**16 core processor**

- Amd Opteron (sixteen-core) Model 6274
- Price: $792.99
- In Stock.

**64 core blade servers ($6K) (shared memory)**

- Amd Opteron (sixteen-core) Model 6274
- Price: $792.99
- In Stock.

\[
\times 4 = 6
\]
Outline

Concurrency vs. Parallelism
Concurrency example
Quicksort example
Nested Parallelism
- fork-join and parallel loops
Cost model: work and span
Techniques:
- Using collections: inverted index
- Divide-and-conquer: merging, mergesort, kd-trees, matrix multiply, matrix inversion, fft
- Contraction: quickselect, list ranking, graph connectivity, suffix arrays

Quad-Core Phones: What to Expect in 2012

Revolutionary a year ago, dual-core mobile processors are now standard; next, chipmakers say, quad-core processors will support mobile multitasking comparable to the performance of a desktop computer.

By Orrin Hess, PCWorld  Dec 11, 2011 8:30 pm
Parallelism in “Real world” Problems

- Optimization
- N-body problems
- Finite element analysis
- Graphics
- JPEG/MPEG compression
- Sequence alignment
- Rijndael encryption
- Signal processing
- Machine learning
- Data mining

Parallelism vs. Concurrency

- **Parallelism**: using multiple processors/cores running at the same time. Property of the machine.
- **Concurrency**: non-determinacy due to interleaving threads. Property of the application.

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>sequential</td>
</tr>
<tr>
<td>parallel</td>
<td>concurrent</td>
</tr>
</tbody>
</table>

**Concurrency : Stack Example 1**

```c
struct link { int v; link* next; }
struct stack {
    link* headPtr;
    void push(link* a) {
        a->next = headPtr;
        headPtr = a;
    }
    link* pop() {
        link* h = headPtr;
        if (headPtr != NULL) {
            headPtr = headPtr->next;
        }
        return h;
    }
}
```

**Concurrency : Stack Example 1**

```c
struct link { int v; link* next; }
struct stack {
    link* headPtr;
    void push(link* a) {
        a->next = headPtr;
        headPtr = a;
    }
    link* pop() {
        link* h = headPtr;
        if (headPtr != NULL) {
            headPtr = headPtr->next;
        }
        return h;
    }
}
**Concurrency : Stack Example 1**

```c
struct link {
    int v;
    link* next;
}

struct stack {
    link* headPtr;
    void push(link* a) {
        a->next = headPtr;
        headPtr = a;
    }
    link* pop() {
        link* h = headPtr;
        if (headPtr != NULL) {
            headPtr = headPtr->next;
            return h;
        }
    }
}
```

**Concurrency : Stack Example 2**

```c
struct stack {
    link* headPtr;
    void push(link* a) {
        do {
            link* h = headPtr;
            a->next = h;
            while (!CAS(&headPtr, h, a));
        } while (!CAS(&headPtr, h, a));
        link* pop() {
            do {
                link* h = headPtr;
                if (h == NULL) return NULL;
                link* nxt = h->next;
                while (!CAS(&headPtr, h, nxt))
                return h;
            }
        }
```
**Concurrency : Stack Example 2**

```
struct stack {
    link* headPtr;

    void push(link* a) {
        do {
            link* h = headPtr;
            a->next = h;
            while (!CAS(&headPtr, h, a));
        } while (!CAS(&headPtr, h, a));

        link* pop() {
            do {
                link* h = headPtr;
                if (h == NULL) return NULL;
                link* nxt = h->next;
                while (!CAS(&headPtr, h, nxt))
                    return h;
            } while (!CAS(&headPtr, h, nxt));
        }
    }
}
```

**Concurrency : Stack Example 2'**

```
P1 : x = s.pop(); y = s.pop(); s.push(x);
P2 : z = s.pop();

Before: A → B → C
After: B → C

The ABA problem
Can be fixed with counter and 2CAS, but...
```

**Concurrency : Stack Example 3**

```
struct link { int v; link* next; }

struct stack {
    link* headPtr;

    void push(link* a) {
        atomic {
            a->next = headPtr;
            headPtr = a;
        }

        link* pop() {
            atomic {
                link* h = headPtr;
                if (h == NULL) return NULL;
                link* nxt = h->next;
                while (!CAS(&headPtr, h, nxt))
                    return h;
            }
        }
    }
}
```
Concurrent Stack Example 3

void swapTop(stack s) {
    link* x = s.pop();
    link* y = s.pop();
    push(x);
    push(y);
}

Queues are trickier than stacks.

Nested Parallelism

nested parallelism =
    arbitrary nesting of parallel loops + fork-join

- Assumes no synchronization among parallel tasks except at joint points.
- Deterministic if no race conditions

Advantages:
- Good schedulers are known
- Easy to understand, debug, and analyze

Nested Parallelism: parallel loops

cilk_for (i=0; i < n; i++)
    B[i] = A[i]+1;

Parallel.ForEach(A, x => x+1);

B = {x + 1 : x in A}

#pragma omp for
for (i=0; i < n; i++)
    B[i] = A[i] + 1;

Nested Parallelism: fork-join

cobegin {
    S1;
    S2;
}

coinvoke(f1,f2)
Parallel.invoke(f1,f2)

#pragma omp sections
{
    #pragma omp section
    S1;
    #pragma omp section
    S2;
}
Nested Parallelism: fork-join

```
spawn S1;
S2;
sync;

(exp1 || exp2)  Various functional
plet
  x = exp1
  y = exp2
in
exp3
```

cilk, cilk+

Serial Parallel DAGs

Dependence graphs of nested parallel computations are series parallel

Two tasks are parallel if not reachable from each other.
A data race occurs if two parallel tasks are involved in a race if they access the same location and at least one is a write.

Cost Model

Compositional:

Work: total number of operations
- costs are added across parallel calls

Span: depth/critical path of the computation
- Maximum span is taken across forked calls

Parallelism = Work/Span
- Approximately # of processors that can be effectively used.

Combining costs

Combining for parallel for:
```
pfor (i=0; i<n; i++)
f(i);
```

```
W_{pexp}(pfor ...) = \sum_{i=0}^{n-1} W_{exp}(f(i))  \quad \text{work}
```

```
D_{pexp}(pfor ...) = \max_{i=0}^{n-1} D_{exp}(f(i))  \quad \text{span}
```

**Why Work and Span**

Simple measures that give us a good sense of efficiency (work) and scalability (span).

Can schedule in $O(W/P + D)$ time on $P$ processors.

This is within a constant factor of optimal.

**Goals in designing an algorithm**

1. Work should be about the same as the sequential running time. When it matches asymptotically we say it is *work efficient*.
2. Parallelism ($W/D$) should be polynomial $O(n^{1/2})$ is probably good enough.

---

**Example: Quicksort**

```plaintext
function quicksort(S) =
if (#S <= 1) then S
else let
    a = S[rand(#S)];
    S1 = {e in S | e < a};
    S2 = {e in S | e = a};
    S3 = {e in S | e > a};
    R = {quicksort(v) : v in [S1, S3]};
in R[0] ++ S2 ++ R[1];
```

How much parallelism?

---

**Quicksort Complexity**

Sequential Partition and appending

Parallel calls

Work = $O(n \log n)$

Span = $O(n)$

Parallelism = $O(\log n)$

Not a very good parallel algorithm

---

*All randomized with high probability*
Quicksort Complexity

Parallel partition
Sequential calls

Span = O(n)
Parallelism = O(log n)

Work = O(n log n)

Not a very good parallel algorithm

*All randomized with high probability

Quicksort Complexity

Parallel partition
Parallel calls

Span = O(lg n)
Parallelism = O(n/log n)

Work = O(n log n)

A good parallel algorithm

*All randomized with high probability

Quicksort Complexity

Caveat: need to show that depth of recursion is O(log n) with high probability

Parallel selection

{e in S | e < a};

S = [2, 1, 4, 0, 3, 1, 5, 7]
F = S < 4 = [1, 1, 0, 1, 1, 1, 0, 0]
I = addscan(F) = [0, 1, 2, 3, 4, 5, 5]

where F
R[I] = S = [2, 1, 0, 3, 1]

Each element gets sum of previous elements. Seems sequential?
**Scan**

![Scan Diagram]

**Scan code**

```plaintext
function addscan(A) =
  if (#A <= 1) then [0]
  else let
    evens = addscan(sums);
    odds = {evens[i] + A[2*i] : i in [0:#a/2]};
  in interleave(evens, odds);

W(n) = W(n/2) + O(n) = O(n)
D(n) = D(n/2) + O(1) = O(log n)
```

**Parallel Techniques**

Some common themes in “Thinking Parallel”
1. Working with collections.
   - map, selection, reduce, scan, collect
2. Divide-and-conquer
   - Even more important than sequentially
     - Merging, matrix multiply, FFT, ...
3. Contraction
   - Solve single smaller problem
     - List ranking, graph contraction
4. Randomization
   - Symmetry breaking and random sampling

**Working with Collections**

redcude ⊕ [a, b, c, d, ...]
= a ⊕ b ⊕ c ⊕ d + ...

scan ⊕ ident [a, b, c, d, ...]
= [ident, a, a ⊕ b, a ⊕ b ⊕ c, ...]

sort compF A

collect [(2,a), (0,b), (2,c), (3,d), (0,e), (2,f)]
= [(0, [b,e]), (2,[a,c,f]), (3,d)]]