CFD on the Cray X1E using Unified Parallel C

Andrew A. Johnson, Ph.D.
Army High Performance Computing Research Center
Network Computing Services, Inc.
Minneapolis, Minnesota
Outline

• CFD methods and code overview

• UPC Benefits
  – Performance
  – Productivity
  – Enables new algorithms and methods
CFD Overview

- Time accurate flow solvers built for unstructured meshes
  - Incompressible or compressible Navier-Stokes
  - Finite Element Method
  - Fully coupled iterative equation solvers
- Developed and used over 15 years
- Fully parallel and scalable
  - Mesh partitioning
  - Fast inter-processor communication
  - MPI-based, but UPC can also be used
- BenchC and Aeolus
  - Written in C
  - Very portable
- Highlights
  - High sustained rates on the Cray X1E
    - Vectorized code
    - 5+ GFlops per MSP
    - 1.25 TFlops sustained overall rate using 252 MSPs
  - Large-scale applications
    - 5 - 50 Million tetrahedral elements are common
    - 1.1 - 2.1 Billion elements used in extreme cases
CFD Overview (large-scale applications)

Parachute aerodynamics.

Mesh partitioning and distribution.

Airflow past a cargo aircraft.

Airflow past an unmanned reconnaissance vehicle.

250 Million Elements

1.1 Billion Elements

QuickTime™ and a Sorenson Video 3 decompressor are needed to see this picture.
History (Parallel Systems/Programming)

- Vectorization/Autotasking ~ 1985
- CMF/CMSSL ~ 1991
- PVM ~ 1994
- MPI ~ 1996
- UPC/CAF/Vector ~ 2003
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
```c
void nscatter(double *bg, int len, int iflag)
{
  int i, j, iloc, num, nreq, il, 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++){
    il = nploc[j+0];
    i2 = nploc[j+1];
    #pragma concurrent
    for (i = il; 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];
    }
  }
}
```

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

  upc_barrier;

  #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;

  for (j = 0; j < npnum; j++){
    il = nploc[j+0];
    i2 = nploc[j+1];
    #pragma concurrent
    for (i = il; 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
Cray X1 Speed-up (2 million elements)
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

```
140. 1--< for (i = 0; i < (npes-1); i++){
141. 1    MPI_Sendrecv(d1, num, MPI_DOUBLE, itoo[i], 111,
142. 1        d2, num, MPI_DOUBLE, ifrom[i], 111,
143. 1        MPI_COMM_WORLD, &stat);
144. 1--> }

183. 1--< for (i = 0; i < (npes-1); i++){
184. 1    MPI_Irecv(d2, num, MPI_DOUBLE, ifrom[i], 222,
185. 1        MPI_COMM_WORLD, &reqL[nreq++]);
186. 1--> }
190. 1--< for (i = 0; i < (npes-1); i++){
191. 1    MPI_Isend(d1, num, MPI_DOUBLE, itoo[i], 222,
192. 1        MPI_COMM_WORLD, &reqL[nreq++]);
193. 1--> }
197.    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)
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 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] ];

    volume += (x1-x3)*(y2-y3) - (x2-x3)*(y1-y3);
}
printf("Volume is %1f\n",volume);
```c
int i, j, k, ip, nod, mypn, npes, smax, rmax, numS, numR;
int nnc, nloc, i1, i2, j1, j2, iloc, itoo, ifrom, nn;
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;
}
```
Parallel (MPI) Version

```c
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;
    il = nsnd[itoo];
    numS = 0;
    for (j = nnpl[itoo]; j < nnpl[itoo+1]; j++) if (ntag[j] > -1){
        isbuff[numS] = j;
        numS++;
    }
    numR = nrec[itoo+1] - nrec[itoo];
    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


def volume = 0.0
for (i = 0; i < nec; i++):
    n1 = n[i*nex + 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];
    volume += (x1-x3)*(y2-y3) - (x2-x3)*(y1-y3);

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);
double 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];

  volume += (x1-x3)*(y2-y3) - (x2-x3)*(y1-y3);
}

volumeSH[MYTHREAD] = volume;
upc_barrier;
If (MYTHREAD == 0){
  for (i = 1; i < THREADS; i++) volume += volumeSH[i];
  printf("Volume is %lf\n",volume);
}
Enabling New Algorithms (Dynamic-Mesh CFD)

Advanced Parallel CFD Methods

Automatic Mesh Generation Technology

Advanced Parallel Programming Models (UPC)

“Dynamic-Mesh CFD” with Advanced Capabilities

Moving Geometry & Changing Domain Shapes

Fully Solution-Adaptive Computations
Dynamic-Mesh CFD (details)

• Fully integrate automatic mesh generation within the parallel flow solver
  – Mesh generation never stops and runs in-conjunction with the flow solver
    • Element connectivity changes as required to maintain a “Delaunay” mesh
    • New nodes added as required to match user-specified refinement values
    • Existing nodes deleted when not needed
  – Mesh continuously changes due to changes in geometry and/or the solution
    • Mesh size can grow or shrink at each time step

• Very complicated method
  – Parallelism (UPC), vectorization, dynamic data structures, solvers (mesh moving and fluid flow), general CFD accuracy, scalability, CAD links, etc.
  – Will take time to fully evaluate
Dynamic-Mesh CFD Applications

- Example: Flapping-wing micro-UAV simulations
  - 3D simulations performed on the Cray X1E using UPC with around 16 MSPs
  - Around 5.5 million tetrahedral elements (symmetric geometry)
  - Forced (algebraic) flapping motion designed like a flexible EAP wing

QuickTime™ and a Sorenson Video 3 decompressor are needed to see this picture.
Challenges (UPC comments and critique)

• Documentation
  – Almost all UPC documentation is based on “defined” array sizes
  – Almost never the case for real-world applications

#define N 1000

shared double x[N][THREADS];
shared int b[100*THREADS];
shared [N] int loc[THREADS][N];
Challenges (UPC comments and critique)

• Dynamic Memory
  – Vague references to pointers in the documentation
  – Vague references to dynamic memory allocation routines
  – No real explanation on how to tie them both together to achieve different layouts of shared data
  – Struggled with this quite a bit in late 2002 / early 2003
    • Had to “invent” much of this on my own based on trial and error

If ‘num’ is only known at run-time...

    shared double x1[num][THREADS];
    shared [num] double x2[THREADS][num];
    (can’t do it this way, since ‘num’ is a variable)

    typedef shared [] double *ptr_shared_dbl;
    shared ptr_shared_dbl x2[THREADS];
    (shared [] double *shared x2[THREADS])

    x2[MYTHREAD] = upc_alloc(num*sizeof(double));

    x2[processor-number][local-index] = 1.2345;
Challenges (UPC comments and critique)

• Block-size limitations
  – Would be nice if block size was a run-time option
  – There would be more data layout possibilities

```c
#define N 1000
int num = 1000;
shared [N ] double x[N *THREADS];
shared [num] double x[num*THREADS];
```

• UPC compilers on clusters
  – Preliminary tests fail (Berkley UPC)
  – Couldn’t get the code to compile
    • Complains about simple things
    • Crashes based on its own internal bugs (translator)