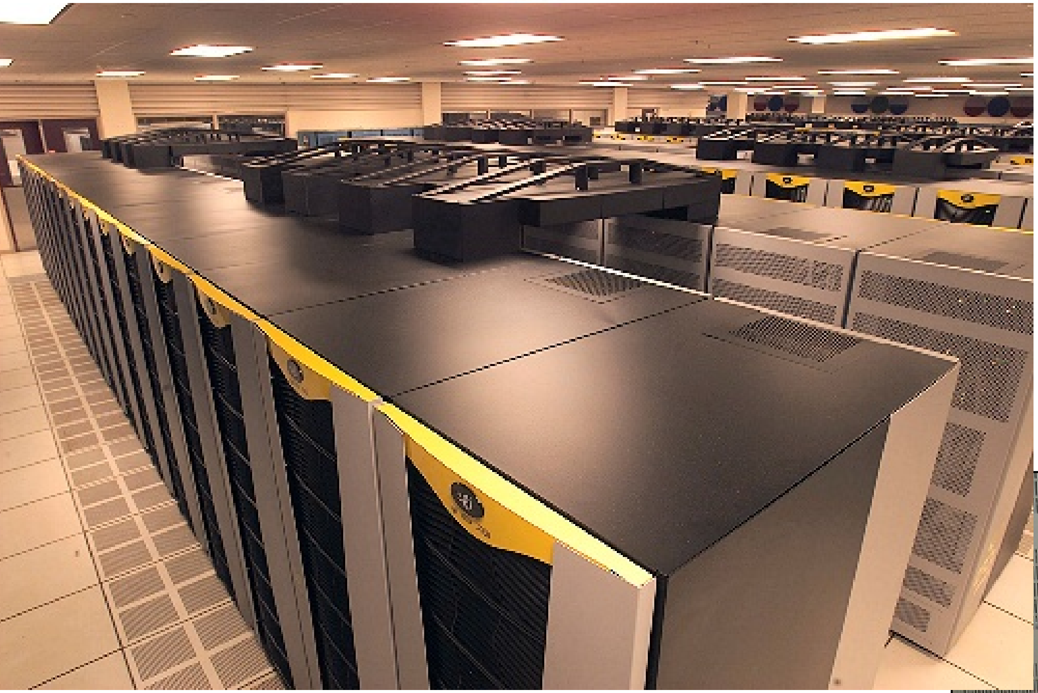
Interactive Tools for Combinatorial Supercomputing

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();
```


UCSB

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;
```

Background

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

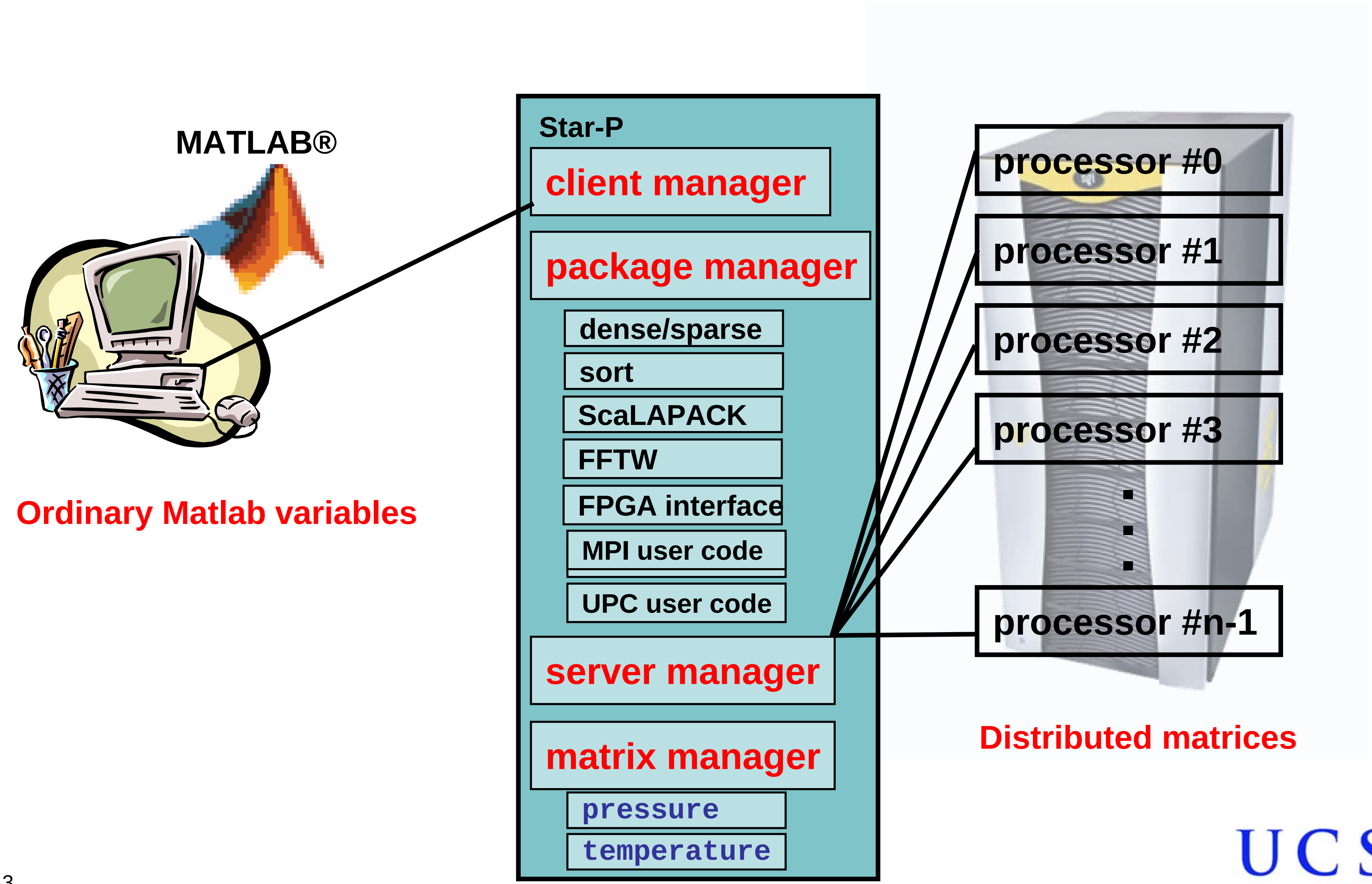
Data-Parallel Operations

```
                    < 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
   0.25000000000000
   0.50000000000000
   0.75000000000000
>> 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



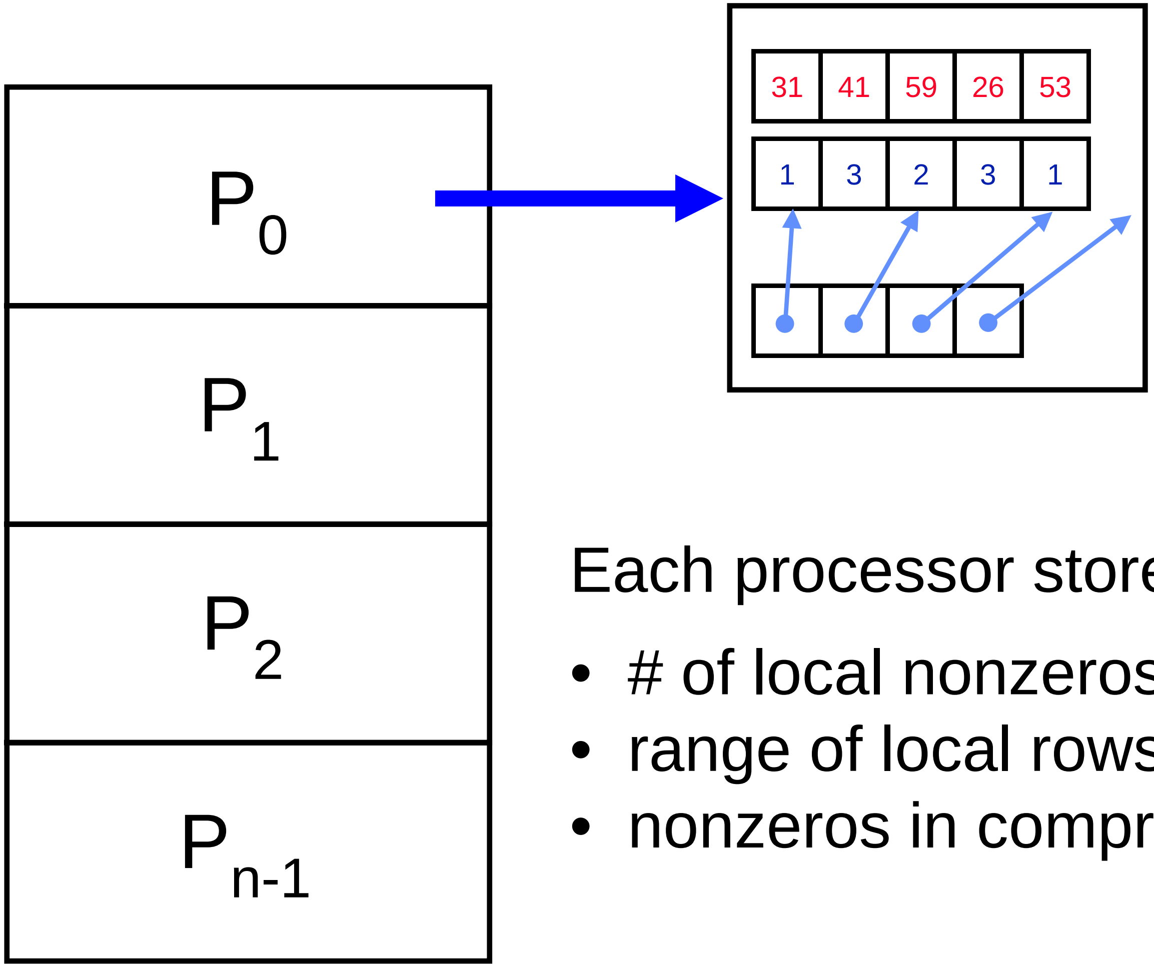
Matlab sparse matrix design principles

- 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(nonzeros)
- Time for a sparse operation should be O(flops) (as nearly as possible)

Star-P dsparse matrices: same principles, but some different tradeoffs

UCSB

Distributed sparse data structure



Each processor stores:

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

The sparse( ) constructor

- `A = sparse (I, J, V, nr, nc)`
- Input: ddense vectors **I**, **J**, **V**, dimensions **nr**, **nc**
- Output: **A**(**I**(k), **J**(k)) = **V**(k)
- Sort triples **<** i, j, v **>** by **<** i, j **>**
- Locally convert to compressed row indices
- Sum values with duplicate indices

Sparse array and matrix operations

- dsparse layout, same semantics as ddense
- Data distribution by rows only, for now
- Matrix arithmetic: +, *, max, sum, etc.
- Matrix indexing and concatenation

  A (1:3, [4 5 2]) = [B(:, J) C] ;

- Linear solvers: x = A \ b; using SuperLU (MPI)
- Eigensolvers: [V, D] = eigs(A); using PARPACK (MPI)

UCSB

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

Combinatorial Scientific Computing

- Sparse matrix methods
- Knowledge discovery
- Web search and information retrieval
- Graph matching
- Machine learning
- Geometric modeling
- Computational biology
- Bioinformatics
- . . .

How will combinatorial methods be used by nonexperts?

UCSB

Analogy: Matrix division in Matlab

x = A \ 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(:, colamd(A))

UCSB

Combinatorics in Star-P

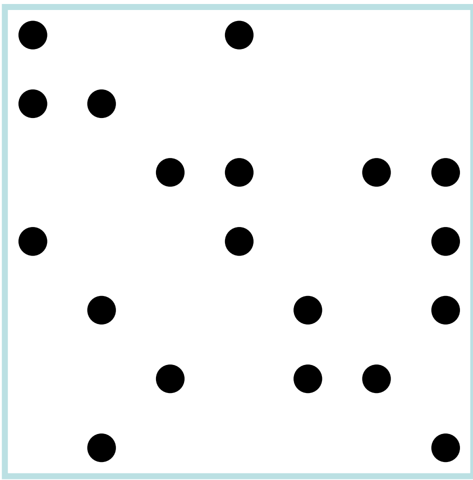
- Represent a graph as a sparse adjacency matrix

- A sparse matrix language is a good start on primitives for computing with graphs

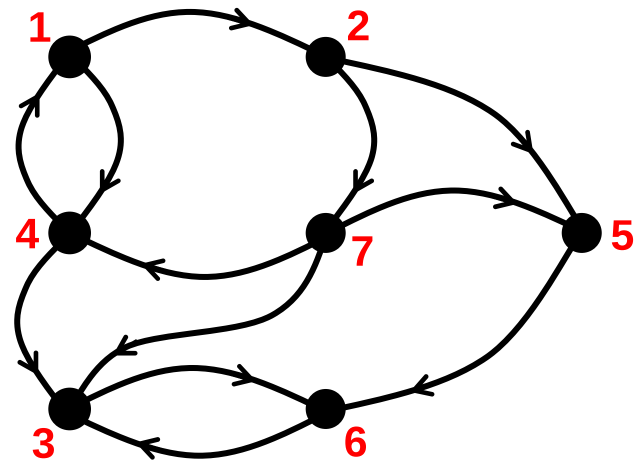
  - Random-access indexing: `A(i,j)`
  - Neighbor sequencing: `find (A(i,:))`
  - Sparse table construction: `sparse (I, J, V)`
  - Breadth-first search step : `A * v`

UCSB

Sparse adjacency matrix and graph

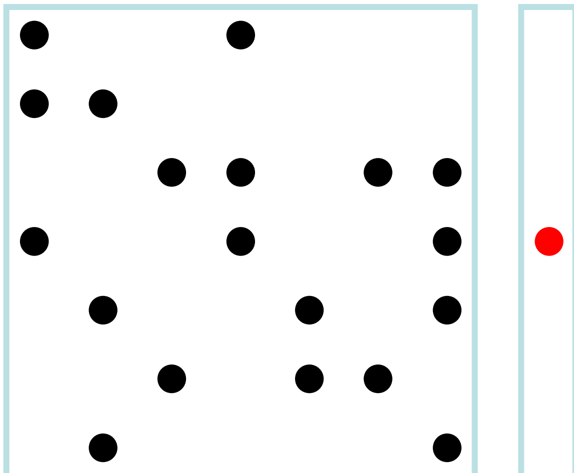


A^T

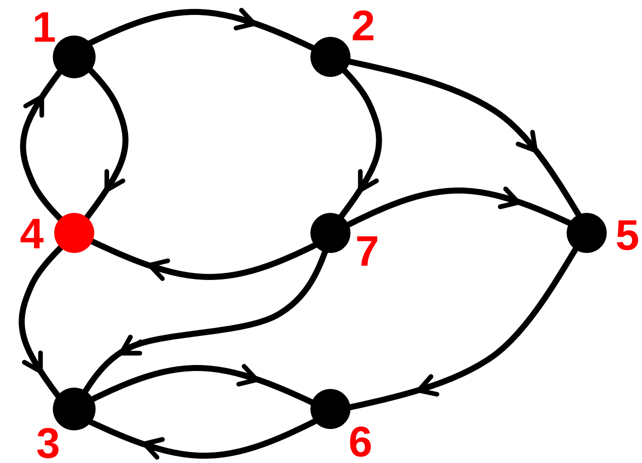


- Adjacency matrix: sparse array w/ nonzeros for graph edges
- Storage-efficient implementation from sparse data structures

Breadth-first search: sparse mat * vec

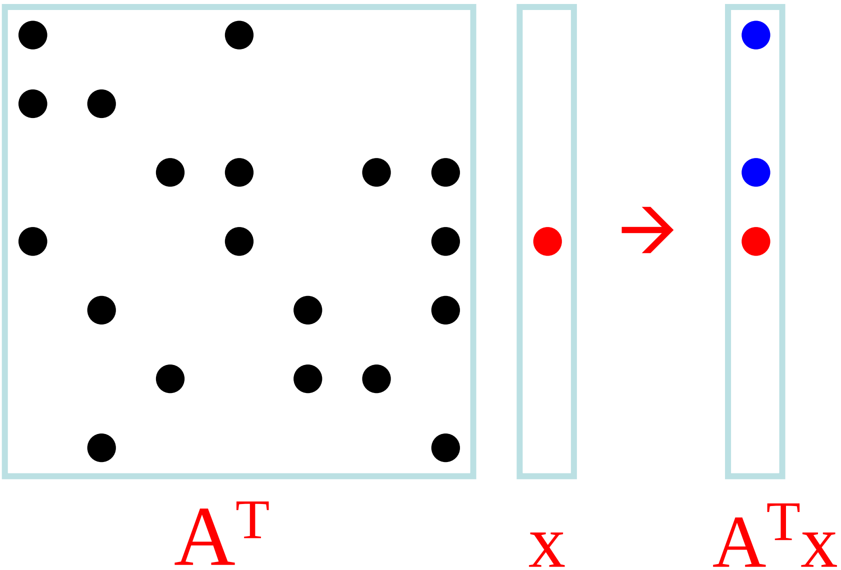


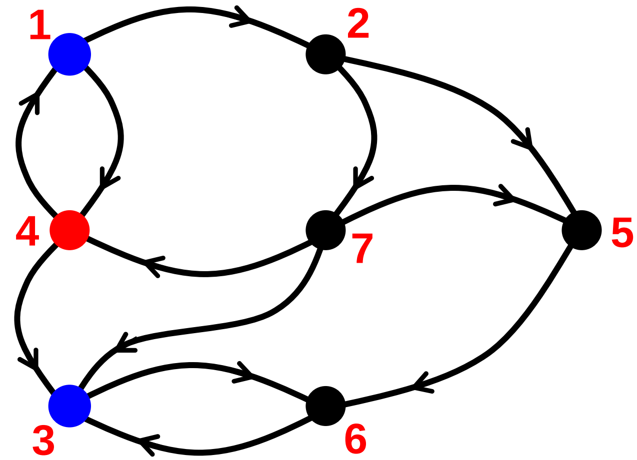
A^T x



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

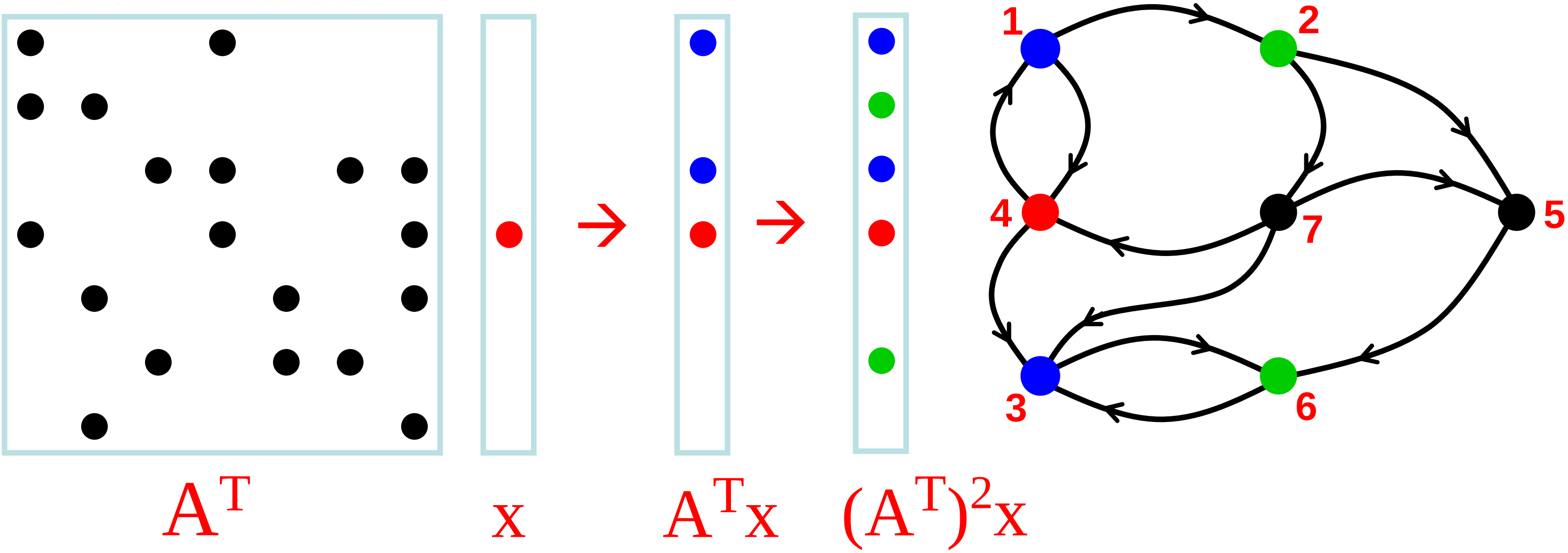
Breadth-first search: sparse mat * vec





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

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
- Pointer-jumping algorithms (Shiloach/Vishkin & descendants)
  - repeat
    - Link every (super)vertex to a neighbor
    - Shrink each tree to a supervertex by pointer jumping
  - until no further change
- Other coming graph kernels:
  - Shortest-path search (after Husbands, LBNL)
  - Bipartite matching (after Riedy, UCB)
  - Strongly connected components (after Pinar, LBNL)

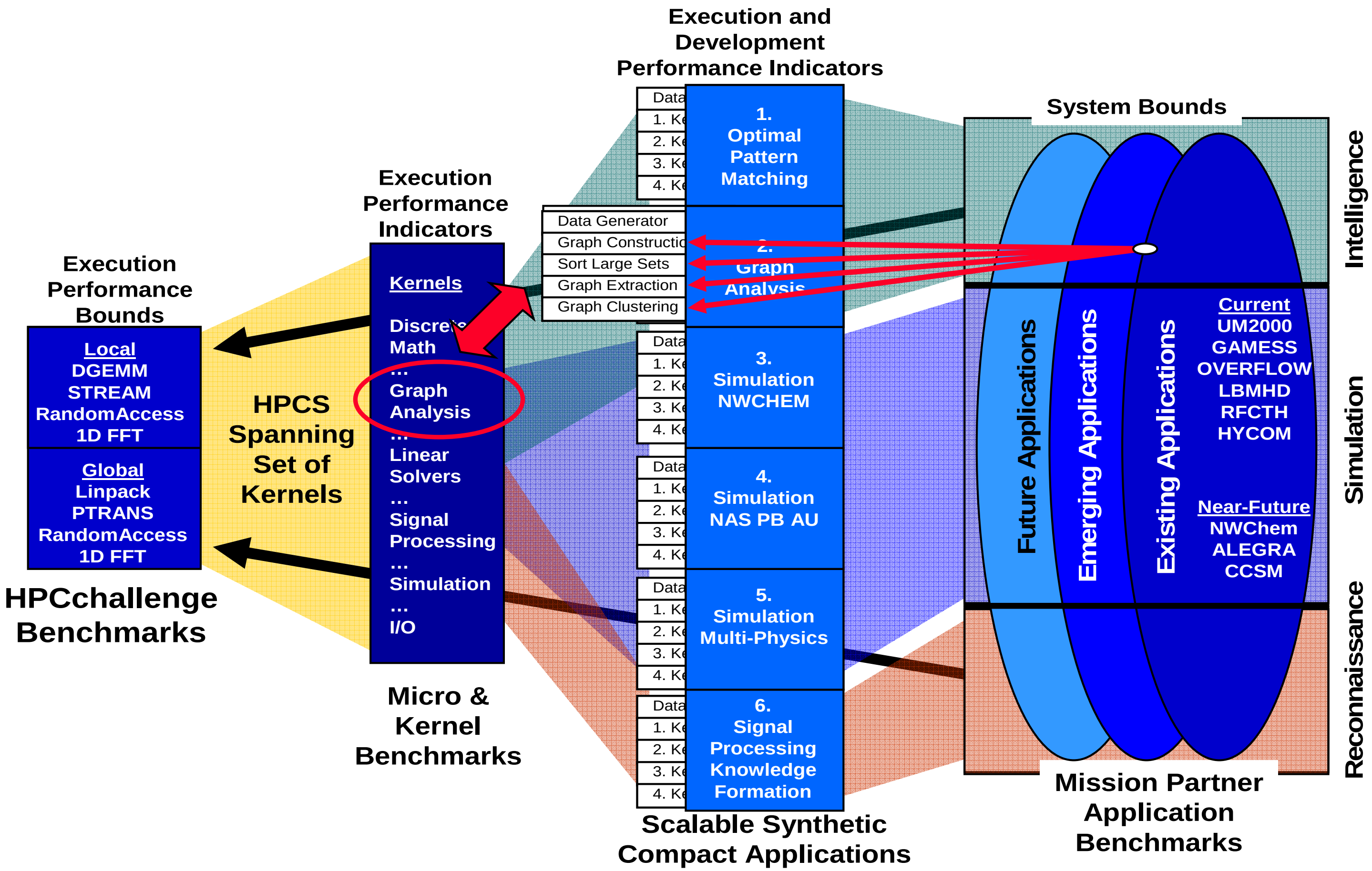
UCSB

HPCS Scalable Synthetic Compact Apps

SSCA#1: Bioinformatics / Pattern Matching

SSCA#2: Graph Analysis

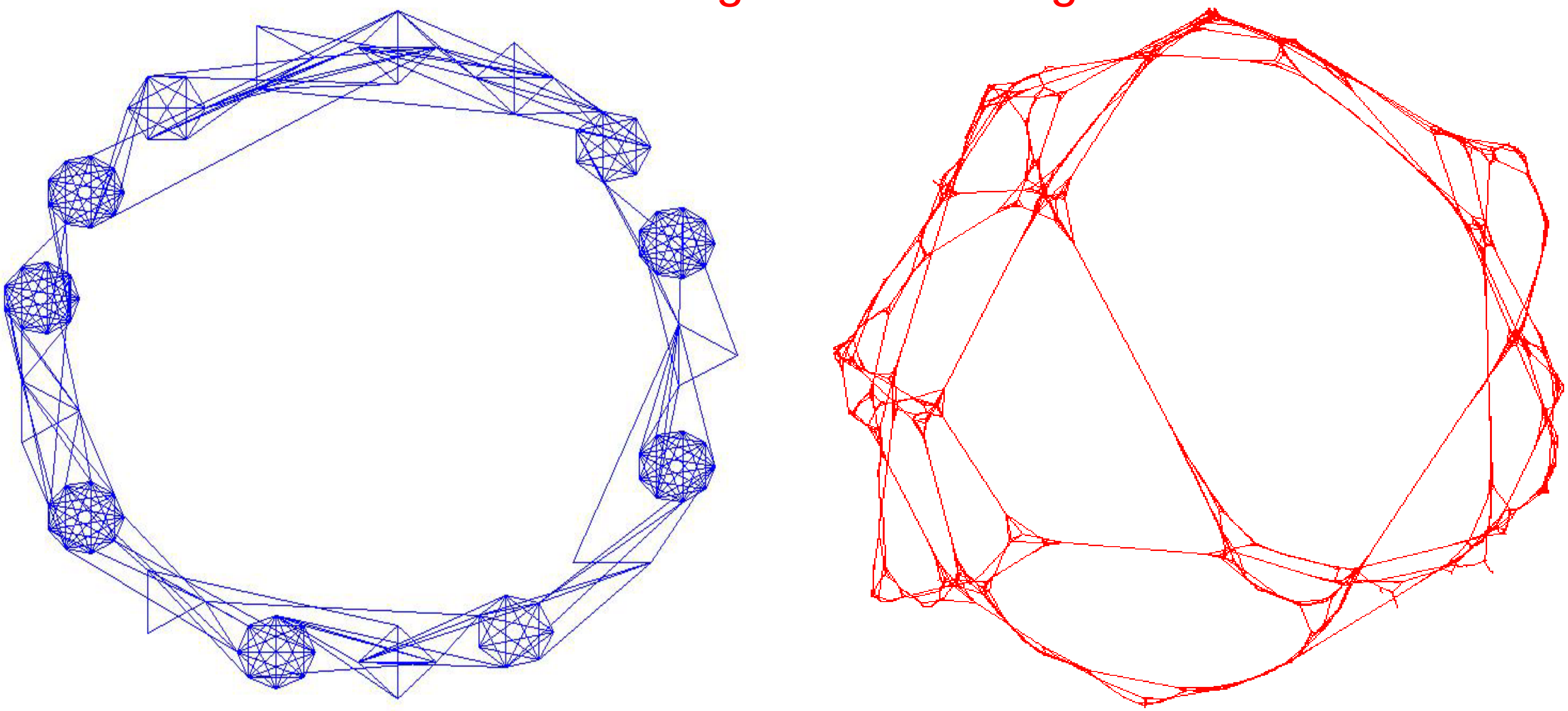
SSCA#3: Sensor Processing / Imaging



SSCA#2: “Graph Analysis”

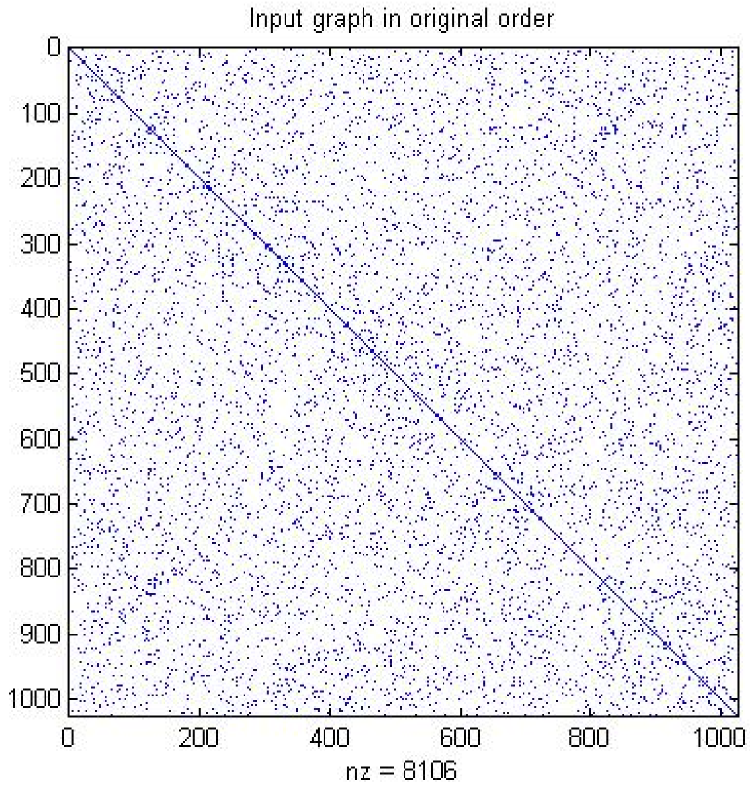
Fine-grained, irregular data access

Searching and clustering



- Many tight clusters, loosely interconnected
- Input data is edge triples < i, j, label(i,j) >
- 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

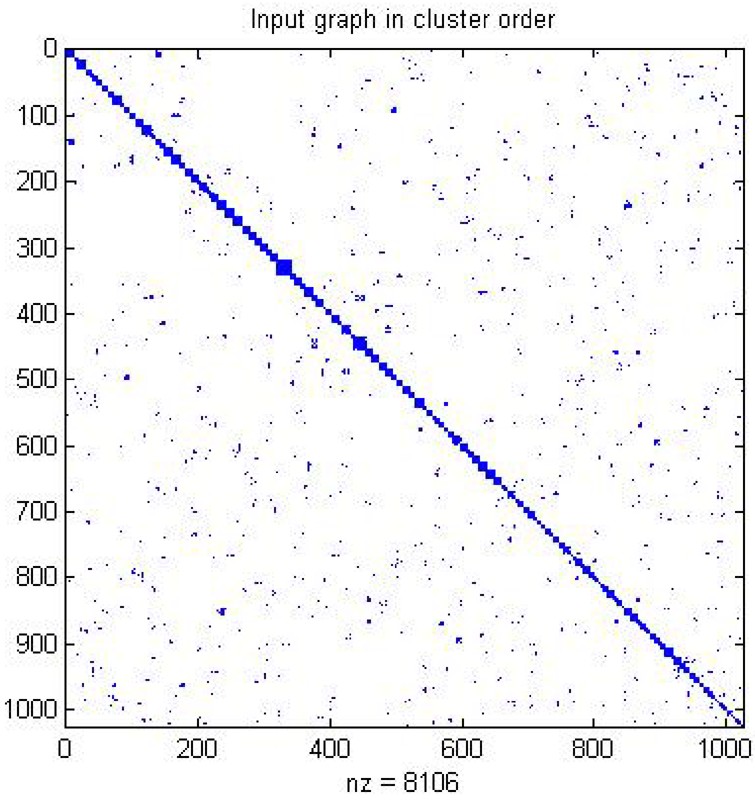
Kernel 3: Extract subgraphs

- 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}(v
    for j = 2:npar
        sg = G.edgeWeights{j}(vtxmap,vtxm
        if nnz(sg) == 0
            break;
        end;
        S.edgeWeights{j} = sg;
    end;
    S.vtxmap = vtxmap;
    subgraphs{i} = S;
end
```

MATLABmpi (91 lines)

```
declareGlobals;


intSubgraphs = subgraphs(G, pathLength, startSetInt);
strSubgraphs = subgraphs(G, pathLength, startSetStr);

%| Finish helping other processors.
if P.Ncpus > 1
  if P.myRank == 0                % if we are the leader
    for unused = 1:P.Ncpus-1
      [src tag] = probeSubgraphs(G, [P.tag.K3.results]);
      [isg ssg] = MPI_Recv(src, tag, P.comm);
      intSubgraphs = [intSubgraphs isg];
      strSubgraphs = [strSubgraphs ssg];
    end
    for dest = 1:P.Ncpus-1
      MPI_Send(dest, P.tag.K3.done, P.comm);
    end
  else
    MPI_Send(0, P.tag.K3.results, P.comm, ...
        intSubgraphs, strSubgraphs);
    [src tag] = probeSubgraphs(G, [P.tag.K3.done]);
    MPI_Recv(src, tag, P.comm);
  end
```

```
      % Wait for a response for each request we sent out.
      for unused = 1:numRqst
        [src tag] = probeSubgraphs(G, [P.tag.K3.dataResp]);
        [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.paral

    % 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
```

Lines of code	Star-P cSSCA2	MATLABmpi 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:
startDests = floor((newStarts - 1) / P.myV);
uniqDests = unique(startDests);
for dest = uniqDests
  starts = newStarts(startDests == dest);

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

Scaling up

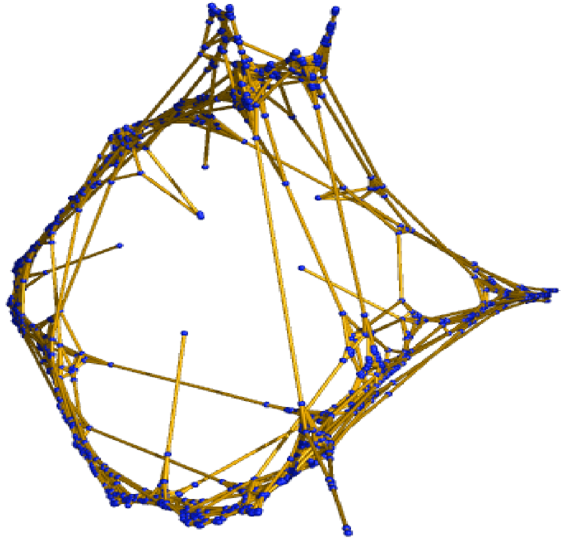
Early results on SGI Altix (up to 64 processors):

- Have run cSSCA2 on graphs with 2^{27} = 134 million vertices
- Have manipulated graphs with 400 million vertices and 4 billion edges
- Timings scale well – for large graphs,
  - 2x problem size → 2x time
  - 2x problem size & 2x processors → same time

Using this benchmark to tune lots of infrastructure

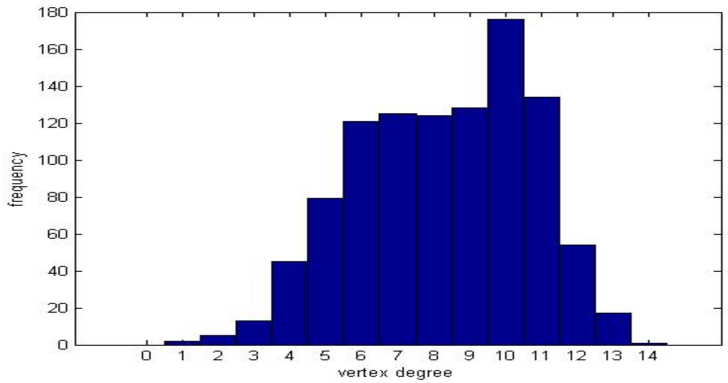
UCSB

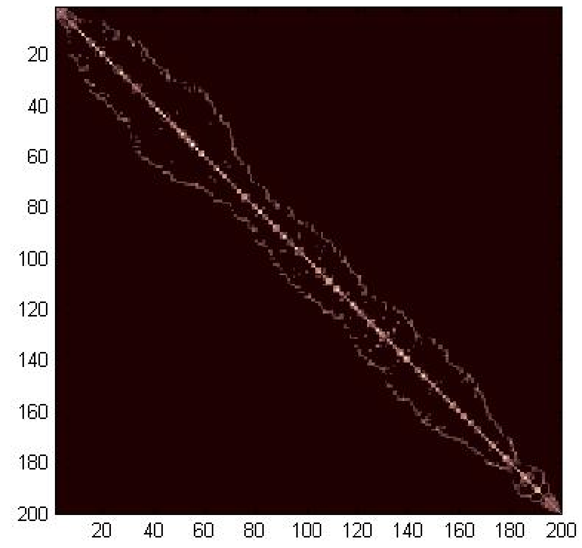
Work in progress: Toolbox for Graph Analysis and Pattern Discovery



Layer 1: Graph Theoretic Tools

- Graph operations
- Global structure of graphs
- Graph partitioning and clustering
- Graph generators
- Visualization and graphics
- Scan and combining operations
- Utilities





Work in progress: Toolbox for Graph Analysis and Pattern Discovery

Layer 1: Graph Theoretic Tools

- Graph operations
- Global structure of graphs
- Graph partitioning and clustering
- Graph generators
- Visualization and graphics
- Scan and combining operations
- Utilities