Implementation of a Portable Nested Data-Parallel Language

Guy E. Blelloch
Jonathan C. Hardwick
Jay Sipelstein
Marco Zagha
Carnegie Mellon University

Siddhartha Chatterjee
RIACS

Data Parallel Languages

Advantages of data-parallel languages

- Fine-grained parallelism at a low cost
- Portable
- Concise and clear algorithmic descriptions
- Deterministic
- Easy to debug and analyze

Complaints against them

Only applicable to a certain class of applications. **Not** good algorithms on:

- Sparse matrices and graphs
- Trees (e.g. tree search, Barnes-Hut)
- Complex pointer-based data structures

The NESL Language

Main Goal:

Keep the advantages of data-parallel languages while extending the range of applications.

Problem with existing DP languages:

Parallel collections (arrays) can only be of atomic types or fixed size structures.

User functions cannot be called in parallel

Nested Parallelism:

- Allow nested structures
- Allow nested parallel calls

CM-Lisp is a nested parallel language

NESL Operations

Sequences as a parallel data-type

```
A = [5 2 0 9 3]    int sequence
B = "porcini"      char sequence
```

Laid out across the processors in block order

Elementwise operations

```
{e + 3: e in A} -> [8 5 3 12 6]
```

Subselection

```
{e in B| e < `n} -> "cii"
```

Parallel functions

```
sum(A)       -> 19
rotate(A,1) -> [3 5 2 0 9]
"big " ++ B -> "big porcini"
```

Nested Parallel Operations

```
A = [[2, 5, 7, 9],[3],[11, 5, 8]]
```

Nested parallel call:

```
{sum(e) : e in A} -> [23, 3, 24]
```

User defined functions:

```
function norm(vect) =
  sqrt(sum({x^2: x in vect}));

norm([2,6,3])      -> 7
{norm(e) : e in A} -> [12.6, 3, 14.5]
```

Intrinsics work on nested sequences:

```
rotate(A,1)

  -> [[11, 5, 8],[2, 5, 7, 9],[3]]
```

Nested Structures

Sparse Matrices:

$$
\begin{bmatrix}
x & x & & x & & \\
& x & & x & & \\
x & & x & x & & x \\
& & & x & & \\
& & x & & x & \\
& & & & & x
\end{bmatrix}
$$

```
[[(1, 3.5) (2, .9) (4, 2.2)]
 [(2, 4.1) (4, 1.9)]
 [(1, 4.6) (3, .7) (4, 2.7) (6, 3.0)]
 [(4, 2.9)] ...]
```

```
{sum({vct[i]*val:(val,i) in r}: r in m}
```

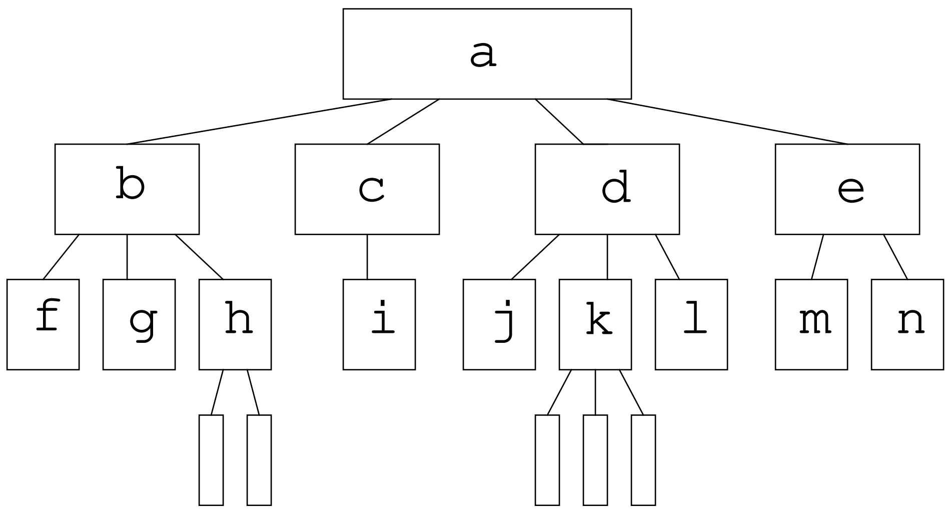
Text: (words, sentences, lines)

```
[“Parallelism” “saves” “the” “world”..]
```

```
{w in words: w[1] = ‘a}
```

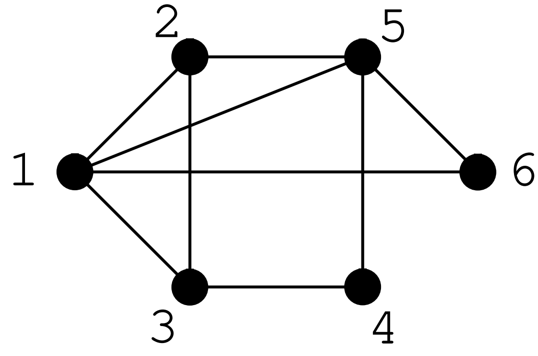
Trees:

```
(a,[(b,[(f,[]),(g,[]),(h,[(o,[]),..)]),
     (c,[(i,[])]) .....
```



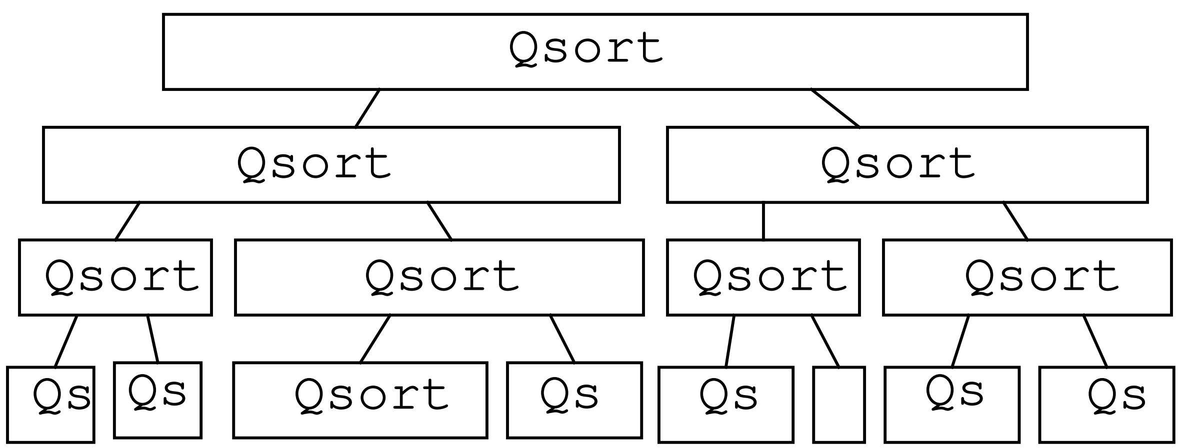
Graphs: (adjacency lists)

```
[[2, 3, 5, 6],[1, 5],[1, 2, 4],[3, 5],
 [1, 2, 4, 6],[1, 5]]
```



Quicksort

```
function qsort(s) =
if (#s < 2) then s
else
  let
    pivot = s[rand(#s)];
    les = {e in s| e < pivot};
    eql = {e in s| e == pivot};
    grt = {e in s| e > pivot};
    result = {qsort(v):v in [les, grt]}
  in result[0] ++ eql ++ result[1];
```



Implementation

Nested parallelism sounds great, but can it be implemented efficiently?

Possible methods:

- Only implement bottom level
  This is what CM-Lisp does
  Not enough parallelism
- Create a job for every parallel call
  Overhead is too large
- Preschedule
  Don't know sizes ahead of time.

Seems as hard as implementing a control parallel language.

Will we loose portability?

Flattening Nested Parallelism

```
A =[[2, 4, 7, 9], [3], [4, 5, 1]]

A.seg  = [4, 1, 3]
A.vals = [2, 4, 7, 9, 3, 4, 5, 1]
```

- Uses segmented functions to operate within segments.
- Uses packs to implement conditionals.

Advantages:

- Multiple levels of parallelism
- No job creation overhead
- Locality (segments can be within a proc).

Potential Problem: too much parallelism

Can be used for any structure and any level of nesting.

```
B = [[(“red”,1),
      (“green”,2),
      (“blue”,2),
     ],
     [(“heavy”,3),
      (“light”,1)
     ]]
```

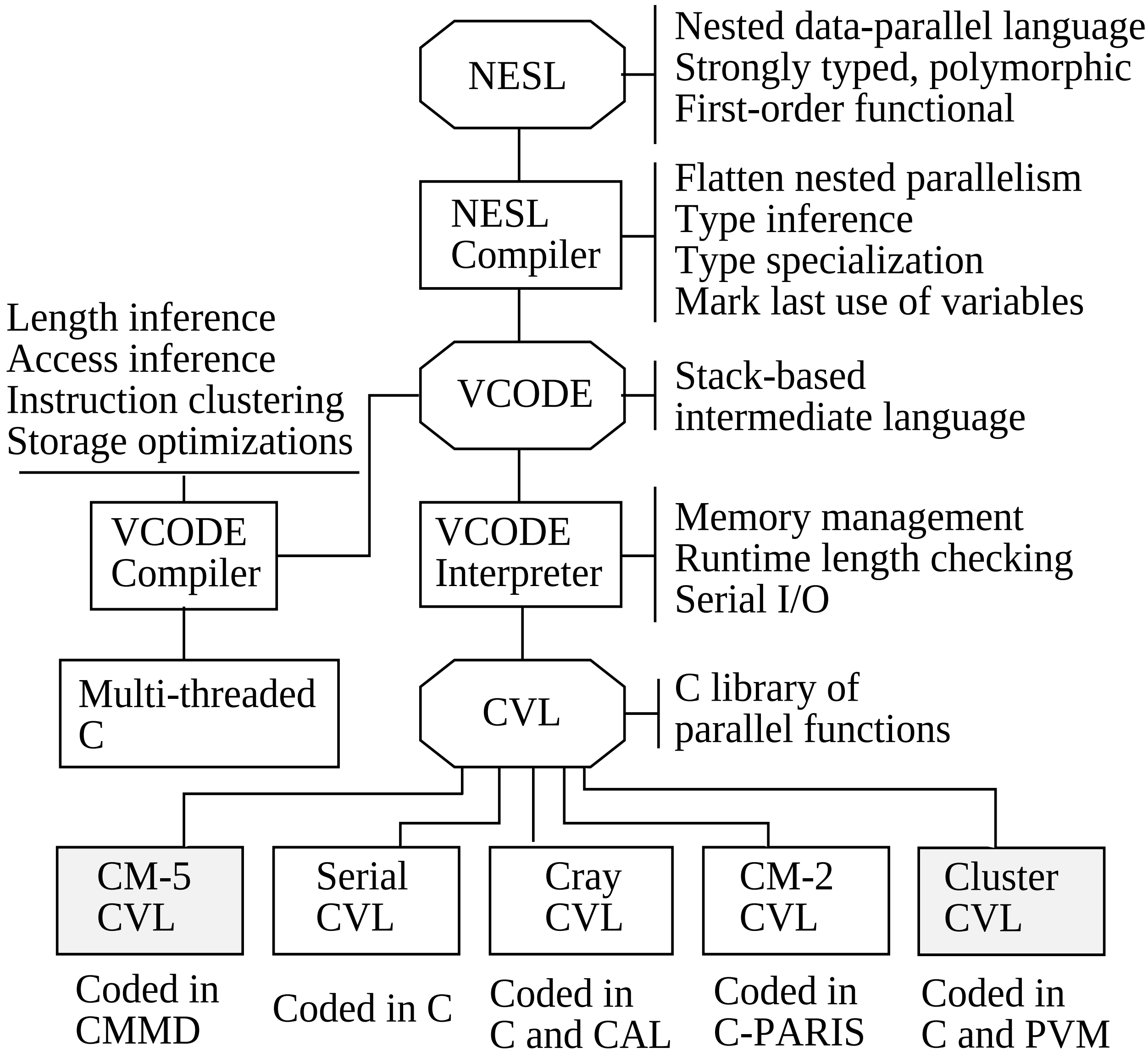
```
B.seg    = [3, 2]
B.v1.seg = [3, 5, 4, 5, 5]
B.v1.v   = “redgreenblueheavylight”
B.v2     = [1, 2, 2, 3, 1]

{{w[2]: (w,c) in words}: words in B}
```

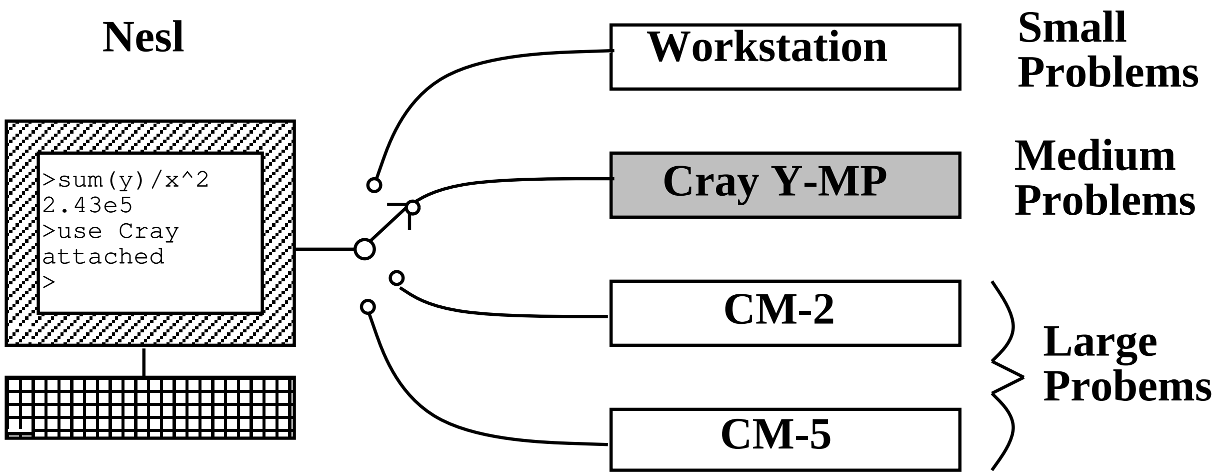
will become

```
gather(B.v1.v, 2 + B.v1.segoffset)
```

System Overview



Environment



About 50 users. Medium size applications:

- Geometric and spectral separators
- Ray tracer , volume renderer , map database
- Speech recognition

```
> #primes(1000000);
78498: int
> use_machine(Cray);
> #primes(1000000);
78498: int
```

Benchmarks

Benchmarks used:

- Least square line fit
- Median (quickmedian algorithm)
- Sparse Matrix-Vector Multiply

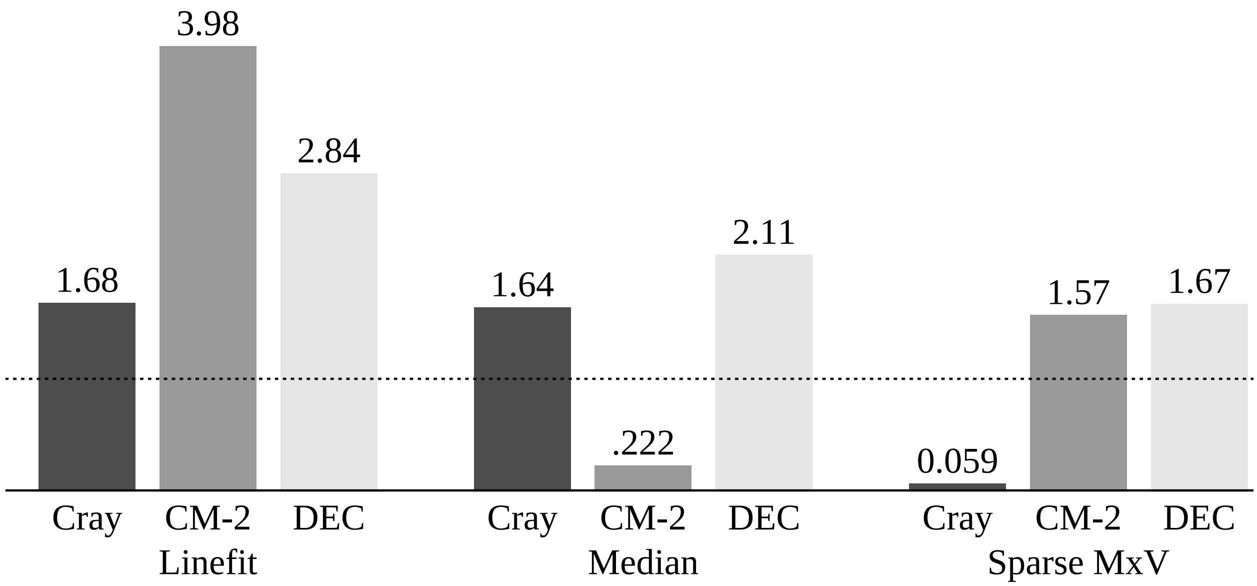
Machines used:

- CM-2 (Nesl vs. CM-Fortran)
- Cray C90 (Nesl vs. Fortran 77)
- Dec 5000 (Nesl vs. C)

Goals of benchmarks: *to measure*

1. performance of nested-parallel code
2. interpretive overhead
3. success of portability

Results



- Factor of 1.5 - 4 interpretive overhead.
- Better on the CM-2 for median because of dynamic load balancing.
- Better than the Cray on Sparse MxV because of nested parallelism.

Details in paper

Status

- Available via anonymous FTP
  (Send email to blelloch@cs.cmu.edu)
- About 50 users
- Used in a course on programming parallel algorithms
- 1week course at Dartmouth
  June 23 -- June 29

Existing Applications:

- Geometric and spectral separators
- Ray tracer , volume renderer
- Pittsburgh map database
- Speech recognition

Line Fitting

```
function linefit(x, y) =
let
  n    = float(#x);
  xa   = sum(x)/n;
  ya   = sum(y)/n;
  Stt  = sum({(x - xa)^2: x});
  b    = sum({(x - xa)*y: x; y})/Stt;
  a    = ya - xa*b;
  chi2 = sum({(y-a-b*x)^2: x; y});
  siga = sqrt((1.0/n+xa^2/Stt)*chi2/n);
  sigb = sqrt((1.0/Stt)*chi2/n)
in
  (a, b, siga, sigb);
```

Median

```
function select_kth(s, k) =
let pivot = s[#s/2];
    les = {e in s | e < pivot}
in
  if (k < #les) then
    select_kth(les, k)
  else
    let grt = {e in s | e > pivot}
    in if (k >= #s - #grt) then
         select_kth(grt,k - (#s - #grt))
       else pivot;

function median(s) =
  select_kth(s,#s/2);
```

Sparse-Matrix Vector Product

```
function MxV(Mval, Midx, Mlen, Vect) =
let v = get(Vect, Midx);
    p = {a * b: a in Mval; b in v}
in
  {sum(row) : row in nest(p, Mlen)};
```