NERSC/LBNL UPC Compiler Status Report

Costin Iancu
and
the UCB/LBL UPC group
Current Status:

- UPC-to-C translator implemented in open64. Compliant with rev 1.0 of the UPC spec.
- “Translates” the GWU test suite and test programs from Intrepid.
UPC Compiler – Future Work

- Integrate with GasNet and the UPC runtime
- Test runtime and translator (32/64 bit)
- Investigate interaction between translator and optimization packages (legal C code)
- UPC specific optimizations
- Open64 code generator
 UPC Optimizations - Problems

- Shared pointer - logical tuple \((\text{addr, thread, phase})\)
  \[
  \{\text{void *addr; int thread; int phase;}}\]

- Expensive pointer arithmetic and address generation
  \[
  p+i \rightarrow \quad p.\text{phase}=(p.\text{phase}+i)\%B
  \quad p.\text{thread}=(p.\text{thread}+(p.\text{phase}+i)/B)\%T
  \]

- Parallelism expressed by forall and affinity test

- Overhead of fine grained communication can become prohibitive
#include <upc.h>
shared float *a, *b;

int main() {
    int i, k;
    upc_forall(k=7; k <234; k++; &a[k]) {
        upc_forall(i = 0; i < 1000; i++; 333) {
            a[k] = b[k+1];
        }
    }
}

k = 7;
while(k <= 233)
{
    Mreturn_temp_0 = upcr_add_shared(a.u0, 4, k, 1);
    __comma1 = upcr_threadof_shared(Mreturn_temp_0);
    if(MYTHREAD == __comma1)
    {
        i = 0;
        while(i <= 999)
        {
            Mreturn_temp_2 = upcr_add_shared(a.u0, 4, k, 1);
            Mreturn_temp_1 = upcr_add_shared(b.u0, 4, k + 1, 1);
            __comma = upcr_get_nb_shared_float(Mreturn_temp_1, 0);
            __comma0 = upcr_wait_syncnb_valget_float(__comma);
            upcr_put_nb_shared_float(Mreturn_temp_2, 0, __comma0);
            _3 ::
            i = i + 1;
        }
    }
}

k = k + 1;
........
UPC Optimizations

- “Generic” scalar and loop optimizations (unrolling, pipelining…)
  - Address generation optimizations
    - Eliminate run-time tests
      - Table lookup / Basis vectors
    - Simplify pointer/address arithmetic
      - Address components reuse
      - Localization
  - Communication optimizations
    - Vectorization
    - Message combination
    - Message pipelining
    - Prefetching for irregular data accesses
Run-Time Test Elimination

- Problem – find sequence of local memory locations that processor P accesses during the computation

- Well explored in the context of HPF

- Several techniques proposed for block-cyclic distributions:
  - table lookup (Chatterjee, Kennedy)
  - basis vectors (Ramanujam, Thirumalai)

- UPC layouts: cyclic, pure block, indefinite block size - particular case of block cyclic
### Table Array Address Lookup

```
upc forall(i=1; i<u; i+=s; &a[i])
    a[i] = EXP();
```

---

#### UPC to C translation

```c
i=1;
while(i<u) {
    t_0 = upcr_add_shared(a, 4, i, 1);
    __comma1 = upcr_threadof_shared(t_0);
    if(MYTHREAD == __comma1) {
        t_2 = upcr_add_shared(a.u0, 4, i, 1);
        upcr_put_shared_float(t_2, 0, EXP());
    }
    _1:
    i+= s;
}
```

---

#### Table based lookup

(Kennedy)
Array Address Lookup

- Encouraging results – speedups between 50:200 versus run-time resolution

- Lookup – time vs space tradeoff. Kennedy introduces a demand-driven technique

- UPC arrays – simpler than HPF arrays

- UPC language restrictions – no aliasing between pointers with different block sizes

- Existing HPF techniques also applicable to UPC pointer based programs
Address Arithmetic Simplification

- **Address Components Reuse**
  - Idea – view shared pointers as three separate components
    \( (A, T, P) : (\text{addr, thread, phase}) \)
  - Exploit the implicit reuse of the thread and phase fields

- **Pointer Localization**
  - Determine which accesses can be performed using local pointers
  - Optimize for indefinite block size

- Requires heap analysis/LQI and a similar dependency analysis to the lookup techniques
Communication Optimizations

- **Message Vectorization** – hoist and prefetch an array slice.

- **Message Combination** – combine messages with the same target processor into a larger message.

- **Communication Pipelining** – separate the initiation of a communication operation by its completion and overlap communication and computation.
Communication Optimizations

- Some optimizations are complementary

- Choi&Snyder (Paragon/T3D - PVM/shmem), Krishnamurthy (CM5), Chakrabarti (SP2/Now)

- Speedups in the range 10%-40%

- Optimizations more effective for high latency transport layers (PVM/Now) ~ 25% speedup vs 10% speedup (shmem/SP2)
Prefetching of Irregular Data Accesses

- For serial programs – hide cache latency

- “Simpler” for parallel programs – hide communication latency

- Irregular data accesses
  - Array based programs: \(a[b[i]]\)
  - Irregular data structures (pointer based)
Prefetching of Irregular Data Accesses

- Array based programs
  - Well explored topic ("inspector-executor" – Saltz)

- Irregular data structures
  - Not very well explored in the context of SPMD programs.
  - Serial techniques: jump pointers, linearization (Mowry)
  - Is there a good case for it?
Conclusions

- We start with a clean slate

- Infrastructure for pointer analysis, array dependency analysis already in open64

- Communication optimizations and address calculation optimizations share common analyses

- Address calculation optimizations are likely to offer better performance improvements at this stage
The End
Address Arithmetic Simplification

- Address Components Reuse
  - Idea – view shared pointers as three separate components (A, T, P) : (addr, thread, phase)
  - Exploit the implicit reuse of the thread and phase fields

shared [B] float a[N], b[N]
upc_forall(i=l; i<u; i+=s; &a[i])
    a[i] = b[i+k];
### Address Component Reuse

#### Table

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Diagram

- $B_i$
- $b_i$
- $e_i$
- $B - k$
- $T_a = T_b$
- $P_b = P_a + k$

#### Equations

- $a[i] = b[i+k]$;
- $a \rightarrow (A_a, T_a, P_a)$
- $b \rightarrow (A_b, T_b, P_b)$
Address Component Reuse

Ta = 0;
for (i=first_block; i<last_block; i=next_block) {
    for(j=bi,Pa=0; j < ei-k; j++,Pa++)
        put(Aa,Ta,Pa, get(Ab,Ta,Pa+k));
    ........
    for(; j<ei; j++)
        put(Aa,Ta,Pa, get(Ab,Ta+1,Pa-j));
    ........
}