Intel Concurrent Collections as a Method for Parallel Programming

UMBC REU Site: Interdisciplinary Program in High Performance Computing
Team members: Richard Adjogah, Randal Mckissack, Ekene Sibeudu
Graduate assistant: Andrew M. Raim Faculty mentor: Matthias K. Gobbert
Client: Loring Craymer, DoD Center for Exceptional Computing

Problem Statement

Computer hardware has become parallel in order to run faster and more efficiently. Intel is developing a new parallel software and translator called CnC (Concurrent Collections) to make coding in parallel easier. With CnC, the user only specifies the work to be done and CnC automatically handles parallelization. **Our research goal was to see if CnC is efficient and usable when creating parallel code and converting serial code to parallel.**

How CnC Works

CnC uses a system of collections comprised of steps, items, and tags. In CnC, computational tasks runs on threads. In order to automatically determine which code can run parallel, CnC uses a graph system. This graph system allows the user to identify dependencies of code segments in the program. The parallelization of each of the segments (called tags) is handled automatically by CnC at runtime. In order to translate this abstract graph into concrete code, the graph is written in Intel's proprietary textual notation. A CnC translator called `cnc` compiles this notation into a C++ header file.

Parameter Study

a	iter	1 Thread Time	8 Threads Time
135.44	594	1.72	2.97
274.75	477	1.38	2.65
486.90	369	1.09	2.35
561.38	367	1.08	2.26
700.98	330	0.97	2.20
840.19	300	0.89	2.02
840.19	300	0.89	2.02
916.46	287	0.86	1.95
Total		8.88	2.98

Times are in seconds. Extention of Poisson example [1] to $-\Delta u + a\,u = f$ with parameter $a > 0$. As a increases, the conjugate gradient method requires fewer iterations and runtimes decrease.

Example CnC Code

```
// The tag collection
<double avalue>;

// Collections of output data
[double errors <double>];
[int iterations <double>];

// Step prescription
<avalue> ::  (compute);

// Step execution
(compute) -> [errors];
(compute) -> [iterations];

// Input from the environment
env -> <avalue>;

// Output to the environment
[errors] -> env;
[iterations] -> env;
```

Example Main Code

```
#include 'main.h'

int compute::execute(const double &
    t, parallel_context & c) const{
  double error, a = t;
  int iter;

  //calls Poisson example function
  poisson(512, 1.0e-06, 99999,
          &error, &iter, a);
  return CnC::CNC_Success;
}

int main(int argc, char *argv[]){
  int j, M = atoi(argv[1]);
  double a;

  //creates a CnC context
  parallel_context c;

  //insert random a values into tags
  //run compute step on each tag
  for (j = 0; j < M; j ++){
    srand(j);
    a = 1000.0 * ((double)rand() /
        (double)RAND_MAX);
    c.avalue.put(a);
  }
  c.wait();
  return 0;
}
```

Results

M	Worst Case	Best Case	Actual Time
8	2.97	2.303	2.98
64	31.75	20.333	21.19
256	137.82	86.835	88.44
1024	556.18	351.908	352.63

M is the number of Poisson runs distributed across 8 threads. Worst case is M times the *maximum* runtime. Best case is M times the *mean* runtime.

Conclusions

- **CnC is a useful method for parallel programming**
- Parallelism constraints are explicit. Parallelization is done automatically at runtime.
- Excels at parameter studies that vary in memory and runtime
- CnC's graph concept is unique and easy to follow

References

Intel Concurrent Collection: User's Guide, Tutorial, and Textual Notation documents `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc`

Full technical report and code: HPCF–2011–14 `www.umbc.edu/hpcf` > Publications.

[1] Raim and Gobbert, Tech. Rep. HPCF–2010–2.

Acknowledgments

- Team members supported by a grant from the National Security Agency (NSA)
- REU Site `www.umbc.edu/hpcreu`
- UMBC High Performance Computing Facility `www.umbc.edu/hpcf`