
- Matrix-matrix product
- Gaussian elimination
- see also TDDC78

Divide-and-conquer

General strategy:

Divide-and-conquer(Problem P of size n):

If P is trivial (i.e., n very small), solve P directly.
otherwise:

divide P into $q \geq 1$ *independent* subproblems P_1,...,P_q of smaller size n_1,...,n_q

solve these recursively:

$S_1 \leftarrow$ Divide-and-conquer(P_1, n_1),

$\vdots$

$S_q \leftarrow$ Divide-and-conquer(P_q, n_q)

combine the subsolutions S_1,...,S_q to a solution S for P.

Can be described by a recursive generic function:
`seq_divide_conquer` skeleton function [PPP p.303]

Parallel divide-and-conquer

The q subproblems (of size n/k, $k > 1$) are independent of each other

$\rightarrow$ solve them in parallel on p processors:

divide the processor group into q subgroups,
assign each subgroup to one subproblem.
Once a subgroup has only 1 processor, switch to the sequential version.
Parallelize also the divide and the combine step if possible.

$$t_{\texttt{divide\_conquer...}}(n,p) = \begin{cases} t_{\texttt{solvetrivial}}(n), & \text{if } \texttt{istrivial}(n) \\ t_{\texttt{solveseq}}(n), & \text{if } p = 1 \\ t_{\texttt{divide}}(n,p) + t_{\texttt{conquer}}(n,p) & \\ \quad + \left\lceil \frac{q}{p} \right\rceil t_{\texttt{divide\_conquer...}}\left(\left\lceil \frac{n}{k} \right\rceil, 1\right), & \text{if } 1 < p < q \\ t_{\texttt{divide}}(n,p) + t_{\texttt{conquer}}(n,p) & \\ \quad + t_{\texttt{divide\_conquer...}}\left(\left\lceil \frac{n}{k} \right\rceil, \left\lfloor \frac{p}{k} \right\rfloor\right), & \text{if } p \geq q \end{cases}$$

Use the Master Theorem [Cormen/Leiserson/Rivest'90] to solve the recurrence.

Parallel Divide-and-conquer — Examples

Algorithm	divide #subpr.	size reduct.	work	time	p
Recursive parallel sum computation	$q=2$	$k=2$	$O(n)$	$O(\log n)$ $O(\log n)$	n $n/\log_2 n$
Upper/lower parallel prefix	$q=2$	$k=2$	$O(n \log n)$	$O(\log n)$	n
Odd/even parallel prefix	$q=1$	$k=2$	$O(n)$	$O(\log n)$	n
Parallel Mergesort	$q=2$	$k=2$	$O(n \log n)$	$O(\log^2 n)$	n
Parallel QuickSort parallel QuickHull	$q=2$	(dyn.)	$O(n \log n)$ exp. $O(n^2)$ worstc	$O(\log n)$ exp. $O(n)$ worstc	n n
Recursive doubling for first-order lin. recurrences	$q=1$	$k=2$	$O(n)$	$O(\log n)$	n
Parallel Strassen matrix-matrix multiplication	$q=7$	$k=2$	$O(n^{\log_2 7})$	$O(\log n)$	$n^{\log_2 7}$

Example: Recursive parallel sum computation ($q = 2,\ k = 2$)

Global sum of n elements of array a:

If a small ($n = 1$), return $a[0]$. Otherwise:

Split a into 2 halves: a_1 of $\lceil n/2 \rceil$ and a_2 of $\lfloor n/2 \rfloor$ elements

Recursively compute in parallel (2 subgroups)

sum s_1 of a_1 with $\lceil p/2 \rceil$ processors

sum s_2 of a_2 with $\lfloor p/2 \rfloor$ processors

return $s_1 + s_2$

When the subgroup size becomes 1, switch to sequential sum algorithm.

$W(n) = 2W(n/2) + O(1) \in O(n), \quad T(n) = T(n/2) + O(1) \in O(\log n)$

With $p \leq n$ processors: Work $O(n/p)$, Cost $O(n/p \log p)$

With generic function: [PPP 7.3.2]

```
divide_conquer( (void ***)&psum, (void **)a, n,
                sizeof(float), seqfloatsum, issmall,
                seqfloatsum, split2, addFloat );
```


Example: Parallel QuickHull ($q = 2$)

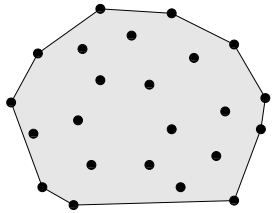
Convex Hull of n points in $\mathbb{R}^2$

given: set S of n points $p_i = (x_i, y_i) \in \mathbb{R}^2$, $i = 1, ..., n$

compute the convex hull $ch(S)$ of S:

$$ch(S) = \bigcap_{K \supset S, K \text{ convex}} K$$

= the smallest convex set containing S.

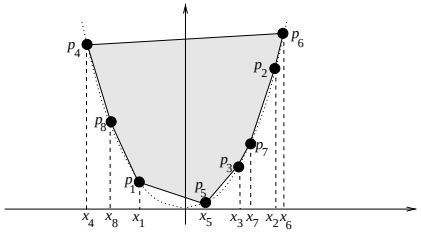


Analogy: nails and rubberband

Example: Parallel QuickHull ($q = 2$) (cont.)

Lower bound for computing the convex hull of n points in $\mathbb{R}^2$

Computing $ch(S)$ needs work $\Omega(n \log n)$.



Reduction to sorting of n real numbers:

Let A be an arbitrary algorithm that computes the convex hull.

Given n real numbers $x_1, ..., x_n$

Set $S = \{p_i := (x_i, x_i^2) : i = 1, ..., n\}$

With A construct $ch(S)$: all p_i appear as vertices!

Linear traversal of the vertices of $ch(S)$, starting at the p_i with least x-coordinate, yields a sorted sequence of the x_i in linear time.

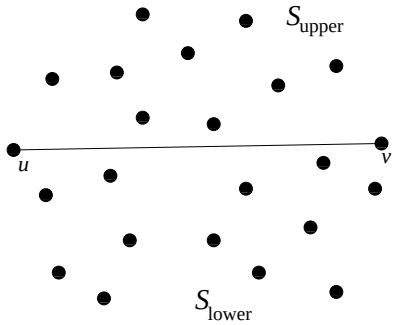
If A were cheaper than $O(n \log n)$ we could accordingly sort faster, contradiction!

Example: Parallel QuickHull ($q = 2$) (cont.)

Quickhull algorithm [Preparata/Shamos'85]

Initialization:

determine the points u, v with minimal resp. maximal x coordinate



u and v belong to the convex hull.

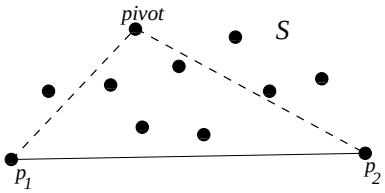
$\rightarrow$ $\overline{uv}$ divides S into S_{upper} and S_{lower}.

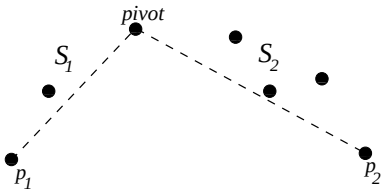
$\rightarrow$ Compute upper and lower hull.

Example: Parallel QuickHull ($q = 2$) (cont.)

Computing the upper hull: (lower hull analogously)

Divide step: determine the pivot point $pivot \in S$
with largest distance from line (p_1, p_2)
i.e. which maximizes the cross-product $|(p_1 - pivot) \times (p_2 - pivot)|$





$pivot$ belongs to the convex hull over (p_1, p_2).

Points *within* the triangle $(p_1, pivot, p_2)$ cannot belong to the convex hull and are eliminated from S. (work linear in $|S|$)

Example: Parallel QuickHull ($q = 2$) (cont.)

$S_1 \leftarrow$ remaining points to the left of line $(p_1, pivot)$
may still belong to the upper hull over $(p_1, pivot)$

$S_2 \leftarrow$ remaining points to the left of line $(pivot, p_2)$,
may still belong to the upper hull over $(pivot, p_2)$

Now apply the method recursively in parallel
to S_1 with $(p_1, pivot)$ and
to S_2 with $(pivot, p_2)$
until the subproblems become trivial.

The points belonging to the convex hull are marked. $\rightarrow$ trivial combine

Expected time (with n processors): $O(\log n)$,
expected work: $O(n \log n)$ (same analysis as for quicksort [Cormen et al.'90])

Worst-case work: $O(n^2)$, worst-case time: $O(n)$

Example: Parallel QuickHull ($q = 2$) (cont.)

Fork code [PPP p.178]

```
sync void qh( sh int *pt, sh int n )
{
 /* We have p1==pt[0],p2==pt[1] */
 sh int pivot,  nprocs_S1;
 sh int p1=pt[0], p2=pt[1]; // the line (p1,p2)
 sh int *S1, *S2, n1, n2;   // to hold subproblems

 if (n==2) return;
 if (n==3) {
    // one point beyond old p1,p2: must belong to hull
    belongs_to_hull[pt[2]] = 1;
    return;
 }

 if (#==1) { seq_qh( pt, n ); return; }
 ...
```


Example: Parallel QuickHull ($q = 2$) (cont.)

```
 ...
 pivot = search_pivot( pt, n );
 belongs_to_hull[pivot] = 1;

 // determine subproblems:
 n1 = n2 = n;
 S1 = delete_right( pt, &n1, p1, pivot );
 S2 = delete_right( pt, &n2, pivot, p2 );

 // determine #processors assigned to each sub-problem:
#define est_work(n) ((float)((n) * (ilog2(n)+1)))
 nprocs_S1 = (int) floor( (float) #
                          * est_work(n1) / est_work(n) );
 if (nprocs_S1==0 && n1>2) nprocs_S1 = 1;

 if ($$<nprocs_S1)    // split group of processors:
    qh( S1, n1 );
                  else     qh( S2, n2 );
}
```

Example: Recursive doubling for first-order linear recurrences ($q = 1,\ k = 2$)

First-order linear recurrence problem:

find solution vector $\vec{x} = (x_0, \ldots, x_{N-1})$ with

$$x_0 = b_0$$
$$x_i = f(b_i, g(a_i, x_{i-1})), \quad 1 \le i < N$$

given the $2N - 1$ parameters $a_1, \ldots, a_{N-1}$, $b_0, \ldots, b_{N-1}$.

Typical case: f addition, g multiplication

```
void seqfolr( float *x, float *a, float *b, int N )
{
  int i;
  x[0] = b[0];
  for (i=1; i<N; i++)
     x[i] = b[i] + a[i] * x[i-1];
}
```


Example: Recursive doubling for FOLR ($q = 1,\ k = 2$) (cont.)

[Kogge/Stone'73]: Divide-and-conquer approach possible if

- f is associative, that is, $f(x, f(y,z)) = f(f(x,y),z)$.
- g distributes over f, that is, $g(x, f(y,z)) = f(g(x,z), g(y,z))$.
- g is associative*, that is, $g(x, g(y,z)) = g(g(x,y),z)$.

Recursive doubling (first step):

$$\begin{aligned} & & x_0 &= b_0 \\ \text{For } 1 \le k \le \frac{N}{2}: & & x_{2k-1} &= f(b_{2k-1}, g(a_{2k-1}, x_{2k-2})) \\ \text{For } 1 \le k < \frac{N}{2}: & & x_{2k} &= f(b_{2k}, g(a_{2k}, f(b_{2k-1}, g(a_{2k-1}, x_{2k-2})))) \\ & & &= f(b_{2k}, f(g(a_{2k}, b_{2k-1}), g(a_{2k}, g(a_{2k-1}, x_{2k-2})))) \\ & & &= f(f(b_{2k}, g(a_{2k}, b_{2k-1})), g(g(a_{2k}, a_{2k-1}), x_{2k-2})) \\ & & &= f(b_k^{(1)}, g(a_k^{(1)}, x_{2k-2})) \end{aligned}$$

with $a_k^{(1)} := g(a_{2k}, a_{2k-1})$, $b_k^{(1)} := f(b_{2k}, g(a_{2k}, b_{2k-1}))$

Example: Recursive doubling for FOLR ($q = 1,\ k = 2$) (cont.)

Substitute $x_k^{(1)} \leftarrow x_{2k}$, $1 \le k \le N/2$, and $b_0^{(1)} \leftarrow b_0$

$\rightarrow$ new first-order linear recurrence problem of size $N/2$:

$$\begin{aligned} & & x_0^{(1)} &= b_0^{(1)} \\ \text{For } 1 \le k < N/2: & & x_k^{(1)} &= f(b_k^{(1)}, g(a_k^{(1)}, x_{k-1}^{(1)})) \end{aligned}$$

$\rightarrow$ partially parallelized algorithm (first step only):

(1) Compute the new parameters $a_k^{(1)}$, $b_k^{(1)}$, $1 \le k < N/2$

(2) Solve the new system (sequentially) $\rightarrow \vec{x}^{(1)}$

(3) Compute the x_{2k-1}, for all $1 \le k \le N/2$ in parallel,
from the $x_{2k} = x_k^{(1)}$ as above

Example: Recursive doubling for FOLR ($q = 1,\ k = 2$) (cont.)

Solve step (2) recursively by another recursive doubling step

$\rightarrow$ logarithmic recursion depth, linear work

Preparation:

$a_i^{(0)} \leftarrow a_i$ for $1 \le i < N$,
$b_i^{(0)} \leftarrow b_i$ for $0 \le i < N$, and
$x_i^{(0)} \leftarrow x_i$ for $0 \le i < N$.
$N^{(d)} \leftarrow 2^{n-d}$ where $n = \log_2 N$, $0 \le d \le n$.

(1) divide step: (level d)
Compute, with $N^{(d)}$ processors in parallel,

$$\begin{aligned} a_k^{(d)} &= g(a_{2k}^{(d-1)}, a_{2k-1}^{(d-1)}) \\ b_k^{(d)} &= f(b_{2k}^{(d-1)}, g(a_{2k}^{(d-1)}, b_{2k-1}^{(d-1)})) \quad \text{for } 1 \le k \le N^{(d)}. \end{aligned}$$

Example: Recursive doubling for FOLR ($q=1,\ k=2$) (cont.)

(2) recursion step:

If $N^{(d)} > 1$, recursively solve the problem for $\vec{a}^{(d)}$ and $\vec{b}^{(d)}$ with size $N^{(d)}$
$\rightarrow$ solution vector $\vec{x}^{(d)}$.

(3) combine step:

compute (the odd values of) $\vec{x}^{(d-1)}$ using up to $N^{(d)}$ processors:

$$\begin{aligned} x_{2k}^{(d-1)} &= x_k^{(d)} \quad \text{for } 0 \le k < N^{(d)} \\ x_{2k-1}^{(d-1)} &= f(b_{2k-1}^{(d-1)}, g(a_{2k-1}^{(d-1)}, x_{2k-2}^{(d-1)})) \\ &= f(b_{2k-1}^{(d-1)}, g(a_{2k-1}^{(d-1)}, x_{k-1}^{(d)})) \quad \text{for } 1 \le k \le N^{(d)} \end{aligned}$$

degree-1 divide $\rightarrow$ no `fork` necessary. [PPP p. 317]

$T(n) = T(n/2) + O(1) \in O(\log n)$
$W(n) = W(n/2) + O(n) \in O(n)$

Accelerated Cascading (1)

Accelerated cascading [Cole/Vishkin'86] [PPP 2.4.4]

turn fast but non-(cost)optimal algorithm into cost-optimal algorithm
by applying blocking and Brent's theorem

1. Use a cost-optimal parallel algorithm to reduce the problem in size
2. Use the fast but nonoptimal algorithm for the reduced problem
3. Use a cost-optimal algorithm to compute the solution
   to the original problem from the solution of the reduced problem

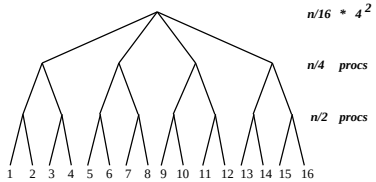
Example problem: Global maximum of n numbers

Degree-2 divide-and-conquer strategy
balanced-binary-tree-like combining (see DC global sum)
$O(\log n)$ time (depth) with n processors
at higher tree levels many processors are idle!

Accelerated Cascading (2)

Idea: use a tree with doubly-logarithmic depth

node u has degree $d_u = \sqrt{n_u}$
where n_u = # leaves in T_u
for $n = 2^{2^k}$, root has degree $2^{2^{k-1}} = \sqrt{n}$



node at level i has degree $2^{2^{k-i-1}}$, for $0 \le i < k$
node at level k has 2 leaves as children
at level i there are $2^{2^k - 2^{k-i}}$ nodes, for $0 \le i < k$
total depth $k+1 = \log\log n + 1$

use a constant-time algorithm for local maximization in each node,
possible on common CRCW PRAM with $O(d^2)$ processors $\rightarrow$ Exercise!

$\rightarrow$ at level $i > 0$ compute $2^{2^k - 2^{k-i}}$ maxima that need $\left(2^{2^{k-i-1}}\right)^2$ processors each

$\rightarrow$ work per level $2^{2^k - 2^{k-i}} \cdot \left(2^{2^{k-i-1}}\right)^2 = O(2^{2^k}) = O(n)$

$\rightarrow$ total work $W(n) = O(n \log\log n)$ still not optimal ...

Accelerated Cascading (3)

... total work $W(n) = O(n \log\log n)$ still not optimal.

Now apply accelerated cascading:

Phase 1:

apply the binary-tree algorithm for the bottommost $\lceil \log\log\log n \rceil$ levels
$\rightarrow$ problem size reduced to $n' = n/\log\log n$
cost-optimal with $O(n)$ operations on $n/\log\log\log n$ processors
in time $O(\log\log\log n)$.

Phase 2:

apply the doubly-logarithmic tree algorithm to the reduced problem
$W(n') = O(n' \log\log n') = O(n)$
in time $T(n') = O(\log\log n') = O(\log\log n)$

(Phase 3 is empty)

In total: work $O(n)$, time $O(\log\log n)$ $\rightarrow$ matches lower bound [JaJa 4.6.3]

Load balancing in irregular parallel divide-and-conquer computations

by Mattias Eriksson

Reference:

Mattias Eriksson, Christoph Kessler, and Mikhail Chalabine:
Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments.
Proc. 8th Workshop on Parallel Systems and Algorithms (PASA 2006), Frankfurt am Main, Germany, March 2006.
GI Lecture Notes in Informatics (LNI), vol. P-81, pp. 313-322, 2006.

http://www.ida.liu.se/~chrke/neststep/eriksson-pasa06-ref.pdf

Pipelining [PPP 7.7]

Split a computation into tasks
→ graph with tasks $v_0, v_1, \ldots, v_k$

arrange these in subsequent stages
→ stages $s_0, s_1, \ldots, s_{d-1}$

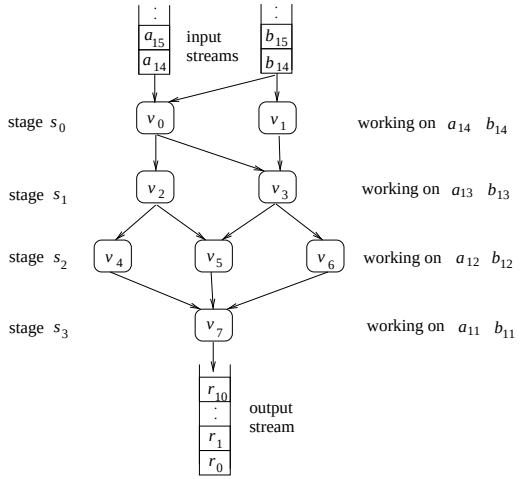
forward results of s_j to inputs of s_{j+1}

process long data streams

Synchronous pipelining
global step signal to all nodes

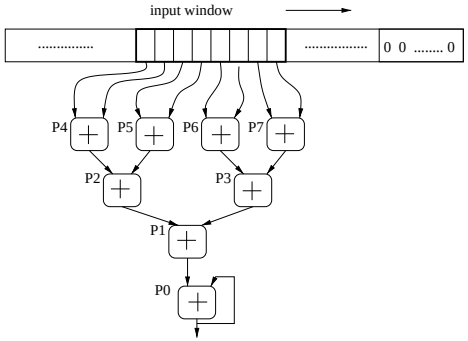
Asynchronous pipelining
data- / event-driven
(e.g., by message passing)

Systolic algorithms
pipelined computations on processor networks



Pipelining (cont.)

Global sum of an array with a tree-shaped pipeline

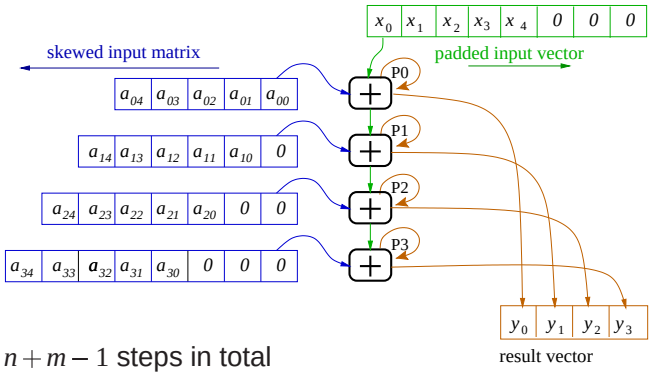


Pipelining (cont.)

Pipelined Matrix-Vector multiplication algorithm [Kung/Leiserson'80]

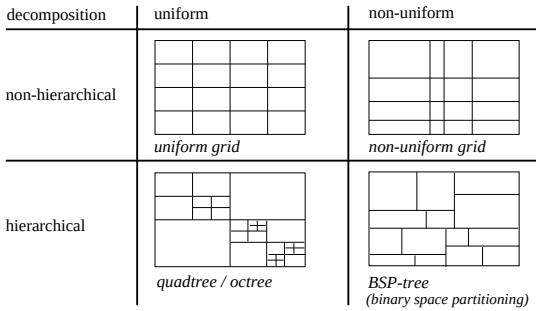
multiply matrix $A \in \mathbb{R}^{n,m}$ by vector $x \in \mathbb{R}^m$ → vector $y \in \mathbb{R}^n$

$$y_i = \sum_{j=0}^{m-1} a_{ij} x_j \qquad i = 0, \ldots, n-1$$



$n + m - 1$ steps in total

Domain Decomposition



decompose problem domain
(discretization/partitioning)
along time or space axes
into clusters

exploit this structure
and locality within clusters
for an efficient solution

e.g. by limiting
the scope of searching

straightforward parallelization:
assign different clusters
to different processors.

Irregular Parallelism

An algorithm is called irregular
if the data access pattern is not known statically
but depends on runtime data.

Example: Sparse matrix-vector multiplication $x = Ab$
with matrix A in column-compressed storage format

```
for (i=0; i<n; i++) // loop over columns
   for (j=firstInCol[i]; j<firstInCol[i+1]; j++)
      x[row[j]] = x[row[j]] + a[j] * b[i];
```

Parallelization of an irregular algorithm produces irregular parallelism.

Requires runtime parallelization techniques $\rightarrow$ overhead, contention

Trade-off:
scalable parallelized naive algorithm vs. non-scalable smart algorithm

N-body problem

Given N particles (e.g., stars, molecules, faces of geometric objects)

Particles interact with each other
by far-field forces (gravity, electrostatic forces, indirect illumination)

Particle i:
current-position vector $\vec{p}_i$ in the space
mass m_i
current velocity vector $\vec{v}_i$

Total gravitational force impact $\vec{f}_i$ on a star i:

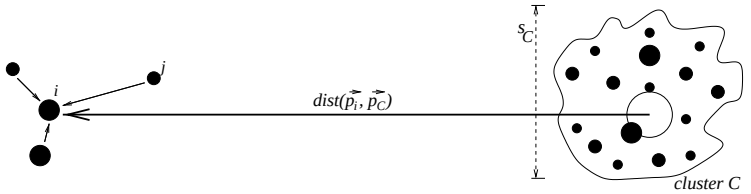
$$\vec{f}_i = \sum_{j \neq i} \vec{f}_{j,i}$$

$$\vec{f}_{j,i} = \gamma \frac{m_i m_j}{(dist(\vec{p}_i, \vec{p}_j))^3} (\vec{p}_i - \vec{p}_j)$$

Can be computed in parallel with work $O(N^2)$, well scalable.

Hierarchical force calculation

Barnes–Hut algorithm [Barnes/Hut'86]
exploit physical locality properties of the particles and the forces
to restrict the accuracy of the overall force computation
e.g. to the precision of the floatingpoint representation used.



Group particles that are relatively close to each other to a cluster C
Keep an artificial *summary particle* ("*multipole*") for each C:

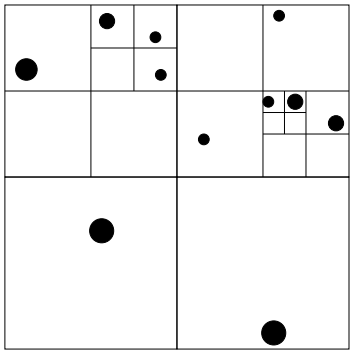
$$m_C = \sum_{j \in C} m_j; \qquad \vec{p}_C = \sum_{j \in C} \frac{m_j}{m_C} \vec{p}_j$$

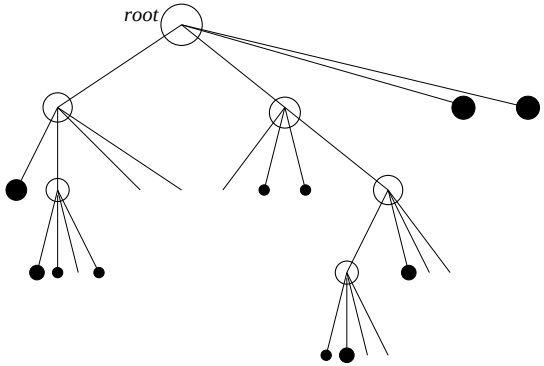
Use the summary particle if C is "far away", ie. $\frac{s_C}{dist(\vec{p}_C, \vec{p}_i)} \leq \theta$
otherwise sum up the individual contributions.

Hierarchical force calculation (cont.)

Recursive uniform space partitioning defines a hierarchy of clusters

→ Octree





The octree can be built sequentially or in parallel

- linear work
- parallel time at least proportional to depth L

Hierarchical force calculation (cont.)

sync vector treeforce(sh int i **, sh node** c **)**

{

if cluster c is empty: $\vec{f}_i = \vec{0}$

if cluster c contains one particle: $\vec{f}_i = \gamma \frac{m_i m_c}{dist(\vec{p}_i, \vec{p}_c)^3} (\vec{p}_i - \vec{p}_c)$

if cluster c contains multiple particles:

if $\frac{s_c}{dist(\vec{p}_i, \vec{p}_c)} < \theta$: $\vec{f}_i = \gamma \frac{m_i m_c}{dist(\vec{p}_i, \vec{p}_c)^3} (\vec{p}_i - \vec{p}_c)$

else: $\vec{f}_i = \sum_{q=0}^{7}$ **treeforce** *(i, c->child[q])*

return $\vec{f}_i$

}

Work depends on accuracy:

If $\theta \geq 1$, inspect $\leq 7L$ summary nodes of all siblings of nodes on path to i

Octree depth L depends on distribution of particles across the space

limited by # bits B used to store particle positions: $N \leq 2^B \rightarrow \log N \leq B$

Total work: $O(N \log N)$ (even distribution) resp. $O(NB)$

Hierarchical force calculation (cont.)

Measurements on SB-PRAM

Force calculation, $N = 100$	p=1	p=4	p=16	p=64	p=256	p=1024
Simple algorithm $O(N^2)$	61.37	15.38	4.89	1.26	0.38	0.21
Tree construction	0.85	0.23	0.09	0.19	0.14	0.14
Summary particles (a)	0.29	0.08	0.04	0.03	0.03	0.03
Summary particles (b)	0.38	0.10	0.04	0.03	0.03	0.03
Treeforce calculation	17.31	4.37	1.19	0.57	0.48	0.48

Force calculation, $N = 500$	p=1	p=4	p=16	p=64	p=256	p=1024
Simple algorithm $O(N^2)$	1546.77	386.9	98.93	24.87	6.32	1.82
Tree construction	4.76	1.21	0.34	0.28	0.53	0.79
Summary particles (a)	1.46	0.44	0.16	0.11	0.10	0.09
Summary particles (b)	1.88	0.50	0.14	0.09	0.09	0.09
Treeforce calculation	123.89	31.13	7.98	3.79	3.75	3.23

for N randomly generated particles.

Parallel data structures for asynchronous computing [PPP 7.4]

Parallel Hashtable [PPP 7.4.1, 7.4.2]

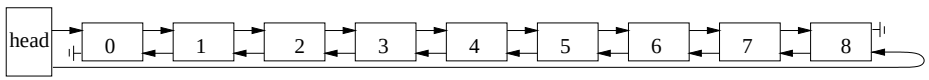
Parallel FIFO queue [PPP 7.4.3]

Parallel Skip list [PPP 7.4.4]

Example:

FIFO Queue:

- sequence of items; operations `put`, `get`, `isempty`
- first-in first-out (global order)



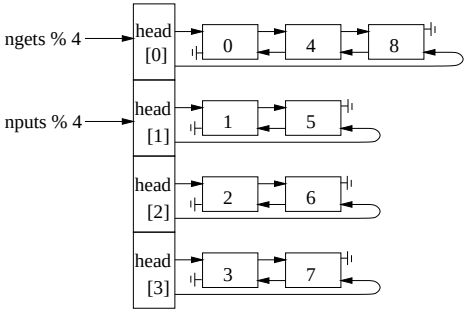
Parallel data structures for asynchronous computing (cont.)

Parallel FIFO Queue [PPP 7.4.3]

k parallel subqueues with cyclic distribution of elements

2 shared counters `ngets`, `nputs`, access by *fetch&incr %k*

fair lock at each subqueue



tricky `isempty` test (due to pending operations)