COMP 633 - Parallel Computing

Lecture 8
September 14, 2021

SMM (3)
Shared Memory Parallel Programming

- Reference material for this lecture
  - OpenMP Tutorial
  - Intel_Cilk++ Programmers Guide

Topics

- Nested parallelism in OpenMP and other frameworks
  - parallel *loops* in OpenMP (2.0)
    - implementation
  - nested parallel *tasks* in Cilk and OpenMP (3.0)
    - task graph and task scheduling
    - Cilk implementation and performance bounds
    - OpenMP directives and implementation
  - nested *data parallelism* in NESL
    - flattening nested parallelism into vector operations

Loop parallelism

- OpenMP annotation of matrix-vector product $R = M^{n \times m} \cdot V^{m}$

```
#pragma omp parallel for private(i,j)
for (i = 0; i < n; i++) {
   R[i] = 0;

   for (j = 0; j < m; j++) {
      R[i] += M[i][j] * V[j];
   }
}
```

– how will it be executed?
  - OpenMP will allocate all available threads to the outer loop
  - Each thread will perform an approximately equal number of iterations (either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$)

– is it safe?
  - each iteration of the outer loop is executed by a specific thread
  - thus R[i] is not subject to concurrent updates

Nested loop parallelism

- OpenMP annotation of matrix-vector product $R = M^{n \times m} \cdot V^m$

```
#pragma omp parallel for private(i)
for (i = 0; i < n; i++) {
   R[i] = 0;

   #pragma omp parallel for private(j) reduction(+:R[i])
   for (j = 0; j < m; j++) {
      R[i] += M[i][j] * V[j];
   }
}
```

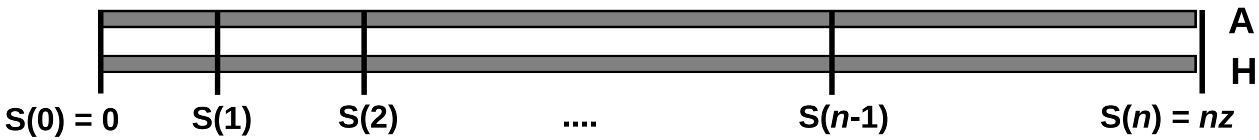
– how should nested parallel regions be executed?
  - each thread in the outer loop becomes the master thread of a team of threads in the inner parallel loop iterations

– how will it be executed?
  - most OpenMP implementations allocate all threads to the outer loop by default
  - the **num_threads(*t*)** clause specifies *t* threads be allocated to a parallel region

– additional consideration
  - Most modern processors have short vector arithmetic units (256 or 512 bit AVX)
    – accelerate the dot product in the inner loop

Irregular loop parallelism: more challenges

- sparse matrix-vector product R = MV
  - *sparse* matrix M is represented using two 1D arrays
    - A[nz], H[nz] arrays of non-zero values and corresponding column indices
    - The nonzeros in S[i] describes the partitioning of A and H into n rows of M



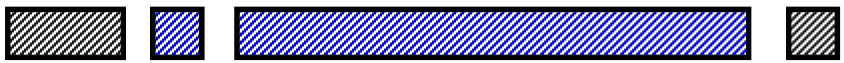

```
#pragma omp parallel for private(i)
for (i = 0; i < n; i++) {
   R[i] = 0;

   #pragma omp parallel for private(j) reduction(+:R[i])
   for (j = S[i]; j < S[i+1]; j++) {
      R[i] += A[j] * V[H[j]];
   }

}
```

How should SPMV be executed?

- Parallelize outer loop?
  - requires dynamic load balancing
    - Poor performance possible when
      - n is not much larger than p
      - there is a large variation in number of non-zeros per row



- Parallelize inner loop?
  - poor performance on “short” rows with few non-zeros

- Both loops must be fully parallelized
  - to achieve runtime bounds of the sort promised by Brent’s theorem
  - $W(nz) = O(nz)$
  - $S(nz) = O(\lg nz)$

Nested parallelism model (a)

- In the W-T model nested parallelism is unrestricted
  - divide & conquer algorithms
    - parallel quicksort, quickhull
  - Other examples, e.g. histogram problem
    - (lg n) reductions of size (n/lg n) run in parallel

- OpenMP recognizes nested parallelism in nested loops, but only implements certain cases
  - typically only the outermost level of parallelism is realized
  - occasional support for orthogonal iteration spaces
    - e.g. {1, … ,n} X {1, … ,m} treated as single iteration space of size nm
    - but how to divide into p equal parts?
  - OpenMP 2.0 directives
    - specify allocation of threads to loops
    - e.g. 16 threads total
      - outermost loop: 4 threads
      - nested loop: respective teams of e.g. 3, 5, 4, 4 threads
    - very tedious and dependent on both problem and machine

Nested parallel model (b)

- Towards the Work-Time model:
  - task parallelism
    - a task is some code for execution and some context for data
      - inputs, outputs, private data
      - dynamically generated and terminated at run time
      - tasks are automatically scheduled onto threads for execution
    - language support for tasks
      - Cilk, Cilk Plus (MIT, Intel)
        - » C or C++ with tasks (and data-parallel operations in Cilk Plus)
        - » runtime scheduler with optimal scheduling strategy
      - OpenMP 3.0
        - » C, C++, Fortran with tasks
  - nested data parallelism
    - generalization of data parallelism
    - implemented in NESL (NEsted Sequence Language)
      - functional language with sequence construction functions (forall)
      - nested sequence construction corresponds to nested parallelism
      - compile-time *flattening transformation* to convert nested sequence operations to simple data-parallel vector operations

Task parallelism: Cilk

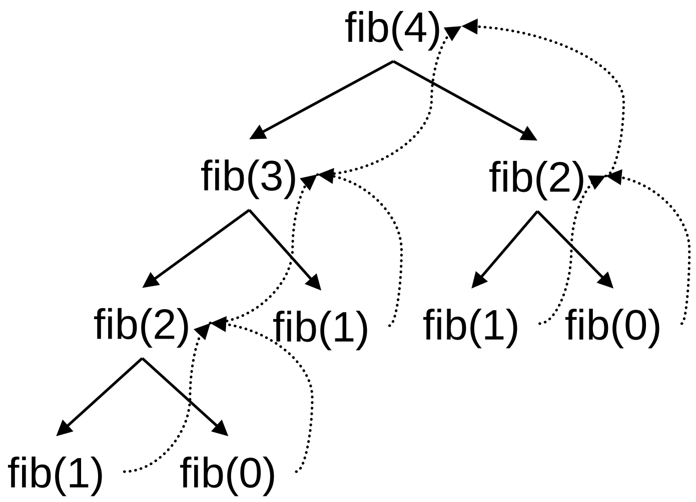
- Cilk fibonacci program
  - Cilk = C + {**cilk**, **spawn**, **sync**}
  - **cilk** declares a procedure to be executable as a task
  - **spawn** starts a cilk task that executes concurrently with creator
  - **sync** waits for all tasks spawned in current procedure to complete

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = spawn fib(n-1);
        y = spawn fib(n-2);

        sync;

        return (x+y);
    }
}
```

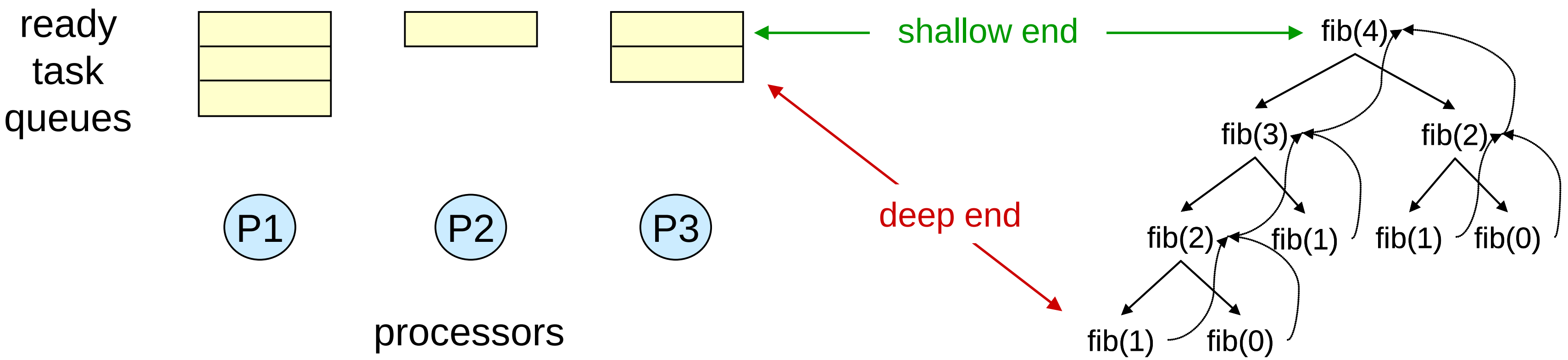


Task dependence graph

CILK runtime task scheduler

- Task dependence graph unfolds dynamically
  - typically far more tasks ready to run than threads available
  - potential blow-up in space

- Scheduling strategy
  - each thread maintains a local double-ended queue of tasks ready to run
    - shallow and deep ends refer to relative positions of tasks in dependence graph
  - if queue is nonempty
    - execute ready task at the *deepest level* in the queue
    - corresponds to sequential execution order, generally friendly to memory hierarchy
  - if queue is empty
    - steal a task at *shallowest level* of the queue in some *randomly chosen* other thread



Cilk execution properties

- Task execution order is parallel depth-first
  - serial order at each processor
  - good fit for parallel memory hierarchy
  - space bound: $Space_p(n) = Space_1(n) + pS(n)$
- Global execution time follows bounds determined by Brent's theorem
  - $T_p(n,p) = O(W(n)/p + S(n))$
- Efficiency
  - work-first principle (busy processors keep working)
    - minimizes interference with useful progress
  - work-stealing principle
    - idle processors steal tasks towards high end of current DAG
      - these tasks are expected to unfold into larger portions of the complete DAG

Sparse matrix-vector product in Cilk++

- Does this solve our problem?

```
double A[nz], V[n],R[n];
int H[nz], S[n+1];

void sparse_matvec() {
   for (int i = 0; i < n; i++) {
      R[i] = cilk_spawn dot_product(S[i],S[i+1]);
   }
   cilk_synch;
}

double dot_product(int j1, int j2) {
   cilk::reducer_opadd<double> sum;
   for (int j = j1; j < j2; j++) {
      cilk_spawn sum += A[j] * V[H[j]];
   }
   cilk_synch;
   return sum.get_value();
}
```

Task creation in loops with Cilk++

- cilk_for creates a set of tasks using recursive division of the iteration space

```
double A[nz], V[n],R[n];
int H[nz], S[n+1];

void sparse_matvec() {
   cilk_for (int i = 0; i < n; i++) {
      R[i] = dot_product(S[i],S[i+1]);
   }
}

double dot_product(int j1, int j2) {
   cilk::reducer_opadd<double> sum;
   cilk_for (int j = j1; j < j2; j++) {
      sum += A[j] * V[H[j]];
   }
   return sum.get_value();
}
```

Divide and conquer algorithms with Cilk

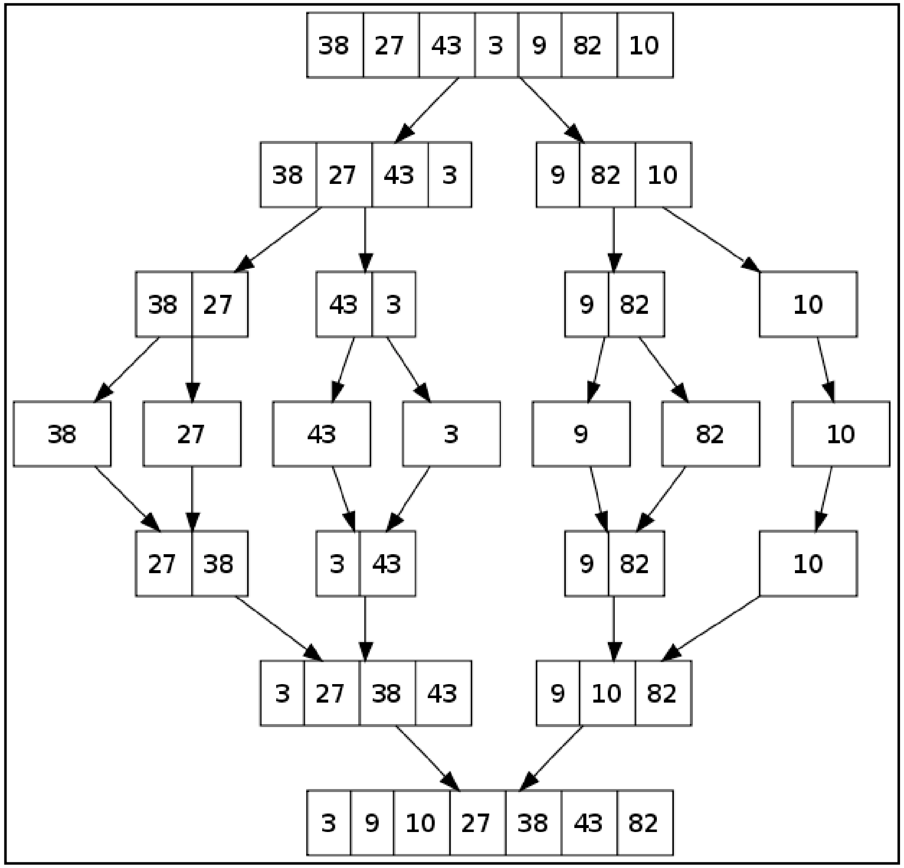
```
cilk void mergesort(int A[], int n) {
  if (n <= 1)
        return
  else {
      spawn mergesort(&A[0],   n/2);
      spawn mergesort(&A[n/2], n/2);
  }
  sync;
  merge(&A[0], n/2, &A[n/2], n/2);
}
```

W(n) =

S(n) =

Why well-suited to the memory hierarchy?

Mergesort Example with Tasks

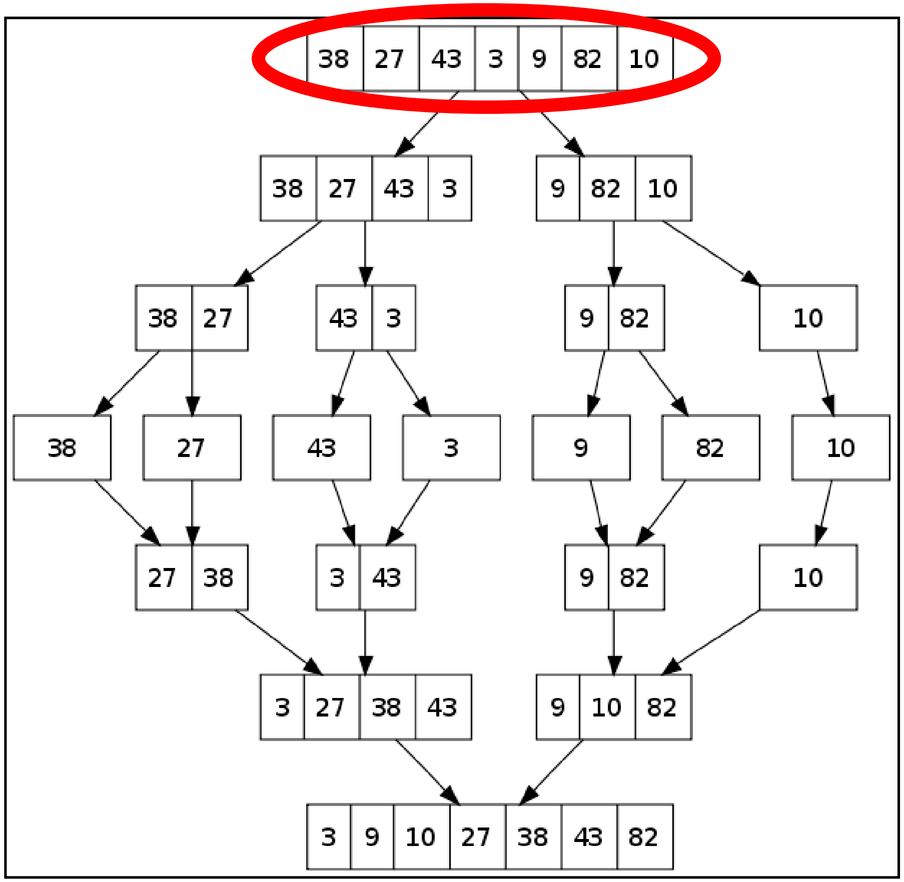


Using two threads:

Thread 0

Thread 1

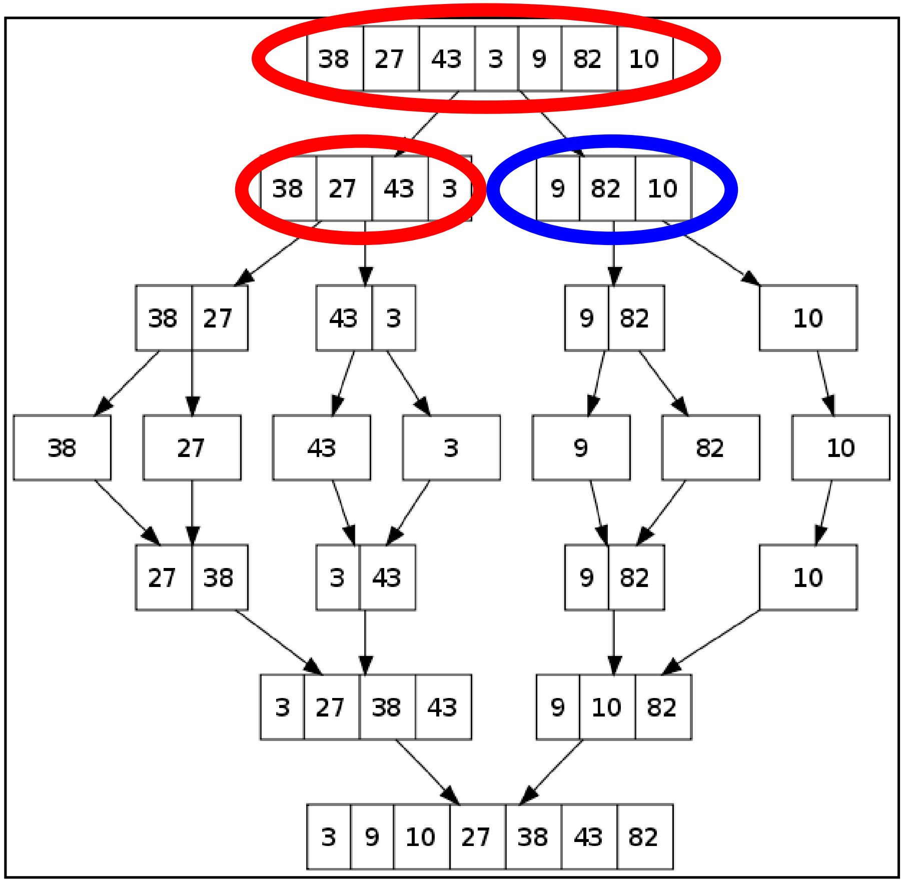
Mergesort Example with Tasks



Thread 0

Thread 1

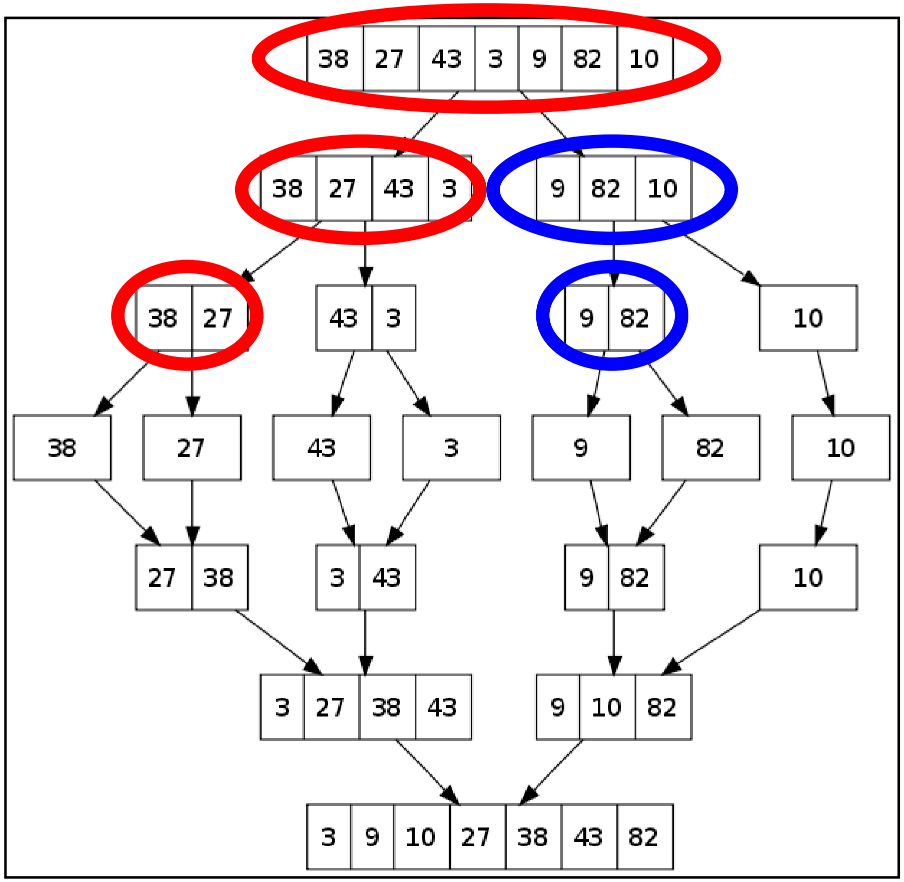
Mergesort Example with Tasks



Thread 0

Thread 1

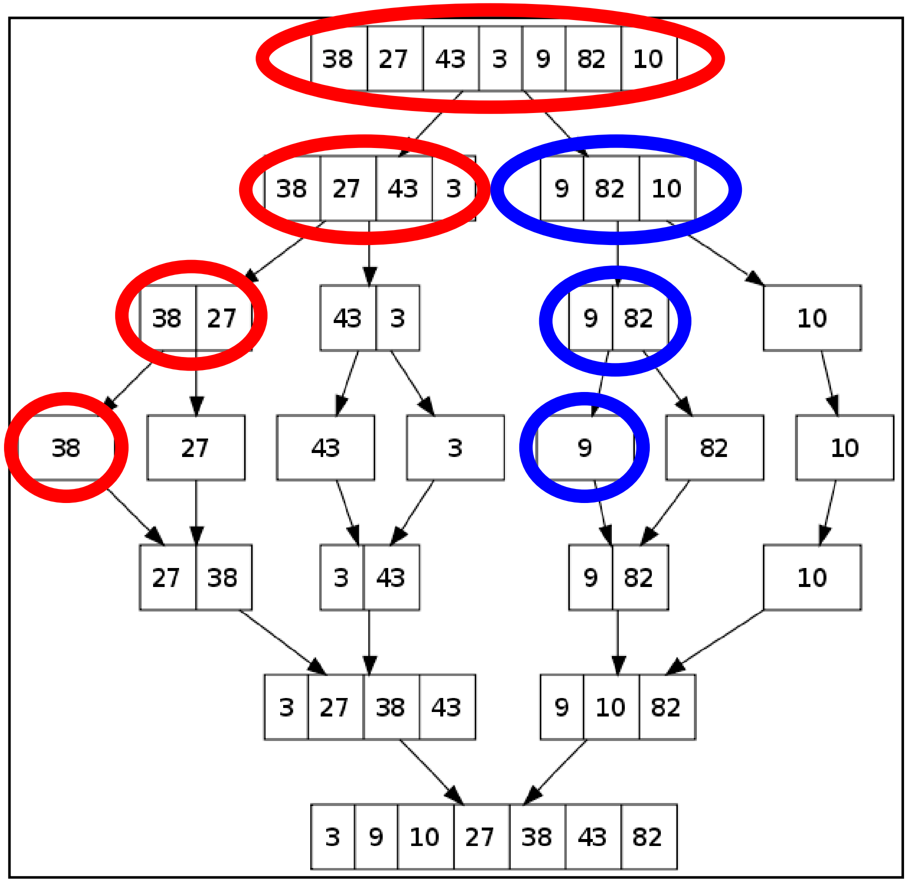
Mergesort Example with Tasks



Thread 0

Thread 1

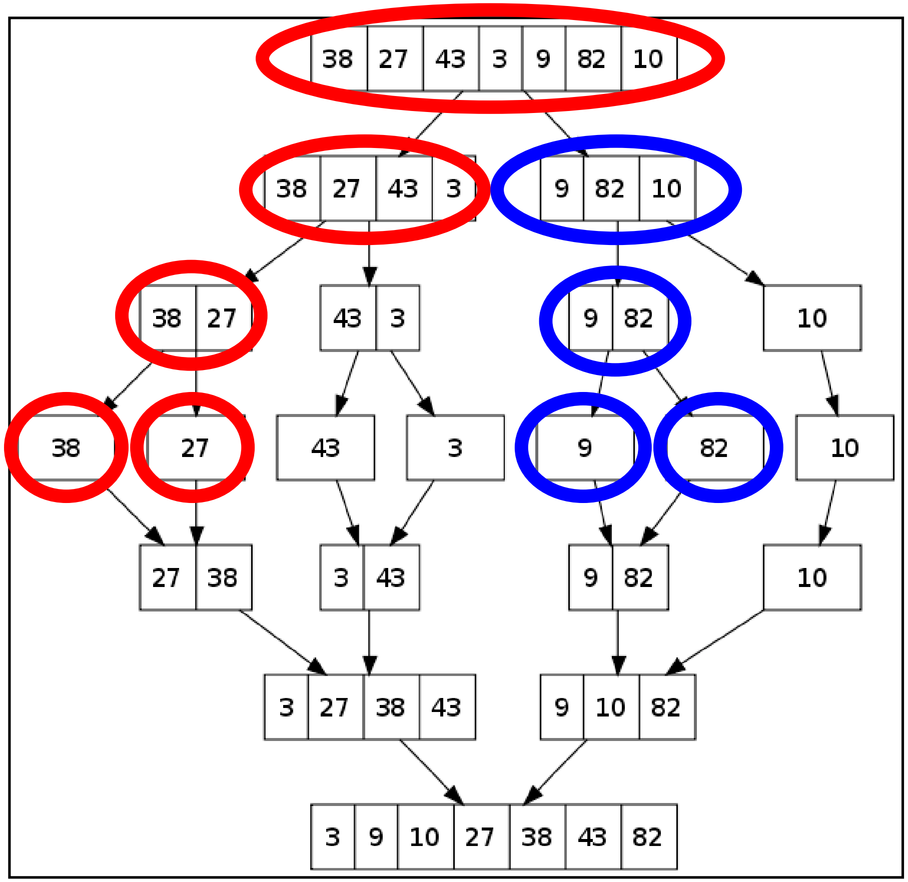
Mergesort Example with Tasks



Thread 0

Thread 1

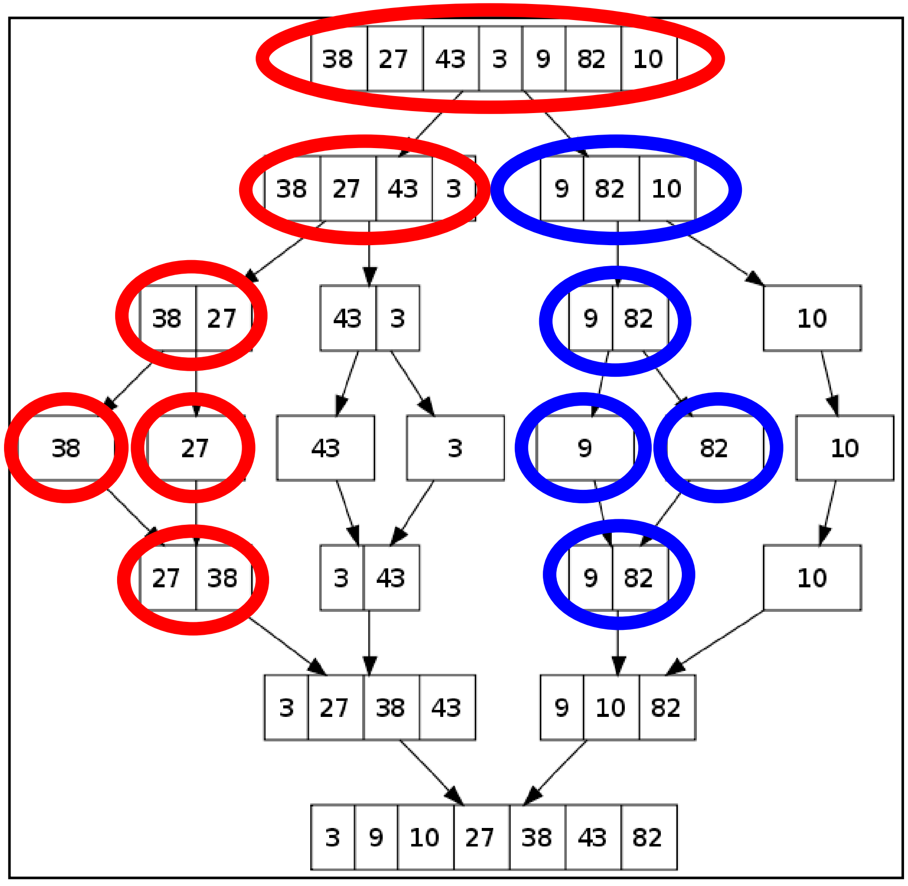
Mergesort Example with Tasks



Thread 0

Thread 1

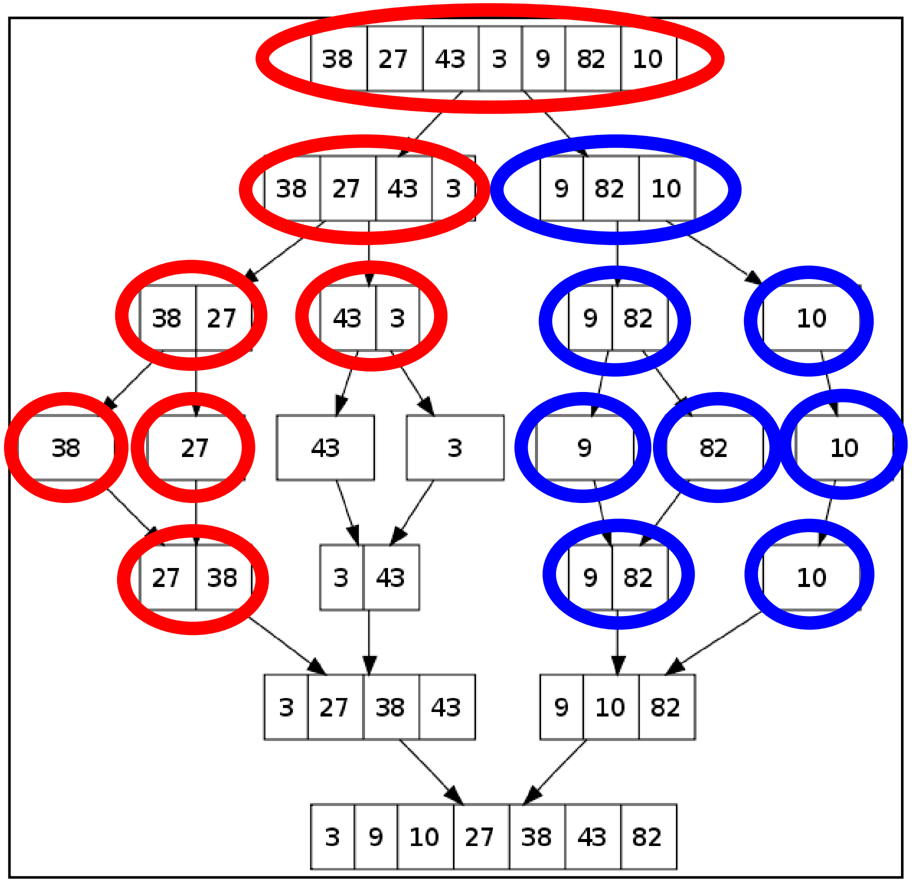
Mergesort Example with Tasks



Thread 0

Thread 1

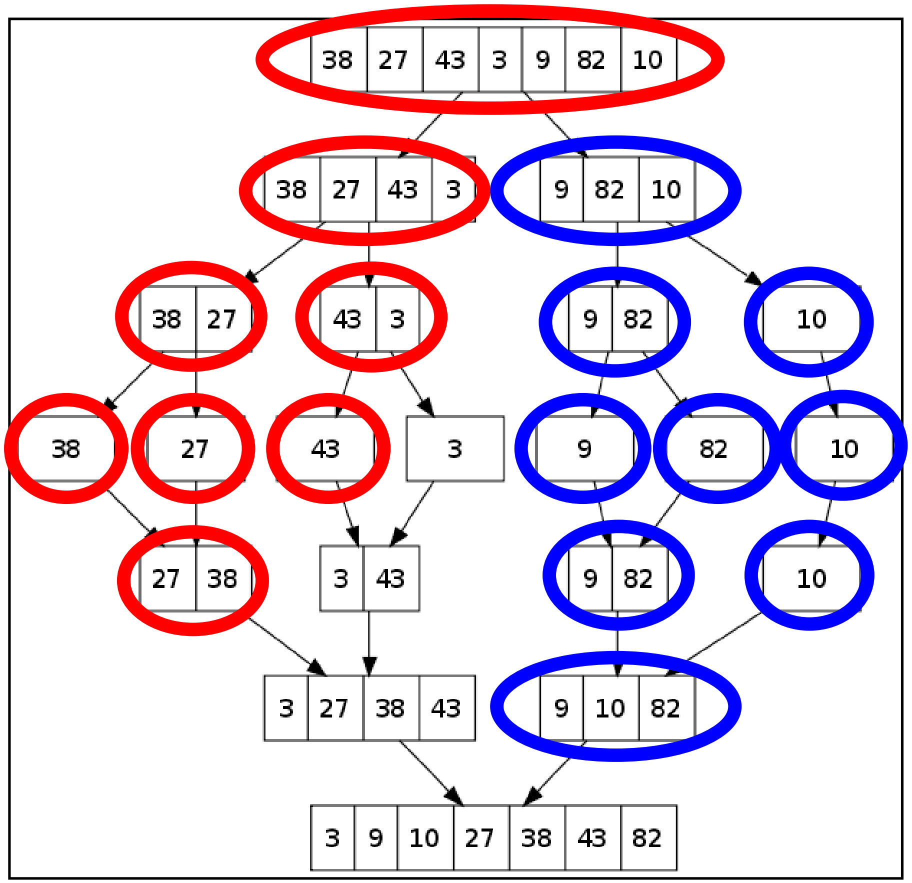
Mergesort Example with Tasks



Thread 0

Thread 1

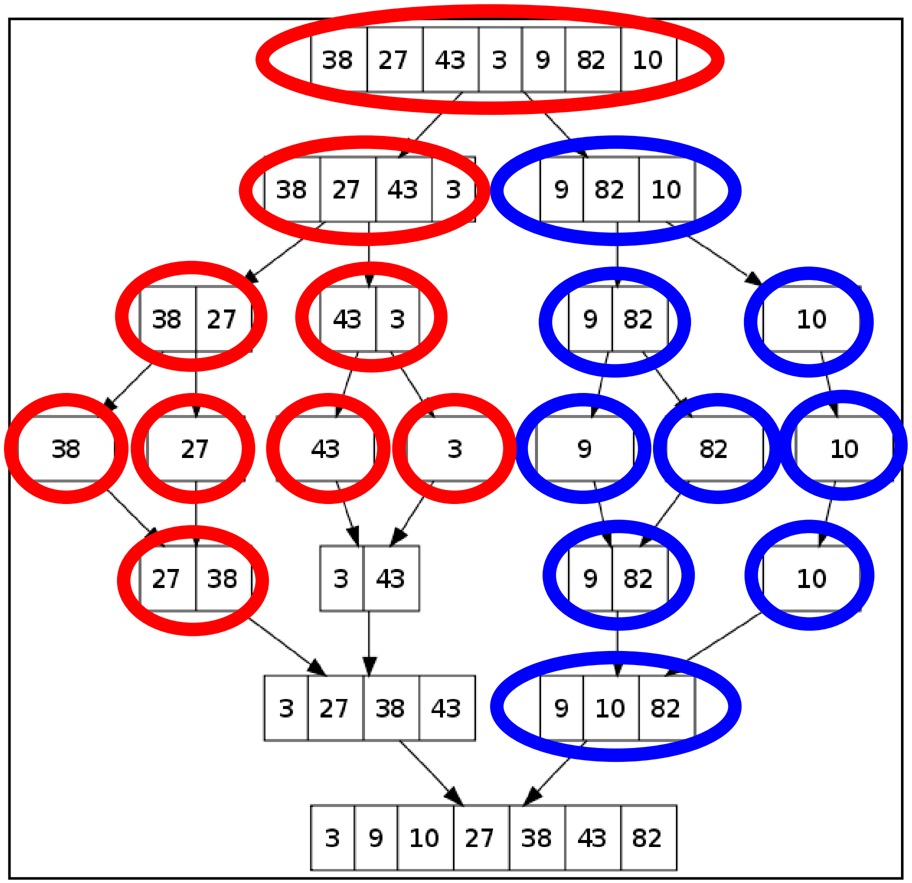
Mergesort Example with Tasks



Thread 0

Thread 1

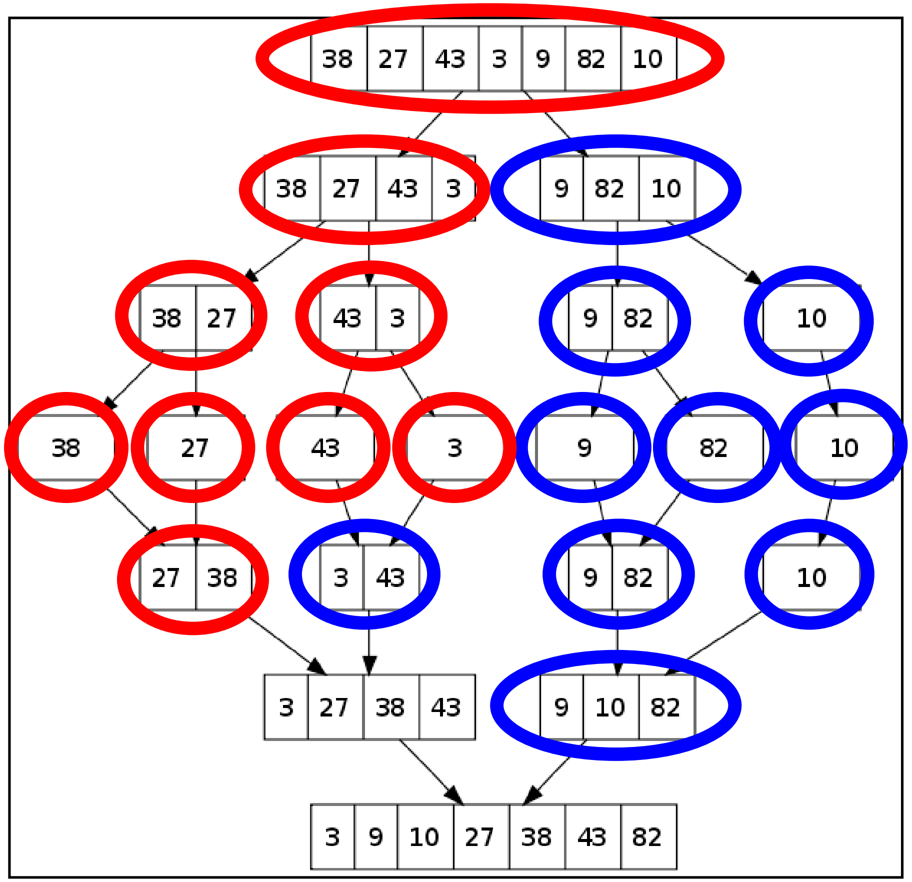
Mergesort Example with Tasks



Thread 0

Thread 1

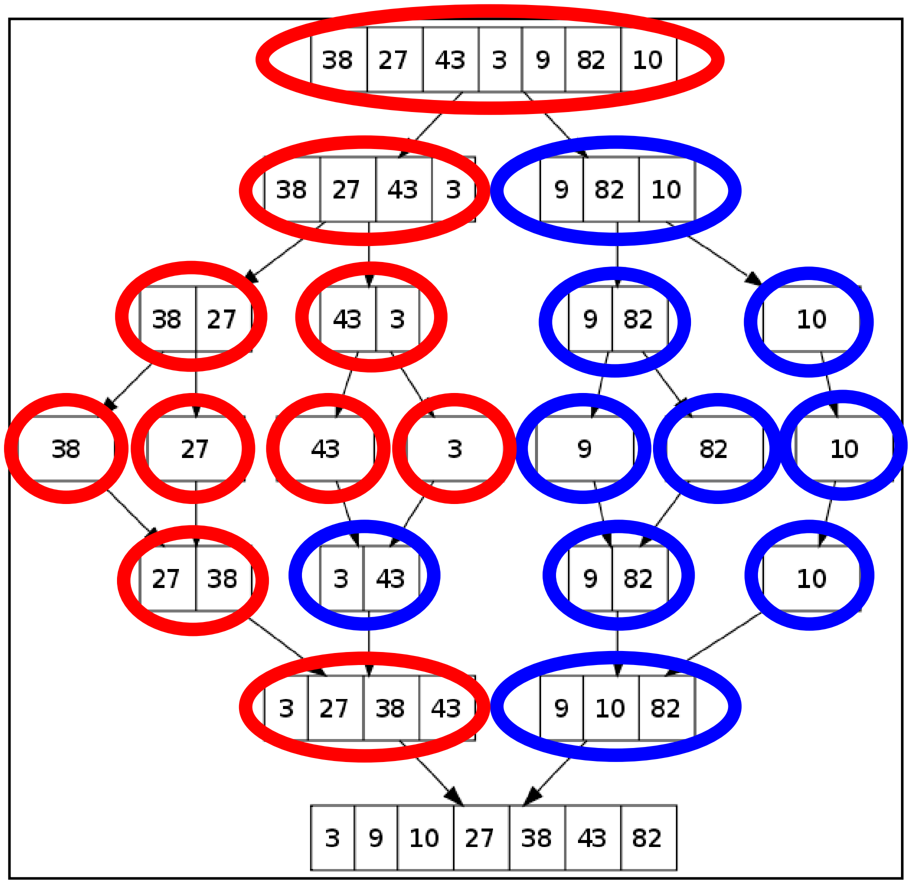
Mergesort Example with Tasks



Thread 0

Thread 1

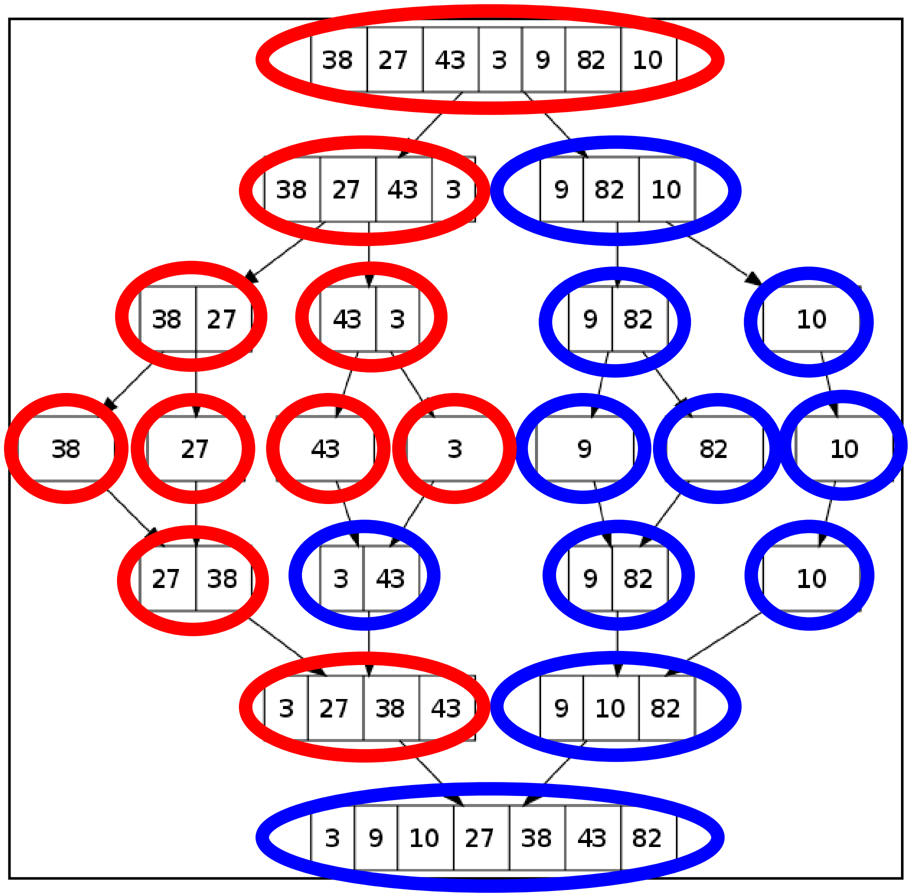
Mergesort Example with Tasks



Thread 0

Thread 1

Mergesort Example with Tasks



Thread 0

Thread 1

A better parallel sort using Cilk

```
cilk void sort(int A[], int n) {
  if (n < 100)
        sort sequentially
  else {
        spawn sort(&A[0],   n/2);
        spawn sort(&A[n/2], n/2);
  }
  sync;
  merge(&A[0], n/2, &A[n/2], n/2);
}

cilk void merge(int A[], int na, int B[], int nb) {
  if (na < 100 || nb < 100)
        merge sequentially
  else {
        int m = binary_search(B, A[na/2]);
        spawn merge(A, na/2, B, m);
        spawn merge(&A[na/2], na/2, &B[m], nb – m);
  }
  sync;
}
```

OpenMP 3.0 includes tasks

- Tasks consist of statements or code blocks
  - basic constructs are task and taskwait

- Works in C, C++, Fortran, supported by many compilers

```
int fib(int n){
   int x, y;

    if (n < 2)
       return n;
    else {
       #pragma omp task
          x = fib(n-1);
       #pragma omp task
          y = fib(n-2);

       #pragma omp taskwait

       return (x+y);
    }
}
```

Scheduling OpenMP Tasks: the Basic Rules

- In general, a task may begin execution on any thread in the team
  - OpenMP does not prescribe a task scheduling strategy
    - generally uses “help first” strategy to create more ready tasks
      - queue the spawned task, and keep going on the parent
      - leads to breadth first evaluation order
    - if(<*cond*>) forces task execution execution when <*cond*> evaluates to true
  - Tied tasks are started on an arbitrary thread and then run to completion in that thread. They can be suspended only at a task spawn or when waiting on a lock.
  - Untied tasks can suspend at any point and may resume on any thread in the team (permits pre-emption – not generally safe)
  - barriers in OpenMP require completion of all outstanding tasks generated by the team of threads encountering the barrier

Scope of variables

- Variables can be shared, threadprivate, or (task) private
  - Shared variables can be accessed concurrently by all tasks
  - Threadprivate variables can be accessed safely within a thread by tied tasks
  - Private variables can only be accessed by the owning task

- Examples where threadprivate variables help
  - Fast summation
  - Dynamic memory allocation

Task parallelism - summary

- Cilk
  - only on Intel systems (and now phased out!)
  - work-first scheduling, generally good for locality
  - cilk_for helps parallelize loops more effectively

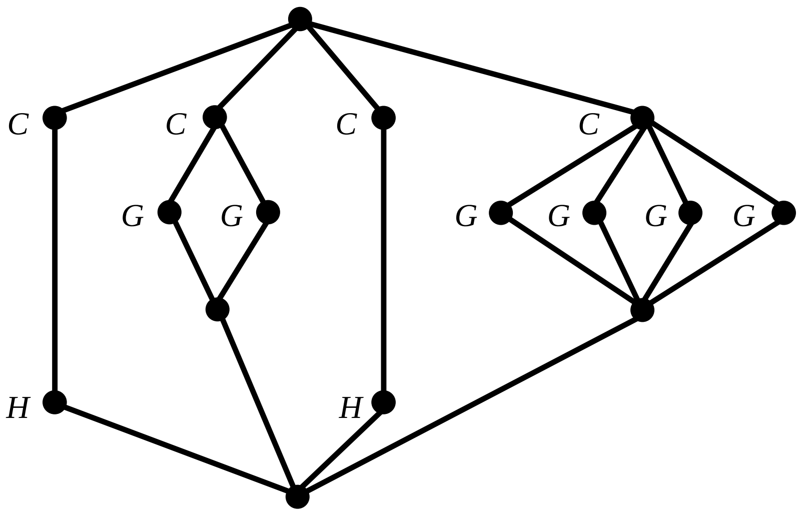
- Open-MP
  - scheduling strategy is not prescribed, generally help-first,
    - not quite as cache-friendly as work-first
  - locality aware schedulers try to schedule tasks on the socket where they were spawned
    - helps increase last-level cache locality

- General
  - task parallelism is well suited to divide & conquer algorithms and irregular parallelism
    - but has higher overheads than pure loop-level parallelization
  - generally insensitive to variation in processor speeds
    - can effectively use hyperthreads and is oblivious to OS interruptions

Nested data parallelism

- Dependence graph reveals available parallelism
  - nodes: computations
  - edges: dependencies
  - dynamic unfolding of graph in execution
    - nested data-parallel loops yield series/parallel graphs

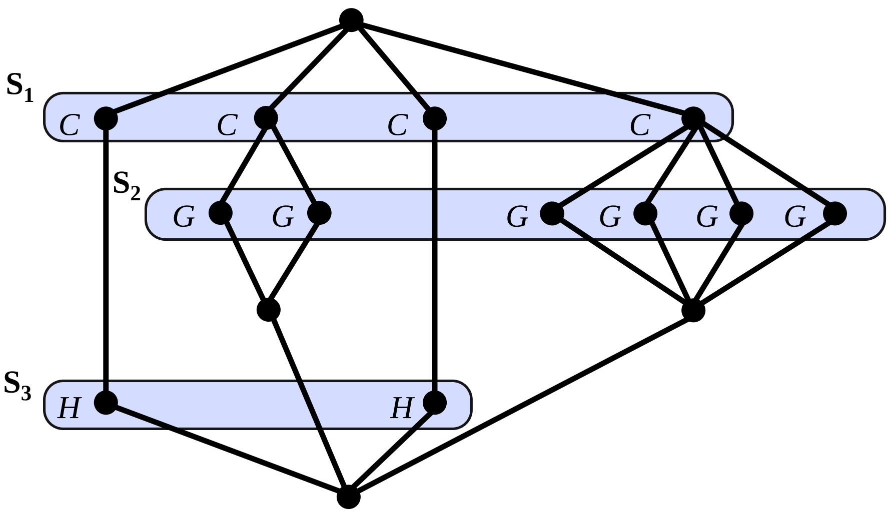
```
FORALL (i = 1,4)
   WHERE C(i) DO
      FORALL (j = 1,i) DO
         G(i,j)
      END FORALL
   ELSEWHERE
      H(i)
   END WHERE
END FORALL
```



Flattening execution strategy

- Each node in the spawn tree is part of a data-parallel operation
  - *flattening* transforms program to a sequence of simple data-parallel operations
    - data-parallel operations have low computational intensity so require high performance parallel memory systems
  - each data-parallel operation is optimally executed using all processors

```
FORALL (i = 1,4)
   WHERE C(i) DO
      FORALL (j = 1,i) DO
         G(i,j)
      END FORALL
   ELSEWHERE
      H(i)
   END WHERE
END FORALL
```



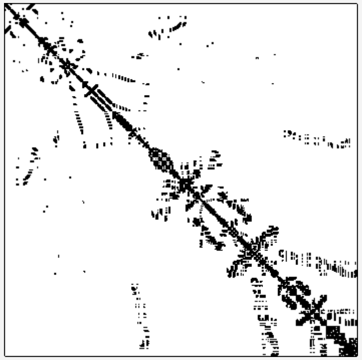
NESL: Sparse matrix-vector product

$R = MV$ where $V, R \in \mathbb{R}^n$ and $M \in \mathbb{R}^{n \times n}$ and M has nz nonzeros

- Nested sequence representation of M
  - Each row is represented by a sequence of pairs
    - (non-zero value a, column index h)
  - M is a sequence of m row representations
- Nested parallel algorithm (NESL)

```
MatVect(M,V) =
    [R in M:
        sum( [(a,h) in R: a * V[h] ] )
    ]
```

a sparse matrix



```
M =
 [
   [(1.0,1), (0.4,3), (0.55,4)],
   [(1.0,2), (0.15,9), (0.18,187)],
      . . .
   [(0.2,3850), (1.0,4165)]
 ]
```

Flattening

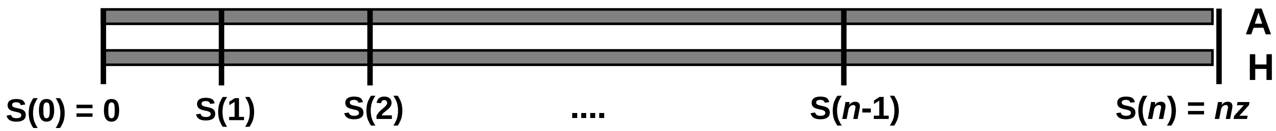
- Compile-time elimination of nested data parallelism
  - Flattening theorem
    - Let F be a set of basic data parallel operations on sequences
    - Let L(F) be a nested data-parallel programming language over F
    - For any program P in L(F), flattening yields a program P' in L(F + F') such that
      - P and P' compute the same function
      - P' contains no nested data-parallel constructs
      - no additional work is introduced and no available parallelism is lost, i.e.

        $W_{P'}(n) = O(W_P(n))$ and $S_{P'}(n) = O(S_P(n))$

  - Example primitives F and F' `V = [1,2,3]` `W = [ [1], [1,2], [1,2,3] ]`

F: $\alpha \to \beta$	F': Seq(α) $\to$ Seq(β)
arithmetic opns	vector arithmetic opns
e.g. `plus(1,1) = 2`	e.g. `plus'(V,V) = [2,4,6]`
`sum(V) = 6`	`sum'(W) = [1,3,6]`
`size(V) = 3`	`size'(W) = [1,2,3]`
`range(3) = [1,2,3]`	`range'(V) = [ [1], [1,2], [1,2,3]]`
`index(V,3)= 3`	`index'(W,V)= [1,2,3]`
`dist(1,3) = [1,1,1]`	`dist'(V,V) = [ [1], [2,2], [3,3,3] ]`

Flattening sparse matrix – vector product



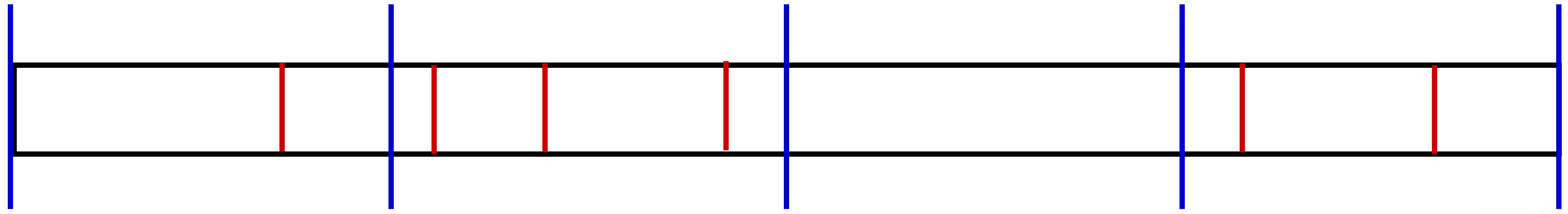

```
#pragma omp parallel do
DO i = 0, n-1
     R(i) = 0
     #pragma omp parallel do reduction(+:R(i))
     DO j = S(i), S(i+1)-1
           R(i) = R(i) + A(j) * V( H(j) )
     ENDDO
ENDDO
```

F77

```
#pragma omp parallel do
DO j = 0, nz-1
     T(j) = A(j) * V( H(j) )
END DO
CALL Segmented_Sum(T,nz,S,R,n)
```

F90

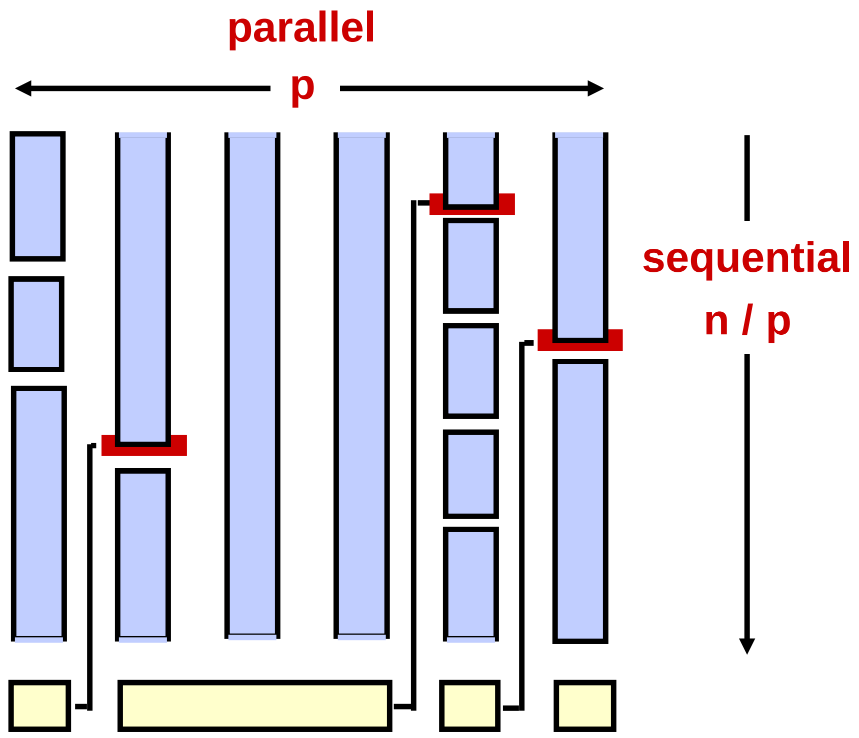
```
R = Segmented_Sum( A * V(H), S )
```



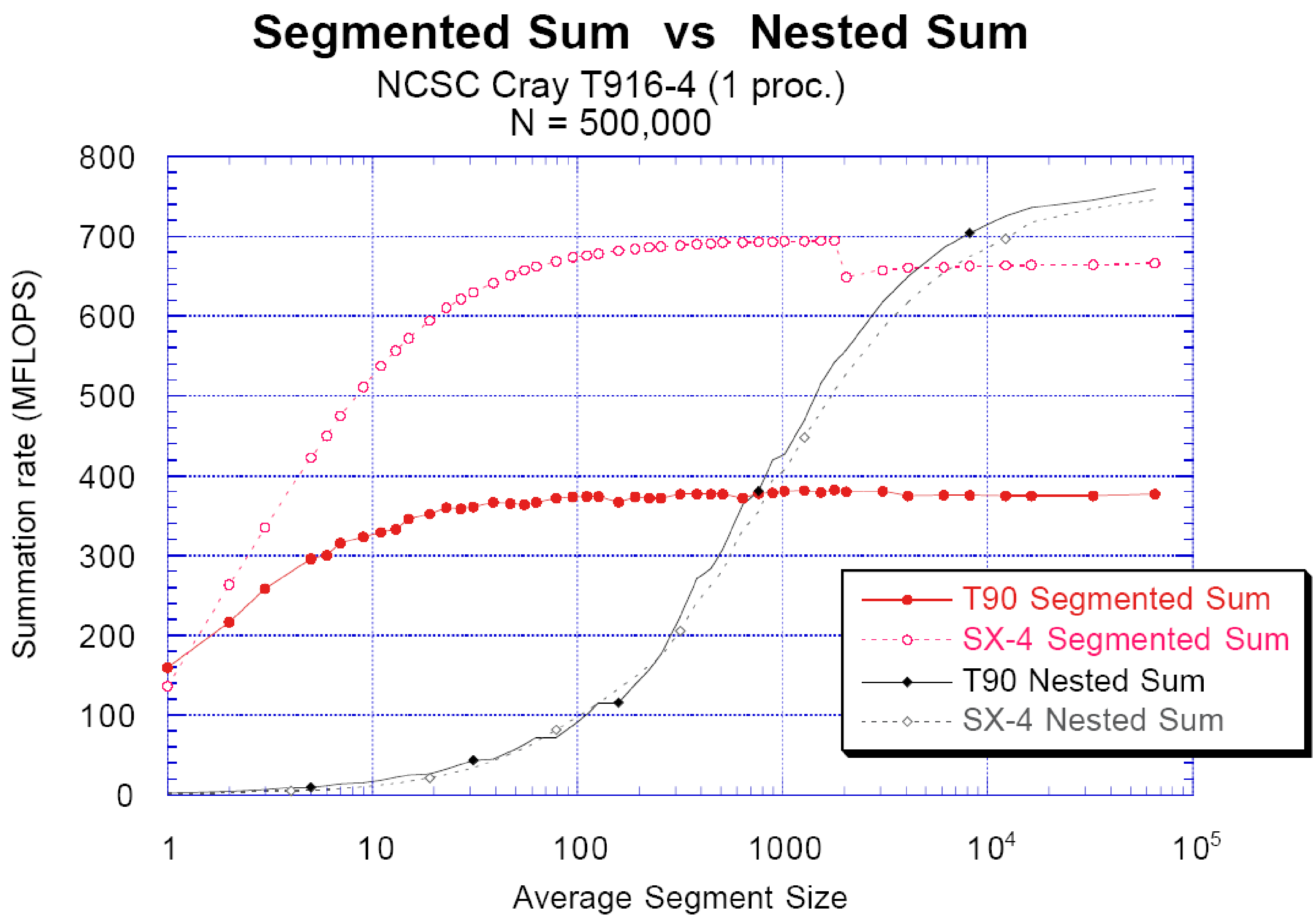
Parallel Implementation of primitives F'

- Goal
  - precise load balance
  - insensitive to
    - number of subproblems
    - size of subproblems

- Example
  - sum' :: Seq(Seq(α)) $\rightarrow$ Seq(α)
  - uses
    - sequential segmented sum of size n/p
    - single parallel segmented sum scan of size p

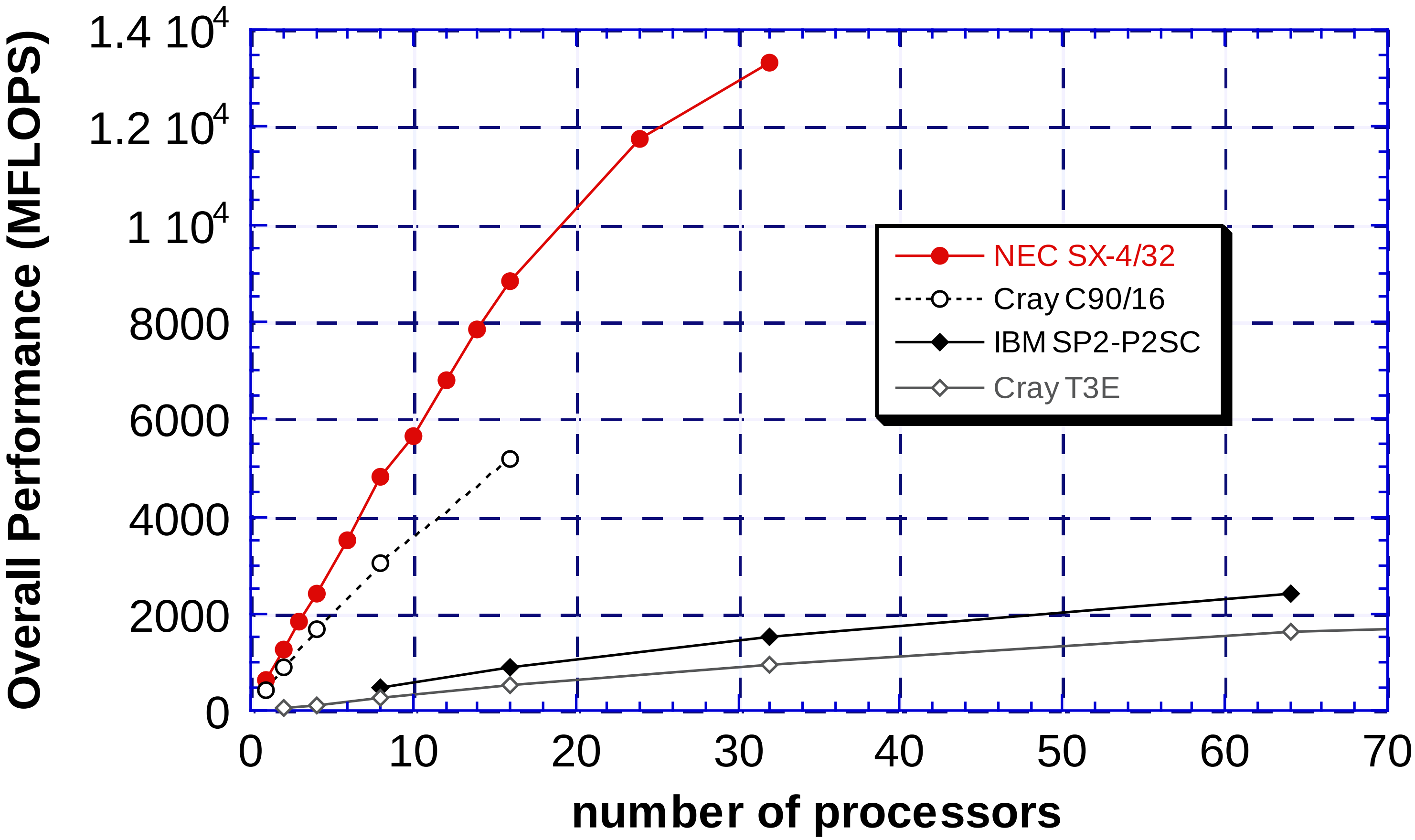


Flattening: Segmented primitives



Flattening: NAS Conjugate Gradient benchmark

- Benchmark: find principal eigenvalue of random sparse linear system using power method
  - repeated use of conjugate gradient method
  - class B benchmark, N = 75,000, average # nz per row = 140, 96% of the work is in sparse matrix – vector product



Comparing execution strategies

- Nested task parallelism
  - few restrictions on program form
  - tasks must be “coarsened” to amortize scheduling overhead
    - load balanced up to granularity of tasks
  - provably good time and space bounds for strict programs
  - can maintain locality (depends on scheduling strategy)

- Nested data parallelism
  - restricted to data parallel programs (subset of all programs)
  - execution is sequence of vector operations
    - easily load-balanced
    - but low computational intensity
  - no run-time scheduler required
  - provably good time bounds, but space bounds are harder