• Reading:
  – The Implementation of the Cilk-5 Multithreaded Language
    • sections 1 – 3
Topics

- Nested parallelism in OpenMP and other frameworks
  - nested 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
Nested loop parallelism

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

```c
#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];
    }
}
```

– what should nested parallel regions mean?
  • each thread in the outer parallel region becomes the master thread of a team of threads in an instance of the inner parallel region

– how will it be executed?
  • most OpenMP implementations allocate all threads to the outer loop
    – so no additional concurrency on the inner loop - when is this a good choice?

– additional consideration
  • Most modern processors have small vector units
    – Accelerate the dot product in the inner loop
Nested parallelism: a more challenging problem

• **sparse** matrix-vector product \( R = MV \)
  - sparse \( M \) is represented using two arrays
    • \( A[nz] \), \( H[nz] \) arrays of non-zero values and column indices
    • \( S[n+1] \) describes the partitioning of \( A \) and \( H \) into \( n \) rows of \( M \)

```c
#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++) {
    }
}
```
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

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

- OpenMP work sharing recognizes nested parallelism in nested loops, but only implements certain cases
  - typically only outermost level of parallelism is realized
  - occasional support for orthogonal iteration spaces
    - e.g. \(\{1, \ldots ,n\} \times \{1, \ldots ,m\}\) 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 parallelism

• 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

fib(4) → fib(3) → fib(2) → fib(1) → fib(0)
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

```
fib(4)    fib(3)    fib(2)    fib(1)    fib(0)
fib(2)    fib(1)    fib(1)    fib(0)
fib(0)    fib(4)    fib(3)    fib(2)    fib(1)
```

<table>
<thead>
<tr>
<th>ready task queues</th>
<th>shallow end</th>
<th>deep end</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cilk execution properties

• Task execution order is parallel depth-first
  – serial order at each processor
  – good fit for parallel memory hierarchy
  – space bound: $\text{Space}_p(n) = \text{Space}_1(n) + pS(n)$

• Global execution time follows bounds determined by Brent’s theorem
  – $T_p(n, p) = O\left( \frac{W(n)}{p} + S(n) \right)$

• 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?

```c
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

```cpp
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
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 can consist of statements or code blocks
  - basic constructs are task and taskwait

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

```c
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
  – 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

```plaintext
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 pipelined parallel memory systems for performance
  - each data-parallel operation is optimally executed using all processors

\[
\begin{align*}
\text{FORALL} & \quad (i = 1, 4) \\
& \quad \text{WHERE } C(i) \ \text{DO} \\
& \quad \quad \text{FORALL} \quad (j = 1, i) \ \text{DO} \\
& \quad \quad \quad G(i, j) \\
& \quad \quad \text{END FORALL} \\
& \quad \text{ELSEWHERE} \\
& \quad \quad H(i) \\
& \quad \text{END WHERE} \\
& \quad \text{END FORALL}
\end{align*}
\]
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)**

\[
\text{MatVect}(M,V) = \left[ R \text{ in } M: \sum \left( (a,h) \text{ in } R: a \times V[h] \right) \right]
\]

\[
M = \begin{bmatrix}
(1,1.0), (3,0.4), (4,0.55), \\
(2,1.0), (9,0.15), (187,0.18), \\
\ddots \\
(3850,0.2), (4165,1.0)
\end{bmatrix}
\]
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:
        $$W_P(n) = O(W_{P'}(n)) \quad \text{and} \quad S_P(n) = O(S_{P'}(n))$$
  - Example primitives $F$ and $F'$
    - $V = [1, 2, 3]$ \hspace{1cm} $W = [[1], [1,2], [1,2,3]]$

<table>
<thead>
<tr>
<th>$F$: $\alpha \rightarrow \beta$</th>
<th>$F'$: $\text{Seq}(\alpha) \rightarrow \text{Seq}(\beta)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic opns</td>
<td>vector arithmetic opns</td>
</tr>
<tr>
<td>e.g. $\mathbf{\text{plus}}(1,1) = 2$</td>
<td>e.g. $\mathbf{\text{plus'}}(V,V) = [2,4,6]$</td>
</tr>
<tr>
<td>$\text{sum}(V) = 6$</td>
<td>$\text{sum'}(W) = [1,3,6]$</td>
</tr>
<tr>
<td>$\text{size}(V) = 3$</td>
<td>$\text{size'}(W) = [1,2,3]$</td>
</tr>
<tr>
<td>$\text{range}(3) = [1,2,3]$</td>
<td>$\text{range'}(V) = [1, [1,2], [1,2,3]]$</td>
</tr>
<tr>
<td>$\text{index}(V, 3) = 3$</td>
<td>$\text{index'}(W,V) = [1,2,3]$</td>
</tr>
<tr>
<td>$\text{dist}(1,3) = [1,1,1]$</td>
<td>$\text{dist'}(V,W) = [1, [2,2], [3,3]]$</td>
</tr>
</tbody>
</table>
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

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

```
#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)
```
Parallel Implementation of primitives $F'$

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

- **Example**
  - $\text{sum'} :: \text{Seq}(\text{Seq}(\alpha)) \rightarrow \text{Seq}(\alpha)$
  - uses
    - sequential segmented sum of size $n/p$
    - single parallel segmented sum scan of size $p$
Flattening: Segmented primitives

Segmented Sum vs Nested Sum
NCSC Cray T916-4 (1 proc.)
N = 500,000

![Graph showing performance comparison between segmented sum and nested sum algorithms.](image-url)
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