Opportunities in Many-Core Algorithms

Guy Blelloch
Carnegie Mellon University
via Michael Scott
via Michael Scott
Parallelism is here... And Growing!

Number of Cores

2006 | 2007 | 2008 | 2009 | 2010 | ... | 2015

Larrabee: 12-32
Nehalem: 8+
Dunnington (6)
Core2 Quad (4)
Core2 Duo (2)

Parallelism for the Masses
"Opportunities and Challenges"

© Intel Corporation

Andrew Chien, 2008
Parallel Algorithms: Where are we at?

**The Good:**

- Lots of great techniques for designing parallel algorithms (contraction, random-mate, cascading, ...)
  most are “robust” across models.
- Many or most problems have good parallel solutions
- Many parallel algorithms have been tested and work well in practice...at least with some engineering
- Good understanding of mapping between models
- Theory of inherently “hard” parallel problems
- Reasonable understanding of locality
Parallel Algorithms: Where are we at?

The Bad:

- No agreed upon model (PRAM, BSP, Circuit, Hypercube, DMM, Scan, ...)
- Programming has diverged from theory
  Shared memory programmers don’t care about procs.
- Some problems still seem hard to parallelize perhaps there is no parallel solution
- Disconnect between “concurrent” and “parallel” algorithms
- Low-level notions of locality
Parallel Algorithms: Where are we at?

The Ugly:

• The theory community has largely given up on doing parallel algorithms and are not keeping up with modern technology and programming technology.

• Many lost opportunities in teaching parallel thinking in our algorithms courses
  • Many algorithms we teach are naturally parallel and we hide it. Others are easy to make parallel and we ignore it.
  • Many of the approaches we teach are naturally parallel (divide-and-conquer)
Today

- Models, programming, and parallel thinking
- High-level notions of locality
- Analyzing transactions
"Bridging" Models

- Memory
  - procs
  - PRAM
    (EREW, CRCW, QRQW, asynch, scan, multiprefix)

- Network
  - procs
  - memory
    (BSP, LogP, hypercube, grid, ...)

- Circuit models

- These are all "hardware" models
Claim

The success of sequential algorithm analysis is largely due to the tight relationship between the standard programming model (e.g. C, Pascal, Java) and the algorithmic cost model (the RAM).

This relationship will **not** be true for the models on the previous page, or probably any hardware model.

It is important to program and analyze algorithms with same thought process.
procedure QUICKSORT(S):
    if S contains at most one element then return S
    else
        begin
            choose an element a randomly from S;
            let $S_1$, $S_2$ and $S_3$ be the sequences of elements in S less than, equal to, and greater than a, respectively;
            return (QUICKSORT($S_1$) followed by $S_2$
                followed by QUICKSORT($S_3$))
        end
Quicksort in NESL

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];
Combining for parallel map:

\[ p_{\text{exp}} = \{ \exp(e) : e \text{ in } A \} \]

\[
W_{p_{\text{exp}}}(A) = \sum_{i=0}^{n-1} W_{\exp}(A_i) \quad \text{work}
\]

\[
D_{p_{\text{exp}}}(A) = \max_{i=0}^{n-1} D_{\exp}(A_i) \quad \text{span}
\]

Can prove runtime bounds for Various models:

\[ T = O(W/P + D \log P) \]
Quicksort in NESL

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];
Parallel selection

\{e \in S \mid e < a\};

\begin{align*}
S &= [2, 1, 4, 0, 3, 1, 5, 7] \\
F = S < 4 &= [1, 1, 0, 1, 1, 1, 0, 0] \\
I = \text{prefixSum}(F) &= [0, 1, 2, 2, 3, 4, 5, 5] \\
\text{where } F &= R[I] = S = [2, 1, 0, 3, 1]
\end{align*}
Prefix Sum (scan) Code

function prefixSum(A) =
if (#A <= 1) then [0]
else let
  E = prefixSum(sums, op);
in interleaved(E, O),

A = [2, 1, 4, 2, 3, 1, 5, 7]
S = [3, 6, 4, 12]
E = [0, 3, 9, 13]
O = [2, 7, 12, 18]
Result = [0, 2, 3, 7, 9, 12, 13, 18]

W(n) = W(n/2) + O(n)
      = O(n)

D(n) = D(n/2) + O(1)
      = O(log n)
Qsort Complexity

Depth of recursion in quicksort is $O(\log n)$ in expectation.

Work = $O(n \log n)$

Span = $O(\log^2 n)$

Expected case
Combining for parallel map:

\[ p_{exp} = \{exp(e) : e \text{ in } A\} \]

\[ W_{p_{exp}}(A) = \sum_{i=0}^{n-1} W_{exp}(A_i) \quad \text{work} \]

\[ D_{p_{exp}}(A) = \max_{i=0}^{n-1} D_{exp}(A_i) \quad \text{span} \]

Can prove runtime bounds for various models:

e.g. \[ T = O(W/P + D) \] for greedy schedules
Formal Cost Model

\[ E \vdash c \xrightarrow{\lambda} c; 1, 1 \]

\[ E \vdash \lambda x.e \xrightarrow{\lambda} cl(E, x, e); 1, 1 \]

\[ \frac{E(x) = v}{E \vdash x \xrightarrow{\lambda} v; 1, 1} \]

\[ E \vdash e_1 \xrightarrow{\lambda} cl(E', x, e'); w_1, d_1 \quad E \vdash e_2 \xrightarrow{\lambda} v_2; w_2, d_2 \]

\[ E'[x \mapsto v_2] \vdash e' \xrightarrow{\lambda} v; w_3, d_3 \]

\[ E \vdash e_1 e_2 \xrightarrow{\lambda} v; w_1 + w_2 + w_3 + 2, \max(d_1, d_2) + d_3 + 2 \]

\[ E \vdash e_1 \xrightarrow{\lambda} c; w_1, d_1 \quad E \vdash e_2 \xrightarrow{\lambda} v_2; w_2, d_2 \]

\[ \delta(c, v_2) = v \]

\[ E \vdash e_1 e_2 \xrightarrow{\lambda} v; w_1 + w_2 + 2, \max(d_1, d_2) + 2 \]
Quicksort in Cilk++

```c
seq quicksort(seq S) {
    if (S.length < 2) return S;
    double a = S[rand(S.length)];
    seq S1, S2, S3;
    cilk_spawn S1 = quicksort(lessThan(S,a));
    cilk_spawn S2 = eqTo(S,a);
    S3 = quicksort(grThan(S,a));
    cilk_sync;
    return S1.append(S2.append(S3));
}
```
Observations

- Quicksort as written by AHU is already naturally parallel.
- Cost analysis closely matches the high-level description of algorithm.
- Never mentioned processor once in the model or analysis.
- No parameters, just multiple cost measures.
- Fully formal model based on “generic” language.
- Lots of work on scheduling this form of parallelism onto processors (work stealing, PDF), opportunities for much more.
- Robust with respect to changing number of processors.
Merging

fun merge(A, B) =
   let
      Node(A_L, m, A_R) = A
      (B_L, B_R) = split(B, m)
   in
      Node(Merge(A_L, B_L), m, Merge(A_R, B_R))

Merge in parallel

\[
\begin{align*}
W(n) &= 2W(n/2) + O(\log n) \\
     &= O(n)
\end{align*}
\]

\[
\begin{align*}
D(n) &= D(n/2) + O(\log n) \\
     &= O(\log^2 n)
\end{align*}
\]
fun split (s, empty) = (empty ,empty)
| split (s, node(v, L, R)) =
  if s < v then
    let val (L1 ,R1) = split(s ,L)
    in (L1,node(v, R1, R)) end
  else
    let val (L1,R1) = split(s ,R)
    in (node (v, L, L1), R1) end;
fun merge(A, B) = 
  let 
  Node(A_L, m, A_R) = A 
  (B_L, B_R) = split(B, m) 
  in 
  Node(Merge(A_L, B_L), m, Merge(A_R, B_R))

\[
W(n) = 2W(n/2) + O(\log n) = O(n)
\]
\[
D(n) = D(n/2) + O(\log n) = O(\log^2 n)
\]
fun fsplit (s, empty) = (empty, empty)
  | fsplit (s, node(v, L, R)) = 
    if s < v then
      let val (L1, R1) = future(fsplit(s, L))
          in (L1, node(v, R1, R)) end
    else
      let val (L1, R1) = future(fsplit(s, R))
          in (node(v, L, L1), R1) end;
function merge(A,B) =
    let
        Node(A_L, m, A_R) = A
        (B_L, B_R) = fsplit(B, m)
    in
        Node(Merge(A_L,B_L), m, Merge(A_R,B_R))

\[
W(n) = 2W(n/2) + O(\log n) = O(n)
\]

\[
D(n) = O(\log(n))
\]
Today

- Models, programming, and parallel thinking
- High-level notions of locality
- Analyzing transactions
High-level notions of locality

Machines do seem to require some locality, but cost of programming is often more important than performance. Cannot burden users.

Most models of locality are either binary (local, non-local), cumbersome, or architecture specific.

High-level model might be portable.
High-level notions of locality

IDEA 1: Leverage work on cache-oblivious algorithms

Approach:
A. Design nested parallel algorithms as described with
   1. Good cache oblivious behavior in seq. order
   2. Low span, optimal work
   Note 3 cost measures: Q, D, W and still “no parameters”
B. Use general bounds to bound parallel cache misses, e.g.
   \( Q_p(Z+PD,L) \leq Q_1(Z,L) \) for a shared cache
   e.g. Quicksort: \( Q_1(n;Z,L) = O(n/L \log (n/Z)) \)
High-level notions of locality

Another very simple model:
1. Nested parallel as before
2. \( Z = \) cache size
3. \( L = \) line size
4. For a function call with footprint of size \( F \),
   \[ Q = F \text{ if } F < Z \text{ and } \text{parent}(F) > Z \]
   \[ Q = 0 \text{ otherwise} \]
Concurrent and Parallel Algorithms

Atomic regions and transactional Memory

```c
int FetchAdd(int* p, int v) {
    atomic {
        t = *p;
        *p = t + v; }
    }

    atomic {
        a = Q.dequeue();
        D.insert(a); }

100s (1000s) of papers on the topic in past 5 or 6 years
```
Concurrent and Parallel Algorithms

Example:

```
parallel for i = 2 to n
    atomic {
        t = a[i-1];
        a[i] = t;
    }
```

Need evaluation methodology and bounds on time.
Conclusions

Opportunities:
• Still many algorithms which we don’t know how to parallelize
• Programming-based cost models
• Scheduling
• High-level locality models