Programming Parallel Language-NESL

Qingyue Liu
Outline

- Introduction
- Features of NESL language
- Model of parallel computation
- Parallel algorithms using NESL
- Efficiency analysis of NESL
- Trend in parallel languages
Different hardware → Different characteristics
  ◦ Ex: GPU is efficient for data parallelism
Need programming models to express abstractions appropriate for HW
A parallel programming model enables users to express programs in a form that enables them to be compiled for execution on parallel systems.
Parallel Programming Models

- **Task parallelism**
  - Emphasize concurrent execution of arbitrary tasks on the same or different data
  - Classified as MIMD/MPMD or MISD

- **Data parallelism**
  - Emphasize concurrent execution of identical tasks on different data elements
  - Classified as SIMD/SPMD
What is NESL?

- NESL: data parallel language developed at Carnegie Mellon by SCandAL project in 1993
- Aim: make parallel programming easy, efficient and portable
New Ideas in NESL

- Nested data parallelism
  - Expressive model for efficient parallel programming

- A language based performance model
  - Support performance analysis independent of any target machine
Features of NESL

- Apply-to-each construct
- Function on sequence
- Nested data parallelism
Features: Apply-to-each Construct

- **Use a set-like notation**
  - e.g. \(\{a*a: a \text{ in } [3, -4, -9, 5]\}\); 
  - Result: [9, 16, 81, 25]

- **Can be applied over multiple sequences**
  - e.g. \(\{a+b: a \text{ in } [3, -4, -9, 5]; b \text{ in } [1, 2, 3, 4]\}\); 
  - Result: [4, -2, -6, 9]

- **Can subselect elements of a sequence**
  - e.g. \(\{a: a \text{ in } [3, -4, -9, 5] \mid a>0\}\); 
  - Result: [3, 5]
Features: Apply-to-each Construct

- Can apply any function to the elements in a sequence
  - e.g. define function:

    ```
    function factorial(n)=
        if(n==1) then 1
        else n*factorial(n-1);
    ```

  - Apply it over elements in a sequence

    ```
    {factorial(a): a in [3, 1, 7]};
    ```

  - Result: [6, 1, 5040]
A set of function on sequences, each of which can be implemented in parallel (sum, reverse, write)

Example: write
- e.g. write([0, 0, 0, 0, 0, 0, 0, 0],[4,-2),(2,5),(5,9)]);
- Result: [0, 0, 5, 0, -2, 9, 0, 0]
Features: Nested Data-parallelism

- **Flat data-parallel vs. nested data-parallel**
  - **Flat**: function can be applied over a set of values
  - **Nested**: parallel function can be applied over a set of values

- **Nested parallelism:**
  - Allow sequences to be nested and allow parallel functions to be used in an apply-to-each

- **Example**: Sum elements over each row in a matrix
  - \{\text{sum}(a): a \text{ in } [[2, 3], [8, 3, 9], [7]]\}
  - Result: [5, 20, 7]
Models of Parallel Computation
Models of Parallel Computation

- **Processor-based models**: Calculate performance in terms of the number of instruction cycles in a computation
  - Function of input size and number of processors

- **Virtual models**: High-level model that can be applied to various machines
  - PRAM
  - Work & depth
Parallel Random Access Machine (PRAM)

- The parallel-computing analogy to the random access machine
- **PRAM Model**
  - P processors connected to a single shared memory
  - All processors perform each instruction simultaneously in unit time
  - Processors may access memory locations simultaneously
PRAM Variants

- Exclusive read exclusive write (EREW)
- Concurrent read exclusive write (CREW)
- Exclusive read concurrent write (ERCW)
- Concurrent read concurrent write (CRCW)
CRCW PRAM

- *Common*—all processors write the same value
- *Arbitrary*—only one arbitrary attempt is successful
- *Priority*—processor rank indicates who gets to write
- *Combining*—the value stored is combination of the values written (+ or max)
PRAM is a Virtual Model

- Viewed as a Virtual Models:
  - Unrealistic to build directly:
    - Assume no limit on the shared memory
    - Assume no limit on the number of processors
    - Assume every processor can access a shared memory in unit time
  - Can be mapped on more realistic machines efficiently
    - Ex: simulating multiple processors of the PRAM on a single processor of a host machine
Work & Depth

- **Work**
  
  The total number of operations executed by computation

- **Depth**
  
  The longest chain of sequential dependencies in the computation

Ex: Summing 16 numbers in a tree

Work: \(n-1\)
Depth: \(\log_2 n\)
Why Work & Depth?

- Easy to describe
- Easy to think about
- Easy to analyze algorithm in terms of work and depth rather than in terms of running time and number of processors
- Perform analysis is closely related to the code and the code provides a clear abstraction of parallelism
**NESL Language-based Model**

- **Work/depth model using NESL language constructs**
- **Specify**
  - Cost of primitive operations
  - Set of rules for composing costs across expressions
- **Work** = sum of work of the tasks
- **Depth** = maximum of the depth of the task
- **Advantage:** describe running time without a machine model
For expression e1+e2

\[ W(e_1 + e_2) = 1 + W(e_1) + W(e_2) \]
\[ D(e_1 + e_2) = \max(D(e_1), D(e_2)) \]

For an apply-to-each expression

\[ W(e_1(a) : a \text{ in } e_2) = 1 + W(e_2) + \sum_{a \text{ in } e_2} W(e_1(a)) \]
\[ D(e_1(a) : a \text{ in } e_2) = 1 + D(e_2) + \max_{a \text{ in } e_2} D(e_1(a)) \]

For if-then-else

\[ W(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \]
\[ = 1 + W(e_1) + \begin{cases} W(e_2) & e_1 = \text{true} \\ W(e_3) & \text{otherwise} \end{cases} \]
Example: The Performance Model

- **Factorial:**
  
  ```
  function factorial(n)=
    if(n==1) then 1
    else n*factorial(n-1);
  e={factorial(n): n in a}, a=[3,1,5,2]
  ```

- **Analyze:**

  \[
  W_{\text{fact}}(n) = \begin{cases} 
  1 & n = 1 \\
  1 + W_{\text{fact}}(n-1) & n > 1
  \end{cases}
  \]

  \[
  W_{\text{fact}}(n) = D_{\text{fact}}(n) = 5n - 2
  \]

  Work: 
  \[
  = 1 + \text{sum}(W_{\text{fact}}(3), W_{\text{fact}}(1), W_{\text{fact}}(5), W_{\text{fact}}(2))
  = 48
  \]

  Depth: 
  \[
  = 1 + \text{max}(D_{\text{fact}}(3), D_{\text{fact}}(1), D_{\text{fact}}(5), D_{\text{fact}}(2))
  = 24
  \]
NESL Examples
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
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];

Work = O(n log n) (expected)
Depth = O(log n) (expected)
Primes Algorithm

- Primes
  - Sieve of Eratosthenes (sequential):
    1. procedure PRIMES(n):
    2. let A be an array of length n
    3. set all but the first element of A to TRUE
    4. for i from 2 to sqrt(n)
    5. begin
    6. if A[i] is TRUE
    7. then set all multiples of i up to n to FALSE
    8. end

  - Serial work: $O(n \log \log n)$
Primes Parallelized

- Parallelize the line “set all multiples of i up to n to FALSE”
- Multiples of a value $i$ can be generated in parallel by $[2*i:n:i]$ and can be written into the array $A$ in parallel with the $write$ function
- $Work = O(n \log \log n)$
- $Depth = O(\sqrt{n})$
Primes Parallelized

- Outer parallelism: `sqr_primes`
- Inner parallelism: Generate the multiples of each prime
- Depth = $O(\log \log n)$
- Work = $O(n \log \log n)$

```plaintext
function primes(n) =
if n == 2 then ([1] int)
else
    let sqr_primes = primes(isqrt(n));
    composites = {2*p:n:p: p in sqr_primes};
    flat_comps = flatten(composites);
    flags = write(dists(true, n), {(i,false): i in flat_comps});
    indices = {i in [0:n]: fl in flags ! fl}
    in drop(indices, 2);
```

Example for primes(20):
- `sqr_primes` = [2,3]
- `composites` = [[4,6,8,10,12,14,16,18] , [6,9,12,15,18]]
- `flat_comps` = [4,6,8,10,12,14,16,18,6,9,12,15,18]
- `flags` = [t,t,t,t,f,t,f,f,t,f,t,f,t,f,t,t]
- `indices` = [0,1,2,3,5,7,11,13,17,19]
- `result` = [2,3,5,7,11,13,17,19]
NESL
Implementation
NESL Implementation Overview

NESL
- Nested data-parallel language
- Strongly typed, polymorphic
- First-order functional

NESL Compiler
- Flatten nested parallelism
- Type inference
- Type specialization
- Mark last use of variables

VCODE
- Stack-based intermediate language

VCODE Compiler

VCODE Interpreter
- Memory management
- Runtime length checking
- Serial I/O

CVL
- C library of parallel functions

Multi-threaded C

Serial CVL
- Coded in C

Cray CVL
- Coded in C and CAL

CM-2 CVL
- Coded in C-PARIS

CM-5 CVL
- Coded in C and CMMD

Cluster CVL
- Coded in C and PVM
NESL Implementation Overview

NESL
- Nested data-parallel language
- Strongly typed, polymorphic
- First-order functional

NESL Compiler
- Flatten nested parallelism
- Type inference
- Type specialization
- Mark last use of variables

VCODE
- Stack-based intermediate language
- Memory management
- Runtime length checking
- Serial I/O

VCODE Compiler
- C library of parallel functions

VCODE Interpreter
- Coded in C
- Coded in C and CAL
- Coded in C and PARIS
- Coded in C and CMMD
- Coded in C and PVM

Multi-threaded C
- Serial CVL
- Cray CVL
- CM-2 CVL
- CM-5 CVL
- Cluster CVL

Length inference
Access inference
Instruction clustering
Storage optimizations

http://www.cs.cmu.edu/~scandal/nesl/nesl_form.html
Experiments
Advantages and Disadvantages of NESL

- Interpreter overhead
- Dynamic allocation
- Nested parallelism
Experiments: Least-squares Line-fit

- **Least-squares line-fit**
  - x and y are sequences of x and y coordinates
  - Returns the intercept, slope and their uncertainties
- **Characteristics:**
  - Little communication
  - No nested parallelism
  - Known size vectors
  - Statically allocate
- **Goal:**
  - Measure the overhead incurred by interpreter-based implementation

```python
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);
```
Experiments: Median Finding

- **Median find:**
  - Function `select-kth` returns the kth smallest element of s
  - Median used `select-kth` to return middle element

- **Characteristics:**
  - Require dynamic memory allocation
  - Need redistributed on each iteration

- **Goal:**
  - Demonstrate the utility and efficiency of NESL’s dynamic allocation

```python
def select_kth(s, k):
    pivot = s[int(len(s)/2)]
    les = {e in s | e < pivot}
    in
    if k < len(les):
        select_kth(les, k)
    else:
        grt = {e in s | e > pivot}
        in
        if k >= len(s) - len(grt):
            select_kth(grt, k - (len(s) - len(grt))
        else:
            pivot;
    return

def median(s):
    return select_kth(s, len(s)/2);
```
Experiments: Sparse-matrix Vector Product

Sparse-matrix vector product:

Characteristics:
- Irregular data structures
- High communication

Goal:
- Demonstrate the power and efficiency of nested parallelism

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

(0 0 0 0) (10) (30)
(0 0 2 0) (20) (60)
(4 0 0 2) (30) (120)
(3 1 0 0) (40) (50)
```

```
Vect = [10 20 30 40]
Midx = [0 2 0 3 0 1]
Vec->Midx = [10 30 10 40 10 20]
Mval = [3 2 4 2 3 1]
p = [30 60 40 80 30 20]
Mlen = [1 1 2 2]
nest(p,Mlen) = [[30] [60] [40] [80] [30] [20]]
rowsums = [30 60 120 50]
```
## Experiments: Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Communication</th>
<th>Dynamic Structures</th>
<th>Nested Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Fit</td>
<td>Low</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Median</td>
<td>High</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Sparse MxV</td>
<td>High</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Experiments: Hardware

- **Alpha benchmarks**: DEC 3000 AXP Model 400 with 32Mbytes of memory
- **Cray C90 benchmarks**: One processor of a C90/16 with 256 Mwords of memory
- **CM-2 benchmarks**: 32K processors of a CM-2 with 1Gbyte of memory
- **CM-5 benchmarks**: 256 processors of a CM-5 with 8Gbyte memory
# Result: Performance Summary

- **Performance of NESL relative to native code in detail**

## Running Times in (Seconds) of the Benchmarks for **NESL** and Native Code

<table>
<thead>
<tr>
<th>$n$</th>
<th>Alpha C</th>
<th>Alpha NESL</th>
<th>C90 F77</th>
<th>C90 NESL</th>
<th>CM-2 CMF</th>
<th>CM-2 NESL</th>
<th>CM-5 CMF</th>
<th>CM-5 NESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>0.0007</td>
<td>0.0029</td>
<td>0.0001</td>
<td>0.0012</td>
<td>0.0018</td>
<td>0.0061</td>
<td>0.0008</td>
<td>0.0063</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>0.0137</td>
<td>0.0468</td>
<td>0.0004</td>
<td>0.0018</td>
<td>0.0019</td>
<td>0.0061</td>
<td>0.0011</td>
<td>0.0063</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td><strong>0.2869</strong></td>
<td><strong>0.9506</strong></td>
<td><strong>0.0058</strong></td>
<td><strong>0.0122</strong></td>
<td><strong>0.0037</strong></td>
<td><strong>0.0133</strong></td>
<td><strong>0.0057</strong></td>
<td><strong>0.0095</strong></td>
</tr>
<tr>
<td>$2^{22}$</td>
<td></td>
<td></td>
<td><strong>0.0927</strong></td>
<td><strong>0.1551</strong></td>
<td><strong>0.0322</strong></td>
<td><strong>0.1283</strong></td>
<td><strong>0.1473</strong></td>
<td><strong>0.1658</strong></td>
</tr>
</tbody>
</table>

**Line fit**

<table>
<thead>
<tr>
<th>$n$</th>
<th>Alpha C</th>
<th>Alpha NESL</th>
<th>C90 F77</th>
<th>C90 NESL</th>
<th>CM-2 CMF</th>
<th>CM-2 NESL</th>
<th>CM-5 CMF</th>
<th>CM-5 NESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>0.0004</td>
<td>0.0059</td>
<td>0.0001</td>
<td>0.0059</td>
<td>0.0293</td>
<td>0.1017</td>
<td>0.0086</td>
<td>0.0376</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>0.0068</td>
<td>0.0773</td>
<td>0.0005</td>
<td>0.0092</td>
<td>0.0623</td>
<td>0.1442</td>
<td>0.0215</td>
<td>0.0544</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td><strong>0.1347</strong></td>
<td><strong>0.4070</strong></td>
<td><strong>0.0080</strong></td>
<td><strong>0.0233</strong></td>
<td><strong>0.2667</strong></td>
<td><strong>0.2163</strong></td>
<td><strong>0.2146</strong></td>
<td><strong>0.0945</strong></td>
</tr>
<tr>
<td>$2^{22}$</td>
<td></td>
<td></td>
<td><strong>0.1276</strong></td>
<td><strong>0.2099</strong></td>
<td><strong>3.7810</strong></td>
<td><strong>8.8389</strong></td>
<td><strong>8.2092</strong></td>
<td><strong>6.5664</strong></td>
</tr>
</tbody>
</table>

**Median**

<table>
<thead>
<tr>
<th>$n$</th>
<th>Alpha C</th>
<th>Alpha NESL</th>
<th>C90 F77</th>
<th>C90 NESL</th>
<th>CM-2 CMF</th>
<th>CM-2 NESL</th>
<th>CM-5 CMF</th>
<th>CM-5 NESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>0.0002</td>
<td>0.0009</td>
<td>0.0002</td>
<td>0.0003</td>
<td>0.0043</td>
<td>0.0142</td>
<td>0.0012</td>
<td>0.0012</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>0.0049</td>
<td>0.0088</td>
<td>0.0037</td>
<td>0.0006</td>
<td>0.0063</td>
<td>0.0152</td>
<td>0.0020</td>
<td>0.0035</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td><strong>0.1503</strong></td>
<td><strong>0.2186</strong></td>
<td><strong>0.0589</strong></td>
<td><strong>0.0038</strong></td>
<td><strong>0.0295</strong></td>
<td><strong>0.0451</strong></td>
<td><strong>0.0175</strong></td>
<td><strong>0.0259</strong></td>
</tr>
<tr>
<td>$2^{22}$</td>
<td></td>
<td></td>
<td><strong>0.9436</strong></td>
<td><strong>0.0557</strong></td>
<td><strong>0.4098</strong></td>
<td><strong>0.6436</strong></td>
<td><strong>0.2791</strong></td>
<td><strong>0.2929</strong></td>
</tr>
</tbody>
</table>

**Sparse-matrix vector product**

*Note. The sparse-matrix vector product uses a row length of 5 and randomly selected column indices. CM-5 NESL results are preliminary.*
Result: Performance Summary

- Performance of NSEL relative to native code

1.0 = native code, smaller numbers are therefore better
Results: Interpreter Overhead

- Interpreter overhead for the line-fitting benchmark

- Interpreter overhead is a main source of inefficiency
  - Cost executing interpreter itself
  - The granularity of the operations performed by the library is too fine

- The vertical lines show sizes at which overhead is 50% of time

- NESL can require large input for close to peak efficiency
Results: Dynamic Load Balancing

- CM-2 median: NESL vs. CM Fortran
- Median algorithm reduced the number of active elements on each step
- In NSEL implementation:
  - Vectors are dynamically allocated
  - The data automatically balanced across the processors
Results: Nested Parallelism

- Runtime of Sparse-matrix vector product for different sparsity
  - NSEL outperforms the native version by over a factor of 10
  - The compilation of nested data parallelism generates code with running time independent of the size of the sub-sequences

The number of nonzero entries in each sparse matrix is fixed at $10^6$
Trend in Parallel Languages

- Task Parallelism
  - Cilk
  - pthreads
  - OpenMP
  - TBB

- Data Parallelism
  - NESL Programming Language
  - NVIDIA CUDA
  - Intel
  - OpenCL
  - ArBB

1993 2007 2008 2010
**Impact**

- Nested data parallelism is an expressive parallel programming model
- The work and depth model is natural and influential
- Experiments show that NESL can achieve high efficiency on both distributed and shared memory systems
  - Interpreted NESL is comparative with native code for large problems
- Greatly influenced data parallelism on GPU
References

- [http://www.cs.cmu.edu/~scandal/nesl/nesl_form.html](http://www.cs.cmu.edu/~scandal/nesl/nesl_form.html)