Global Address Space Language for Distributed Memory Machines

Arvind Krishnamurthy
Fall 2004

Overview of Split-C

- Parallel systems programming language based on C
  - Also be used as a “low-level” language for “systems programming”
- Can run on most distributed memory machines
- Creating Parallelism: SPMD
- Memory Model
  - Global address space via global pointers and spread arrays
- Split phase communication
- Running example: EM3D
SPMD Control Model (as in MPI)

- **PROCS threads of control**
  - independent
  - explicit synchronization
  - "extern" variables replicated

**Synchronization**
- global barrier
- locks

```c
int x;
main() {
  x = MYPROC;
  if (x == PROCS/2)
    ...
  barrier();
}
```

**Global Pointers**

A **global** pointer may refer to an object anywhere in the machine.

- global ptr = (proc#, local ptr)

Each object (C structure) pointed to lives on one processor

Global pointers can be dereferenced, incremented, and indexed just like local pointers.

```c
int *global gp1;  /* global ptr to an int */
typedef int *global g_ptr;
gptr gp2;    /* same */
typedef double foo;
foo *global *global gp3; /* global ptr to a global ptr to a foo*/
int *global *gp4; /* local ptr to a global ptr to an int*/
```
Memory Model

Processor 0

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC000</td>
<td>int x</td>
</tr>
<tr>
<td>0xC004</td>
<td>int *g_P</td>
</tr>
</tbody>
</table>

Processor 2

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC000</td>
<td>int x</td>
</tr>
<tr>
<td>0xC004</td>
<td>int *g_P</td>
</tr>
</tbody>
</table>

int x;      // extern var
int *g_P;

on_one {
    g_P = toglobal(2, &x);
    *g_P = 6;
}

What can global pointers point to?

- Anything, but not everything is useful
- Extern variables are common use
- Malloced heap values are useful in having inter-processor data structures
  - Imagine graphs where graph links cross processor boundaries
  - Graph nodes are “malloc”ed by each processor locally
  - Edges are setup by creating global pointers to the nodes
- Data in stack
  - Dangerous: After routine returns, pointer no longer points to valid address
  - OK if program logic dictates that the same stack frame is active on all processors
- Global pointers to locally allocated arrays are fine too
  
  ```c
  int *global arr_ptr = toglobal(MYPROC, malloc(100*sizeof(int)));
  int *global arr_ptr = (int *global)malloc(100*sizeof(int));
  ```
Discussion on Global Pointers

- To implement "*local ptr1 = *global ptr"
  - Assume code running on proc 0
  - Suppose global ptr = (proc#,local ptr2) = (1,local ptr2)
  - An active message is sent from proc 0 to proc 1, containing
    - name of message handler to run on arrival at proc 1
    - local ptr 2
    - `return address` (local ptr1)
  - When proc 1 receives the active message, it calls the message handler with local ptr2 and return address as arguments
  - The message handler retrieves the data at local ptr 2 and sends it to the return address as another active message
  - Received by the generating processor and it stores returned value in local ptr1

Discussion

- Look like pointers and act like pointers
  - Useful for building graphs, trees etc (that span the machine)
  - Gives you “naming” and “asynchrony”
  - Can write code from a single processor’s point of view

- But aren’t really the shared memory pointers we are used to
  - Fetched value is not cached
  - Subsequent request also goes over the network
  - If you want to cache it, use: “x = *gp;”
  - But then consistency of “x” is not guaranteed by the system

- Intermediate design point which makes sense on message passing machines
Spread Arrays

Spread Arrays are spread over the entire machine
- spreader “::” determines which dimensions are spread
- dimensions to the right define the objects on individual processors
- dimensions to the left are linearized and spread in cyclic map

Example 1: double A[PROCS]::[10],

Example 2: double A[n][r]::[b][b]

A[i][j] is j-th word on processor i

A[i][j] is a b-by-b block living on processor i*r + j mod P

The traditional C duality between arrays and pointers is preserved through spread pointers.

Spread Pointers

° Global pointers, but with index arithmetic across processors (cyclic)

  - If Sptr = (proc, addr) then
    Sptr+1 points to (proc+1, addr)
    (or (0, addr+1) if proc=P-1)

° In contrast if Gptr = (proc, addr) then Gptr+1 points to (proc, addr+1)

No communication:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

double A[PROCS]::;
for_my_1d (i, PROCS) {
    A[i] = i*2;
}
Communication

- Can be used in assignment statements
- Assignment statements can wait to complete, or just initiate communication
  - local object = *global ptr
    - waits to complete read of remote data
  - local ptr := *global ptr
    - initiates get of remote data
    - can continue computing while communication occurs
    - can do sync later to wait for data to arrive
- global ptr := *local ptr
  - similar, but called put

Message Types in Split-C
Irregular Problem: EM3D

Maxwell's Equations on an Unstructured 3D Mesh

Basic operation is to subtract weighted sum of neighboring values

for all E nodes
for all H nodes

Irregular Bipartite Graph of varying degree (about 20) with weighted edges

Uniprocessor Version

typedef struct node_t {
    double value;
    int edge_count;
    double *coeffs;
    double *(values);
} node_t;

void all_compute_E() {
    node_t *n;
    int i;
    for (n = e_nodes; n; n = n->next) {
        for (i = 0; i < n->edge_count; i++)
            n->value = n->value - (n->values[i]) * (n->coeffs[i]);
    }
}

How would you optimize this for a uniprocessor?
– minimize cache misses by organizing list such that neighboring nodes are visited in order
EM3D: Simple Parallel Version

Each processor has list of local nodes
typedef struct node_t {
    double value;
    int edge_count;
    double *coeffs;
    double *global (*values);
    struct node_t *next;
} node_t;

void all_compute_e() {
    node_t *n;
    int i;
    for (n = e_nodes; n; n = n->next) {
        for (i = 0; i < n->edge_count; i++)
            n->value = n->value - *(n->values[i]) * (n->coeffs[i]);
    }
    barrier();
}

How do you optimize this?
- Minimize remote edges
- Balance load across processors:
  \[ C(p) = a*\text{Nodes} + b*\text{Local Edges} + c*\text{Remote Edges} \]

Eliminate Redundant Remote Access

void all_compute_e() {
    ghost_node_t *g;
    node_t *n;
    int i;
    for (g = h_ghost_nodes; g; g = g->next) {  g->value = *(g->rval);
        for (n = e_nodes; n; n = n->next) {
            for (i = 0; i < n->edge_count; i++)
                n->value = n->value - *(n->values[i]) * (n->coeffs[i]);
        }
        barrier();
    }
}
# Overlap Remote Reads

```c
void all_compute_e()
{
    ghost_node_t *g;
    node_t *n;
    int i;
    for (g = h_ghost_nodes; g; g = g->next) g->value := *(g->rval);
    sync();
    for (n = e_nodes; n; n = n->next) {
        for (i = 0; i < n->edge_count; i++)
            n->value = n->value - *(n->values[i]) * (n->coeffs[i]);
    }
    barrier();
}
```

# Signaling Stores

- Reads/writes incur two-way communication
- Use signaling stores
  - *global_ptr :- local value;
- Saves reply message
- How do you synchronize?
  - Destination processor can wait for "x" number of stores to have completed (using "store_sync")
  - "all_store_sync" waits for all the stores in the system to be complete (enhanced version of barrier)
- Em3d can be optimized further to use stores instead of "gets"
Coordinated Operations

```c
int all_bcast(int val) {
    /* broadcast val from processor 0 to all processors */
    int left = 2*MYPROC+1; /* left child in processor tree */
    int right = 2*MYPROC+2; /* right child in processor tree */
    if (MYPROC > 0) { /* wait for val from parent */
        while (spread_flag[MYPROC] == 0) {}  
        spread_flag[MYPROC] = 0;
        val = spread_buf[MYPROC];
    }
    if ( left < PROCS) { /* if I have a left child, send val */
        spread_buf[left] = val;
        spread_flag[left] = 1; /* tell child val has arrived */
    }
    if ( right < PROCS) { /* if I have a right child, send val */
        spread_buf[right] = val;
        spread_flag[right] = 1; /* tell child val has arrived */
    }
    return val;
}
```

Broadcast using Stores

```c
int all_bcast(int val) {
    /* broadcast val from processor 0 to all processors */
    int left = 2*MYPROC+1; /* left child in processor tree */
    int right = 2*MYPROC+2; /* right child in processor tree */
    if (MYPROC > 0) { /* wait for val from parent */
        store_sync(sizeof(int));
        /* wait until one int has arrived */
        val = spread_buf[MYPROC];
    }
    if ( left < PROCS) { /* if I have a left child, send val */
        spread_buf[left] = val;
    }
    if ( right < PROCS) { /* if I have a right child, send val */
        spread_buf[right] = val;
    }
    return val;
}
**Stores and Global Communication**

Transpose from A to B: A has elements interleaved at “m” elements per processor, B has elements interleaved at “1” element per processor.

**Transpose operation:** \( A[i][j] \rightarrow B[m*i+j] \)

```c
void all_transpose ( int m ,
        double B[PROCS*m][],
        double A[PROCS][m])
{
    double *a = &A[MYPROC];
    for (i = 0; i < m; i++) {
        B[m*MYPROC+i] := a[i];
    }
    all_store_sync();
}
```

---

**Sequential Blocked Matrix Multiply**

```c
void all_mat_mult_blk(int n, int r, int m, int b,
        double C[n][m][b][b],
        double A[n][r][b][b],
        double B[r][m][b][b])
{
    int i,j,k,l;
    double (*la)[b], (*lb)[b];
    for (i=0; i<n; i++)  for (j=0; j<m; j++) {
        double (*lc)[b] = &(C[i][j]);
        for (k=0;k<r;k++) {
            la = &(A[i][k]);
            lb = &(B[k][j]);
            matrix_mult(b,b,b,lc,la,lb);
        }
    }
    barrier();
}
```

- **Configuration independent use of C arrays**
- **Points to subblocks**
- **Highly optimized local routine**
- **Blocking improves performance because the number of cache misses is reduced.**
Parallel Blocked Matrix Multiply

```c
void all_mat_mult_blk(int n, int r, int m, int b,
    double C[n][m]:[b][b],
    double A[n][r]:[b][b],
    double B[r][m]:[b][b]){
    int i,j,k,l;
    double la[b][b], lb[b][b];
    for_my_2D(i,j,1,n,m) {
        double (*lc)[b] = &C[i][j];
        for (k=0;k<r;k++) {
            bulk_read (la, A[i][k], b*b*sizeof(double));
            bulk_read (lb, B[k][j], b*b*sizeof(double));
            matrix_mult(b,b,b,lc,la,lb);
        }
    }
    barrier();
}
```

Configuration independent use of spread arrays

Local copies of subblocks

Communication can be split-phase or non-blocking

Highly optimized local routine

Blocking improves performance because the number of remote accesses is reduced.

Summary

- Performance tuning capabilities of message passing
  - Nuts and bolts language
  - Shares C’s philosophy
- Support for shared data structures
- Available on many platforms
  - Follow-up language is “UPC”
- Consistent with C design
  - arrays are simply blocks of memory
  - no linguistic support for data abstraction
    - interfaces difficult for complex data structures
  - explicit memory management