Cluster Computing using Intel Concurrent Collections

Randal Mckissack
Senior Thesis Spring 2012
Department of Mathematics and Statistics
University of Maryland, Baltimore County

Abstract

The Intel Corporation is developing a new parallel software and compiler called Concurrent Collections (CnC) to make programming in parallel easier for the user. CnC provides a system of collections comprised of steps, items, and tags. A CnC user specifies their algorithm in a graph representation using these constructs. Using this graph of dependencies, CnC automatically identifies parallelizable code segments and executes code in parallel.

The present work focuses on the distributed version of CnC, where parallel code is run across multiple compute nodes. Specific accomplishments included getting distributed CnC working on the cluster tara in the UMBC High Performance Computing Facility, running timing tests, analyzing the data, and creating a generalized portable version of the distributed CnC code. This work allows a user in the distributed mode to have independent control over the number of threads, cores, and nodes to be used by a program. Several performance studies were ran in order to analyze the efficiency of the parallelism. Results for a parameter study show that Distributed CnC achieves a near-ideal speed-up for an increasing number of nodes.

1 Introduction

To better explain how parallel programming works and why CnC is a good software choice for it, it can be compared to the current standard for parallel coding, Message Passing Interface (MPI). MPI requires the programmer to explicitly declare what data gets sent and received by what process. MPI also provides methods to determine how many processes the code will run on and a designating number of the current process that is executing code. When coding, one must identify which parts of the code to divide up between processors (parallelize) and manually divide them up. This involves calls to very complicated C/C++/Fortran function that have multiple arguments. This adds another unwelcome layer of complexity to parallel programming, which already requires a significant amount of thought for algorithm design. More details on the MPI method of parallelization can be found here [6].

Intel's Concurrent Collections [4] is an easier way for the user to think about parallelizing programs. Instead of explicitly sending messages to processes the way MPI does, CnC uses a system of collections comprised of steps, items, and tags [4]. A user specifies the work to be done and CnC automatically sends to the work out to the processes. This in theory reduces the amount of coding the user has to do, allowing them to spend more time improving their algorithms. Early performance studies of CnC for some numerical algorithms on multi-core computers are provided in [2]. In this work, we investigate using CnC on multiple nodes on a distributed memory computing cluster. We evaluate CnC from the perspective of a novice user whose goal is to run parameter studies for a serial code in parallel. Our focus was to see if this new software is efficient, concise, and user-friendly.

In order to automatically determine which code can run parallel, CnC uses a graph system. This graph system is what the programmer uses to design their algorithms. Unlike MPI where the programmer defines what data gets passed to what process, CnC users create a list of

independent, parallelizable code segments identified by step collections. Each code segment (step) gets assigned to an item in a tag collection, which controls when and on which process the code gets executed at run-time. The final collection in the graph system is an item collection, which can hold any user defined data and gets send to and from steps. In addition to defining various collections, the user also defines the relationships between the collections in the graph. This set of collections and relationships represents the graph for the users' program.

After designing their algorithm using a graph, the graph must be translated into Intel's textual notation. A special CnC translator simply called `cnc` compiles this notation into a C++ [3, 5, 8] header file which the user includes in their code. The translator also generates a text file which contains hints for implementing the header file in code. In CnC, the place where each task runs on is called a thread as opposed to being called a process in MPI.

Through previous research [1], I found CnC to be a useful method for parallel computing using multi-threading on one multi-core/multi-processor compute node. By making the parallelization process as abstract as possible, the amount of coding a programmer has to do is reduced and task distribution can be done as effectively as possible at run-time. While the graph concept of how CnC works is a very different way of thinking than how parallelization is done in MPI, it is easy to follow once understood. The nature of CnC's parallelization makes operations that require accessing parallel elements in order counter productive and time costly. But CnC excels at parameter studies where multiple runs of a method each may vary in memory and run-time in unknown ways [1].

The remainder of this report is organized as follows: Section 2 documents how to use CnC on the cluster tara in the UMBC High Performance Computing Facility (HPCF). Section 3 describes the Poisson problem we chose to solve that shows how CnC works with parameter studies. Section 5 explains how Distributed CnC works using the same Poisson problem as the example. The report ends with the conclusions found in Section 6.

2 Using CnC on Tara at UMBC

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for high performance computing available to all researchers at UMBC. Started in 2008 by more than 20 researchers from more than ten departments and research centers from all three colleges, it is supported by faculty contributions, federal grants, and the UMBC administration. More information on HPCF is available at `www.umbc.edu/hpcf`. Installed in Fall 2009, HPCF has an 86-node distributed-memory cluster, consisting of 82 compute nodes, 2 development nodes, 1 user node, and 1 management node. Each node has two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8192 kB cache) and 24 GB of memory. All components are connected by a state-of-the-art InfiniBand (QDR) interconnect.

More information and a tutorial on how to run CnC programs on the tara cluster at UMBC can be found in the technical report [1]. This report comes from the research done at the REU Site: Interdisciplinary Program in High Performance Computing `www.umbc.edu/hpcreu` during the Summer of 2011.

3 Poisson Equation

This example provides a parameter study where the execute method in CnC requires a substantial amount of work which may vary from one task to the next. Parameter studies have variables, answers, amount of work, and run times that can vary in unknown ways. Because of this, parallelizing multiple runs of such a code can be difficult due to the fact that it is unknown how long any task will take, making it impossible to evenly divide up the work evenly before run-time. Fixing this requires using a master-slave system where the master process sends tasks to the other processes after they finish the task they are currently working on. In MPI, coding this is very involved and can lead to logical errors. In addition, the master process normally only handles the coordination of the program, so it is not used in the actual computational work. In contrast, this type of parameter study is easily done in CnC because CnC divides up the tasks among the threads by itself at run-time. CnC's advantage over MPI is that it only needs to know what code and data are independent, and then handles the distribution itself when the program is run. With MPI, the programmer would need to decide how to do this and do so beforehand.

For this parameter study, the problem solved was the partial differential equation (PDE)

$$-\Delta u(x,y) + a\,u(x,y) = f(x,y) \quad \text{for } (x,y) \in \Omega$$

for the equation $u(x,y)$ using several values of the parameter $a \geq 0$. This problem generalizes the Poisson equation $-\Delta u = f$ solved in [7]. That report discretizes the PDE by the finite difference method and uses the iterative conjugate gradient (CG) method to solve the resulting linear system. This implementation uses the same methodology, with an additional variable a. Setting $a = 0$ obtains the same results as the original Poisson equation described in [7]. As a grows larger, the system matrix becomes more diagonally dominant and the CG method will require fewer iterations to compute the results, which in turn decreases the run time.

4 CnC: Multi-Threading on One Node

This section describes the work done during Summer 2011 [1], getting CnC to parallelize multiple threads on one tara compute node. A serial version of the above Poisson function was created and a separate file was used to have CnC parallelize multiple calls to the function. In the code, inside the `compute` step there is a call to our Poisson function and the resulting iteration count and error calculation data is placed into two collections. The a value that gets passed into the Poisson function was determined using a random number generator that ranged from 0 to 1000. Timers were placed inside of the `compute` step to find the time for each `compute` step as well as a timer for the entire program. The tests aimed to see if CnC could successfully allocate different executions of the Poisson code to different processes and minimize running time. CnC is fit for this because when one process finishes its Poisson calculation, it starts the next one and does not depend on the previous one to do its work. In this way, all calculations of Poisson are parallel and CnC can optimally distribute the work.

In the CnC file, the graph algorithm is written out with the items, tags, and steps. The tag collection for this problem is the collection of all a values, and the compute step just calls the serial Poisson function from the pre-existing file where it is written. There are two output collections, one to hold the `error` and the other for the number of iterations, `iter`.

The Poisson function being written in another file is significant because it means that the CnC related files that the programmer has to create can remain completely separate from the code that solves the problem. In this parameter study example, the programmer already has working serial code that solves their problem so CnC is just a tool to run it with varying parameters as efficiently as possible in parallel. A user can approach a code, knowing only which variables need to be parallelized, and modify pre-existing CnC code to fit the given problem. This opens up the idea of parallel computing to anyone who has access to the hardware. This hardware can be a multi-core CPU or a multi-node cluster.

First shown are results from running this Poisson code with eight different a values, on an $N \times N$ mesh with $N = 512$, using a single thread. This is output captured directly from stdout after running the program:

```
  a           error       iter       time
486.90   9.794116e-07    369        1.09
135.44   2.569999e-06    594        1.72
274.75   1.489192e-06    477        1.38
916.46   5.078839e-07    287        0.86
561.38   8.260621e-07    367        1.08
700.98   5.619581e-07    330        0.97
840.19   5.165500e-07    300        0.89
840.19   5.165500e-07    300        0.89
Total time =                        8.88
```

The `a` column shows the value of the parameter a for each call to the Poisson function. `error` represents the maximum error between the true and computed solutions on the $N \times N$ mesh and `iter` represents the number of iterations required by the CG method to solve the problem for this a. The values in the column `time` in each row with an a value show the wall clock time in seconds as measured in the `compute` step, and the last row shows the total wall clock time in seconds as measured in the `main` function. It can be seen that as a increases, the number of iterations and time decrease. The errors are all small and within the tolerance given to the function, showing that the output is correct. The total time for this essentially serial run is just the addition of all of the individual Poisson calculation run times. Also, by printing directly to the screen, it is shown that the calculations are done and printed out in a random order based on a seed in the program, as seen by the a values.

Next, are the same eight a values (generated by the random number generator using the same seed) on 8 threads:

```
  a           error       iter       time
916.46   5.078839e-07    287        1.95
840.19   5.165500e-07    300        2.02
840.19   5.165500e-07    300        2.02
700.98   5.619581e-07    330        2.20
561.38   8.260621e-07    367        2.26
486.90   9.794116e-07    369        2.35
274.75   1.489192e-06    477        2.65
135.44   2.569999e-06    594        2.97
Total time =                        2.98
```

Upon inspection it can be seen that the `error` and `iter` for each a are identical to the corresponding case in the previous output. But the values of a appear ordered now; this reflects the

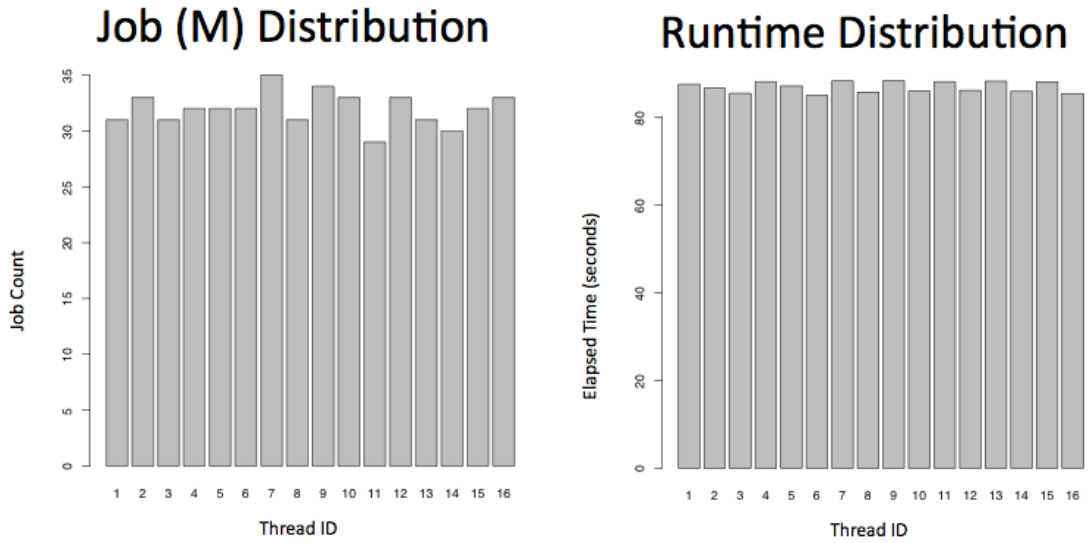
fact that stdout printed faster from those threads that completed faster, which are those for the largest a values, since then iteration count and wall clock time are lowest. It is noticeable that there is an overhead associated with using CnC on several threads, since each individual time for the Poisson function is larger than when using only one thread. But it is also apparent that the total wall clock time is only slightly longer than the time from the longest process. This shows that the parallelization was effective in decreasing the total run time to as small as possible, namely controlled by the slowest thread.

5 Distributed CnC: Multi-Threading on Several Nodes

This section describes the new work I did this semester in order to to use CnC on multiple nodes. The distributed version of CnC, called Distibuted CnC required a tuner struct to be written in the CnC header file. The tuner is what allows for individual jobs to be sent between nodes. The tuner used for the Poisson problem used a round-robin method for distributing jobs among nodes. Because the CnC header file is normally auto-generated, it is best to keep the tuner struct in a separate header file and just include it in each new CnC header file. Edits could also be made to prevent the CnC header file from being overwritten as a solve for this.

An example of the round robin method worked for the Poisson parameter study would be the run with 4096 jobs using 8 nodes. Each node could be viewed as a separate non-distributed CnC parameter study with 512 jobs. That is due to the fact that while jobs were manually assigned to each node, CnC still automatically distributed all jobs within that node to the respective cores and threads.

For the parameter study, the Poisson function was run numerous times, under the same conditions as in the previous section of the report. However, this time the focus was on how Distributed CnC performed. The variables tested were the number of nodes and the number of threads used for each run. Each run used 1, 2, 4, or 8 nodes, 1, 2, 4, or 8 threads per node, and 8, 64, 256, 512, 1024, 2048, or 4096 jobs (Poisson solves). In the following tables and graphs, M and Number of Jobs are used interchangeably to denote the number of Poisson solves. The number of threads per node used is written as Threads and the number of nodes used is written as Nodes. All times are in seconds.



(A) Job Distribution (B) Time Distribution

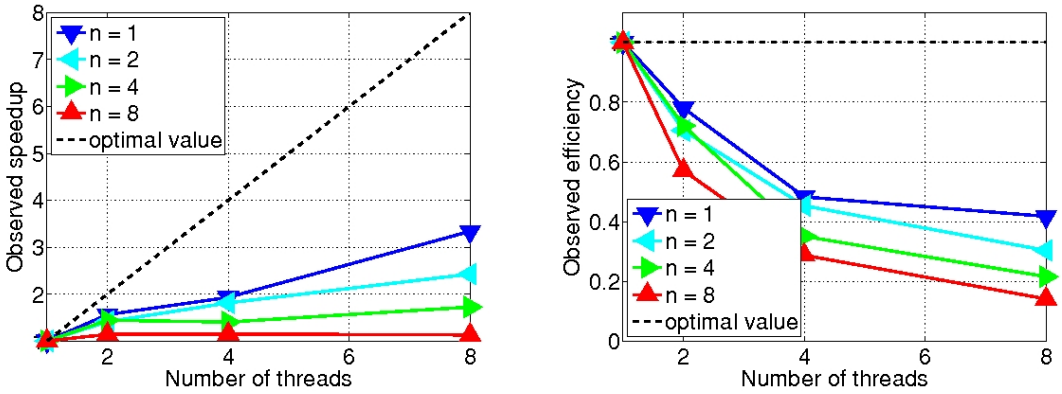
Figure 5.1: Poisson parameter study with $M = 512$, nodes = 2, and threads = 8. (A) Number of jobs per thread. (B) Elapsed time per thread.

Figure 5.1 (A) shows how effective CnC is at sending jobs to the different threads. Because 2 nodes are being used with 8 threads running on each, the horizontal axis is each of the 16 total threads. 8 of the threads are on one node and the other 8 threads are on the other node. The vertical axis is the total number of jobs each thread received out of a total of 512 jobs. The distribution was done automatically by CnC and it is clear that the number of jobs per thread is uneven. Combining this image with Figure 5.1 (B) shows why. The second graph displays the elapsed run-time of each thread. So even though thread 7 had 35 jobs and thread 8 had 30 jobs, they both took about 90 seconds to complete. CnC is able to distribute work among threads in a way that is most efficient in terms of run-time.

Table 5.1 shows the raw timing data in seconds for each M size's elapsed run-time given a certain number of threads and nodes. The table makes it clear that there are two trends; one between run-time and the number of threads, and the other between run-time and the number of nodes. In general, increasing the number of threads only brings 50% of the expected speed up. For instance, going from $M = 4096$, nodes = 1, threads = 1 to $M = 4096$, nodes = 1, threads = 2 is a change from 5499.21 seconds to 3458.57 seconds. The latter is only 1.59 times as fast as the former, even though twice as many threads were used. This trend can be seen all over the table and the other graphs. Thread efficiency is only around 50%. In contrast, increasing the number of nodes brings about almost 100% of the expected speed up. Going from $M = 4096$, nodes = 1, threads = 1 to $M = 4096$, nodes = 2, threads = 1 is a change from 5499.21 seconds to 2806.27 seconds. The latter is 1.96 times as fast as the former, where twice as many nodes were used. Analyzing the rest of the data shows that increasing the number of nodes approaches 100% efficiency. By combining both multiple nodes and multiple threads, Distributed CnC can achieve impressive speedups.

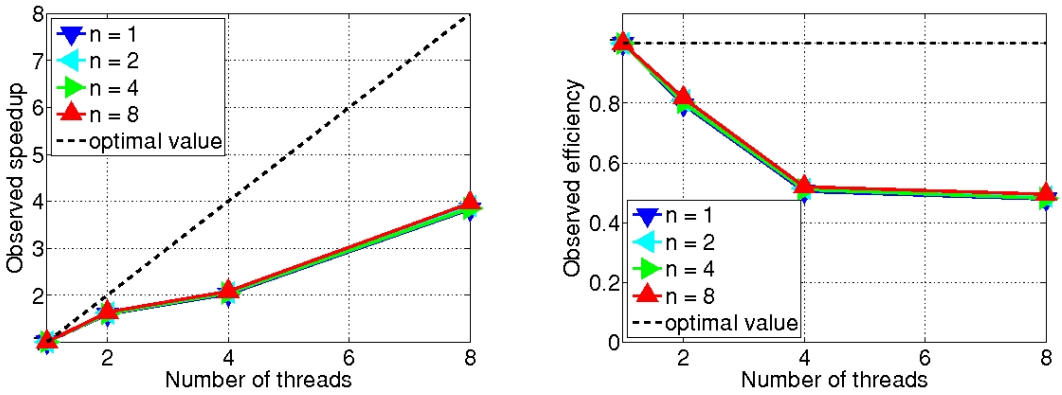
$M = 8$	threads=1	threads=2	threads=4	threads=8
nodes=1	9.38	6.01	4.86	2.81
nodes=2	5.02	3.56	2.77	2.07
nodes=4	3.10	2.15	2.21	1.80
nodes=8	1.78	1.56	1.55	1.58
$M = 64$	threads=1	threads=2	threads=4	threads=8
nodes=1	84.33	53.63	42.29	22.10
nodes=2	45.22	27.99	22.11	12.04
nodes=4	23.99	14.99	12.34	6.53
nodes=8	12.27	8.09	6.49	3.77
$M = 256$	threads=1	threads=2	threads=4	threads=8
nodes=1	337.19	211.97	166.99	86.28
nodes=2	173.91	108.77	84.36	43.66
nodes=4	90.20	55.47	43.97	22.83
nodes=8	45.88	29.32	23.06	12.65
$M = 512$	threads=1	threads=2	threads=4	threads=8
nodes=1	679.16	427.58	336.72	173.79
nodes=2	345.02	216.79	169.90	88.29
nodes=4	180.63	112.66	86.88	45.37
nodes=8	91.89	56.41	44.65	23.99
$M = 1024$	threads=1	threads=2	threads=4	threads=8
nodes=1	1360.61	870.49	677.47	349.64
nodes=2	689.81	436.70	343.46	178.20
nodes=4	348.02	223.41	172.44	89.79
nodes=8	182.76	114.48	88.45	46.30
$M = 2048$	threads=1	threads=2	threads=4	threads=8
nodes=1	2743.01	1734.68	1367.93	707.83
nodes=2	1418.20	870.98	684.77	354.02
nodes=4	708.78	445.80	345.77	183.09
nodes=8	361.05	223.83	175.77	91.32
$M = 4096$	threads=1	threads=2	threads=4	threads=8
nodes=1	5499.21	3458.57	2724.56	1436.37
nodes=2	2806.27	1735.74	1367.89	719.19
nodes=4	1407.09	878.34	688.84	365.53
nodes=8	719.21	440.08	345.94	181.72

Table 5.1: Total elapsed wall clock time in seconds for the Poisson problem parameter study.



(A) Speedup (B) Efficiency

Figure 5.2: Poisson parameter study results for threads with $M = 8$. 'n' is the number of nodes.



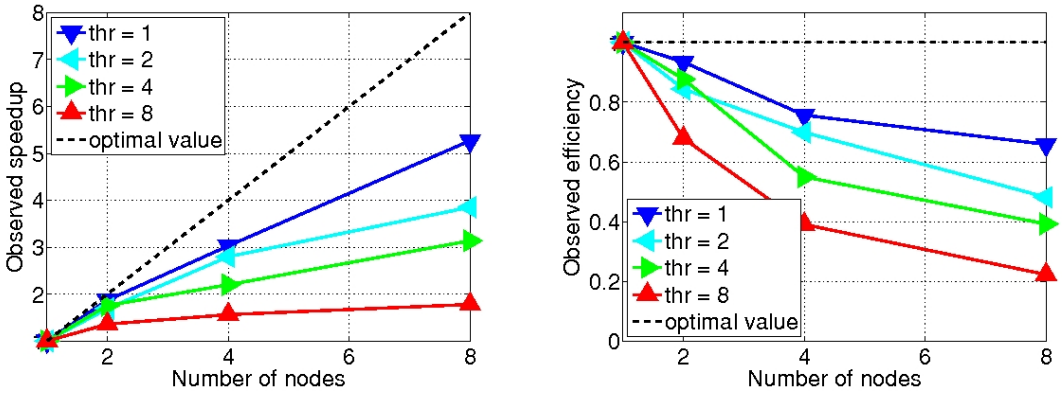
(A) Speedup (B) Efficiency

Figure 5.3: Poisson parameter study results for threads with $M = 4096$. 'n' is the number of nodes.

The speedup and efficiency plots Figures 5.2, 5.3, 5.4, and 5.5) are a visual representation of Table 5.1. For any given graph, M is held constant. For the graphs with 'Number of Nodes' as the horizontal axis, each position (1, 2, 4, and 8) is the corresponding number of nodes used at that point. Then, each line is a different number of threads. So the entire red line is has threads set to 8 and each point on the line is a different number of nodes. The graphs with 'Number of Threads' as the horizontal axis are set up in the opposite manner. For the speed up plots, the optimal value is one to one relationship between the horizontal and vertical axes, so that twice as much hardware should produce twice as much speed up, and so on. The efficiency plots show the efficiency on the vertical axis from 0 to 1 which translates to 0% to 100%.

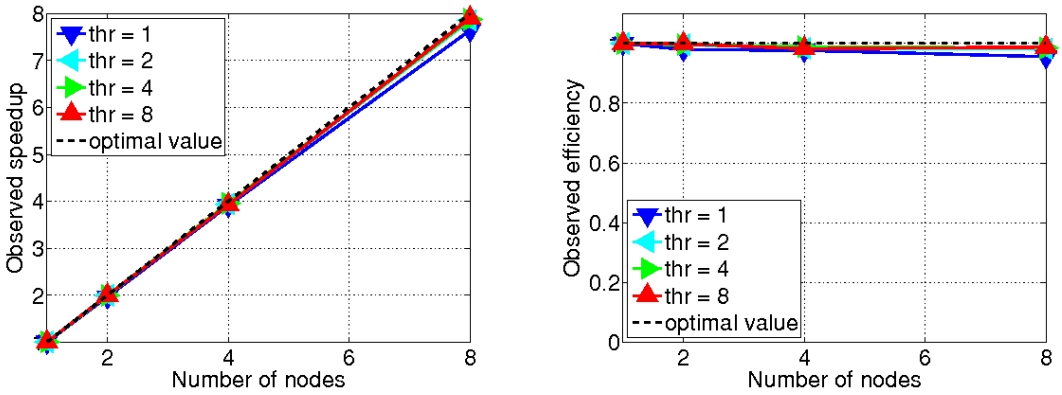
6 Conclusions

Through my research, CnC has shown itself to be a viable option for parallel programming. The graph concept is unique and easy to follow once understood. Parallelism constraints are



(A) Speedup (B) Efficiency

Figure 5.4: Poisson parameter study results for nodes with $M = 8$. 'thr' is the number of threads per node.



(A) Speedup (B) Efficiency

Figure 5.5: Poisson parameter study results for nodes with $M = 4096$. 'thr' is the number of threads per node.

explicit and do not require knowledge on how a program works, only on which variables and sections will be parallelized. The amount of parallel code needed to be written by the user is very minimal and can be reused between programs with little to no change in structure. Distributed CnC allows the user to have full control over the number of threads, cores, and nodes used in a program. Distribution of parallel jobs is done automatically at run-time for all jobs within a node. Distribution between nodes is done effectively with a manual round-robin style loop. I found it disappointing that CnC did not automatically handle the node to node distribution like it does with threads and cores within a node. My testing shows that while parallelization within a single node is only around 50% to 60% effecient, parallelization across multiple nodes is around 100% efficient. Combining the speed up produced by both nodes and threads prove that Distributed CnC is a great improvement to the existing CnC system.

Acknowledgments

The research during the 2011–2012 academic school year was conducted at UMBC through the Department of Mathematics and Statistics using the High Performance Computing Facility (HPCF) (`www.umbc.edu/hpcf`). Dr. Matthias K. Gobbert was my faculty mentor and HPCF Research Assistant Andrew Raim assisted in working on the project. Our research built upon work done during Summer 2011 REU Site: Interdisciplinary Program in High Performance Computing (`www.umbc.edu/hpcreu`) in the UMBC Department of Mathematics and Statistics with fellow undergraduates Richard Adjogah and Ekene Sibeudu. We were supported by a grant to UMBC from the National Security Agency (NSA) through the Meyerhoff Scholarship Program. UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and HPCF also supported this program. The computational hardware in HPCF is partially funded by the National Science Foundation through the MRI program (grant no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC.

References

[1] Richard Adjogah, Randal Mckissack, Ekene Sibeudu, Andrew M. Raim, Matthias K. Gobbert, and Loring Craymer. Intel Concurrent Collections as a method for parallel programming. Technical Report HPCF–2011–14, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2011.

[2] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12, Atlanta, GA, April 2010.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[4] Kathleen Knobe, Melanie Blower, Chih-Ping Chen, Leo Treggiari, Stephen Rose, and Ryan Newton. Intel Concurrent Collections for C++ 0.7 for Windows and Linux. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc, accessed February 3, 2012.

[5] Rob McGregor. *Using C++*. Que Corporation, 1999.

[6] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[7] Andrew M. Raim and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster tara. Technical Report HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[8] Walter Savitch. *Absolute C++*. Pearson Education, 2002.