



INTEL Software Network:

Parallel programming was once the sole concern of extreme programmers worried about huge supercomputing problems. With the emergence of multi-core processors for mainstream applications, however, parallel programming is well poised to become a technique every professional software developer must know and master.

<http://softwarecommunity.intel.com/articles/eng/1252.htm>

INTEL Software Network:



Parallel programming can be difficult, ... It includes all the characteristics of more traditional, serial programming, but in parallel programming, there are three additional and well defined steps

- * Identify parallelisms: Analyze a problem to identify tasks that can execute in parallel**
- * Expose parallelisms: Restructure a problem so tasks can be effectively exploited. This often requires finding the dependencies between tasks and organizing the source code so they can be effectively managed.**
- * Express parallelisms: Express the parallel algorithm in source code using a parallel programming notation**

INTEL Software Network:



Parallel programming can be difficult, ... It includes all the characteristics of more traditional, serial programming, but in parallel programming, there are three additional and well defined steps



*** Identify parallelisms: Analyze a problem to identify tasks that can execute in parallel**

*** Expose parallelisms: Restructure a problem so tasks can be effectively exploited. This often requires finding the dependencies between tasks and organizing the source code so they can be effectively managed.**

*** Express parallelisms: Express the parallel algorithm in source code using a parallel programming notation**

INTEL Software Network:



Parallel programming can be difficult, ... It includes all the characteristics of more traditional, serial programming, but in parallel programming, there are three additional and well defined steps



*** Identify parallelisms: Analyze a problem to identify tasks that can execute in parallel**






*** Expose parallelisms: Restructure a problem so tasks can be effectively exploited. This often requires finding the dependencies between tasks and organizing the source code so they can be effectively managed.**

*** Express parallelisms: Express the parallel algorithm in source code using a parallel programming notation**

INTEL Software Network:



Parallel programming can be difficult, ... It includes all the characteristics of more traditional, serial programming, but in parallel programming, there are three additional and well defined steps

-  *** Identify parallelisms: Analyze a problem to identify tasks that can execute in parallel**
-  *** Expose parallelisms: Restructure a problem so tasks can be effectively exploited. This often requires finding the dependencies between tasks and organizing the source code so they can be effectively managed.**
-  *** Express parallelisms: Express the parallel algorithm in source code using a parallel programming notation**

- **SequenceL was developed at Texas Tech University, funded by NASA**
 - ❖ Developed over a 20 year period by Dr. Daniel Cooke at Texas Tech.
 - ❖ Over \$10 million of NASA grants.
 - ❖ Over 50 man years of research.
- **An executable language for documenting software requirements.**
 - ❖ SequenceL transparency met NASA documentation requirements.
 - ❖ Being executable enabled NASA to eliminate prototyping.
 - ❖ NASA continues funded research at Texas Tech with objective to develop on board navigation and control software.
- **Texas Multicore Technologies was based on the SequenceL Technology.**
 - ❖ Formed in 2009.
 - ❖ Licensed SequenceL technology from Texas Tech University.
 - ❖ Commercialized as a parallelizing development tool for the multicore industry.

SequenceL Overview



Simple

Transparent

High Order / Functional / Declarative

Statically Typed

Strict Evaluation

Turing Complete

15 or so Grammar Rules

Two computational laws:

- **Consume-Simplify-Produce**
- **Normalize Transpose**



Tableau

Series of Tableaus comprise the execution of a problem solution

Example:

$$(20-5) * ((10-3) + (6 / 2))$$

$$15 * (7 + 3)$$

$$15 *$$

$$10$$

$$150$$

Example: Recursion free of charge

$\text{fact}(n(0)) := \text{fact}(n-1) * n$ when $n > 1$ else 1;

$\text{fact}(3)$

$\text{fact}(3-1) * 3$ when $3 > 1$ else 1

$\text{fact}(3-1) * 3$ when true else 1

$\text{fact}(3-1) * 3$

$\text{fact}(2) * 3$

$(\text{fact}(2-1) * 2$ when $2 > 1$ else 1) * 3

$(\text{fact}(2-1) * 2$ when true else 1) * 3

$\text{fact}(2-1) * 2 * 3$

$\text{fact}(1) * 2 * 3$

$(\text{fact}(1-1) * 1$ when $1 > 1$ else 1) * 2 * 3

$(\text{fact}(1-1) * 1$ when false else 1) * 2 * 3

$1 * 2 * 3$

6

Possible Simplification Step of CSP

[1, 2, 3] * 5

[1*5, 2*5, 3*5]

Normalize: [[1, 2, 3], [* , * , *], [5, 5, 5]]

Transpose: [1*5, 2*5, 3*5]

[5, 10, 15]

SCALES!

$[[1, 2, 3], [4, 5, 6], [7, 8, 9]] * 2$

$[[1, 2, 3] * 2, [4, 5, 6] * 2, [7, 8, 9] * 2]$

$[[1 * 2, 2 * 2, 3 * 2], [4 * 2, 5 * 2, 6 * 2], [7 * 2, 8 * 2, 9 * 2]]$

$[[2, 4, 6], [8, 10, 12], [14, 16, 18]]$

SCALES! - Maximally Overtyped

[[1, 2, 3], [4, 5, 6], [7, 8, 9]] * [10, 20, 30]

[[1, 2, 3]*[10, 20, 30], [4, 5, 6]*[10, 20, 30], [7, 8, 9]*[10, 20, 30]]

[[1*10, 2*20, 3*30], [4*10, 5*20, 6*30], [7*10, 8*20, 9*30]]

[[10, 40, 90], [40, 100, 180], [70, 160, 270]]

SCALES! - Functions

`fact(n(0)) := fact(n-1) * n when n>1 else 1;`

`fact([1, 2, 3, 4])`

`[fact(1), fact(2), fact(3), fact(4)]`

`[1, 2, 6, 24]`

The tableau, combined with the inherent properties of the CSP and the NT, presents a tremendous opportunity for parallelisms to be generated automatically

Experiments

This section presents comparison data from experiments focused on the respective speeds of SequenceL and parallel Haskell on three problems: matrix multiplication, word search, and Quicksort.

The data we present in this paper uses the Haskell compiler GHC version 6.10.1 running on Xeon Dual Quad Core Processors.

The SequenceL compiler is written in a sequential version of Haskell and generates multi-threaded C++ code for the same machine.

Both languages have a runtime component for the multi-core processors. For each experiment we ran 20 trials on each configuration (i.e., 20 on 1 processor, 20 on 2 processors, etc.).

Experiments

Matrix Multiplication

The matrix multiplication was performed in both languages on a 1000 x 1000 matrix. We found the parallel Haskell version on a Haskell website (www.haskell.org).

```
multMat :: [[Int]] -> [[Int]] -> [[Int]]
multMat m1 m2 = (multMatT m1 (transpose m2))
```

```
multMatT :: [[Int]] -> [[Int]] -> [[Int]]
multMatT m1 m2T =
[[multVec row col|col <- m2T]|row <- m1]
```

```
multVec :: [Int] -> [Int] -> Int
multVec v1 v2 = sum (zipWith (*) v1 v2)
```

```
multMatPar :: Int -> [[Int]] -> [[Int]] -> [[Int]]
multMatPar z m1 m2 =
(multMat m1 m2) `using` strat z
```

Parallel Haskell



```
strat = blockStrat
lineStrat c = parListChunk c rnf
blockStrat c matrix -- best?
    = let blocks = concat
      (splitIntoClusters numB matrix) -- result splitted
                                     -- in numB * numB blocks
numB = round (sqrt (fromIntegral (length
matrix) / fromIntegral c))
-- approx. same num/granularity of sparks as in others...
    in parList rnf block

type Vector = [Int]
type Matrix = [Vector]

splitIntoClusters :: Int -> Matrix -> [[Matrix]]
splitIntoClusters c m | c < 1 = splitIntoClusters 1 m
splitIntoClusters c m1 = mss
    where bh = kPartition (length m1) c
bhsplit [] [] = []
bhsplit [] _ = error
"some elements left over"
```



Parallel Haskell



```
bhsplit (t:ts) xs = hs : (bhsplit ts rest)
                    where (hs,rest) = splitAt t xs

ms = bhsplit bh m1 -- blocks of rows
mss = map (colsplit bh) ms
colsplit [] _ = []
colsplit (t:ts) rs
  | head rs == [] = []
  | otherwise =
(cab:colsplit ts resto)
    where (cab,resto) = unzip
(map (splitAt t) rs)
-- helper for splitIntoClusters (formerly bresenham)
kPartition :: Int -> Int -> [Int]
kPartition n k = zipWith (+) ((replicate (n `mod` k) 1) ++ repeat 0)
  (replicate k (n `div` k))
```



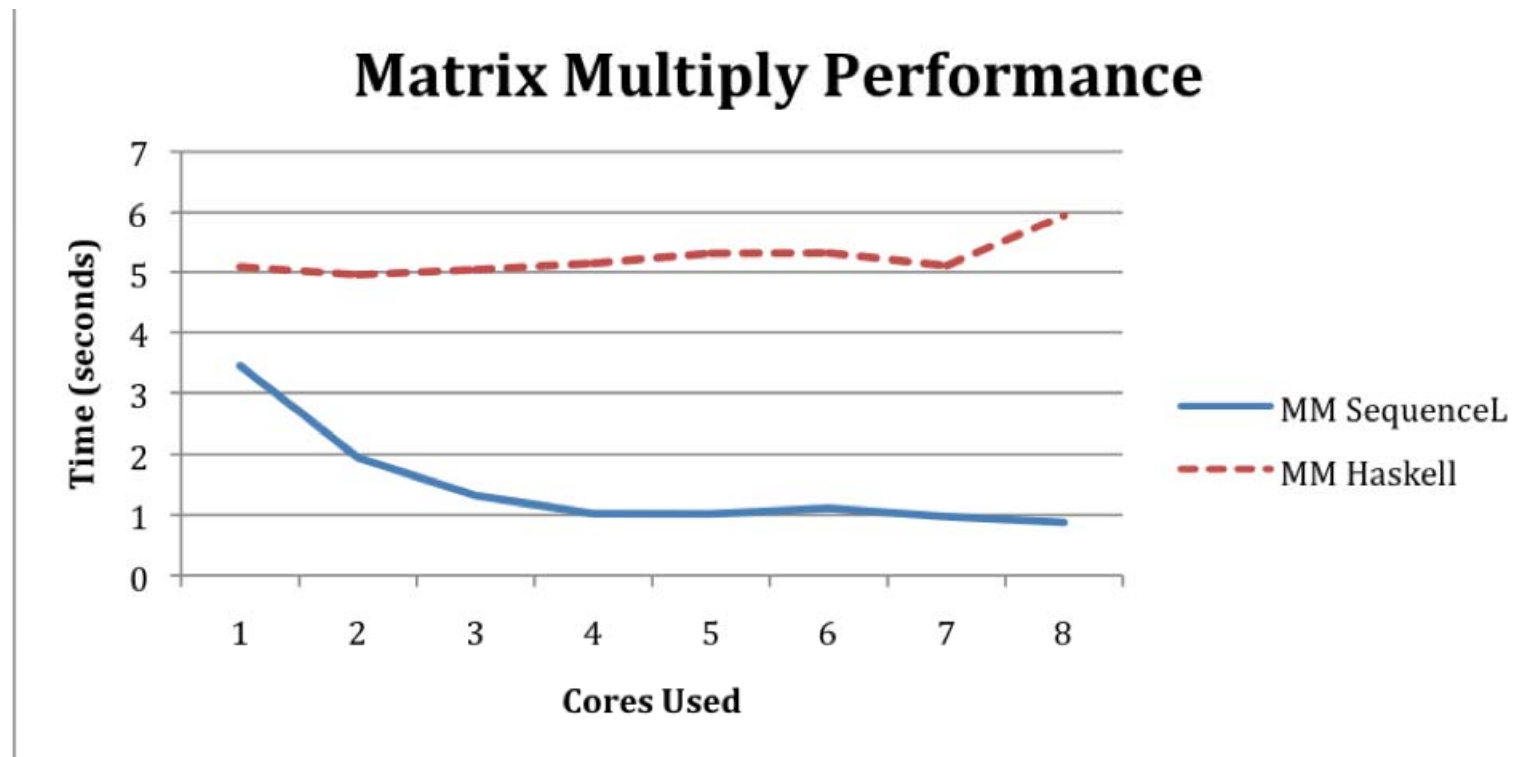
SequenceL



```
matmul(x(2), y(2))[i, j] := sum( x[i, all] * y[all, j]);
```



Performance Results



Experiments

Word Search

We were unable to find parallel Haskell codes for Word Search. So our best Haskell programmer wrote the Parallel Haskell version of a simple Word Search (i.e., no regular expressions). We mention this, because we expect that there are better performing versions in parallel Haskell.

We experimented with adding `par` and `seq` commands to different parts of the program and show the results for the version with the best performance.

The Word Search was performed on a 10,000,000 character file searching for a 5 character word.

```
grep :: [String] -> String -> [String]
```

```
grep lines key = filter (substring key) lines
```

```
substring :: String -> String -> Bool
```

```
substring [] _ = True
```

```
substring _ [] = False
```

```
substring (x:xs) (y:ys) = checkFront `par` (checkRest `pseq` (checkFront ||  
checkRest))
```

where

```
checkFront = isPrefix (x:xs) (y:ys)
```

```
checkRest = substring (x:xs) ys
```

```
isPrefix :: String -> String -> Bool
```

```
isPrefix [] _ = True
```

```
isPrefix _ [] = False
```

```
isPrefix (x:xs) (y:ys) = (x==y) && (isPrefix xs ys)
```

SequenceL



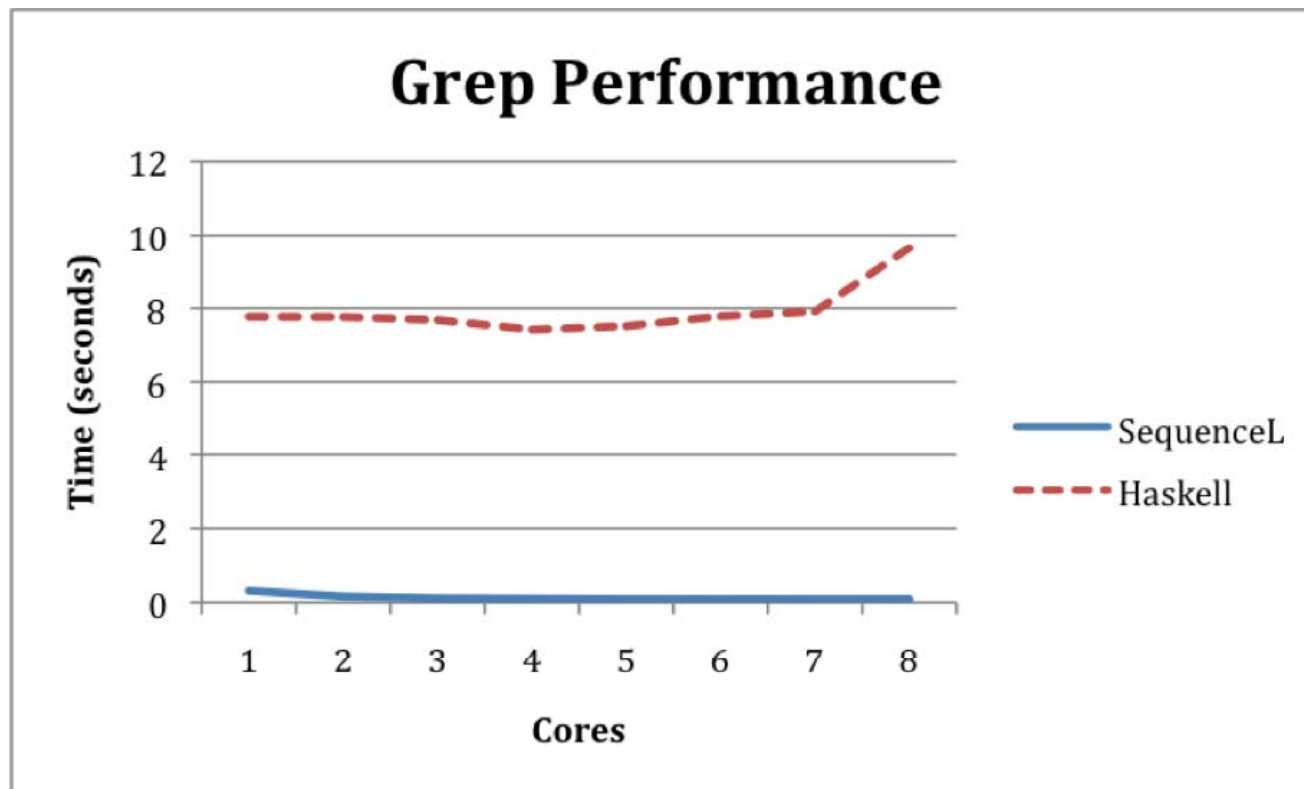
```
ws(a(1), b(1)) := word_search(a, b, 1...(size(a)-size(b)+1));
```

```
word_search(a(1), b(1), n) :=
```

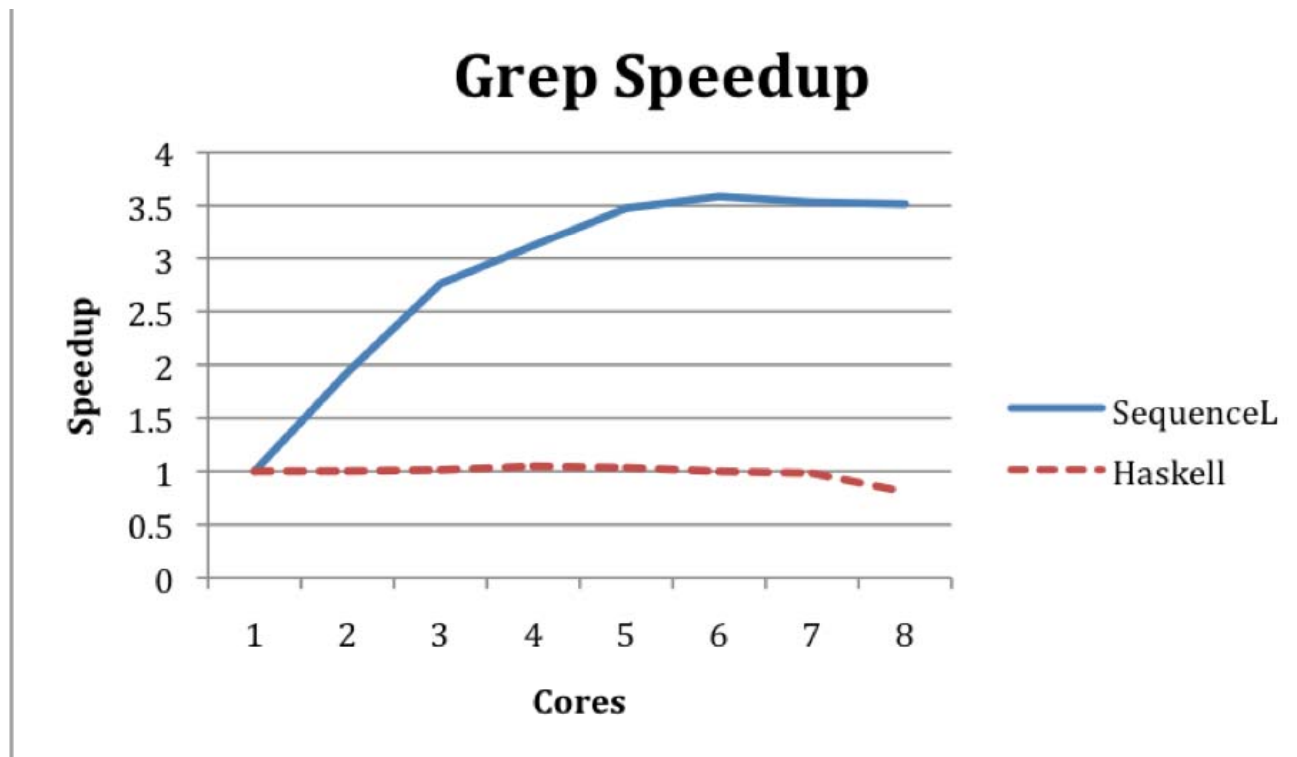
```
    let str := a[n...(n+size(b)-1)];
```

```
    in str when eq_list(str,b);
```





Performance Results



Experiments

Quicksort

The Quicksort is an interesting parallel problem, because parallelisms are dynamic (i.e., you cannot predict parallelisms before execution because of the pivot).

The experiment involved a list of 5,000,000 integers. We found the parallel Haskell version on a Haskell website: (<http://www.macs.hw.ac.uk/~dsg/gph/docs/Gentle-GPH/sec-gph.html>).

```
quicksortS [] = []
quicksortS [x] = [x]
quicksortS (x:xs) = losort ++ (x:hisort) `using` strategy
  where
    losort = quicksortS [y | y<- xs, y<x]
    hisort = quicksortS [y|y<-xs, y>= x]
    strategy result =
      rnf losort `par`
      rnf hisort `par`
      rnf result `par`
      ()
```

`lessThan(p,s) := s when (s < p);`

`greaterThan(p,s) := s when (s >= p);`

`quicksort(x(1)) :=`

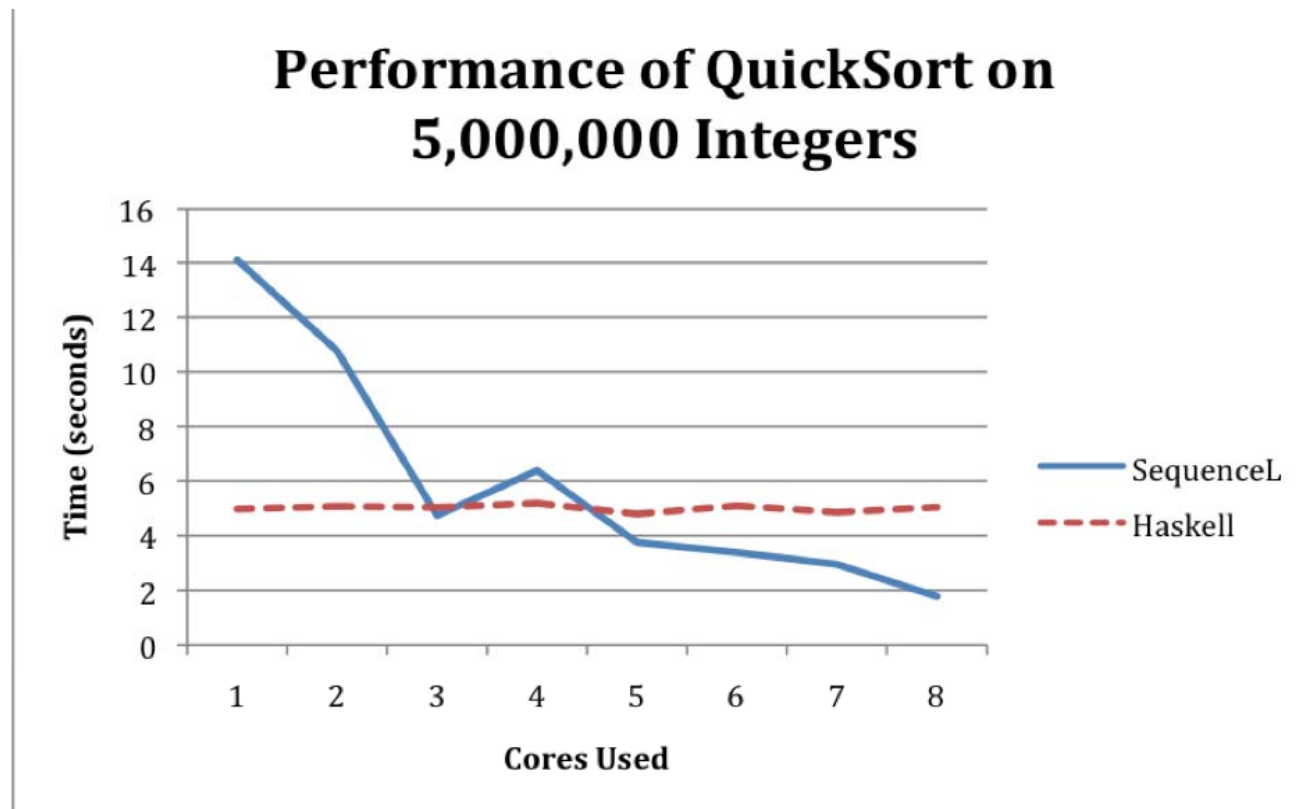
`quicksort (lessThan(head(x), tail(x))) ++`

`[head(x)] ++`

`quicksort (greaterThan(head(x), tail(x)))`

`when size(x)>1`

`else x;`



- NASA Guidance, Navigation, and Control

- Papers:

DAMP: Nemanich, Brad, Dan Cooke, and Nelson Rushton. Proc. of DAMP (Declarative Aspects of Multi-core Programming) 2010, Madrid, Spain. Print.

Computer Magazine: Cooke, Daniel E., and J. Nelson Rushton. "Taking Parnas's Principles to the Next Level: Declarative Language Design." *Computer* Sept. 2009: 46-53. Print.

ACM TOPLAS: Cooke, Daniel E., J. Nelson Rushton, Brad Nemanich, Robert G. Watson, and Per Andersen. "Normalize, Transpose, and Distribute: An Automatic Approach for Handling Non-scalars." *ACM Transactions on Programming Languages and Systems* 30.2 (March 2008): 9:1-9:49. Print.

- Industry Experience

- Oil & Gas
- Process Control
- High Performance Database
- CAD/CAM

Transparency (MATLAB Vector Match)

```
function [aIndex, bIndex] = vfind_scalar(avec, bvec)
```

```
avecLen = length(avec);
```

```
bvecLen = length(bvec);
```

```
% Size aIndex and bIndex to be large enough
```

```
outlen = min(avecLen, bvecLen);
```

```
aIndex = zeros(outlen,1);
```

```
bIndex = zeros(outlen,1);
```

```
n = 0;
```

```
ai = 1;
```

```
bi = 1;
```

```
while (ai <= avecLen || bi <= bvecLen)
```

```
    % Get vector elements at indices ai and bi
```

```
    A = avec(ai);
```

```
    B = bvec(bi);
```

Transparency (MATLAB Vector Match)

```
% If equal, record indices where elements match
if A == B
    n = n + 1;
    aIndex(n) = ai;
    bIndex(n) = bi;
end
```

```
% Advance index of avec, when appropriate
if A <= B
    if ai < avecLen
        ai = ai + 1;
    else
        break;
    end
end
```

```
% Advance index of bvec, when appropriate
if A >= B
    if bi < bvecLen
        bi = bi + 1;
    else
        break;
    end
end
```

```
end
```



Transparency (SequenceL Vect Match)

$\text{index}(a(1), b(1)) := \text{appends}(\text{Index}(a, b));$

$\text{Index}(a(1), b(1)) [i, j] := [i, j] \text{ when } a[i] = b[j];$



What about other non-Haskell, non-Matlab alternatives?

Transparency (Erlang Map-Reduce)



```
-module(mapreduce).  
-export([reduce_task/2, map_task/2,  
        test_map_reduce/0]).
```

```
repeat_exec(N,Func) ->  
  lists:map(Func, lists:seq(0, N-1)).
```

```
find_reducer(Processes, Key) ->  
  Index = erlang:phash(Key, length(Processes)),  
  lists:nth(Index, Processes).
```

```
find_mapper(Processes) ->  
  case random:uniform(length(Processes)) of  
    0 ->  
      find_mapper(Processes);  
    N ->  
      lists:nth(N, Processes)  
  end.
```

```
collect(Reduce_proc) ->  
  Reduce_proc ! {collect, self()},  
  receive  
    {result, Result} ->  
      Result  
  end.
```

```
map_reduce(M, R, Map_func,  
          Reduce_func, Acc0, List) ->
```

```
  Reduce_processes =  
    repeat_exec(R,  
      fun(_) ->  
        spawn(mapreduce, reduce_task,  
              [Acc0, Reduce_func])  
      end),
```

```
  io:format("Reduce processes ~w are started~n",  
            [Reduce_processes]),
```

Transparency (Erlang Map-Reduce)



```
Map_processes =
  repeat_exec(M,
    fun(_) ->
      spawn(mapreduce, map_task,
        [Reduce_processes, Map_func])
    end),

io:format("Map processes ~w are started~n",
  [Map_processes]),

Extract_func =
  fun(N) ->
    Extracted_line = lists:nth(N+1, List),
    Map_proc = find_mapper(Map_processes),
```

```
io:format("Send ~w to map process ~w~n",
  [Extracted_line, Map_proc]),
  Map_proc ! {map, Extracted_line}
end,

repeat_exec(length(List), Extract_func),

timer:sleep(2000),
io:format("Collect all data from reduce
processes~n"),
All_results =
  repeat_exec(length(Reduce_processes),
    fun(N) ->
      collect(lists:nth(N+1, Reduce_processes))
    end),
  lists:flatten(All_results).
```

Transparency (Erlang Map-Reduce)



```
map_task(Reduce_processes, MapFun) ->
receive
  {map, Data} ->
    IntermediateResults = MapFun(Data),
    io:format("Map function produce: ~w~n",
              [IntermediateResults ]),
    lists:foreach(
      fun({K, V}) ->
        Reducer_proc =
          find_reducer(Reduce_processes, K),
          Reducer_proc ! {reduce, {K, V}}
        end, IntermediateResults),
      map_task(Reduce_processes, MapFun)
    end.
```

```
reduce_task(Acc0, ReduceFun) ->
receive
  {reduce, {K, V}} ->
    Acc = case get(K) of
            undefined ->
              Acc0;
            Current_acc ->
              Current_acc
          end,
    put(K, ReduceFun(V, Acc)),
    reduce_task(Acc0, ReduceFun);
  {collect, PPid} ->
    PPid ! {result, get()},
    reduce_task(Acc0, ReduceFun)
end.
```

Transparency (SequenceL Map-Red.)



```
ws(a(1), b(1)) := word_search(a, b, 1...(size(a)-size(b)+1));
```

```
word_search(a(1), b(1), n) :=
```

```
    let str := a[n...(n+size(b)-1)];
```

```
    in str when eq_list(str,b);
```

```
(from before...)
```

```
map_reduce(a(2), b(1)) := (word: b, count: size(ws(a, b)));
```



Transparency (OpenMP Jacobian)



```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "SSTimer.h"
#ifdef _OPENMP
#include <omp.h>
#endif

using namespace std;

int main(int argc, char** argv)
{
    int x,y,z,n,procs;

    if (argc == 1)
    {
        x = 100;
        y = 100;
        z = 100;
        n = 10;
        procs = 1;
    }
    else if (argc < 4)
    {
        cout<<"Provide x-size y-size z-size num-
            iterations"<<endl;
        return -1;
    }
    else
    {
        x = atoi(argv[1]);
        y = atoi(argv[2]);
        z = atoi(argv[3]);
        n = atoi(argv[4]);
    }

#ifdef _OPENMP
    int nthreads = 1;
#endif
}
```

Transparency (OpenMP Jacobian)



```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads =
omp_get_num_threads();
        cout<<"Number of openMP
threads: "<<nthreads<<endl;
    }
}
#endif
```

```
double*** Un = new double**[x];
double*** U = new double**[x];
double*** b = new double**[x];
```

```
#pragma omp parallel for
```

```
for (int i=0; i<x; i++)
{
    Un[i] = new double*[y];
    U[i] = new double*[y];
    b[i] = new double*[y];
    for (int j=0; j<y; j++)
    {
        Un[i][j] = new double[z];
        U[i][j] = new double[z];
        b[i][j] = new double[z];
        for (int k=0; k<z; k++)
        {
            Un[i][j][k] = 2.0;
            U[i][j][k] = 2.0;
            b[i][j][k] = 5.0;
        }
    }
}
```

Transparency (OpenMP Jacobian)



```
SSTimer t1;

t1.start();
int i,j,k;
float c = 1/ 6.0;
for (int it= 0; it<n; it++)
{
    #pragma omp parallel
    shared(U,Un,b,x,y,z,c) private(i,j,k)
    #pragma omp for schedule(static)
    for (i=1; i<x-1; i++)
        for (j=1; j<y-1; j++)
            for (k=1; k<z-1; k++)
            {
                Un[i][j][k] = c * (U[i-1][j][k] +
                U[i+1][j][k] + U[i][j-1][k] +
                U[i][j+1][k] + U[i][j][k-1] +
                U[i][j][k+1] - b[i-1][j-1][k-1]);
            }
    double*** tmp = U;
    U = Un;
    Un = tmp ;
}
}
```

```
t1.stop();

cout<<"Time: "<<t1.getTime()<<endl;

for (int i=0; i<x; i++)
{
    for (int j=0; j<y; j++)
    {
        delete [] Un[i][j];
        delete [] U[i][j];
        delete [] b[i][j];
    }
    delete [] Un[i];
    delete [] U[i];
    delete [] b[i];
}
delete [] Un;
delete [] U;
delete [] b;
```

```
iterate(U(3),b(3),n) := U when n=0 else iterate(compute(U,b), b, n-1);
```

```
compute(U(3),b(3))[i,j,k] :=
```

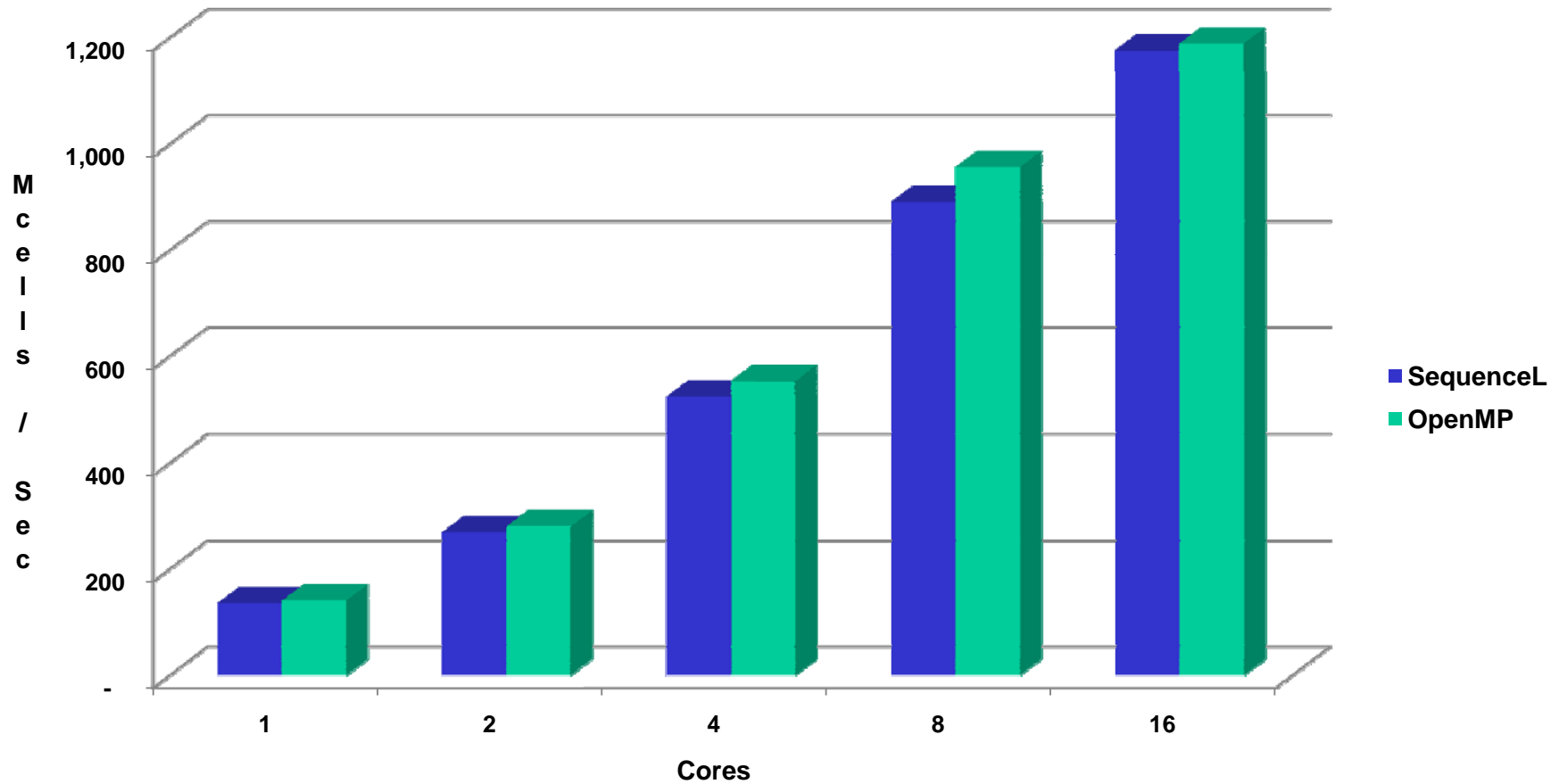
```
  U[i,j,k]
```

```
  when (i=1 or i=size(U) or j=1 or j=size(U[i]) or k=1 or k=size(U[i,j]))
```

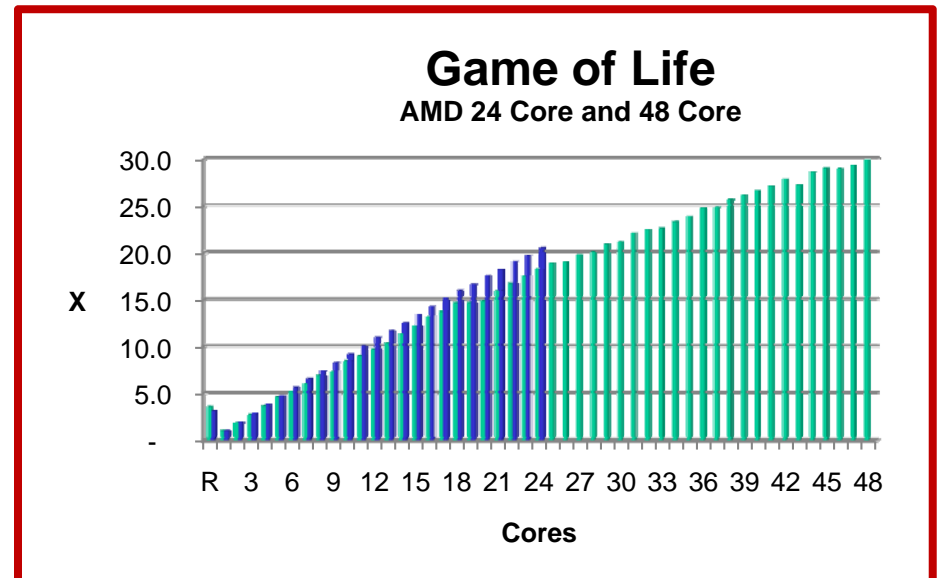
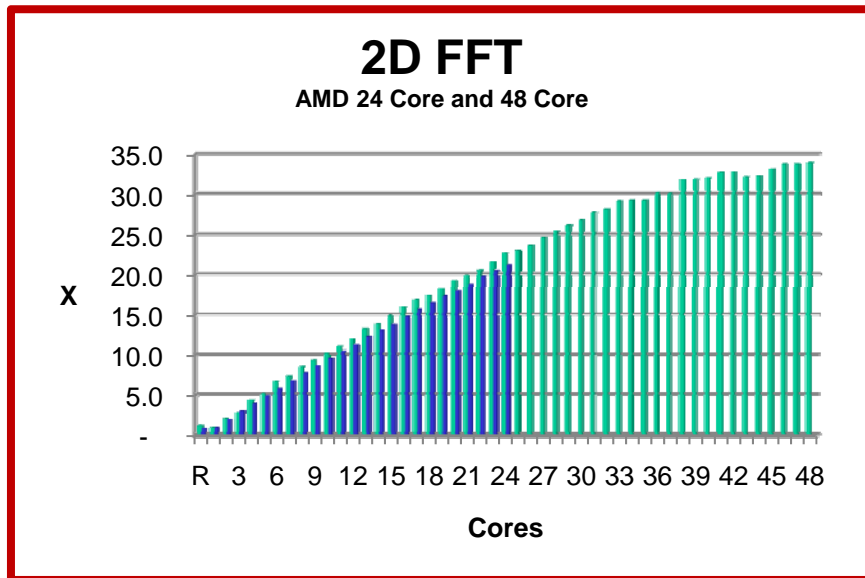
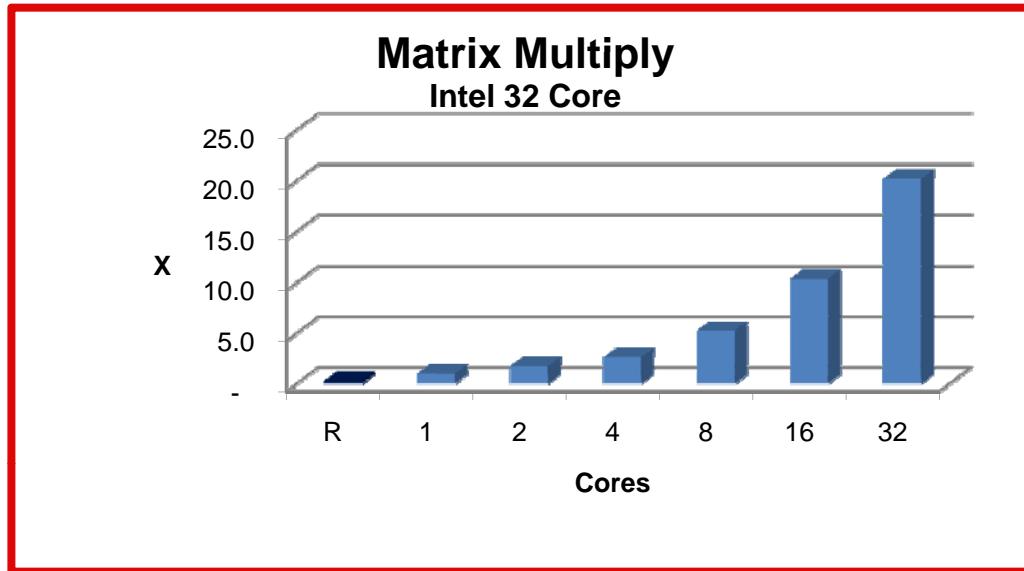
```
  else
```

```
  c*(U[i-1,j,k] + U[i+1,j,k] + U[i,j-1,k] + U[i,j+1,k] + U[i,j,k-1] + U[i,j,k+1] - b[i-1,j-1,k-1]);
```

3D Jacobi Algorithm 256x256x256 (100 Iterations)



SequenceL Benchmark Tests



□ Declarative

- ❖ Write the computation.
- ❖ Don't worry about how to compute.

□ Transparent

- ❖ Easy to read.
- ❖ Automatic Documentation

□ Automatic Threading

- ❖ High performance applications.
- ❖ High efficiency.

□ Transportable

- ❖ Develop once.
- ❖ Utilize evolving hardware performance.

ROI Sources

Software Development

- > Fewer coders
- > Greater innovation
- > Fewer bugs
- > Shorter lead time

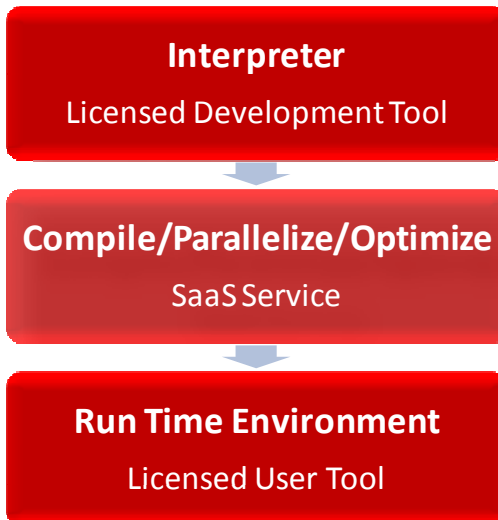
Data Center Efficiency

- > Reduced power
- > Reduced cooling
- > Lower capital
- > Smaller real estate

Application Performance

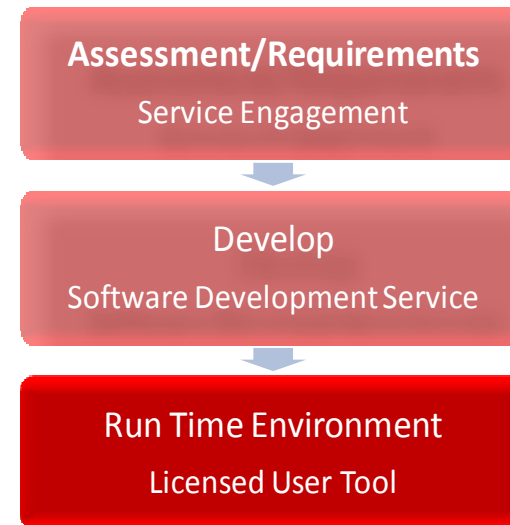
- > Faster run time
- > More capability
- > Better transportability

SEQUENCEL



- Develop your C++ software with the SequenceL Interpreter .
- Use SequenceL services to compile/parallelize and optimize it.
- Run the application on the SequenceL Run Time Environment.

THREADED



- Begin with an assessment for legacy code accelerations or requirements development for new application development.
- Fixed price commitment for the actual acceleration or development.
- ThreadedS developed software is fully parallelized and runs on the SequenceL Run Time Environment.