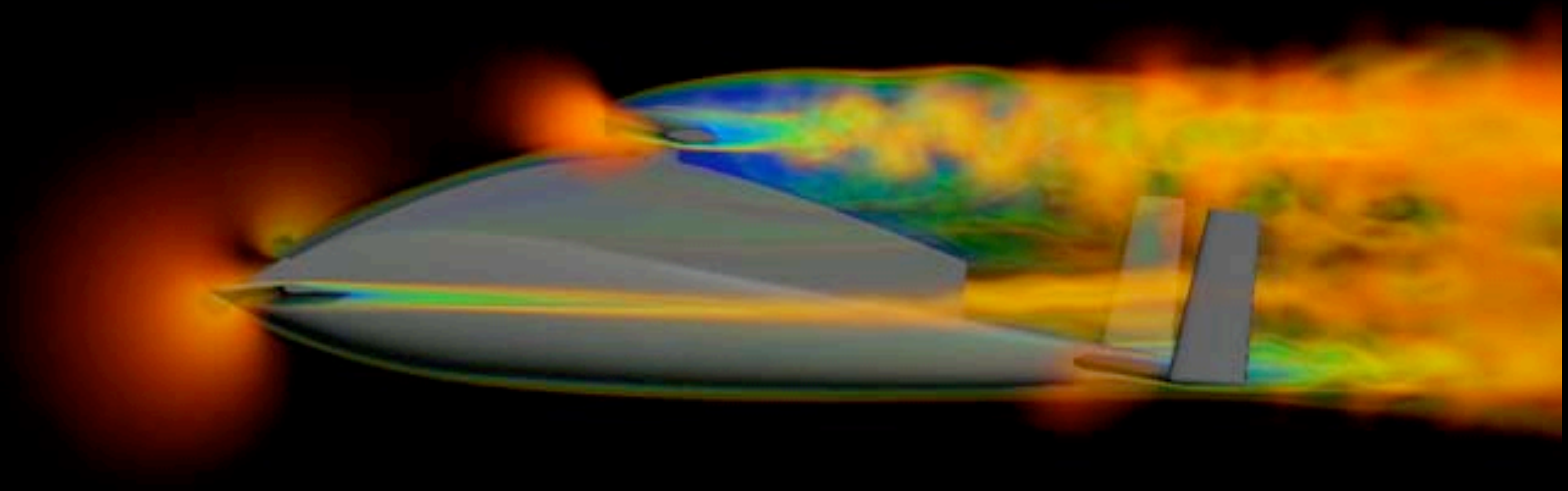
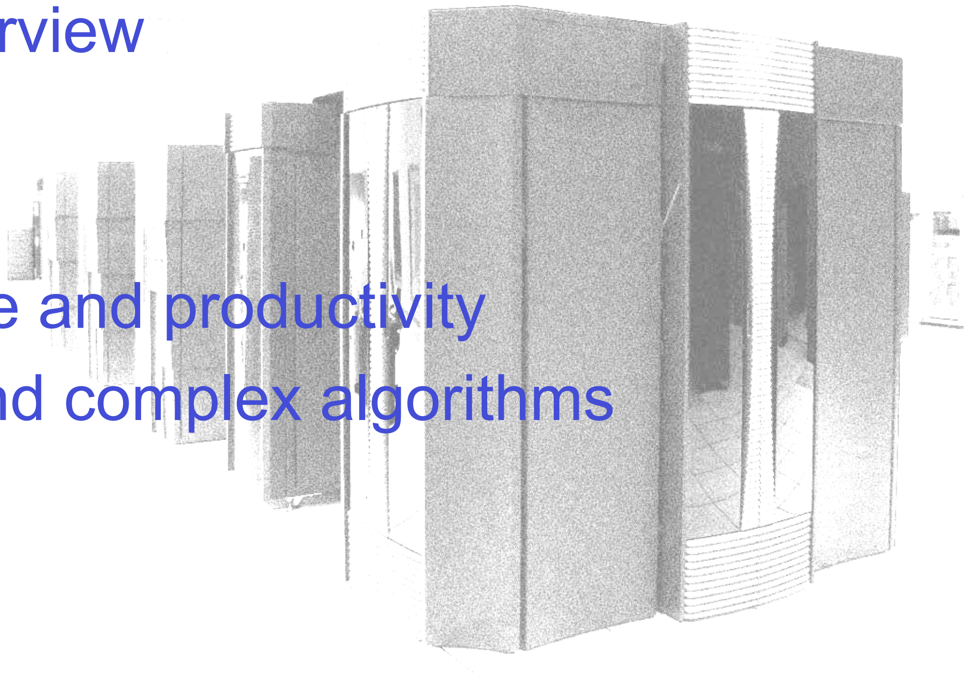
Unified Parallel C within Computational Fluid Dynamic Applications on the Cray X1(E)



Andrew A. Johnson, Ph.D.
Army HPC Research Center / NetworkCS, Inc.
Minneapolis, Minnesota

Outline

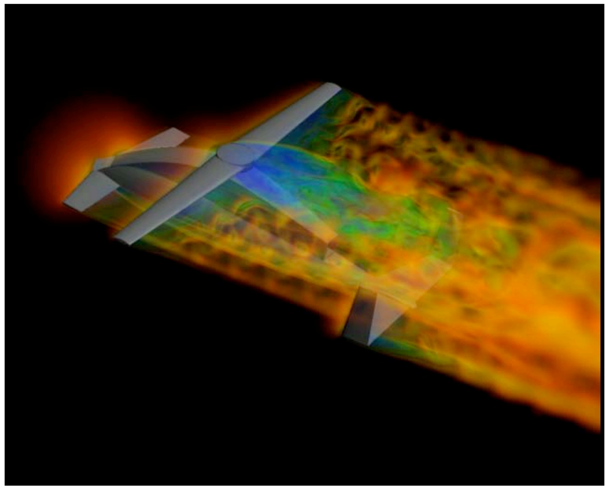


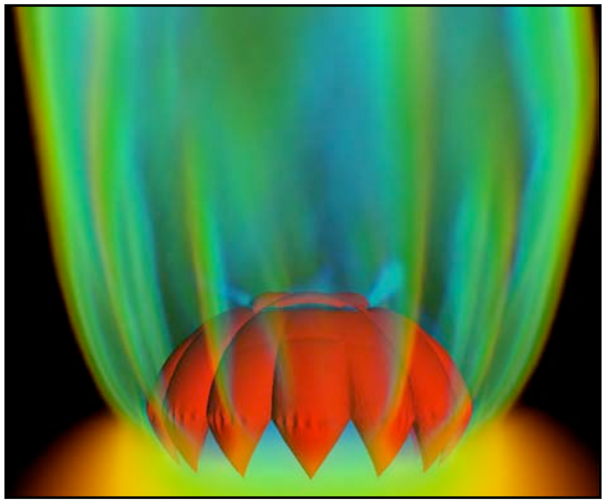
- CFD methods and code overview
- CFD performance
- UPC language concepts
- Benefit 1: Programming style and productivity
- Benefit 2: Implement new and complex algorithms
- Benefit 3: Performance

Army High Performance Computing Research Center
Contract DAAD19-03-D-0001

Computational Fluid Dynamics

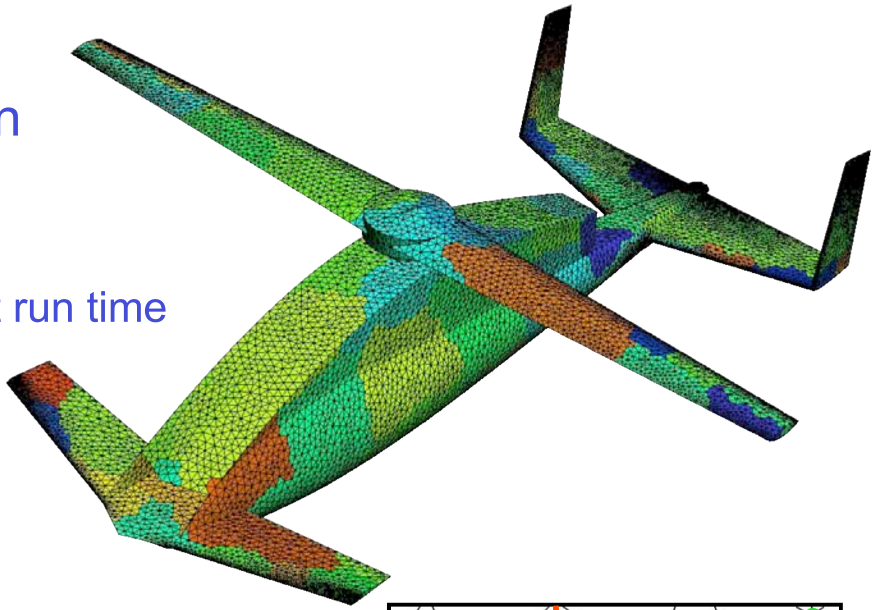
- Time accurate incompressible flow solvers built for unstructured meshes
  - Tetrahedral element meshes (mainly)
  - Finite Element method
  - Fully coupled GMRES-based iterative equation solver
    - Matrix-Free mode (most used)
    - Sparse-Matrix mode
- Developed at the AHPCRC
  - In use for over 14 years
  - Used for a wide variety of applications
- Fully parallel and scalable
- **BenchC** and Aeolus
  - Written entirely in C
  - Roughly 8,000 lines
  - Built-in performance monitoring

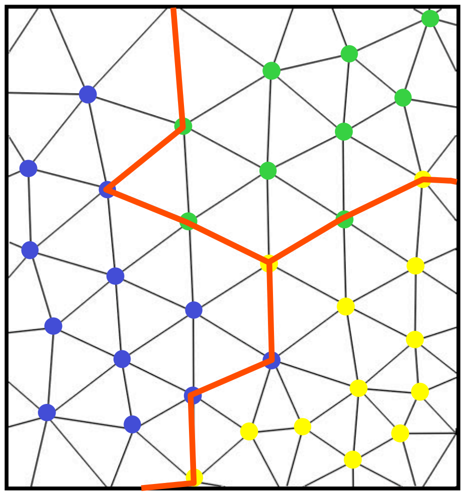

Tactical-Unmanned Aerial Vehicle


Parachute Aerodynamics

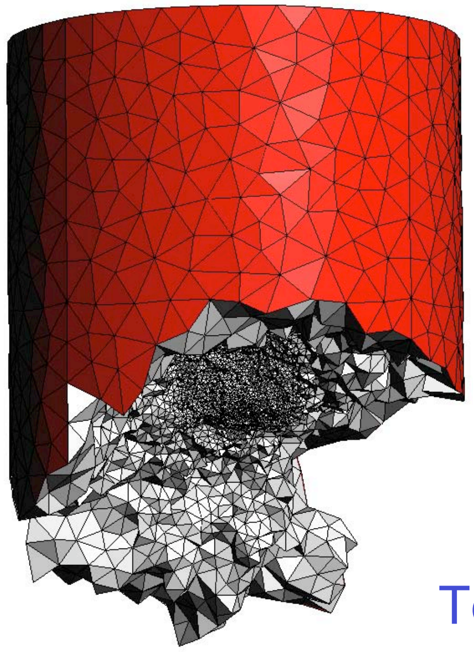
CFD Description (parallelism)

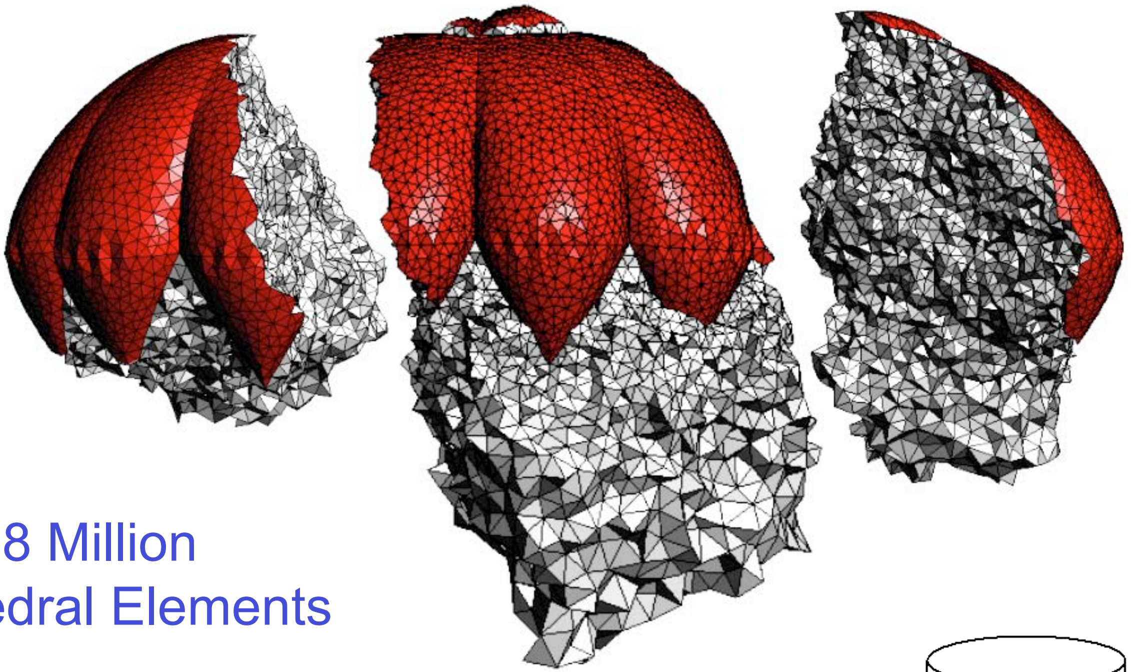
- Fully Parallel Based on MPI
  - Mesh partitioning and re-distribution
    - In-house RCB algorithm
    - Entirely built-in to the code
      - Number of processors specified at run time
  - Fast and efficient inter-processor communication
    - Non-blocking routines
  - Fully scalable
    - Both computation and memory
- Portable to All Parallel Systems
  - Requires only C and MPI
  - Tested on X1(E),T3E,SGI,IBM,Clusters,Mac
- Can also use UPC for the Core Communication Mechanism



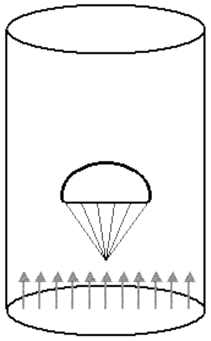


CFD Description (continued)

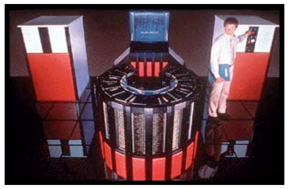




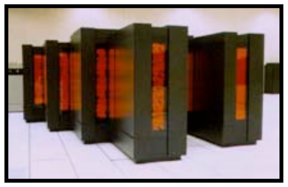
1.8 Million
Tetrahedral Elements



History (Parallel Systems/Programming)



Vectorization/Autotasking ~ 1990



CMF/CMSSL ~ 1991

PVM ~ 1994

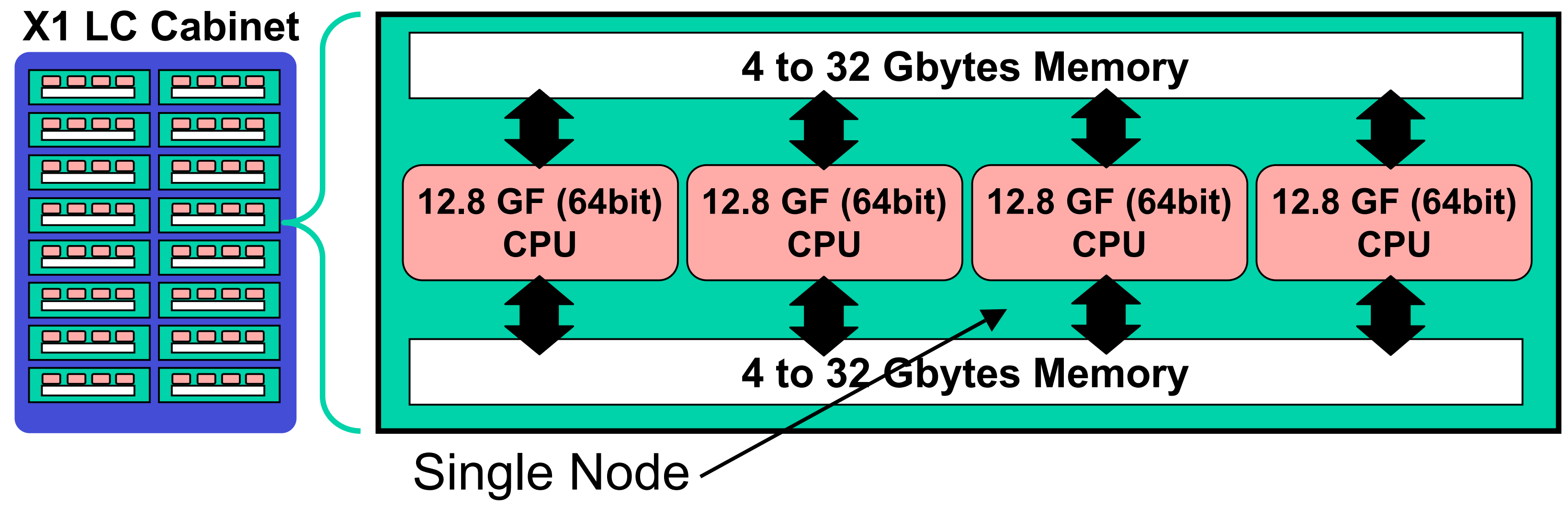
MPI ~ 1996

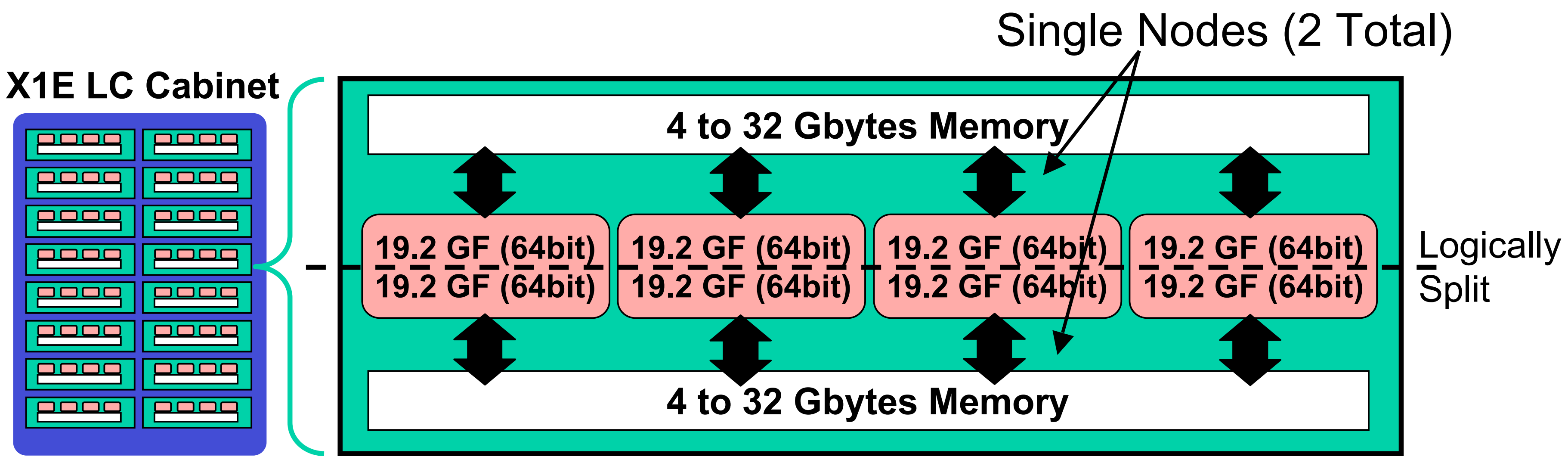
UPC/CAF/Vector ~ 2003

CFD Description (vectorization)

- Fully vectorized on the Cray X1(E)
  - 256 Multi-Streaming Processors
    - Vector processing elements
    - Very high sustained computational rates
  - 19.2 GFlops peak per processor (MSP)
  - Very fast interconnect network
    - Supports globally addressable memory
- Enables large-scale applications
  - 1 to 10 million elements have been common
  - Regularly using meshes with around 100 million elements these days
  - Applications with meshes containing up to 2.1 billion elements have been used

Cray X1 and X1E Differences





Vectorization of CFD Kernel

- One big vectorization challenge in unstructured mesh CFD codes
- Assembly (scatter) of element results to the mesh nodes

```
do i=1,number_of_elements
      node1 = ien(1,i)
      x1 = x(1,node1)
      y1 = x(2,node1)
      z1 = x(3,node1)
      …Several more lines like these…

      …Lots of calculations using these "localized" variables…


      d(1,node1) = d(1,node1) + result1
      d(2,node1) = d(2,node1) + result2
      d(3,node1) = d(3,node1) + result3
      …Several more lines like these…
enddo
```

Total number of mesh elements on this processor. Usually in the 100 thousands or more range.

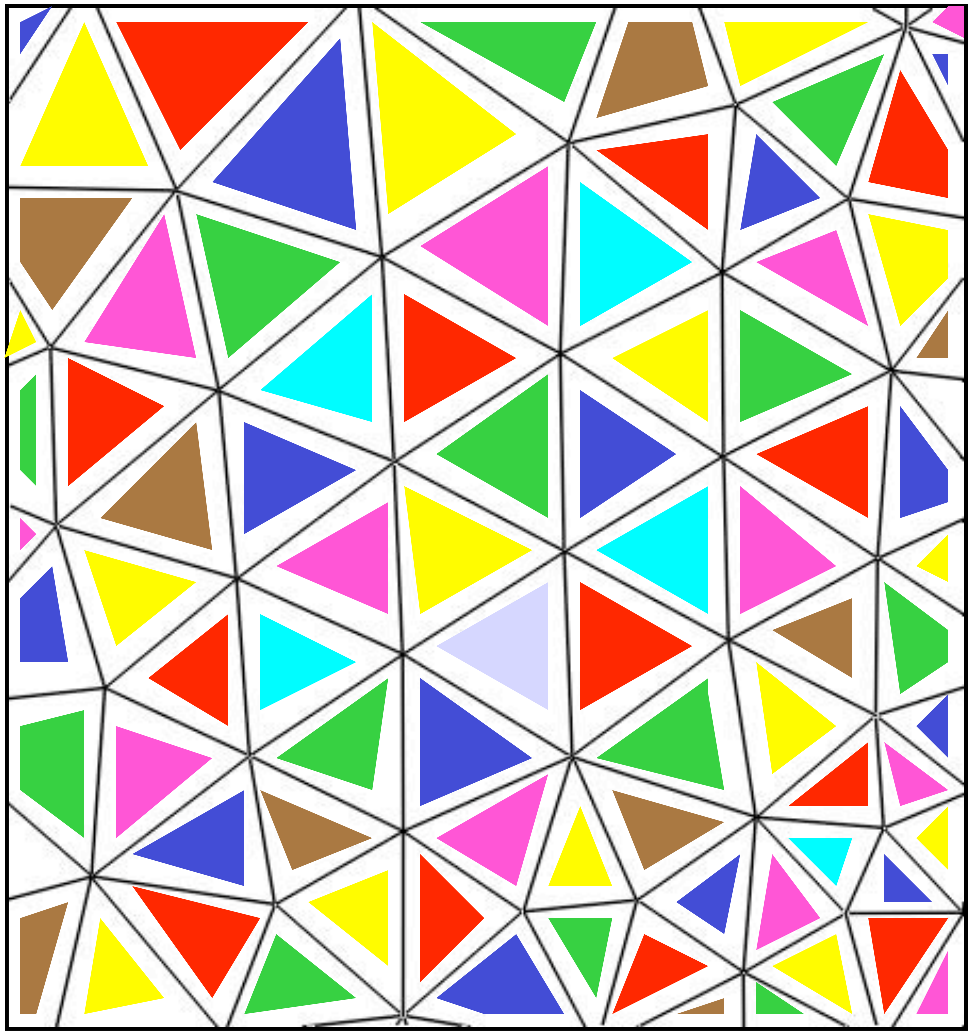
"Gather" values from global memory.

In the range of 1100 flops per iteration.

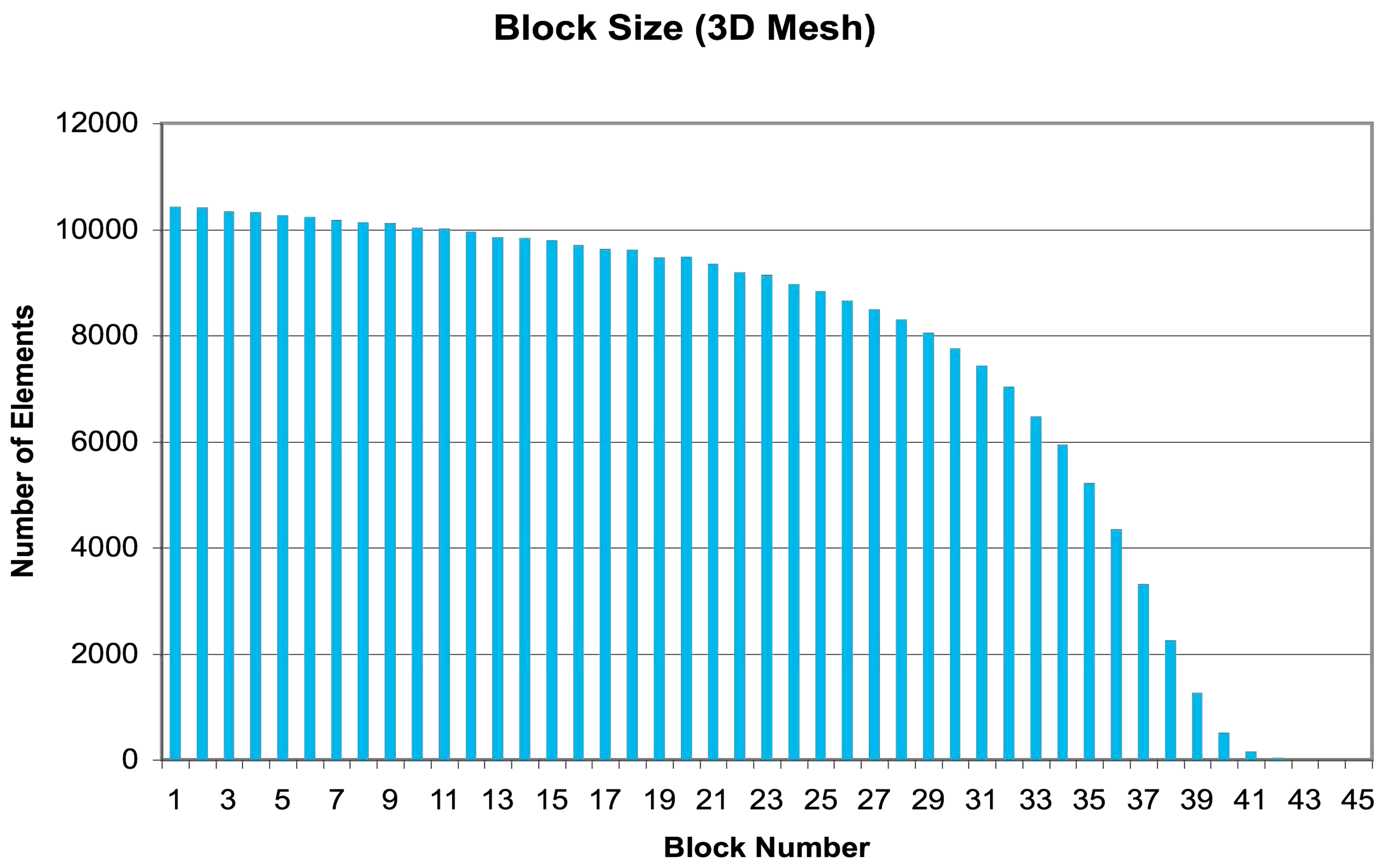
"Scatter" values back into global memory. (Traditional Finite Element Assembly Operation)

Element Coloring Algorithm

- No two elements in a group (color) can touch each other
- Fairly simple algorithm to compute the coloring
- **8 Colors in this example**
- Around 40 colors in 3D meshes



Element Coloring Algorithm (continued)



Vectorization of CFD Kernel (modification)

```
do ig=1,number_of_groups
   i1 = gg_begin(ig)
   i2 = gg_end(ig)

!DIR$ CONCURRENT
   do i=i1,i2
      node1 = ien(1,i)
      x1 = x(1,node1)
      y1 = x(2,node1)
      z1 = x(3,node1)
      …Several more lines like these…

      …Lots of calculations using these "localized" variables…

      d(1,node1) = d(1,node1) + result1
      d(2,node1) = d(2,node1) + result2
      d(3,node1) = d(3,node1) + result3
      …Several more lines like these…
   enddo
enddo
```

Outer group loop not vectorized.

Groups are pre-computed during the set-up stage.

Inner group loop can now be fully vectorized and multi-streamed. Iteration count is usually in the 10k or larger range. A few small group sizes exist at the end.

Can guarantee that these values will always be different for all iterations of this loop.

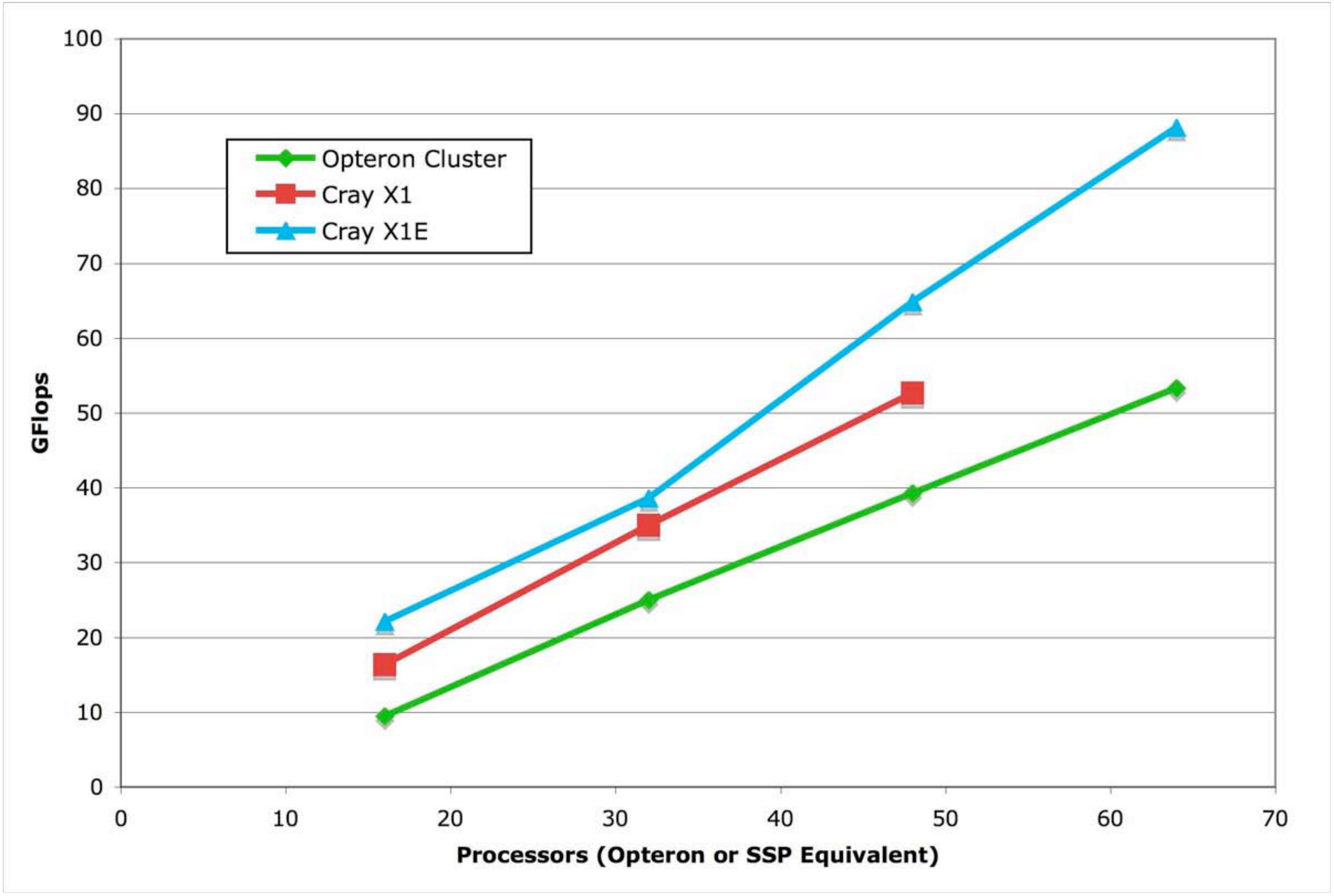
AHPCRC

Network Computing Services, Inc.
© 2005

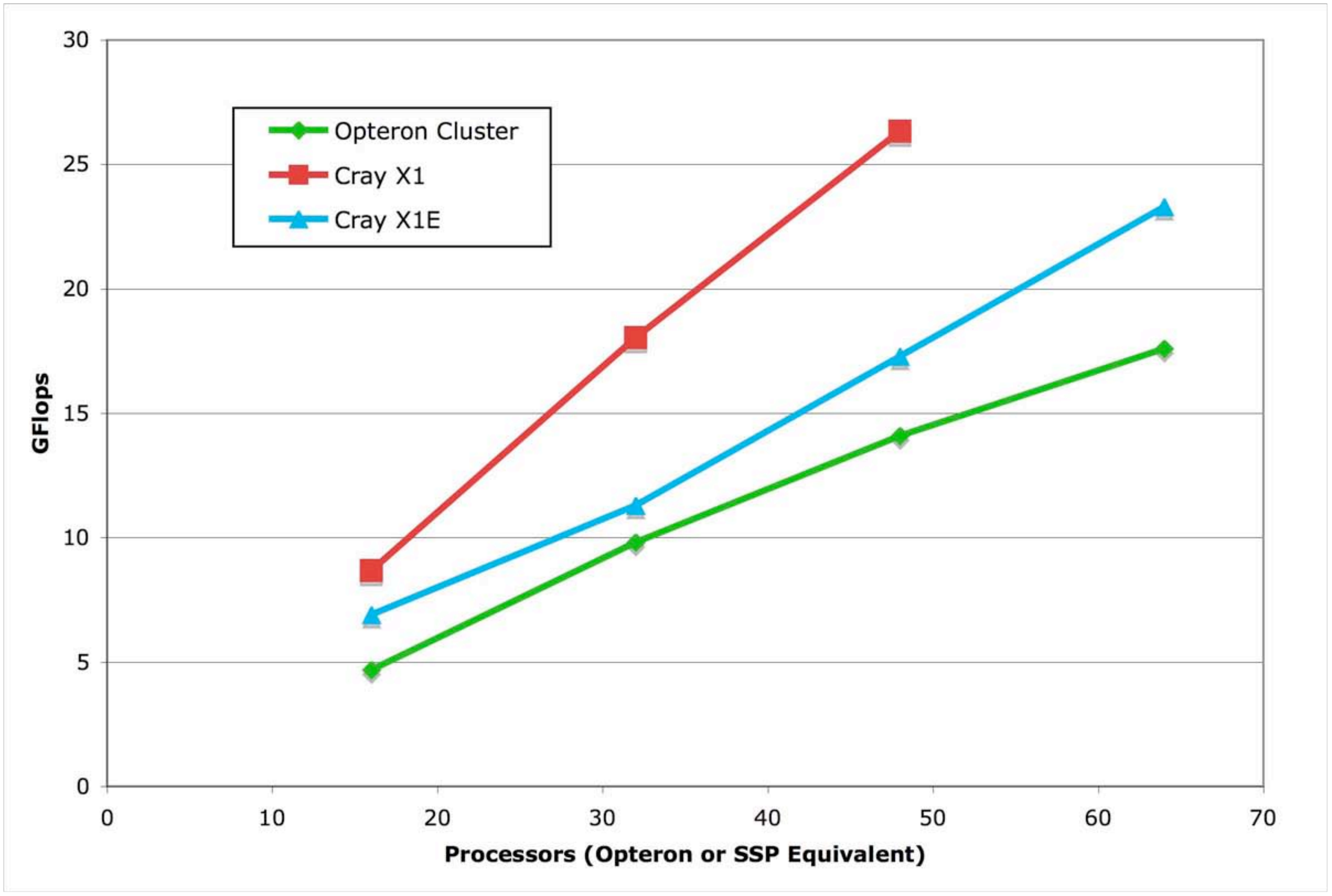
Performance

- Cray X1 and Cray X1E
  - BenchC application runs in MSP “mode”
  - Compared SSP “equivalents”
- Atipa Opteron Cluster
  - 2.2 GHz, 2 Processors per node
  - Myrinet network (switched)
- Tested a CFD data set with 2 million tetrahedral elements
  - 16, 32, 48, 64 Opterons
  - 4, 8, 12, 16 Cray X1(E) MSPs
- Report GFlops based on our estimation of total operations performed in each run
- No I/O or set-up time measured

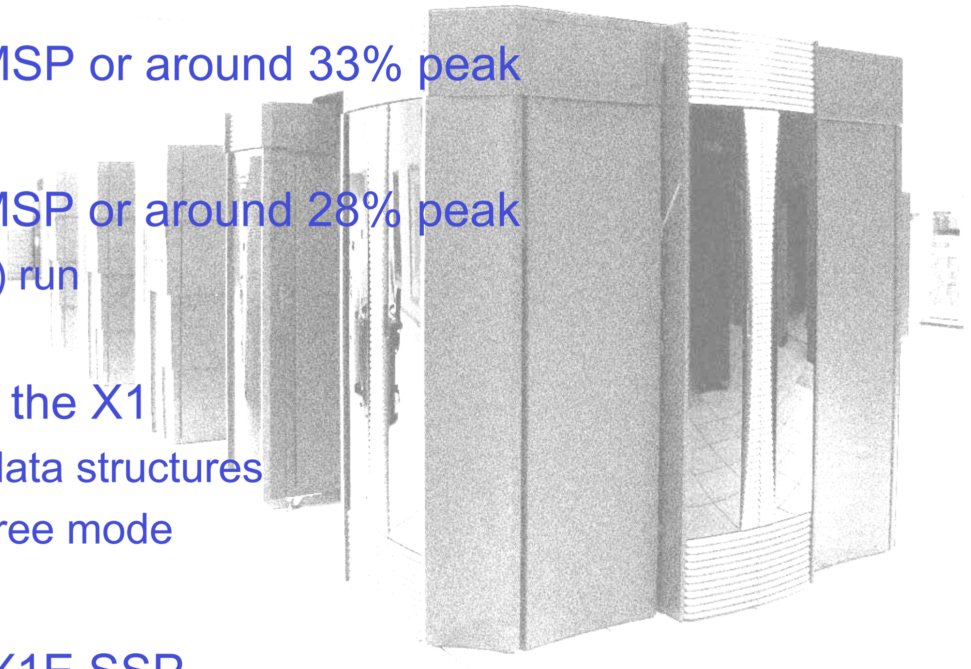
Performance - Matrix Free Mode



Performance - Sparse Matrix Mode

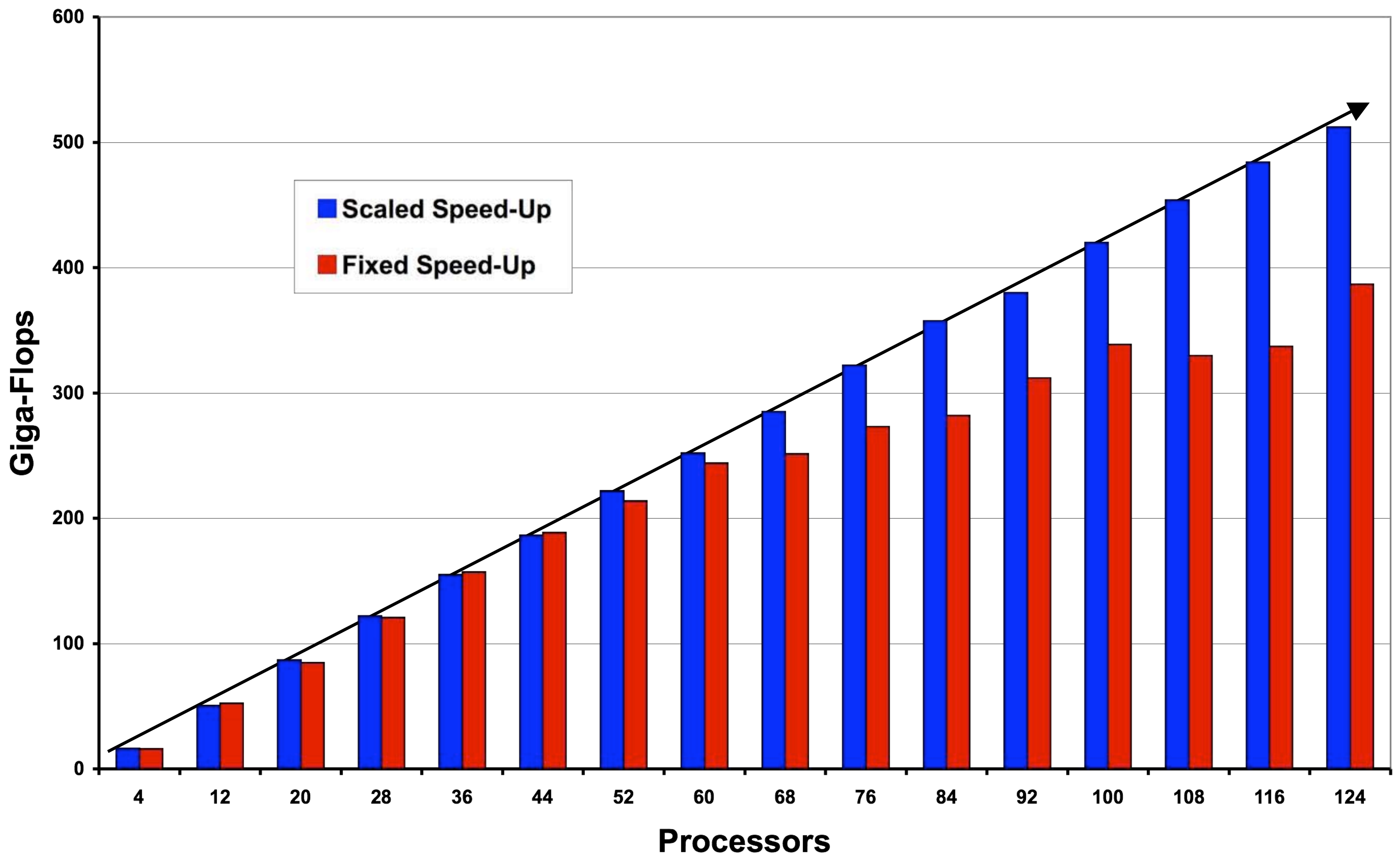


Performance - Summary

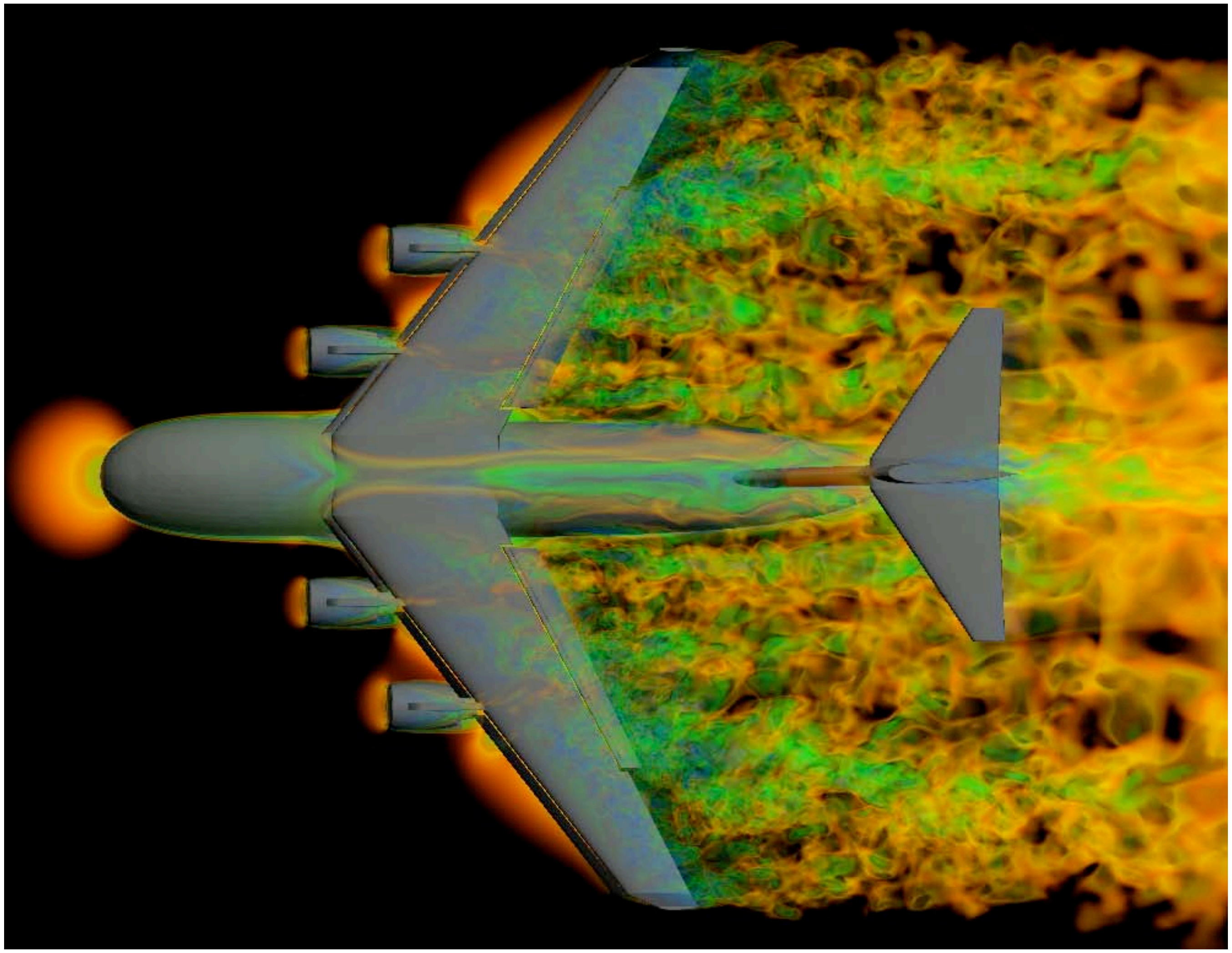


- Cray X1
  - About 4.2 GFlops sustained per MSP or around 33% peak
- Cray X1E
  - About 5.5 GFlops sustained per MSP or around 28% peak
    - Aberration at the 8 MSP (32 SSP) run
    - Various kernel routines run faster
  - Sparse mode behaves worst than the X1
    - More complex vectorization and data structures
    - Not used as much as the matrix-free mode
- Opteron Cluster
  - Roughly 60% performance of an X1E SSP
  - Wide variability in performance
    - 10% to 30% variability in performance
      - 4 separate runs within a 2 hour time frame
    - Hampers our efforts for large-scale scalability tests
- Communication times were a few percent of total run time in all cases

Application Scalability (X1)

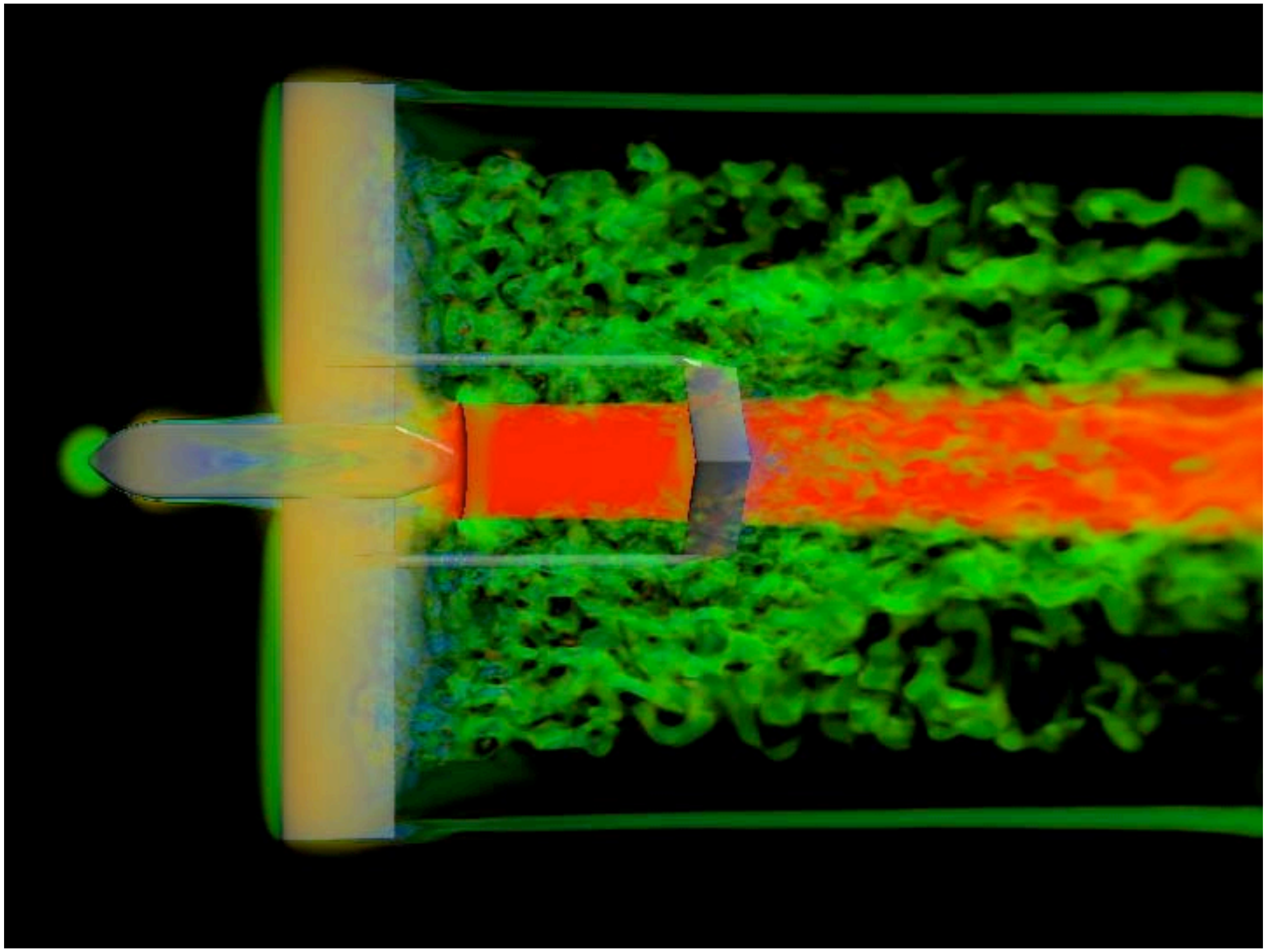


Large CFD Calculations



Volume rendering of airflow past a cargo aircraft in a take-off configuration. (243 million tetrahedral elements)

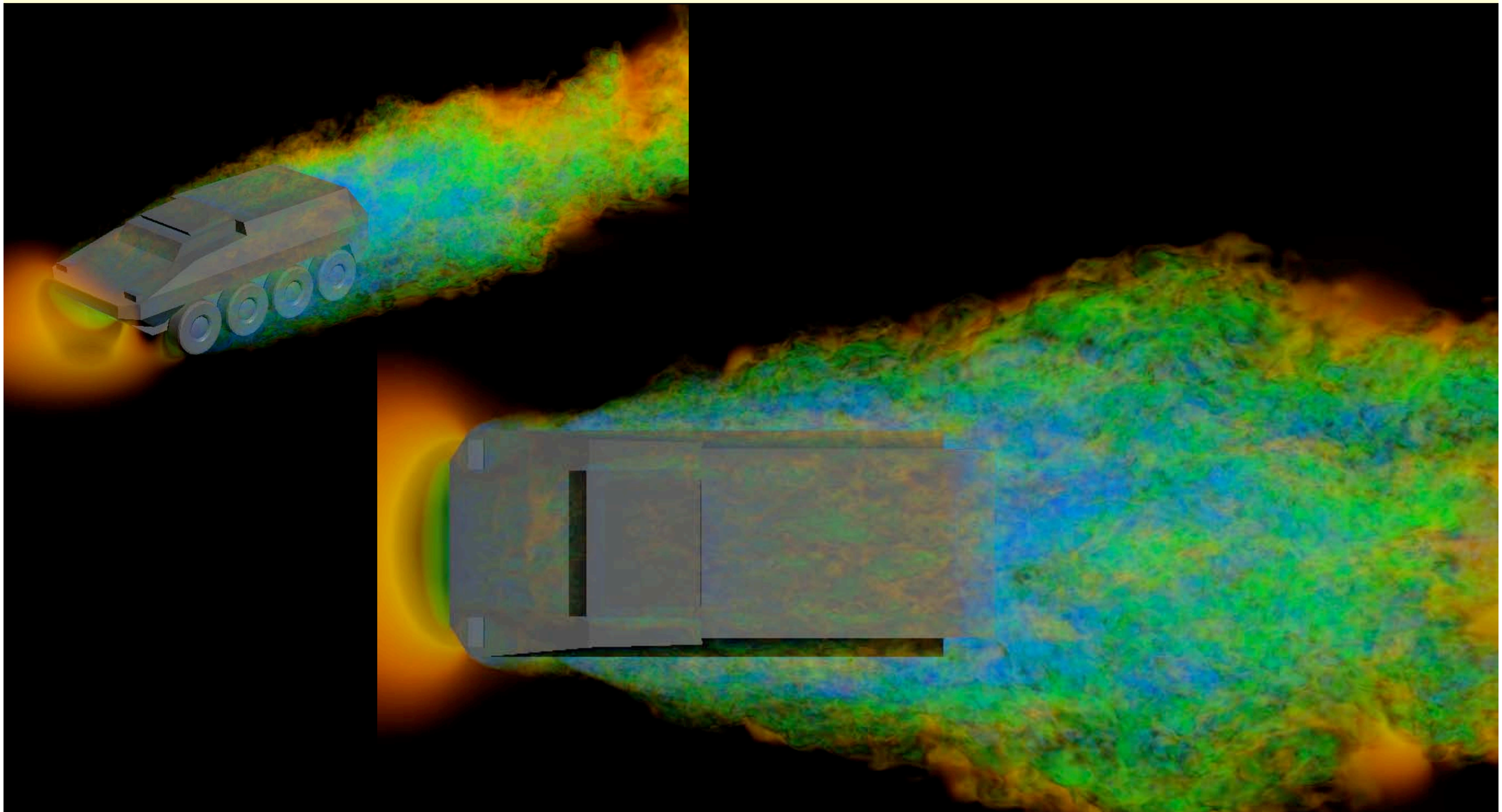
Large CFD Calculations (continued)



Volume rendering of airflow past a tactical unmanned aerial vehicle (TUAV). (450 Million tetrahedral elements)

AHPCRC

Large CFD Calculations (continued)

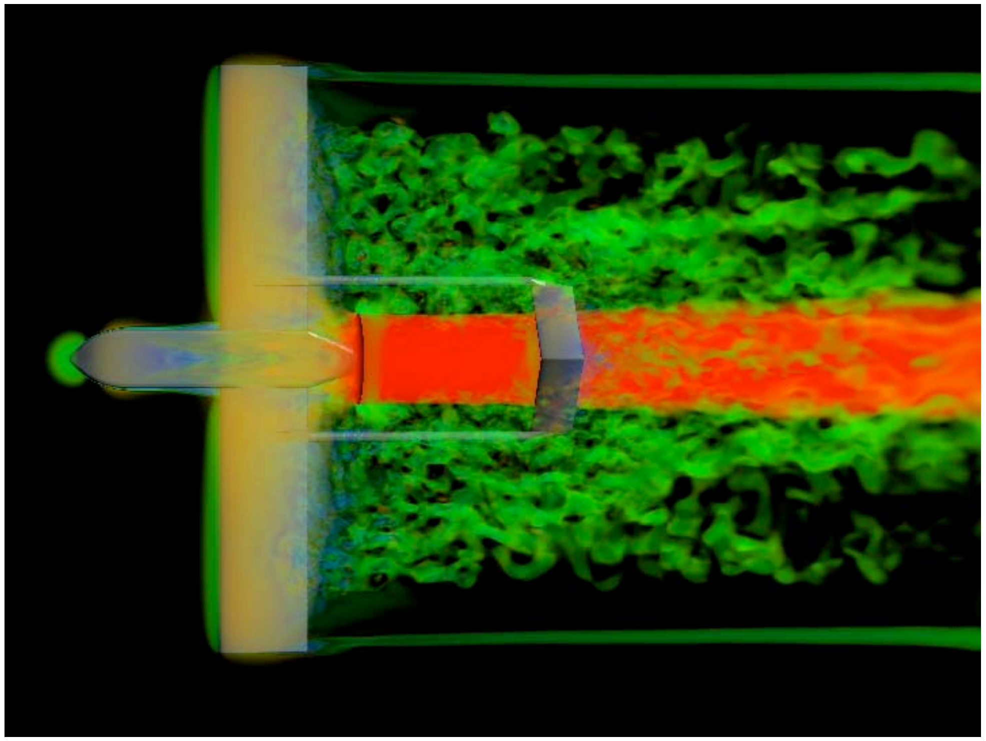


Volume rendering of airflow past a military ground vehicle.
(1.1 Billion tetrahedral elements)

Large CFD Calculations (continued)

- 1.25 Teraflops sustained overall performance
  - 252 application processors on the Cray X1E
    - Upgraded AHPCRC system as of 2 months ago
- 1.1 Billion element mesh
  - Unstructured mesh
  - Tetrahedral elements
  - GMRES Iterative equation solver
  - Matrix-Free Mode

This application, but with a larger mesh.


Tactical Unmanned Aerial Vehicle

UPC and CAF

- Extensions to the C (UPC) and FORTRAN (CAF) languages that make distributed data “visible” and accessible by all processors
  - No (very few) library functions are needed (i.e. it is part of the language)
  - In general, <u>can be mixed with other models such as MPI</u>

- Very low overhead for one-sided reads and writes
- Able to take advantage of the Cray X1’s global address space
- No subroutine calls
  - Compiler can optimize across GETs and PUTs
  - Can make coding simpler (more elegant) in many cases
  - Can implement advanced algorithms which would be difficult or impossible with message-passing models such as MPI

- Fairly new and supported by only a few vendors
- Will require changes to existing codes

CAF and UPC (Language Overview)

UPC

```
shared double d1[N][THREADS];
shared double d2[N][THREADS];
shared int nn[THREADS], ne[THREADS];
shared int norder;

double x1[N], x2[N];


if (MYTHREAD == 0){
   printf("Hello world!\n");
   for (i = 1; i < THREADS; i++){
      nn[i] = 9381;
      ne[i] = 3813;
   }
}
upc_barrier;

for (i = 0; i < N; i++){
   x1[i] = d1[i][MYTHREAD - 1];
   d2[i][MYTHREAD + 1] = x2[i];
}
upc_barrier;
```

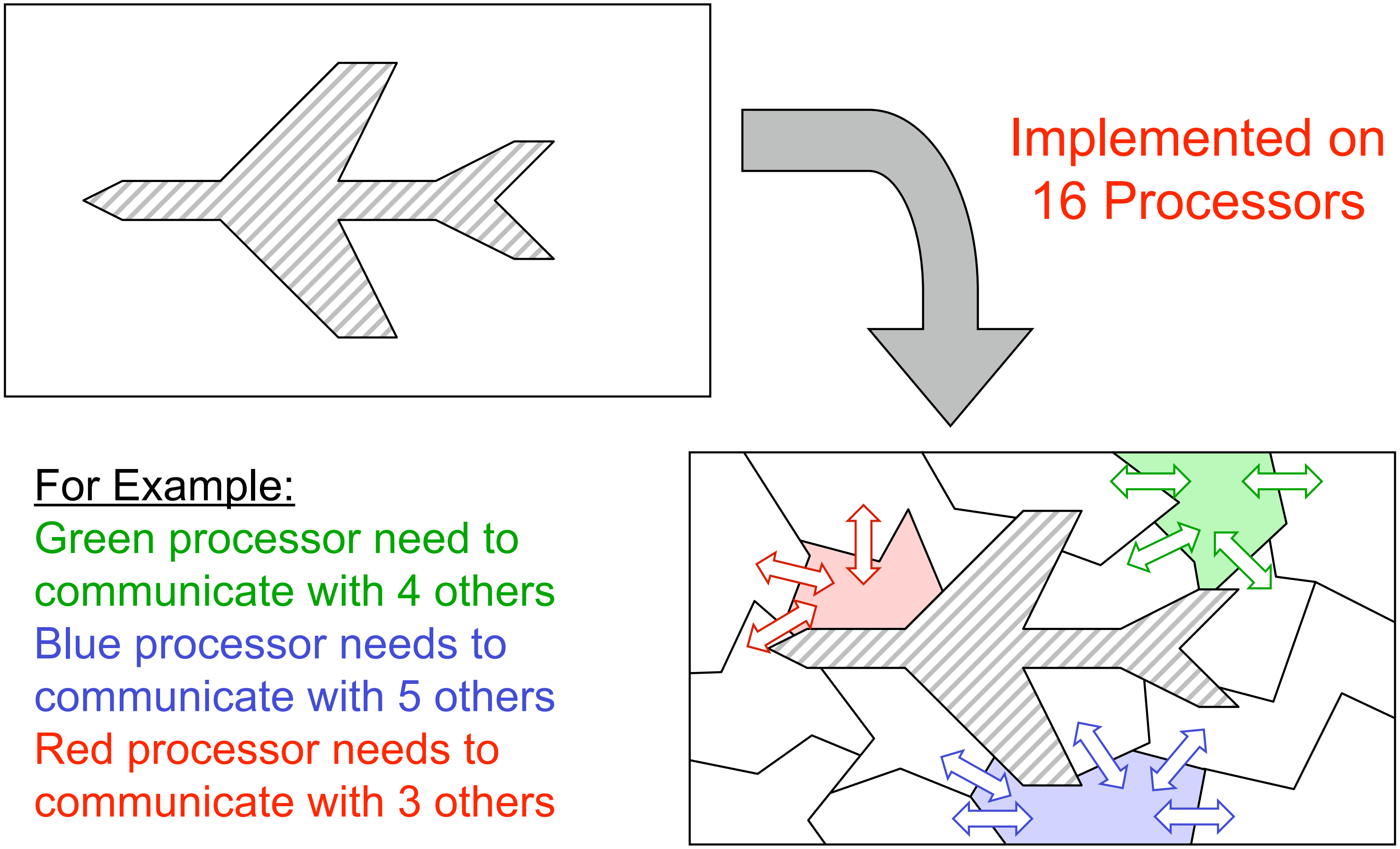
CAF

```
real*8 d1(N)[*]
real*8 d2(N)[*]
integer nn[*], ne[*]
integer npes, mypn
real*8 x1(N)[*], x2(N)[*]

mypn = this_image()
npes = num_images()
if (mypn .eq. 1) then
   print*,"Hello world!"
   do i=2,npes
      nn[I] = 9381
      ne[I] = 3813
   enddo
endif
call sync_all()

do i=1,N
   x1(i) = d1(iI)[mypn - 1]
   d2(i)[mypn + 1] = x2(i)
enddo
call sync_all()
```

Programming Style and Productivity

- It is often easier and more efficient to implement parallel communication algorithms using UPC than with MPI
  - Avoids function calls
  - Compiler can perform better optimizations
  - More readable and elegant coding
- Performance increases are also observed using UPC, at least on the Cray X1(E)
- Example case; unstructured-mesh inter processor communication for CFD (and many CSM) codes

Partitioned Mesh Communication Pattern



Implemented on
16 Processors

<u>For Example:</u>
Green processor need to communicate with 4 others
Blue processor needs to communicate with 5 others
Red processor needs to communicate with 3 others

“Scatter” Communication Function

```
void nscatter(double *bg, int len, int iflag){
   int i, j, iloc, num, nreq, i1, i2;

   for (i = nreq = 0; i < npnum; i++){
      iloc = nploc[i];
      num  = nploc[i+1] - iloc;
      MPI_Irecv(&buff[iloc*len],len*num,
                MPI_DOUBLE,np[i],MPI_ANY_TAG,
                MPI_COMM_WORLD,&ereq[nreq++]);
   }
   for (i = 0; i < epnum; i++){
      iloc = eploc[i];
      num  = eploc[i+1] - iloc;
      MPI_Isend(&bg[iloc*len],len*num,
                MPI_DOUBLE,ep[i],999,
                MPI_COMM_WORLD,&ereq[nreq++]);
   }
   if (nreq > 0) MPI_Waitall(nreq, ereq, estat);

   for (j = 0; j < npnum; j++){
      i1 = nploc[j+0];
      i2 = nploc[j+1];
#pragma concurrent
      for (i = i1; i < i2; i++){
         iloc = ibuff[i]*4;
         bg[iloc + 0] += buff[i*4 + 0];
         bg[iloc + 1] += buff[i*4 + 1];
         bg[iloc + 2] += buff[i*4 + 2];
         bg[iloc + 3] += buff[i*4 + 3];
      }
   }
}
```

```
void nscatter(double *bg, int len, int iflag){
   int i, j, iloc, iloc1, iloc2, num, ip, i1, i2;

   upc_barrier 567;

#pragma csd parallel for private(i,j,iloc1,iloc2,num,ip)
   for (i = 0; i < epnum; i++){
      iloc1 = eploc [i]*len;
      iloc2 = eploc2[i]*len;
      num = (eploc[i+1] - eploc[i+0])*len;
      ip = ep[i];

#pragma ivdep
      for (j = 0; j < num; j++)
         buffSH[ip][iloc2+j] = bg[iloc1+j];
   }

   upc_barrier 568;

   for (j = 0; j < npnum; j++){
      i1 = nploc[j+0];
      i2 = nploc[j+1];
#pragma concurrent
      for (i = i1; i < i2; i++){
         iloc = ibuff[i]*4;
         bg[iloc + 0] += buff[i*4 + 0];
         bg[iloc + 1] += buff[i*4 + 1];
         bg[iloc + 2] += buff[i*4 + 2];
         bg[iloc + 3] += buff[i*4 + 3];
      }
   }
}
```

Write data on to another processor

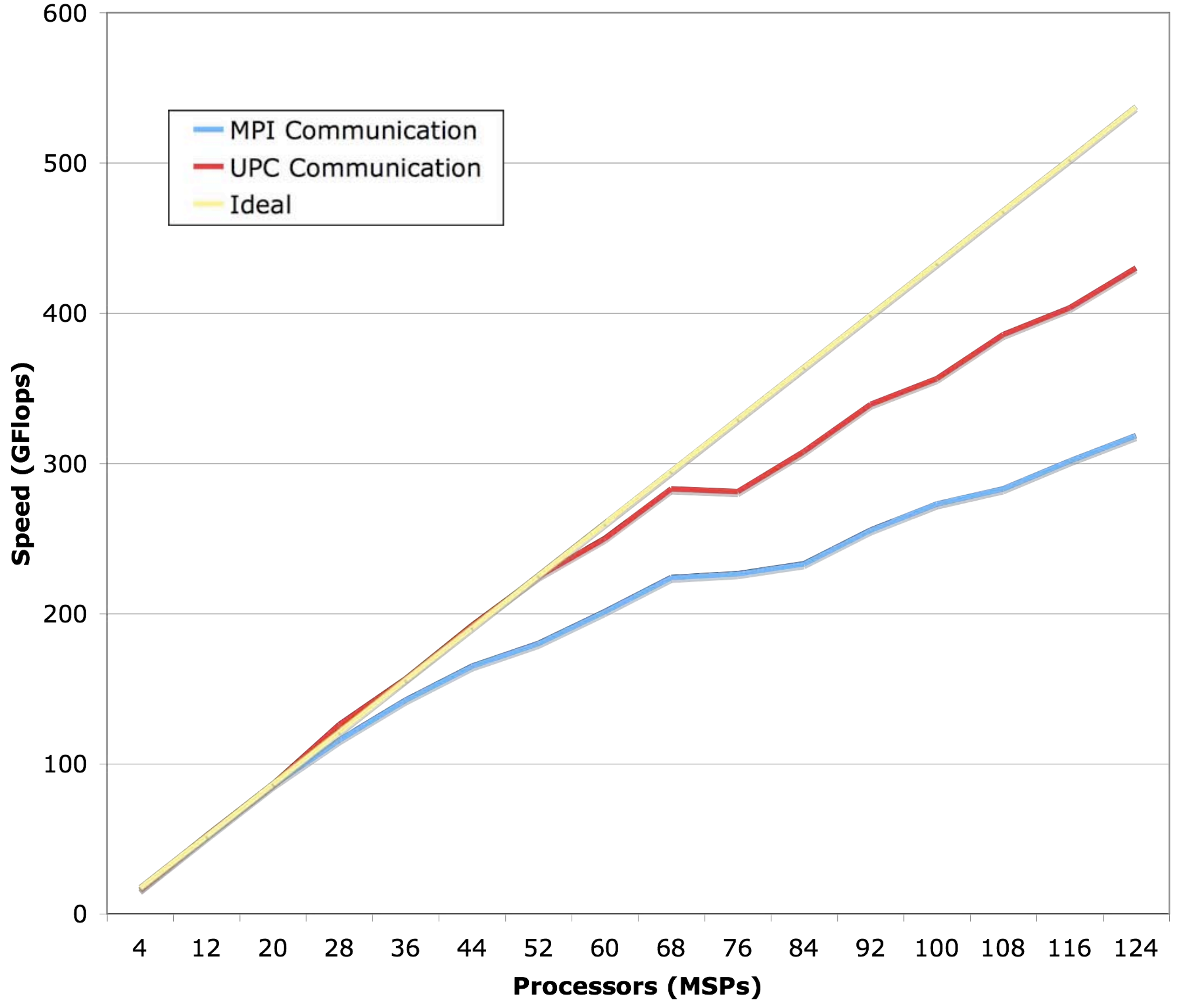
AHPCRC

Compiler Listing (simplified form)

```
633.  1------------< for (i = 0; i < epnum; i++){
634.  1                 iloc1 = eploc [i]*len;
635.  1                 iloc2 = eploc2[i]*len;
636.  1                 num = (eploc[i+1] - eploc[i+0])*len;
637.  1                 ip = ep[i];
639.  1                 #pragma concurrent
640.  1 MV---------<    for (j = 0; j < num; j++){
641.  1 MV                 buffSH[ip][iloc2+j] = bp[iloc1+j];
642.  1 MV---------     }
643.  1------------> }
644.               upc_barrier;
```

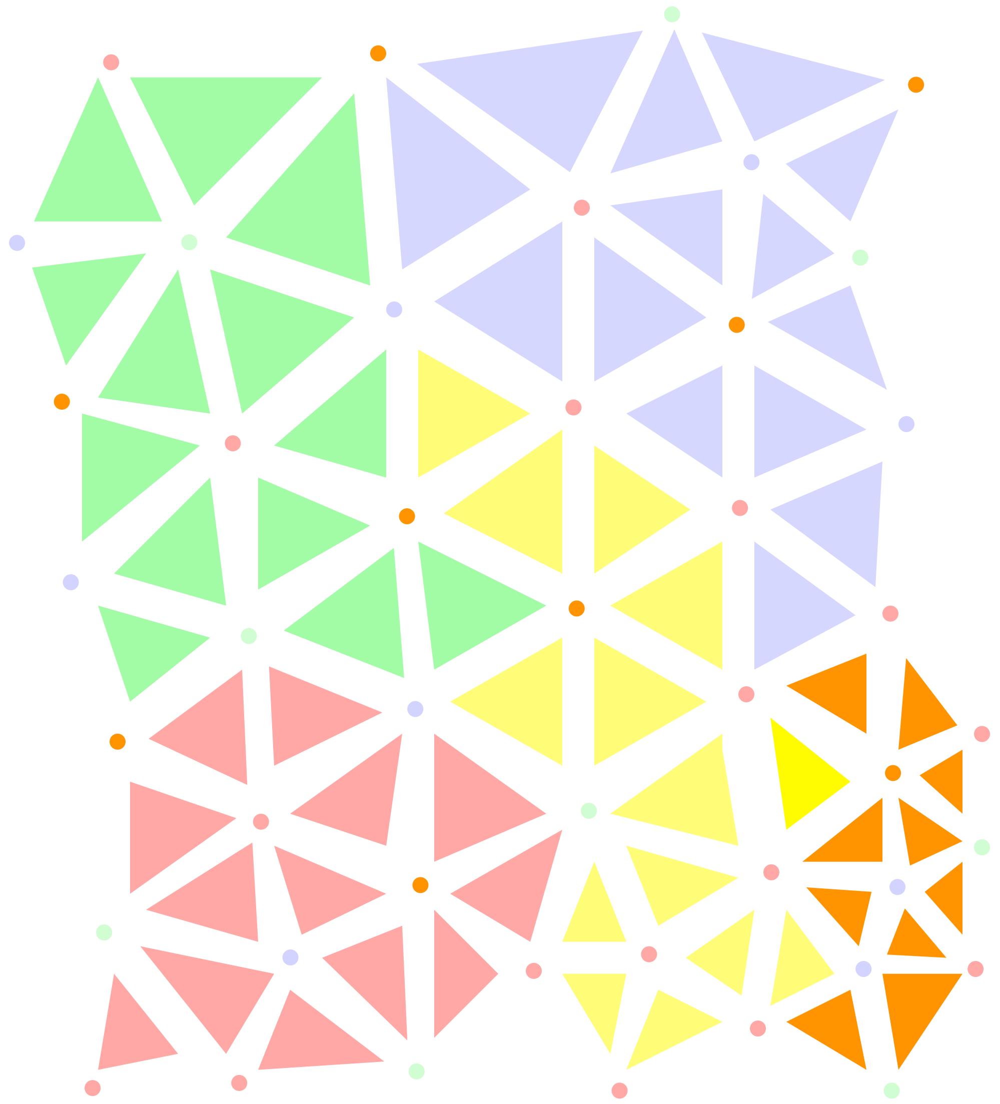
```
632.               #pragma csd parallel for private(i, j, iloc1, iloc2, num, ip)
633.  M------------< for (i = 0; i < epnum; i++){
634.  M                 iloc1 = eploc [i]*len;
635.  M                 iloc2 = eploc2[i]*len;
636.  M                 num = (eploc[i+1] - eploc[i+0])*len;
637.  M                 ip = ep[i];
639.  M                 #pragma ivdep
640.  M MV---------<    for (j = 0; j < num; j++){
641.  M MV                 buffSH[ip][iloc2+j] = bp[iloc1+j];
642.  M MV---------     }
643.  M------------> }
644.               upc_barrier;
```

Cray X1 Speed-up (2 million elements)



Implementing Parallel Algorithms

- Volume calculation
  - Each mesh element calculates its volume and then sums them all up
  - Representative of more complex calculation and numerical methods
- Implemented on a distributed mesh
  - Elements and nodes



Serial (not-parallel) version

```
double vol, volume;
double x[nn], y[nn];
int n1[ne], n2[ne], n3[n3];

volume = 0.0;
for (i = 0; i < ne; i++){
   x1 = x[ n1[i] ];  y1 = y[ n1[i] ];
   x2 = x[ n2[i] ];  y2 = y[ n2[i] ];
   x3 = x[ n3[i] ];  y3 = y[ n3[i] ];

   vol = (x1-x3)*(y2-y3) - (x2-x3)*(y1-y3);
   volume += vol;
}
printf("Volume is %lf\n",volume);
```

Parallel (MPI) Version

- Need to localize “relevant” mesh coordinates on each processor
  - Figure out which node references are off-processor
  - Gather up those nodes and sort them based on processor ownership
  - Send out requests for these node coordinates to these other processors
  - Give give out these node coordinates
  - Receive the node coordinates back
  - *Lots of memory allocations and bookkeeping involved throughout*
- Can then calculate the element volumes in a normal manner
- Call an “MPI_Allreduce” at the end to sum-up the values
  - The only part of the algorithm which is actually easier to do with MPI than with UPC (not much easier though)

Parallel (MPI) Version

```
int i, j, k, ip, nod, mypn, npes, smax, rmax, numS, numR;
int nnc, nloc, i1, i2, j1, j2, iloc, itoo, ifrom, nn;
int *ntag, *nsnd, *nrec, *ans, *irbuff, *isbuff;
double xp, yp, zp, xmax, xmin, ymax, ymin, zmax, zmin, *xtmp, *xbuff;
MPI_Status stat;

MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &mypn);

nsnd = (int *) malloc(sizeof(int) * (npes+1));
nrec = (int *) malloc(sizeof(int) * (npes+1));
for (i = 0; i <= npes; i++) nsnd[i] = nrec[i] = 0;

ntag = (int *) malloc(sizeof(int) * nn);
for (i = 0; i < nn; i++) ntag[i] = -1;
for (i = 0; i < (nen*nec); i++) if ((nod = n[i]) > -1) ntag[nod] = 0;

nloc = 0;
for (ip = 0; ip < npes; ip++){
   for (i = nnpl[ip]; i < nnpl[ip+1]; i++) if (ntag[i] == 0){
      ntag[i] = nloc;
      nsnd[ip]++;
      nloc++;
   }
}

MPI_Alltoall(nsnd, 1, MPI_INT, nrec, 1, MPI_INT, MPI_COMM_WORLD);
smax = rmax = -1;
for (i = 0; i < npes; i++) if (i != mypn){
   if (nsnd[i] > smax) smax = nsnd[i];
   if (nrec[i] > rmax) rmax = nrec[i];
}
isbuff = (int    *) malloc(sizeof(int)          * smax);
irbuff = (int    *) malloc(sizeof(int)          * rmax);
xbuff  = (double *) malloc(sizeof(double) * NSD * rmax);
xtmp   = (double *) malloc(sizeof(double) * NSD * nloc);

for (i = j1 = j2 = 0; i <= npes; i++){
   k = nsnd[i];  nsnd[i] = j1;  j1 += k;
   k = nrec[i];  nrec[i] = j2;  j2 += k;
}
```

Page 1

Smallest font supported by PowerPoint

AHPCRC

Parallel (MPI) Version

Page 2

```
for (i = nnpl[mypn]; i < nnpl[mypn+1]; i++) if ((j = ntag[i]) > -1){
   nod = i - nnpl[mypn];
   xtmp[j*NSD + X] = x[nod*NSD + X];
   xtmp[j*NSD + Y] = x[nod*NSD + Y];
   xtmp[j*NSD + Z] = x[nod*NSD + Z];
}

for (i = 1; i < npes; i++){
   MPI_Barrier(MPI_COMM_WORLD);

   itoo  = mypn + i;  if (itoo >= npes) itoo  -= npes;
   ifrom = mypn - i;  if (ifrom < 0)    ifrom += npes;

   i1 = nsnd[itoo];
   numS = 0;
   for (j = nnpl[itoo]; j < nnpl[itoo+1]; j++) if (ntag[j] > -1){
      isbuff[numS] = j;
      numS++;
   }

   numR = nrec[ifrom+1] - nrec[ifrom];
   MPI_Sendrecv(isbuff, numS, MPI_INT,  itoo, 111,
                irbuff, numR, MPI_INT, ifrom, 111, MPI_COMM_WORLD, &stat);

   for (j = 0; j < numR; j++){
      nod = irbuff[j] - nnpl[mypn];

      xbuff[j*NSD + X] = x[nod*NSD + X];
      xbuff[j*NSD + Y] = x[nod*NSD + Y];
      xbuff[j*NSD + Z] = x[nod*NSD + Z];
   }

   MPI_Sendrecv(        xbuff, NSD*numR, MPI_DOUBLE, ifrom, 222,
                &xtmp[i1*NSD], NSD*numS, MPI_DOUBLE,  itoo, 222,
                MPI_COMM_WORLD, &stat);
}
free(irbuff);  free(isbuff);  free(xbuff);
```

Parallel (MPI) Version

```
volume = 0.0
for (i = 0; i < nec; i++){
   n1 = n[I*nen + N1];  n1Loc = ntag[ n1 ];
   n2 = n[I*NEN + N2];  n2Loc = ntag[ n2 ];
   n3 = n[I*NEN + N3];  n3Loc = ntag[ n3 ];
   x1 = xtmp[n1Loc*NSD + X];
   y1 = xtmp[n1Loc*NSD + Y];
   x2 = xtmp[n2Loc*NSD + X];
   y2 = xtmp[n2Loc*NSD + Y];
   x3 = xtmp[n3Loc*NSD + X];
   y3 = xtmp[n3Loc*NSD + Y];

   vol_Loc = (x1-x3)*(y2-y3) - (x2-x3)*(y1-y3);
   volume += vol_Loc
}
free( ntag );  free( xtmp );

MPI_Allreduce(&volume, &volumeTot, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
if (mypn == 0){
   printf("Volume is %lf\n",volumeTot);
}
MPI_Barrier(MPI_COMM_WORLD);
```

Page 3

Parallel (UPC) Version

```
double vol, volume;
shared double volumeSH[THREADS];
shared double x[nnLocal][THREADS], y[nnLocal][THREADS];
int n1[neLocal], n2[neLocal], n3[n3Local];

upc_barrier;

volume = 0.0;
for (i = 0; i < neLocal; i++){
   p1 = n1[i] / nnLocal;  n1Loc = n1[i] % nnLocal;
   p2 = n2[i] / nnLocal;  n2Loc = n2[i] % nnLocal;
   p3 = n3[i] / nnLocal;  n3Loc = n3[i] % nnLocal;
   x1 = x[n1Loc][p1];  y1 = y[n1Loc][p1];
   x2 = x[n2Loc][p2];  y2 = y[n2Loc][p2];
   x3 = x[n3Loc][p3];  y3 = y[n3Loc][p3];

   vol = (x1-x3)*(y2-y3) - (x2-x3)*(y1-y3);
   volume += vol;
}
volumeSH[MYTHREAD] = volume;
upc_barrier;
If (MYTHREAD == 0){
   for (i = 1; i < THREADS; i++) volume += volumeSH[i];
   printf("Volume is %lf\n",volume);
}
```

Performance Of UPC (Cray X1)

- Special-purpose benchmark code I wrote
- Every processor sends data (of varying size) to every other processor at the same time
- Compare MPI times with UPC

```
1--< for (i = 0; i < (npes-1); i++){
1        MPI_Sendrecv(d1, num, MPI_DOUBLE,  itoo[i], 111,
1                     d2, num, MPI_DOUBLE, ifrom[i], 111,
1                     MPI_COMM_WORLD, &stat);
1--> }

1--< for (i = 0; i < (npes-1); i++){
1        MPI_Irecv(d2, num, MPI_DOUBLE, ifrom[i], 222,
1                  MPI_COMM_WORLD, &reqL[nreq++]);
1--> }
1--< for (i = 0; i < (npes-1); i++){
1        MPI_Isend(d1, num, MPI_DOUBLE, itoo[i], 222,
1                  MPI_COMM_WORLD, &reqL[nreq++]);
1--> }
     MPI_Waitall(nreq, reqL, statL);
```

Performance Tests (UPC Version)

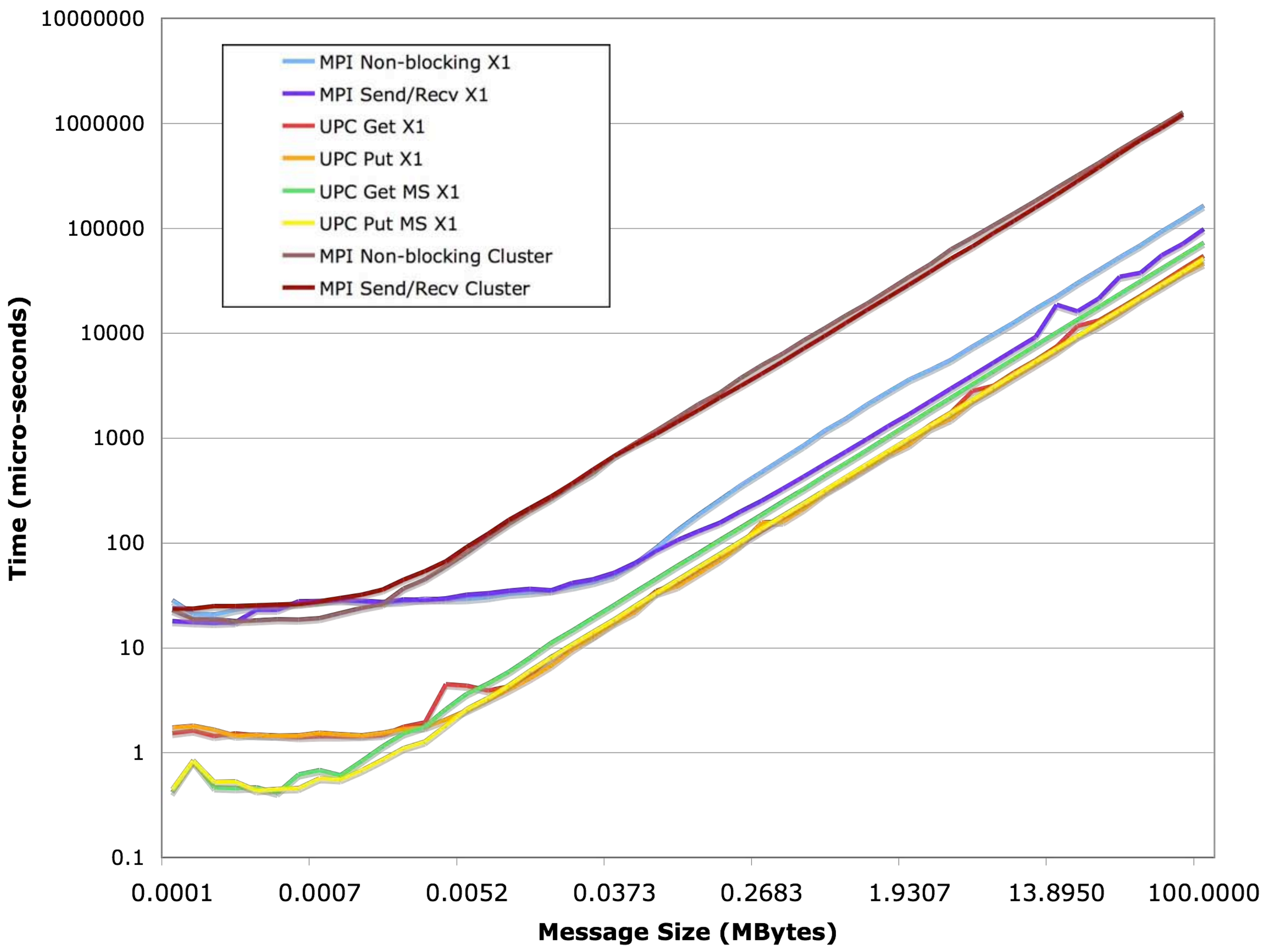
- The entire data transfer loop is vectorized and multi-streamed

```
156.   1-----<   for (i = 0; i < (npes-1); i++){
157.   1 MV--<      for (j = 0; j < num; j++){
158.   1 MV            d2[j] = d1S[ ifromLR[i] ][j];
159.   1 MV-->      }
160.   1----->   }
```

- In an alternate form, the outer “processor” loop is multi-streamed and the inner data transfer loop is vectorized
  - Cray Streaming Directives (CSD) are used to govern this behavior

```
258.             #pragma csd parallel for private(i, j)
259.   M-----<   for (i = 0; i < (npes-1); i++){
260.   M MV--<      for (j = 0; j < num; j++){
261.   M MV            d2[j] = d1S[ ifromLR[i] ][j];
262.   M MV-->      }
263.   M----->   }
```

All-to-All Data Transfer (Cray X1, 32 MSP)



AHPCRC

NETWORK COMPUTING SERVICES, INC.
© 2005

The Good and the Bad

- Distributed memory (shared nothing) - MPI
  - **Portable to most (all) systems**
  - Many codes have been ported to this model
  - Fairly simple concepts
  - Nice collective operations are available
  - Library calls may limit optimization and performance
  - Complex routines and coding required for many algorithms
- Distributed shared memory (globally addressable) - UPC/CAF
  - Simple concepts and relatively easy to program
  - Can achieve high performance
  - Can implement complex routines and algorithms much easier
  - Not fully portable yet
  - Don't having nice collective operations available (have to do it yourself)
    - Are adding some collective routines to the standard