



An Interactive Environment for Numerical and Combinatorial Supercomputing

John R. Gilbert

University of California, Santa Barbara

Viral Shah, Imran Patel (UCSB)

Alan Edelman (MIT and Interactive Supercomputing)

Ron Choy, David Cheng (MIT)

Parry Husbands (Lawrence Berkeley Lab)

Steve Reinhardt, Todd Letsche (SGI)

Support: DOE Office of Science, DARPA, SGI, ISC

Parallel Computing Today



Columbia,
NASA Ames Research Center

Departmental Beowulf cluster



But!

How do you program it?

C with MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#define A(i,j) ( 1.0/(((1.0*(i)+(j))*(1.0*(i)+(j)+1)/2 + (1.0*(i)+1)) )
void errorExit(void);
double normalize(double* x, int mat_size);
int main(int argc, char **argv)
{
    int num_procs;
    int rank;
    int mat_size = 64000;
    int num_components;
    double *x = NULL;
    double *y_local = NULL;
    double norm_old = 1;
    double norm = 0;
    int i,j;
    int count;
    if (MPI_SUCCESS != MPI_Init(&argc, &argv)) exit(1);
    if (MPI_SUCCESS != MPI_Comm_size(MPI_COMM_WORLD,&num_procs)) errorExit();
```

C with MPI (2)

```
if (0 == mat_size % num_procs) num_components = mat_size/num_procs;
else num_components = (mat_size/num_procs + 1);
mat_size = num_components * num_procs;
if (0 == rank) printf("Matrix Size = %d\n", mat_size);
if (0 == rank) printf("Num Components = %d\n", num_components);
if (0 == rank) printf("Num Processes = %d\n", num_procs);
x = (double*) malloc(mat_size * sizeof(double));
y_local = (double*) malloc(num_components * sizeof(double));
if ( (NULL == x) || (NULL == y_local) )
{
    free(x);
    free(y_local);
    errorExit();
}
if (0 == rank)
{
    for (i=0; i<mat_size; i++)
    {
        x[i] = rand();
    }
    norm = normalize(x,mat_size);
}
```

C with MPI (3)

```
if (MPI_SUCCESS !=
    MPI_Bcast(x, mat_size, MPI_DOUBLE, 0, MPI_COMM_WORLD)) errorExit();
count = 0;
while (fabs(norm-norm_old) > TOL) {
    count++;
    norm_old = norm;
    for (i=0; i<num_components; i++)
    {
        y_local[i] = 0;
    }
    for (i=0; i<num_components && (i+num_components*rank)<mat_size; i++)
    {
        for (j=mat_size-1; j>=0; j--)
        {
            y_local[i] += A(i+rank*num_components,j) * x[j];
        }
    }
    if (MPI_SUCCESS != MPI_Allgather(y_local, num_components, MPI_DOUBLE, x,
        num_components, MPI_DOUBLE, MPI_COMM_WORLD)) errorExit();
```

C with MPI (4)

```
    norm = normalize(x, mat_size);
}
if (0 == rank)
{
    printf("result = %16.15e\n", norm);
}
free(x);
free(y_local);

MPI_Finalize();
exit(0);
}
void errorExit(void)
{
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("%d died\n", rank);
    MPI_Finalize();
    exit(1);
}
```

C with MPI (5)

```
double normalize(double* x, int mat_size)
{
    int i;
    double norm = 0;
    for (i=mat_size-1; i>=0; i--)
    {
        norm += x[i] * x[i];
    }
    norm = sqrt(norm);
    for (i=0; i<mat_size; i++)
    {
        x[i] /= norm;
    }
    return norm;
}
```

Star-P

```
A = rand(4000*p, 4000*p);  
x = randn(4000*p, 1);  
y = zeros(size(x));  
while norm(x-y) / norm(x) > 1e-11  
    y = x;  
    x = A*x;  
    x = x / norm(x);  
end;
```

History

- Matlab*P 1.0 (1998): Edelman, Husbands, Isbell (MIT)
- Matlab*P 2.0 (2002-): MIT / UCSB / LBNL
- Star-P (2004-): Interactive Supercomputing / SGI



Data-Parallel Operations

< M A T L A B >

Copyright 1984-2001 The MathWorks, Inc.

Version 6.1.0.1989a Release 12.1

```
>> A = randn(500*p, 500*p)
```

```
A = ddense object: 500-by-500
```

```
>> E = eig(A);
```

```
>> E(1)
```

```
ans = -4.6711 +22.1882i
```

```
e = E(:, :);
```

```
>> ppwhos
```

Name	Size	Bytes	Class
A	500px500p	688	ddense object
E	500px1	652	ddense object
e	500x1	8000	double array (complex)



Task-Parallel Operations

```
>> quad('4./(1+x.^2)', 0, 1);
```

```
ans = 3.14159270703219
```

```
>> a = (0:3*p) / 4
```

```
a = ddense object: 1-by-4
```

```
>> a(:, :)
```

```
ans =
```

```
0
```

```
0.2500000000000000
```

```
0.5000000000000000
```

```
0.7500000000000000
```

```
>> b = a + .25;
```

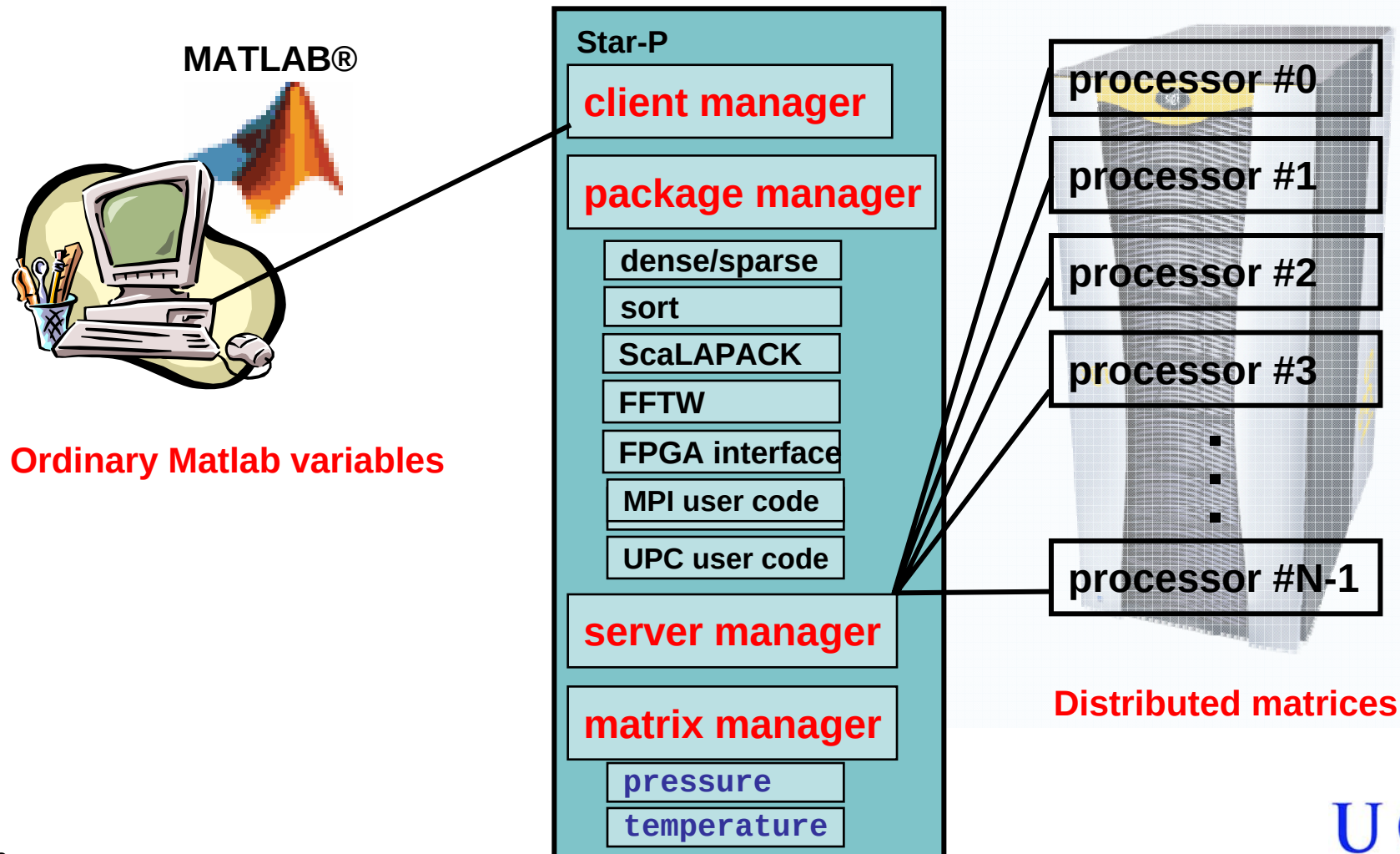
```
>> c = ppeval('quad', '4./(1+x.^2)', a, b);
```

```
c = ddense object: 1-by-4
```

```
>> sum(c)
```

```
ans = 3.14159265358979
```

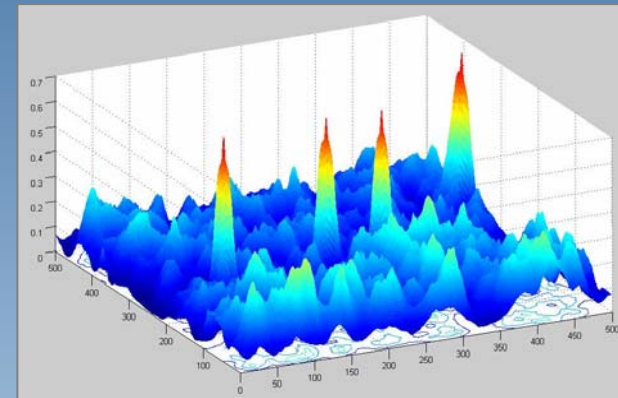
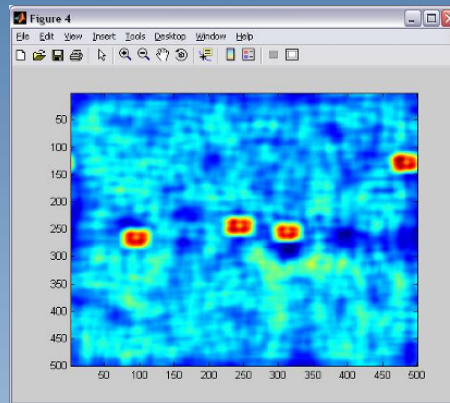
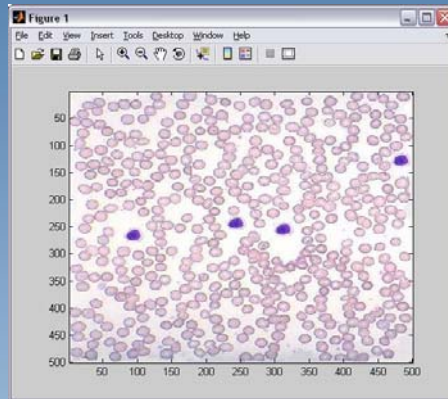
Star-P Architecture



Language Philosophy

- An interactive parallel programming system
- User interacts with Star-P via a front end running Matlab®
- Distributed dense & sparse arrays are first-class Matlab objects
- Data stays on back-end server until retrieved explicitly
- Two views of the same data:
 - Data-parallel SIMD
 - Task-parallel SPMD
- Server supplies parallel primitives and software libraries
(ScaLAPACK, SuperLU, PARPACK, FFTW, . . .)

Application Example: Image Processing in MATLAB®



by Vern Schrauger & Ilya Mirman, ISC

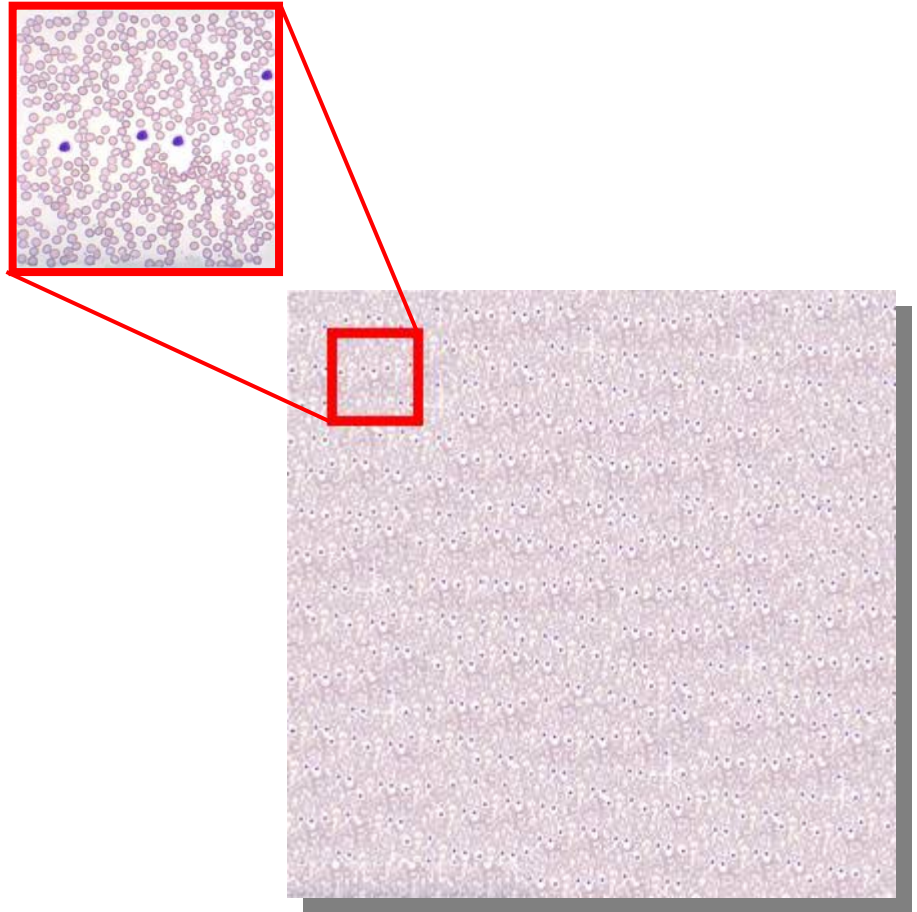
Interactive Supercomputing

• 135 Beaver Street, FL 2 • Waltham, MA 02452

781.398-0010 • www.interactivesupercomputing.com

Example: Blood Cells

- Blood cell image
- Goal: Identify stained blood cells
- Challenges
 - Potentially very large images
 - Want to explore multiple detection algorithms



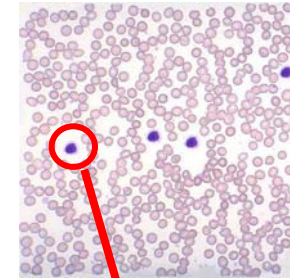
Target:



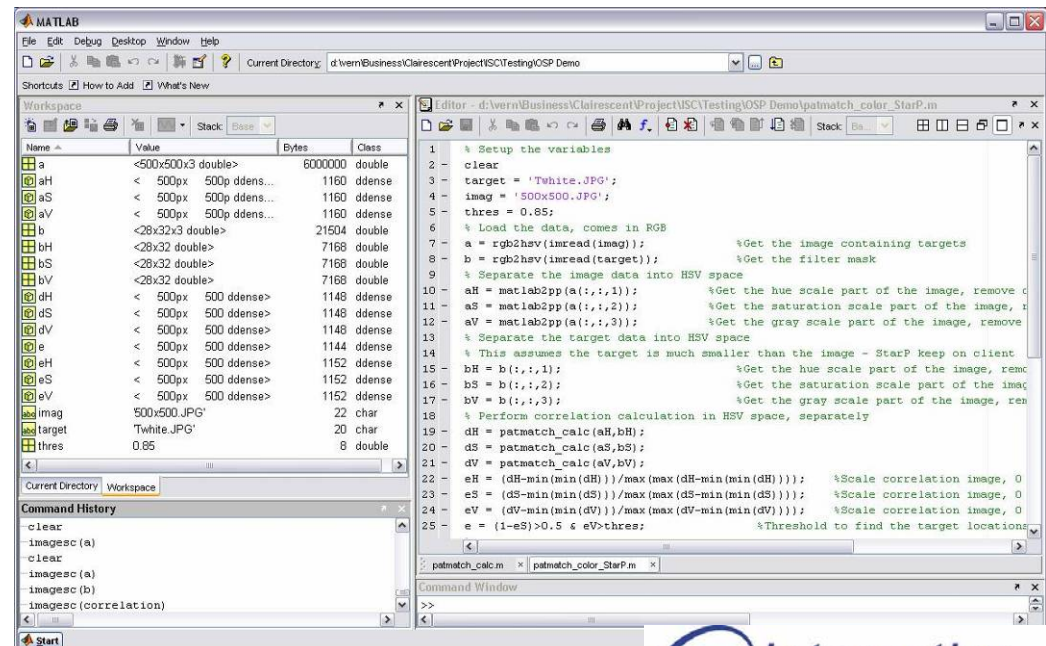
Simple MATLAB Script

- Load search and target images
- Separate into HSV color space variables
- Pattern match on each color variable using fast transforms
- Use thresholding on correlation plots to detect cells

Search image:

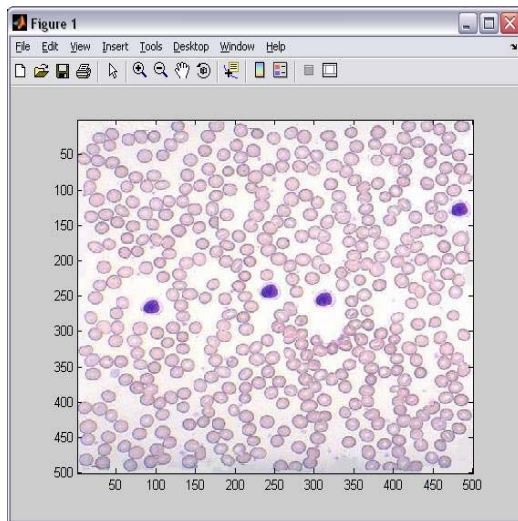


Target image:

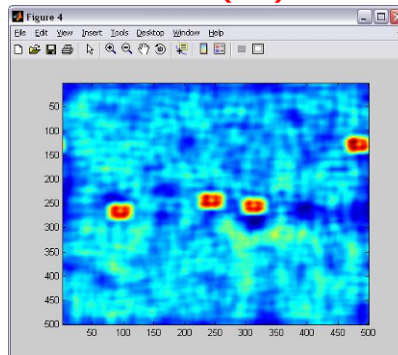


Correlation Plots

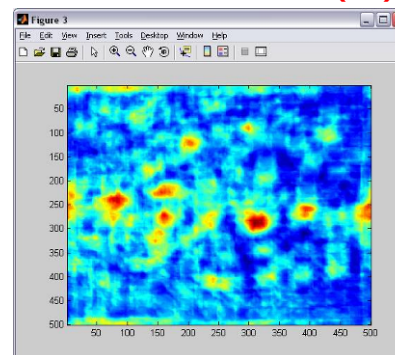
input image



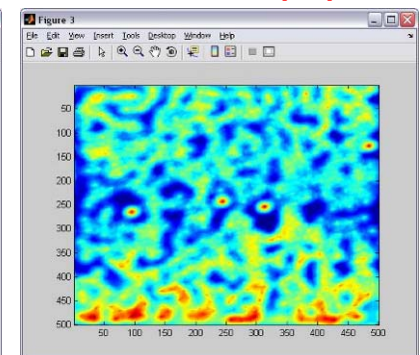
Hue (H)



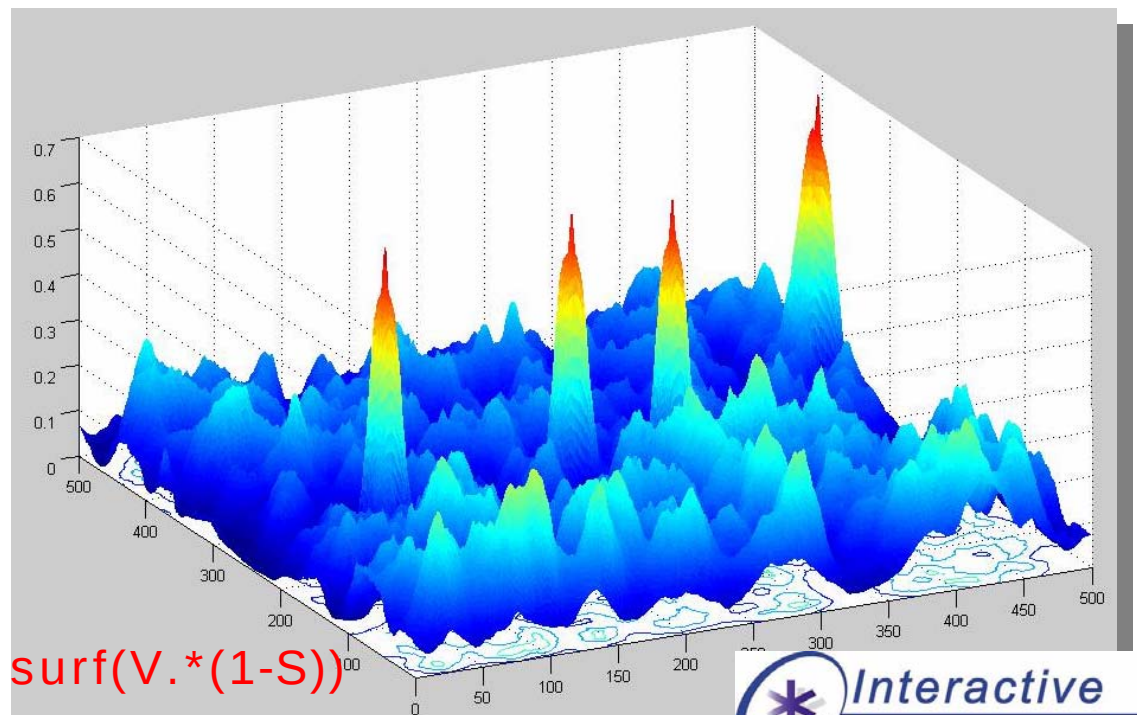
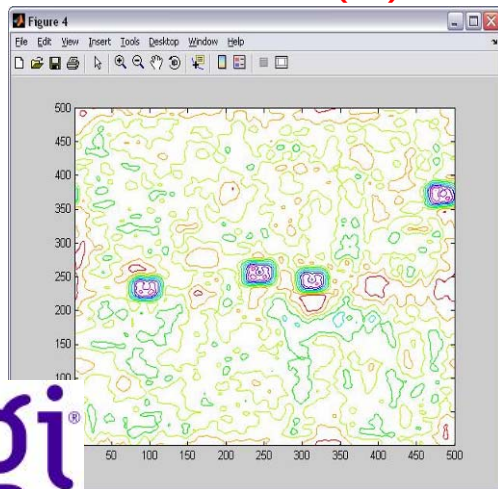
Saturation (S)



Value (V)



contour (V)

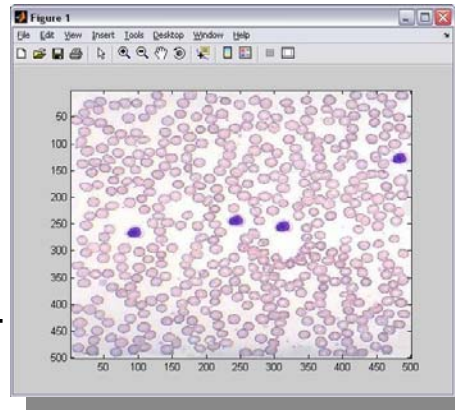


surf(V.*(1-S))

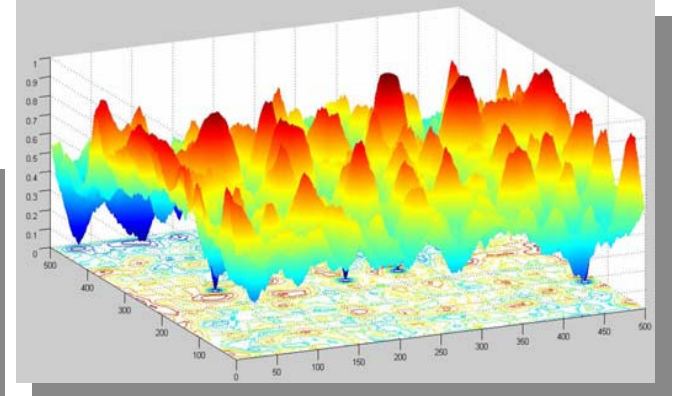
Proprietary and Confidential

Interactivity Key for Refinement

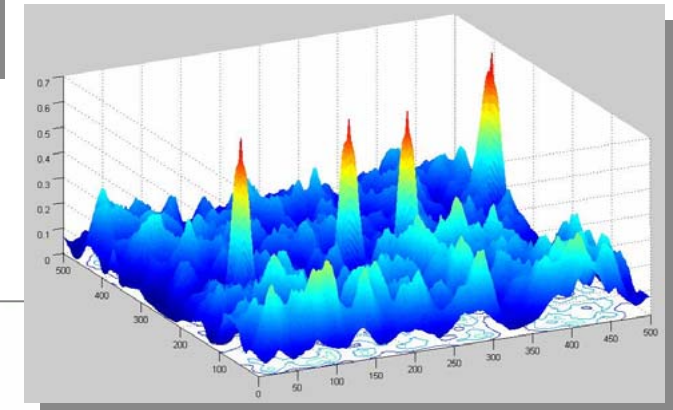
- Vary basis of color space (HSV, RGB, etc.)
- Vary threshold parameters
- Vary transform algorithms
- Fine-grained control of execution (e.g., with breakpoints)
- Fine-grained control of data (access & compute on areas of interest)



poor threshold value



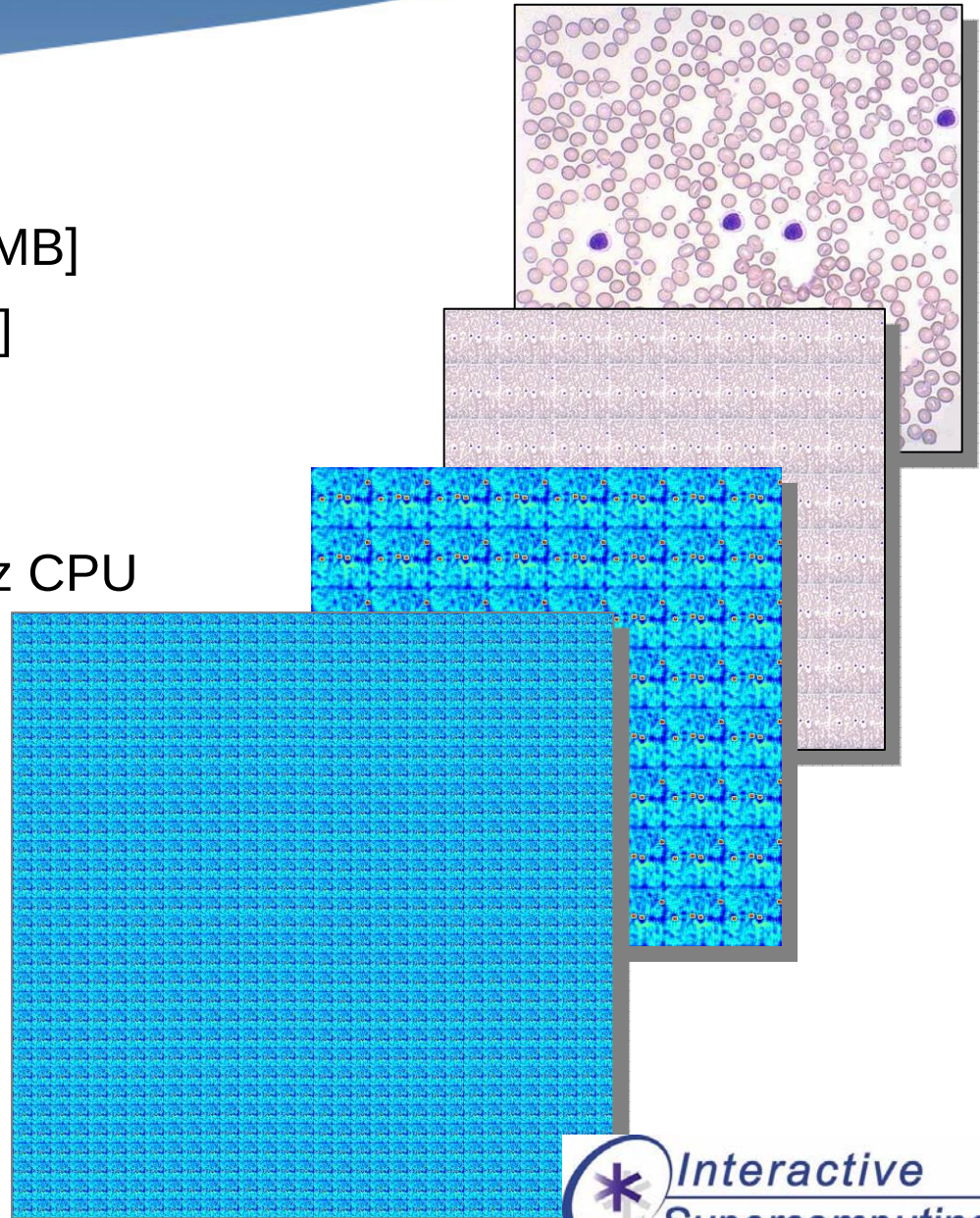
good threshold value



```
14 % Perform correlation calculation in HSV space, separately
15 - dH = patmatch_calc(aH,bH);
16 - dS = patmatch_calc(aS,bS);
17 - dV = patmatch_calc(aV,bV);
18 - eH = (dH-min(min(dH)))/max(max(dH-min(min(dH))))); %Scale correlation image, 0 to 1
19 - eS = (dS-min(min(dS)))/max(max(dS-min(min(dS))))); %Scale correlation image, 0 to 1
20 - eV = (dV-min(min(dV)))/max(max(dV-min(min(dV))))); %Scale correlation image, 0 to 1
21 - e = (1-eS)>0.5 & eV>thres; %Threshold to find the target locations within input image
22
```

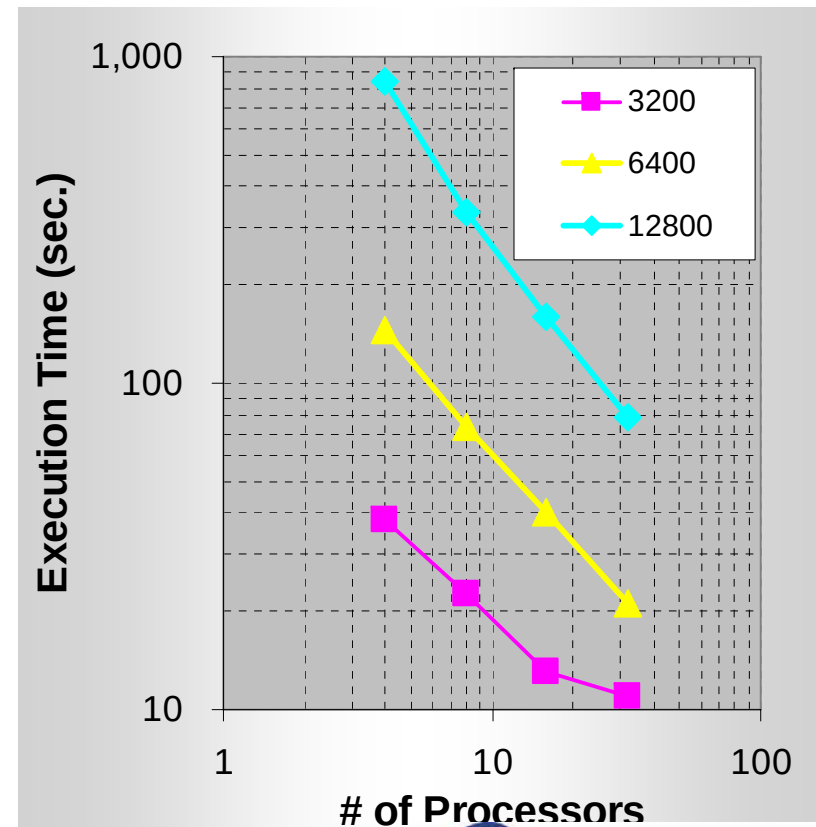
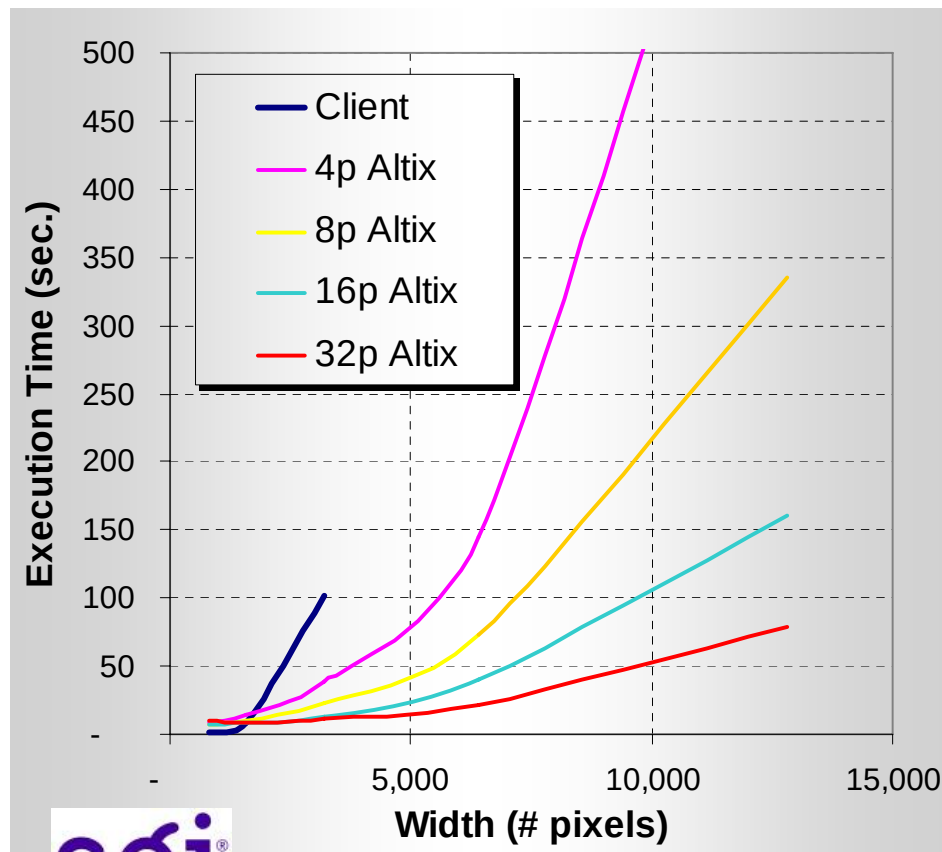
Benchmarking

- Tiled input images:
From: 400 x 400 pixels [1.2 MB]
To: 12,800 x 12,800 [1.2 GB]
(Arrays: 1x1 to 32x32)
- Desktop PC:
 - Dual-processor 3.2 GHz CPU
 - 2GB RAM
- SGI Altix HPC:
 - 4-32 processors
 - 1.5 GHz Itanium
 - 128 GB shared RAM

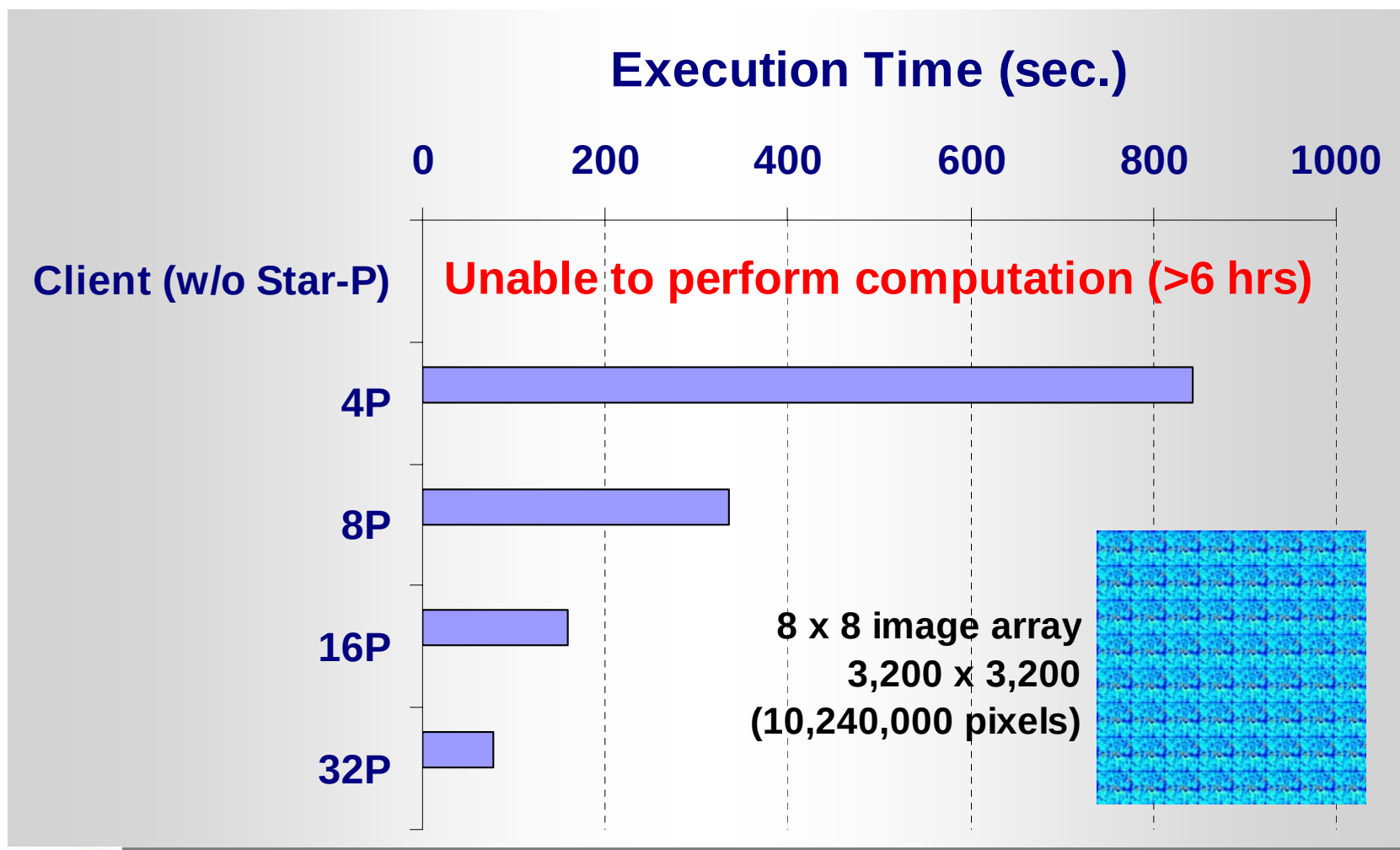


Performance Benchmarks

- Cross-over point at 1,600 pixels
- Well-behaved speed-up at 4P/8P/16P/32P



Scalability Example: 10,000,000 Pixel Array





Sparse matrices and graphs in Star-P

Matlab sparse matrix design principles

- All operations should give the same results for sparse and full matrices (almost all)
- Sparse matrices are never created automatically, but once created they propagate
- Performance is important -- but usability, simplicity, completeness, and robustness are more important
- Storage for a sparse matrix should be $O(\text{nonzeros})$
- Time for a sparse operation should be $O(\text{flops})$ (as nearly as possible)

Matlab sparse matrix design principles

- All operations should give the same results for sparse and full matrices (almost all)
- Sparse matrices are never created automatically, but once created they propagate
- Performance is important -- but usability, simplicity, completeness, and robustness are more important
- Storage for a sparse matrix should be $O(\text{nonzeros})$
- Time for a sparse operation should be $O(\text{flops})$ (as nearly as possible)

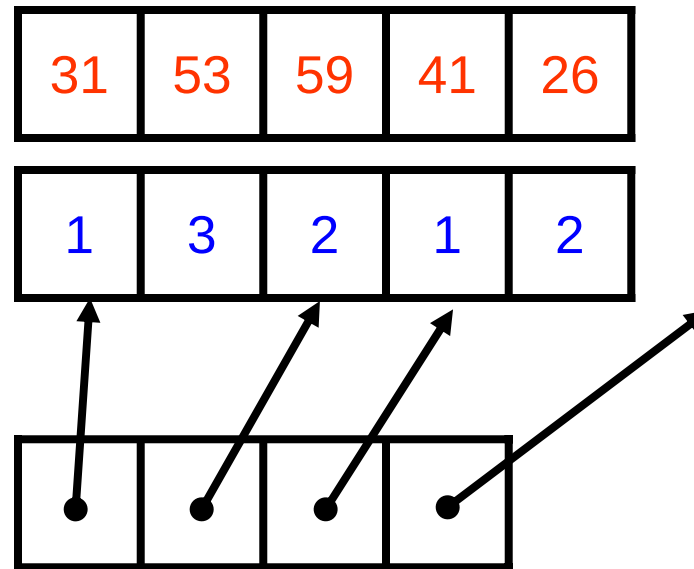
**Matlab*P dspase matrices: same principles,
but some different tradeoffs**

Sparse matrix operations

- **dsparse** layout, same semantics as **ddense**
- For now, only row distribution
- Matrix operators: **+**, **-**, **max**, etc.
- Matrix indexing and concatenation
$$A(1:3, [4\ 5\ 2]) = [B(:, 7)\ C];$$
- **A \ b** by direct methods

Sparse data structure

31	0	53
0	59	0
41	26	0



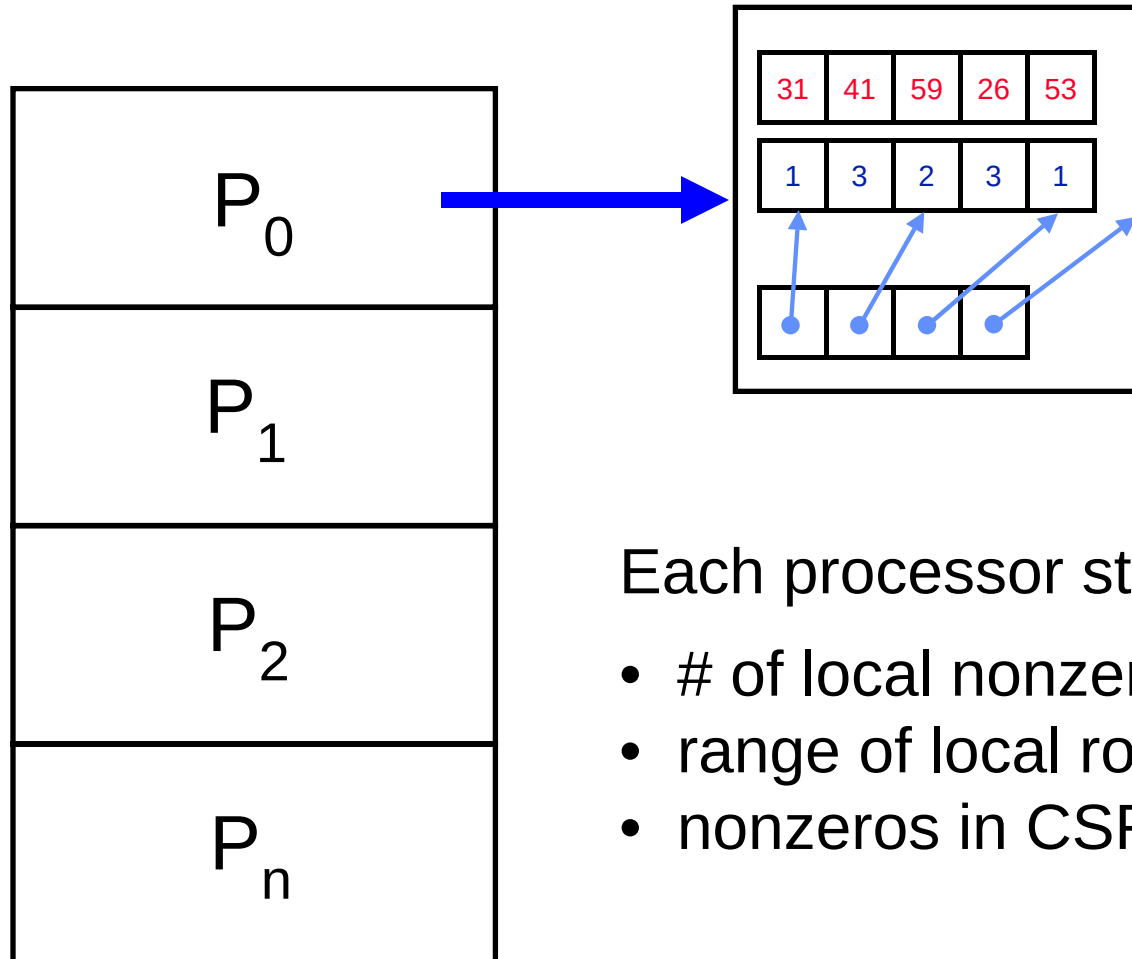
- Full:

- 2-dimensional array of real or complex numbers
- $(nrows \times ncols)$ memory

- Sparse:

- compressed row storage
- about $(2 \times nzs + nrows)$ memory

Distributed sparse data structure



Each processor stores:

- # of local nonzeros
- range of local rows
- nonzeros in CSR form

The sparse() constructor

- **$A = \text{sparse}(\mathbf{I}, \mathbf{J}, \mathbf{V}, \text{nr}, \text{nc})$**
- Input: ddense vectors **$\mathbf{I}, \mathbf{J}, \mathbf{V}$** , dimensions **$\text{nr}, \text{nc}$**
- Output: **$A(\mathbf{I}(k), \mathbf{J}(k)) = \mathbf{V}(k)$**
- Sort triples **$\langle i, j, v \rangle$** by **$\langle i, j \rangle$**
- Locally convert to compressed row indices
- Sum values with duplicate indices

Sorting in Star-P

- `[V, perm] = sort (V)`
- Common primitive for several sparse array algorithms, including the `sparse` constructor
- Star-P uses a parallel sample sort

Sparse matrix times dense vector

- $y = A * x$
- First matvec with A caches a communication schedule
- Later matvecs with A use the cached schedule
- Communication and computation overlap

Sparse linear systems

- $x = A \setminus b$
 - Matrix division uses MPI-based direct solvers:
 - SuperLU_dist: nonsymmetric, static pivoting
 - MUMPS: nonsymmetric, multifrontal
 - PSPASES: symmetric positive definite
- `ppsetoption('SparseDirectSolver', 'SUPERLU')`
- Iterative solvers implemented in Star-P

Sparse eigenvalue solvers

- $[V, D] = \text{eigs}(A);$ etc.
- Uses MPI-based PARPACK
[Lehoucq, Maschhoff, Sorensen, Yang]

Combinatorial Scientific Computing

- Sparse matrix methods; graph matching; machine learning; search and information retrieval; adaptive multilevel modeling; geometric meshing; computational biology; . . .
- How will combinatorial methods be used by people who don't understand them in complete detail?

Analogy: Matrix division in Matlab

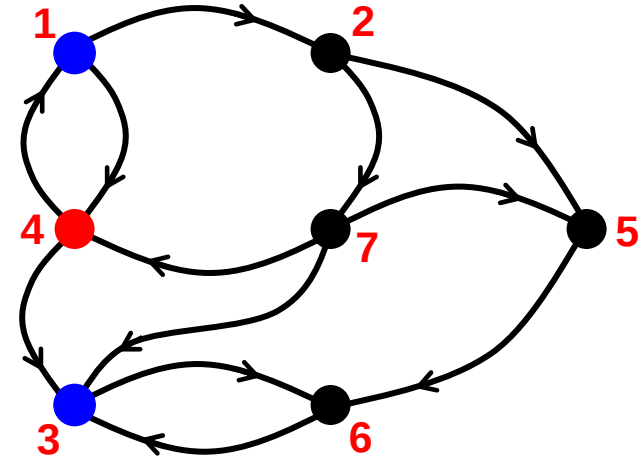
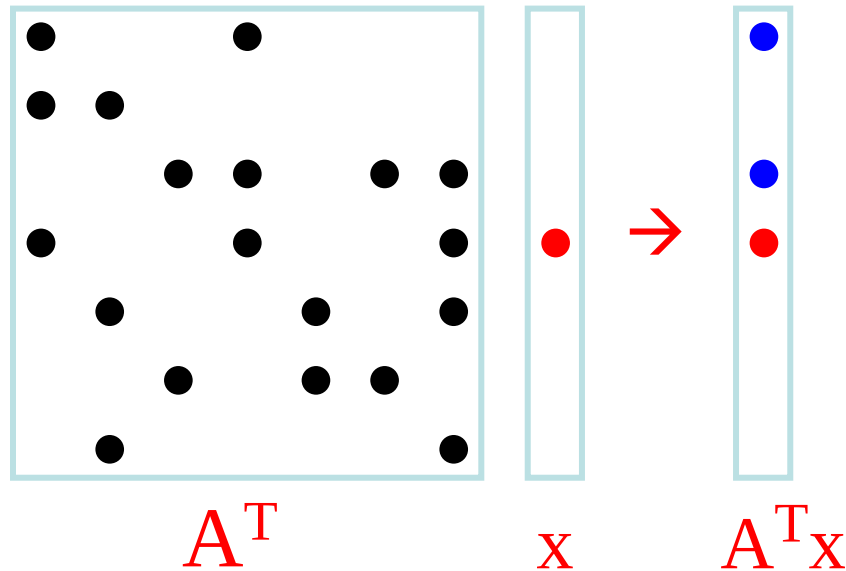
$$x = A \setminus b;$$

- Works for either full or sparse A
- Is A square?
 - no** => use QR to solve least squares problem
- Is A triangular or permuted triangular?
 - yes** => sparse triangular solve
- Is A symmetric with positive diagonal elements?
 - yes** => attempt Cholesky after symmetric minimum degree
- Otherwise
 - =>** use LU on $A(:, \text{colamd}(A))$

Combinatorics in Star-P

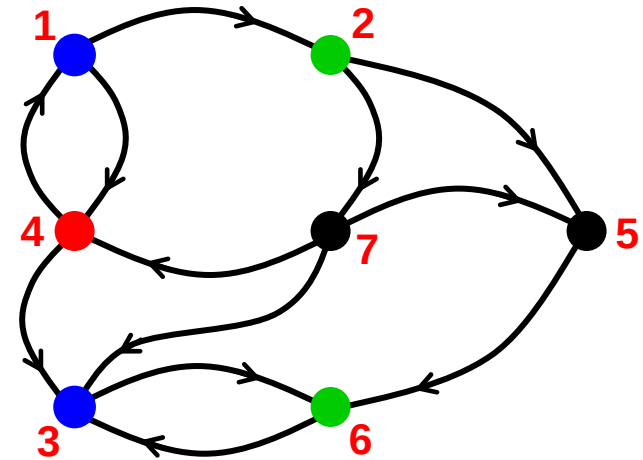
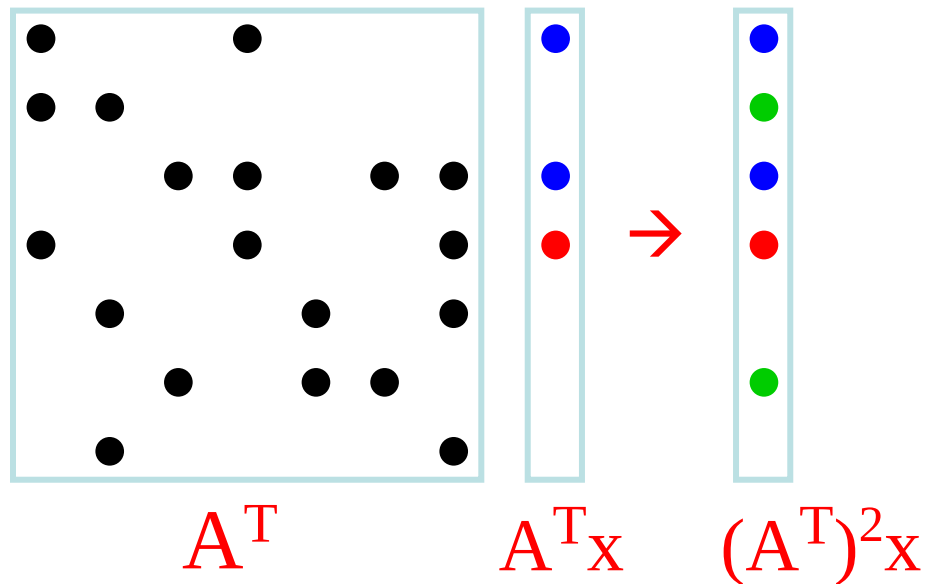
- Represent a graph as a sparse adjacency matrix
- A sparse matrix language is a good start on primitives for computing with graphs
 - Random-access indexing: **$A(i, j)$**
 - Neighbor sequencing: **$\text{find } (A(i, :))$**
 - Sparse table construction: **$\text{sparse } (I, J, V)$**
 - Breadth-first search step : **$A * v$**

Breadth-first search: sparse mat * vec



- Multiply by adjacency matrix → step to neighbor vertices
- Efficient implementation from sparse data structures

Breadth-first search: sparse mat * vec



- Multiply by adjacency matrix → step to neighbor vertices
- Efficient implementation from sparse data structures

Connected components of a graph

- Sequential Matlab uses depth-first search (**dmperm**), which doesn't parallelize well
- Shiloach-Vishkin pointer-jumping algorithm:
 - repeat
 - Link every (super)vertex to a neighbor
 - Shrink each tree to a supervertex by pointer jumping
 - until no further change
- Hybrid SV / search method under construction
- Other possible graph kernels:
 - Shortest-path search (after Husbands, LBNL)
 - Bipartite matching (after Riedy, UCB)
 - Strongly connected components (after Pinar, LBNL)

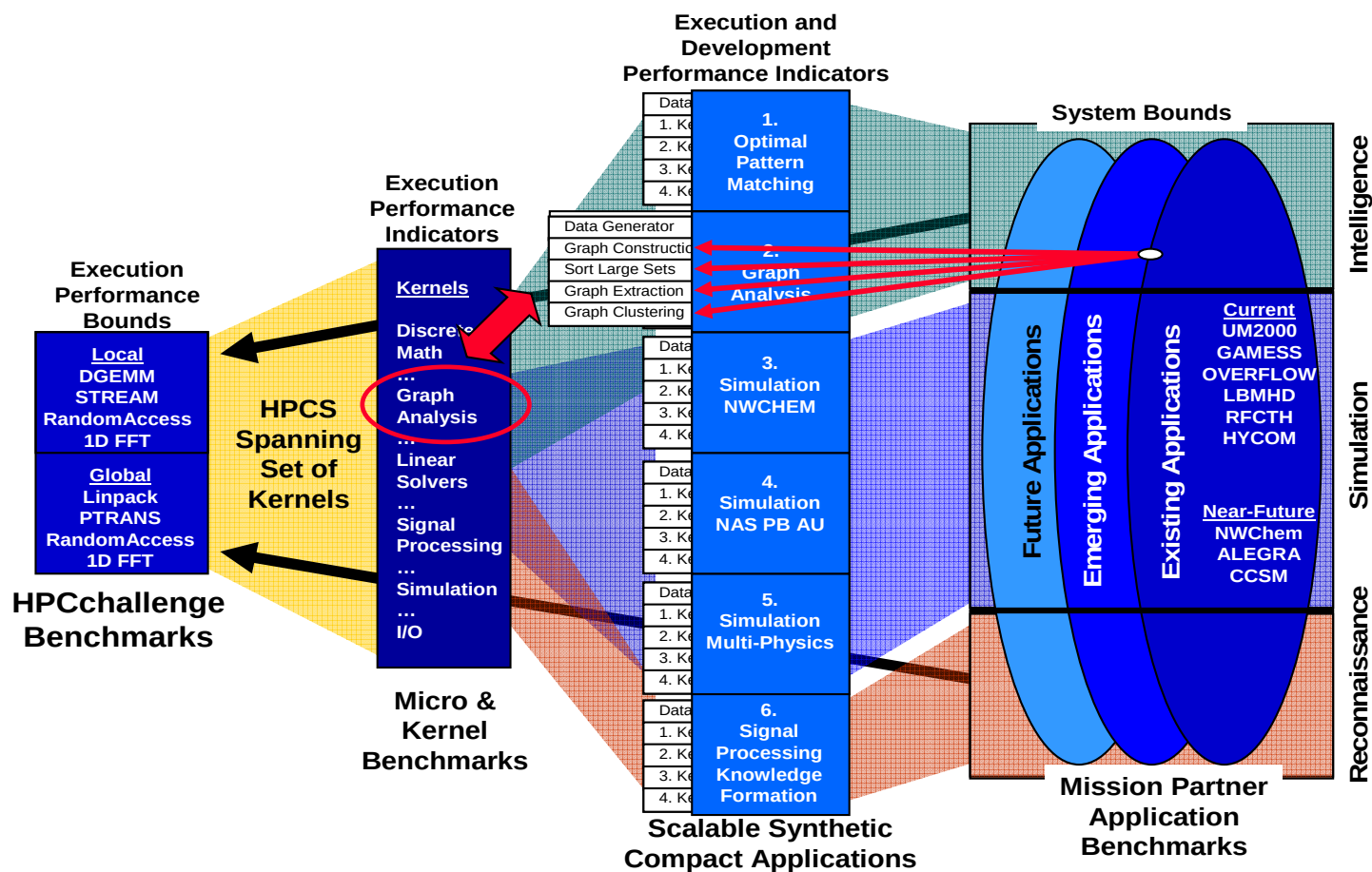
HPCS Scalable Synthetic Compact Apps



SSCA#1: Bioinformatics / Pattern Matching

SSCA#2: Graph Analysis

SSCA#3: Sensor Processing / Imaging



SSCA#2: “Graph Analysis”



- Fine-grained, irregular data access
- Searching and clustering
- Goal is scaling to very large graphs
- Four computational kernels:

Kernel 1: Build graph data structure

Kernel 2: Search by edge labels

Kernel 3: Extract subgraphs

Kernel 4: Partition into clusters

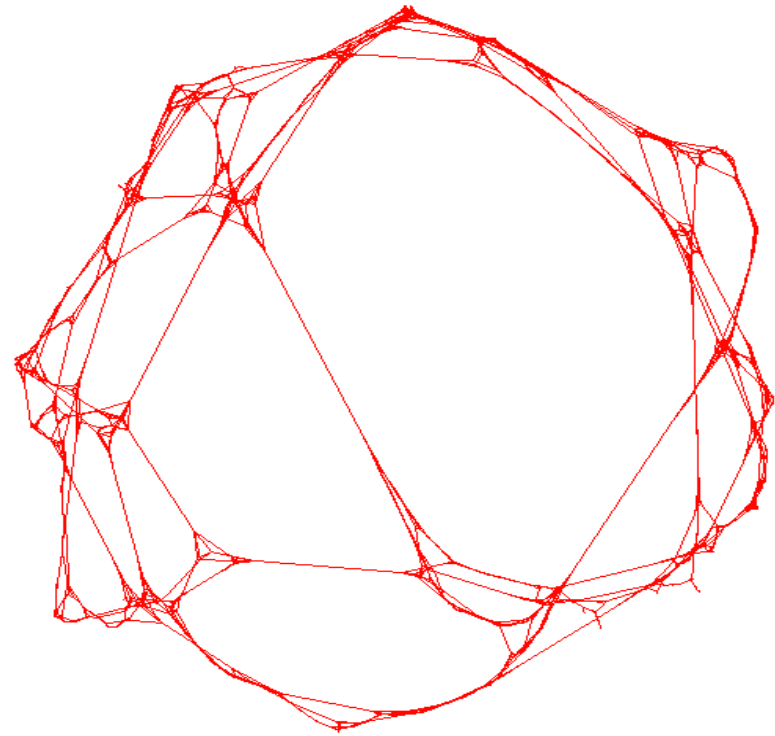
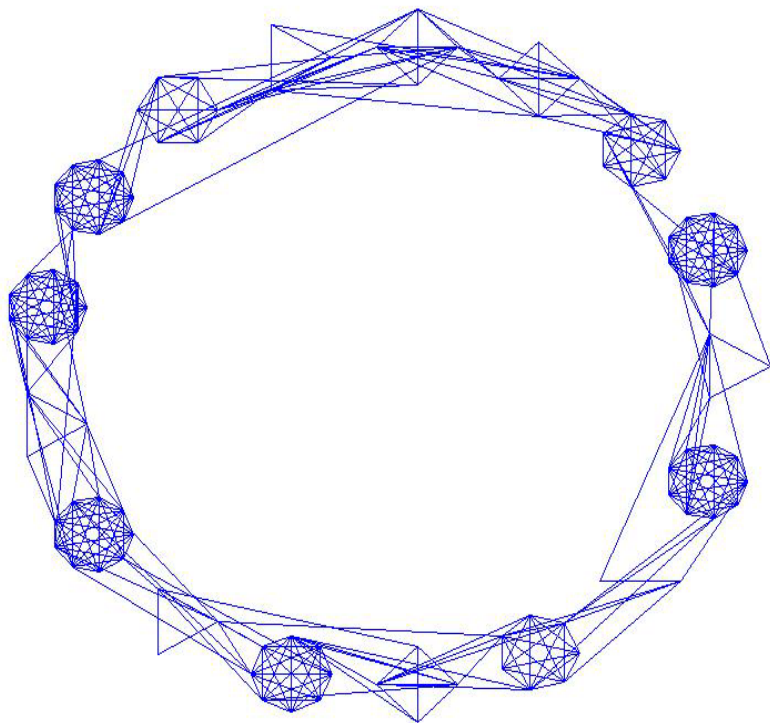
SC05 SSCA#2 OpenMP Contest:

<http://www.openmp.org/drupal/sc05/omp-contest.htm>

First prize: \$1000 plus a 60GB iPod.

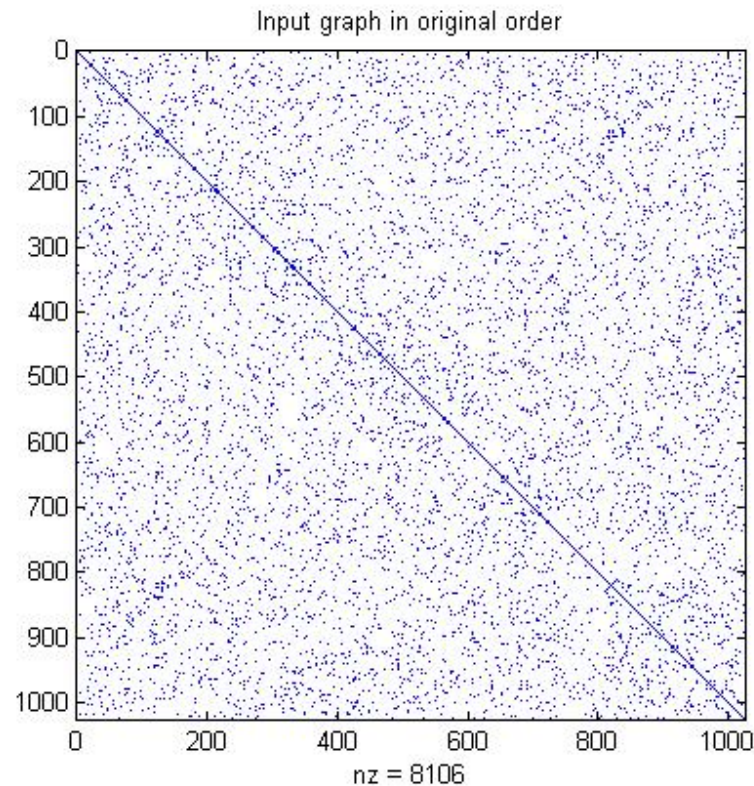


SSCA#2: “Graph Analysis”



- Many tight clusters, loosely interconnected
- Input data is edge triples $\langle i, j, \text{label}(i,j) \rangle$
- Vertices and edges permuted randomly

Concise SSCA#2 in Star-P



Kernel 1: Construct graph data structures

- Graphs are dsparse matrices, created by `sparse()`

Kernels 2 and 3

Kernel 2: Search by edge labels

- About 12 lines of executable Matlab or Star-P

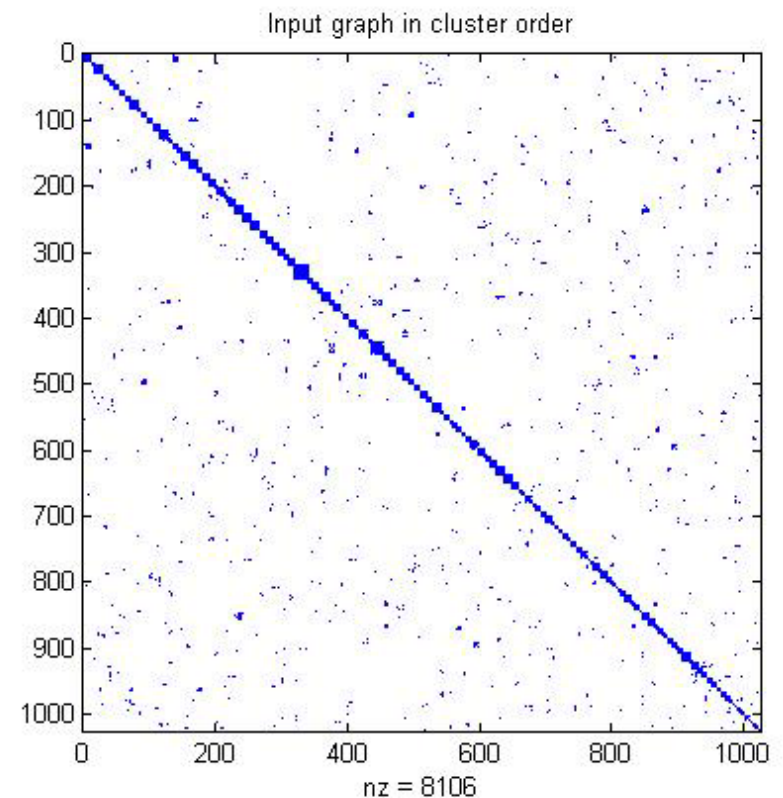
Kernel 3: Extract subgraphs

- Returns subgraphs consisting of vertices and edges within fixed distance of given starting vertices
- Sparse matrix-vector product for breadth-first search
- About 25 lines of executable Matlab or Star-P

Kernel 4: Vertex clustering

- Grow local clusters from many seeds in parallel
- Breadth-first search by sparse matrix * matrix

```
% Grow each seed to vertices  
%   reached by at least two  
%   paths of length 1 or 2  
  
C = sparse(seeds, 1:ns, 1, n, ns);  
C = A * C;  
C = C + A * C;  
C = C > 1;
```



Expressive Power: SSCA#2 Kernel 3

Star-P (25 lines)

```
A = spones(G.edgeWeights{1});
nv = max(size(A));
npar = length(G.edgeWeights);
nstarts = length(starts);
for i = 1:nstarts
    v = starts(i);
    % x will be a vector whose nonzeros
    % are the vertices reached so far
    x = zeros(nv,1);
    x(v) = 1;
    for k = 1:pathlen
        x = A*x;
        x = (x ~= 0);
    end;
    vtxmap = find(x);
    S.edgeWeights{1} = G.edgeWeights{1}(vtxmap);
    for j = 2:npar
        sg = G.edgeWeights{j}(vtxmap, vtxmap);
        if nnz(sg) == 0
            break;
        end;
        S.edgeWeights{j} = sg;
    end;
    S.vtxmap = vtxmap;
    subgraphs{i} = S;
end
```

MATLABmpi (91 lines)

```
declareGlobals;

% Wait for a response for each request we sent out.
for unused = 1:numRqst
    [src tag] = probeSubgraphs(G, [P tag K3 dataReq]);
    [starts newEdges] = MPI_Recv(src, tag, P, comm);
    subg.edgeWeights{1}(:, starts) = newEdges;
    [newEnds unused] = find(newEdges);
    allNewEnds = [allNewEnds; newEnds];
end
end % of if -P para

% Eliminate any new ends already in the all starts list.
newStarts = setdiff(allNewEnds, allStarts);
allStarts = [allStarts; newStarts];

if ENABLE_PLOT_K3DB
    plotEdges(subg.edgeWeights{1}, startVertex, endVertex, k);
end % of ENABLE_PLOT_K3DB

if isempty(newStarts) % if empty we can quit early.
    break;
end
end
```

<u>Lines of code</u>	cSSCA2	executable spec	C/Pthreads/SIMPLE
Kernel 1	29	68	256
Kernel 2	12	44	121
Kernel 3	25	91	297
Kernel 4	44	295	241

```
% For each processor which has any of the vertices we need.
startDest = floor((newStarts - 1) / P.myV);
uniqDest = unique(startDest);
for dest = uniqDest
    starts = newStarts(startDest == dest);

    if dest == P.myRank
        newEdges = G.edgeWeights{1}(:, starts - P.myBase);
        subg.edgeWeights{1}(:, starts) = newEdges;
        [allNewEnds unused] = find(newEdges);
        allNewEnds = [allNewEnds; newEnds];
    end
    MPI_Send(dest, P tag K3 dataReq, P, comm, starts);
    numRqst = numRqst + 1;
end
```

Scaling up

Recent runs of cSSCA#2 on SGI Altix (up to 128 processors):

- Have run kernels 1 – 3 on graphs with $2^{26} = 67$ million vertices
- Have run kernel 4 on graphs with $2^{24} = 16$ million vertices
(held the record until two weeks ago)
- Have built graphs with 400 million vertices and 4 billion edges
- Timings scale well – for large graphs,
 - 2x problem size $\rightarrow$ 2x time
 - 2x problem size & 2x processors $\rightarrow$ same time

Using this benchmark to tune lots of internal implementations

Remarks

- Easy-to-use interactive programming environment
 - Rapid prototyping tool for algorithm development
 - Harness for assembling libraries, data, and visualization
 - Alan E: “parallel computing is fun again 😊”
- Interface to existing parallel packages
- Combinatorial methods toolbox built on parallel sparse matrix infrastructure
- A few other research issues . . .
 - Dynamic resource management
 - Fault management
 - Programming in the large