An Empirical Performance Study of Chapel Programming Language

Nan Dun† and Kenjiro Taura

The University of Tokyo

†dun@logos.ic.i.u-tokyo.ac.jp
Background

- Modern parallel machines
  - Massive parallelism: 100K~ cores
  - Heterogenous architecture: CPUs + GPGPUs
- Modern parallel programming languages
  - Programmability, portability, robustness, performance
  - Chapel, X10, and Fortress, etc.
Motivation

- Programmability has been well illustrated
  - Abstract of parallelism
- Performance is yet unknown
  - Performance implications
- Performance tuning
  - Language improvements

The performance should not surprise newbies...

My First FMM Program in Chapel

Relative Elapsed Time

- Chapel
- C

Monday, May 21, 12
Agenda

- Short overview of Chapel
- Approach
- Evaluation
  - Microbenchmark results
  - Suggestions for writing efficient Chapel programs
  - N-body FMM results
- Conclusions
The Chapel Language

- Developed by Cray Inc, initiated by HPCS in 2003
- Designed to improve programmability
  - Global view model vs. fragmented model
  - Abstract of parallelism (task, data parallelism, etc.)
  - Object-oriented, generic programming
- For more details: http://chapel.cray.com
Evaluation Approach

Chapel benchmarks: data structures, language features, etc.

Intermediate C code \rightarrow \text{Comparisons} \rightarrow \text{Assembly code} \rightarrow \text{Comparisons} \rightarrow \text{Executable} \rightarrow \text{Performance Results} \rightarrow \text{Executable}
Environment

- Xeon 2.33GHz 8 core CPU, 32GB MEM
- Linux 2.6.26, GCC 4.6.2, Chapel 1.4.0
- Compile options
  - $ chpl -o prog --fast prog.chpl // Chapel
  - $ gcc -o prog -O3 -lm prog.c // C
- Use `--savec` to keep intermediate C code
- "$CHPL_COMM=none" for single locale, malloc series used
- Synthesized benchmarks from N-Body simulations
Primitive Types (1/3)

```plaintext
var res: int(32);
for i in 1..N do res = res + i;
```

```plaintext
while (...) {
    T1 = ((_real32)(i);
    T2 = (resReal32 + T1);
    resReal32 = T2;
    i = ...;
}
```
Primitive Types (2/3)

var arr: [1..N] int; // int and real
for d in arr.domain do
  res = res + arr(d); // read only

while (T80) {
  _ret42 = arrInt;
  _ret43 = (_ret42->origin);
  _ret_10 = (&(_ret42->blk));
  _ret_x110 = (*_ret_10)[0];
  T82 = (i5 * _ret_x110);
  T83 = (_ret43 + T82);
  _ret44 = (_ret42->factoredOffs);
  T84 = (T83 - _ret44);
  T85 = (_ret42->data);
  T86 = (&((T85)->_data[T84]));
  _ret45 = *(T86);
  T87 = (resInt / _ret45);
  resInt = T87;
  T88 = (i5 + 1);
  i5 = T88;
  T89 = (T88 != end5);
  T80 = T89;
}

$ gcc ... -ftree-vectorize -ftree-vectorizer-verbose=5

Relative Performance (vs. Cref)

int vs. C int  real vs. C double

colored bars for add, sub, mul, div
var arr: [1..N] int;  // int and real
for d in arr.domain do
    arr(d) = arr(d) + d; // read + write

Relative Performance (vs. Cref)

LEA instruction is executed by a separate addressing unit
Structured Types (1/3)

**Tuple**

```plaintext
var Tuple:
    (real, real, real);

var 2D_Tuple:
    (Tuple, Tuple, Tuple);
```

**Record**

```plaintext
record Record {
    var x, y, z: real
}

record 2D_Record {
    var x, y, z: Record;
}
```

**C Mapping of Tuple**

```c
double Tuple[3];

double Tuple[3][3];
```

**C Mapping of Record**

```c
struct Record {
    double x, y, z;
}

struct 2D_Record {
    struct Record x, y, z;
}
```
Structured Types (2/2)

- tuple vs. C array
- record vs. C struct
- 2D-tuple vs. C 2D-array
- 2D-record vs. C 2D-struct
- tuple+ vs. C array
- record+ vs. C struct
- 2D-tuple+ vs. C 2D-array
- 2D-record+ vs. C 2D-struct

Walk through the array and manipulate each element.

Relative Performance (vs. Cref)

- asg
- add
- sub
- mul
- div

Monday, May 21, 12
Structured Types (3/3)

- Redundant address substitution in 2D-Tuple
  - Asm: 197 vs. 33 of $C_{ref}$
  - Complex for GCC to optimize
  - Data references
  - Redundant operations
  - May be related to construction of heterogenous tuple

```c
while (...) {
    _tmp_37 = &(_ret57[0]);
    _tmp_x139 = *__tmp_37[0];
    _tmp_x239 = *__tmp_37[1];
    _tmp_x339 = *__tmp_37[2];
    ...
    chpl__tupleRestHelper(...)
    ...
    T297[0] = _tmp_x139;
    T297[1] = _tmp_x239;
    ...
}
```
Iterators for Loops (1/2)

```python
iter myIter(min: int, max: int, step: int = 1) {
    while min <= max {
        yield min;
        min += step;
    }
}
```

// Nested loops
var dom = [1..N]; // or 1..N
for i in 1..M do
    for j in [1..N] do ...; // domain
    for j in 1..N do ...; // range
    for j in dom do ...;  // pre-defined domain
    for j in myIter(1, N) do ...; // iterator
```
Iterators for Loops (2/2)

// Domain
chpl__buildDomainExpr(...);
while (loop_variable) { ... }
chpl__autoDestroy(...);

// Range
_build_range(...);
while (loop_variable) { ... }

// Pre-defined domain
_ret10 = dom;
...
_ret12 = (T45._low);
_ret13 = (T45._high);
...
while (loop_variable) { ... }

// User defined iterator
while (loop_variable) { ... }
Domain and Array

```
var rctDom3D: domain(3) = [1..N, 1..N, 1..N]; // rectangular domain
var rctArr3D: [rctDom3D] real;
var irrDom3D: domain(3*int);     // irregular domain
var irrArr3D: [irrDom3D] real;
```

---

**Domain i.e. index set**

**Array i.e. space allocation**
Domain Maps (1/2)

```plaintext
var space = [1..N, 1..N];
var blockSpace = space dmapped Block(space);
var arrBlock: [blockSpace] real;
var cyclicSpace = space dmapped Cyclic(space);
var arrCyclic: [cyclicSpace] real;
var blkCycSpace = space dmapped BlockCyclic(space);
var arrBlkCyc: [blkCycSpace] real;
var replicatedSpace = space dmapped ReplicatedDist();
var arrRep: [replicatedSpace] real;

for d in arr.domain do on Locales(here.id) do
    /* arithmetic on arr(d) */
```

Block Distribution

Cyclic Distribution
Domain Maps (2/2)

**Single Locale**

**Two Locales** *300Kbps achieved < 434Mbps measured by Iperf*
Speedup FMM Application

- Manipulate a large array of structured elements
  - Use record instead of tuple
  - Optimize small inner loop
- Auxiliary data structure
  - Use rectangular domain instead of associative domain
- Reduce locks to improve scalability
  (increase computation in some cases)
Molecular Dynamics (1/2)

- Fast Multipole Method

- Calculate the $N$-body interactions in $O(N)$ time

Relative Performance (vs. Serial Cref)

- Parallel Version
- Serial Version

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Total</th>
<th>Init</th>
<th>LeapFrog(1)</th>
<th>BuildNebrList</th>
<th>EvalForces</th>
<th>MultipoleCalc</th>
<th>WallForces</th>
<th>ApplyThermo</th>
<th>LeapFrog(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Version</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Serial Version</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Monday, May 21, 12
Molecular Dynamics (2/2)

![Graph showing speedup vs. number of threads for different system sizes (N=8^3, N=16^3, N=32^3, N=64^3). The graph demonstrates an increasing speedup with an increasing number of threads for all system sizes.](graph.png)

Monday, May 21, 12
Related Work

- Evaluations of the Chapel language
  - Programmability [Chamberlain et al. ’06,’07,’08,’11]
  - Performance potential [Barrett et al. ’08]
  - HPCC benchmark [Chamberlain et al. ’11]
  - 95% for EP STREAM & 50% for Random Access
  - Task parallel feature [Weiland et al. ’09]
  - On GPGPU [Ren et al. ’11]
Conclusions

- Chapel can achieve comparable performance to C
  - 70%~ on single locale (w/ current v1.4.0)
- User should be aware of performance implications
  - Choose proper data structure
  - Write program in proper structure
- Current performance penalties are FIXABLE
  - By improving the Chapel compiler
Questions?