Intel Concurrent Collections as a Method for Parallel Programming

Richard Adjogah*, Randal Mckissack*, Ekene Sibeudu*, Undergraduate Researchers
Andrew M. Raim**, Graduate Assistant
Matthias K. Gobbert**, Faculty Mentor
Loring Craymer***, Client

*Department of Computer Science and Electrical Engineering,
University of Maryland, Baltimore County
**Department of Mathematics and Statistics, University of Maryland, Baltimore County
***Department of Defense: Center for Exceptional Computing

Technical Report HPCF–2011–14, `www.umbc.edu/hpcf` > Publications

Abstract

Computer hardware has become parallel in order to run faster and more efficient. One of the current standard parallel coding libraries is MPI (Message Passing Interface). The Intel Corporation is developing a new parallel software and translator called CnC (Concurrent Collections) to make programming in parallel easier. When using MPI, the user has to explicitly send and receive messages to and from different processes with multiple function calls. These functions have numerous arguments that need to be passed in; this can be error-prone and a hassle. CnC uses a system of collections comprised of steps, items, and tags to create a graph representation of the algorithm that defines the parallelizable code segments and their dependencies. Instead of manually assigning work to processes like MPI, the user specifies the work to be done and CnC automatically handles parallelization. This, in theory, reduces the amount of work the programmer has to do. Our research evaluates if this new software is efficient and usable when creating parallel code and converting serial code to parallel.

To test the difference between the two methods, we used benchmark codes with both MPI and CnC and compared the results. We started with a prime number generator provided by Intel as sample code that familiarizes programmers with CnC. Then we moved on to a π approximation, for which we used a MPI sample code that uses integration to approximate π. We ran it in MPI first, then stripped it of all MPI, ported it to C++, and added our own CnC code. We then ran performance studies to compare the two. Our last two tests involved doing parameter studies on a variation of the Poisson equation using the finite difference method and a DNA entropy calculating project. We used existing serial code for the two problems and were easily able to create a couple of new files to run the studies using CnC. The studies ran multiple calls to the problem functions in parallel with varying parameters. These last two tests showcase a clear advantage CnC has over MPI in parallelization of these types of problems. Both the Poisson and the DNA problems showed how useful techniques from parallel computing and using an intuitive tool such as CnC can be for helping application researchers.

1 Introduction

The current standard for parallel coding, 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/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 [5].

Intel's Concurrent Collections[1] changes the way the user thinks 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 [3]. A user specifies the work to be done but 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. Our goal in research is to see if this new software is more efficient and concise when trying to create parallel code or when converting serial code to parallel.

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, in which the programmer defines what data gets passed to what process, in CnC a list of independent, parallelizable code segments identified by step collections is created. 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 runtime. 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. It is worth noting that unlike MPI, there is no way to find out the number of processes CnC has access to unless it is explicitly set by the user beforehand.

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++ [2, 4, 7] 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 our research, we have found CnC to be a useful method for parallel computing. 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 runtime. 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. However, CnC excels at parameter studies where multiple runs of a method each may vary in memory and run-time in unknown ways.

[1] `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`

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). The following four sections present results from studies for four sample problems: Section 3 on the `primes` program that comes with the CnC installation, Section 4 on the `pi` program distributed with most MPI installations, and Sections 5 and 6 on two parameter studies. The report ends with our conclusions in Section 7.

The required codes for our tests in Sections 3, 4, and 5 are posted along with a PDF version of this report itself at the HPCF webpage `www.umbc.edu/hpcf` under Publications. The files are bundled in the tar ball `REU2011Team4_code.tgz`, which expands into a `REU2011Team4_code` directory with sub-directories for each example.

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.

In order to use CnC on tara, several commands must first be executed.

```
[user@tara-fe1 ~]$ module load tbb
[user@tara-fe1 ~]$ module load intel-cnc
[user@tara-fe1 ~]$ source $CNC_INSTALL_DIR/bin/intel64/cncvars.sh
```

The first command loads the library `tbb` (Intel Thread Building Blocks), the second loads CnC, and the third sets the user's environment for use with CnC. These commands should be issued once in the user's SSH session in order to compile and run CnC programs. The library `tbb` is used directly, for example, when timing CnC code. Here the `tbb::tick_count` function should be used rather than through usual C++ calls.

3 Prime Number Generator

The first program we worked with was `primes`. The original version of this code is an example that comes bundled in the CnC installation. This code takes a value n and returns the number of primes between 1 and n. We analyzed the code and made slight changes to it to ensure that it is working in parallel on tara. The code for this example is provided in the files:

- `code/primes/primes.cpp`
- `code/primes/FindPrimes.cnc`
- `code/primes/Makefile`
- `code/primes/run.slurm`

In the `primes.cpp` file, the first function is the compute step. This is where CnC does the parallelization process. A number `t`, provided in the main function, and a parallel context `c`, the `FindPrimes.cnc` file, are the parameters. Inside the compute step, the number `t` is checked to see if it is prime and then placed inside of `c.primes` if it is. `c.primes` is the collection of values holding the output. At the end of the compute step, `return CnC::CNC_Success` is written to acknowledge the successful termination of the parallel computation. In the main method, the parallel context (`FindPrimes_context c`) is the first CnC piece of code that has to be written in main. The name must match the `.cnc` file. Afterwards, a for loop places all the odd numbers to be tested as primes as tags into the `c.oddNums` tag collection. As soon as tags are placed into the collection, the compute step runs, although there is no explicit call to start the parallel computing process. Rather, as soon as the tag collection begins to fill, CnC automatically runs the parallel code from the compute step on each tag, if a computational resource (thread) is available for it. The `c.wait()` command is used to synchronize the rest of the program with CnC. It ensures that all parallel processes finish before moving forward with the code in main.

Looking at the `FindPrimes.cnc` file, the graph system for the parallel parts of the code is written out. The first two items are collections that hold the odd numbers as tags and prime numbers as outputs, respectively. The next line specifies that the compute step is ran for each value in the tag collection. Following that is the step execution which shows that the compute step may return a prime number output that would belong in the primes collection. Finally, the tag collection is identified as an input coming from the main method in `primes.cpp` and the primes collection is identified as an output to the main in `primes.cpp`.

During our work on `primes`, we discovered differences between running CnC and MPI programs. When scheduling a run of CnC code on tara using a slurm file, the number of nodes does not need to be specified as the standard CnC version only runs on one node. There exists a distributed version of CnC that runs on more than one node, but we have not been able to make it function on tara. Though the number of processes does not need to be specified to the program, it can be manually set using the command `cnc::debug::set_num_threads(np)`, where `np` is the number of processes. We noticed that any given `np` above 8 defaults to 8. Unfortunately, when the `cnc::debug::set_num_threads(np)` command is not used, there is no way to control many processes CnC is using and it will default to 8 threads.

Table 3.1: Wall clock time in seconds using CnC for generating prime numbers for increasing values of n using 1, 2, 4, and 8 threads.

num threads	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	0.0208	0.8612	68.29	5824.60
2	0.0130	0.4395	34.23	2931.77
4	0.0114	0.2349	17.35	1457.80
8	0.0070	0.1427	9.01	753.92

Table 3.1 shows the results of a timing study on `primes`. The columns represent four settings of n at increasing orders of magnitude, and the rows represent 1, 2, 4, and 8 processes per node (i.e., the number of threads). The entries themselves are the elapsed wall clock times in seconds to execute the `primes` program. We can see that for $n \geq 10^5$, the run times are roughly halved when the thread count is doubled, so that CnC is effectively parallelizing the program.

4 Pi Calculation

Next we consider a `pi` program which approximates the value of the number π. The original version of this code is an example that comes bundled in most MPI installations. This code is based on the mathematical observation that $\tan(\pi/4) = 1$ and hence $\pi = 4\arctan(1)$. Using the fact that

$$\int_0^1 \frac{1}{1+x^2}\,dx = \arctan(1),$$

we have thus that π is given by an integral:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx.$$

In turn, this integral is approximated numerically using the midpoint rule, splitting the integration domain $0 \leq x \leq 1$ into n subintervals. Thus, the code returns better and better approximations for increasing values of the integer n.

We started by obtaining this well known sample code written in C, and using MPI for parallel processing. We converted this code to CnC for comparison purposes. We first removed all the MPI calls and changed the needed syntax. Then we used `tbb` to do all of the timing for the code. We tested this now serial code and it functioned correctly. Our next step was to add in the needed code to make it work in parallel using CnC. Namely, we worked on the sets, tags, and collections needed for parallelization in CnC. We used a function called `compute` to create a collection called `fvalues` that held the height of each subinterval that was calculated by the compute step. Our tags were values from 1 to n. We calculated π by summing up all the values in `fvalues` in a `for` loop. The code for the `pi` MPI program is available in the following files:

- `code/pi/mpi/cpi.c`
- `code/pi/mpi/cpi.h`
- `code/pi/mpi/cpilog.c`
- `code/pi/mpi/cpilog_pack.c`
- `code/pi/mpi/Makefile`
- `code/pi/mpi/run.slurm`

and the CnC version is available in the files:

- `code/pi/cnc/parallel2.cnc`
- `code/pi/cnc/cpi2.cpp`
- `code/pi/cnc/cpi2.h`
- `code/pi/cnc/Makefile`
- `code/pi/cnc/run.slurm`

The CnC `pi` code is structured identically to the `primes` code when looking at the parallel elements. In the `cpi2.cpp` file, the compute step takes in some value to be operated on and a parallel context. In this example, the number `t` is the x-value of the center of one of the trapezoids from the trapezoids tag collection. The integral function value $f(x)$ for that x-value is put in the `c.fvalues` collection. In the main method, the parallel process begins as soon as values are placed in the `c.trapezoids` tag collection inside of the for loop. In the `parallel2.cnc` file, the collections are created first. Then the specifications for the compute step are given. Finally, the main program sums up all $f(x)$ values and computes the integral approximation.

It can be noted that these series of steps are common to all CnC problems of this type. It is possible to vary the details of the CnC graph and have different `.cnc` and `.cpp` files, but these two benchmarks shown follow the same structure.

Tables 4.1 (a) and (b) show the results of a performance study using the MPI and CnC code, respectively. Both codes were run with 1, 2, 4, and 8 processes per node on a single node, since as mentioned earlier we are using a non-distributed version of CnC. We chose $8^6 = 262{,}144$, $8^8 = 16{,}777{,}216$, and $8^{10} = 1{,}073{,}741{,}824$ for n because the numbers are both large and divisible by 8. For the MPI code, as the number of processes per node increase the time is halved, which follows a predictable and correct pattern. For CnC, our results showed

Table 4.1: Wall clock time in seconds using the midpoint rule for increasing values of n using 1, 2, 4, and 8 threads. The dashes indicate a failed job due to lack of memory.

num threads	$n = 8^6$	$n = 8^8$	$n = 8^{10}$
1	0.0101	0.2936	18.32
2	0.0048	0.1553	9.18
4	0.0025	0.0791	4.84
8	0.0006	0.0369	2.34

(a) Using MPI

num threads	$n = 8^6$	$n = 8^8$	$n = 8^{10}$
1	.2202	16.15	—
2	.2708	19.45	—
4	.2423	17.29	—
8	.2187	27.14	—

(b) Using CnC

that the code was correctly calculating π, but that the timing was severely off. Our times were significantly longer than when using MPI. We formulated that the issues we had with our timing was caused by the `for` loop that calculated π because it was done serially. In order to check that this was in fact the issue we tested the code without the `for` loop. We found out that in fact it did increase our time, but that was not the only issue with timing. Our times for the code did not decrease as the number of processes per node increased as it did when running the MPI version. The reason for CnC taking an unreasonable amount of time for this code is the fact that it distributes all the data to threads individually. Therefore, the data is only available in isolation on each thread and no partial sums of several function values can be computed, which is what makes the MPI code effective for this problem. The partial sums also avoid the need to collect all function values in the main program on one thread. Also note that there was insufficient memory to run the $n = 8^{10}$ case in CnC. This is partially caused both by the need to collect all function values in the main program; the other contributing factor to the memory problems is the need to store all tag values simultaneously, instead of computing them on the fly.

We conclude from the experience with this code that CnC should perform well if the compute step contains a serial function that requires significant run time and whose tag is independent of any other tag.

5 Poisson Equation

This example provides a parameter study where the execute method 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, we solved 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 [6]. That report discretizes the PDE by the finite difference method and uses the iterative conjugate gradient (CG) method to solve the resulting linear system. Our implementation uses the same methodology, with an additional

variable a. Setting $a = 0$ obtains the same results as the original Poisson equation described in [6]. 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.

We created a serial version of the above Poisson function and had CnC parallelize multiple calls to the function. In the code, inside the `compute` step we have a call to our Possion 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. We put timers in the `compute` step to find the time for each `compute` step as well as a timer for the entire program. Our 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. The code for the `poisson` CnC program is available in the following files:

- `code/poisson/parallel.cnc`
- `code/poisson/main.cpp`
- `code/poisson/main.h`
- `code/poisson/p2.cpp`
- `code/poisson/p2.h`
- `code/poisson/Makefile`
- `code/poisson/run.slurm`

The fundamental structure of the `main.cpp` and `parallel.cnc` files are the same as the previous two sections with one exception. This Poisson code is more akin to what a CnC code would look like for someone who is working with an existing serial, stand-alone code. We took this serial code (`p2.cpp`) and modified it so that input parameters could be given and the outputs are returned by its main method. The `parallel.cnc` and the main method from the `main.cpp` file follow the format described in the `primes` and `pi` sections. The tag collection for this problem is the collection of all a values, and the compute step in `main.cpp` just calls the serial Poisson function from the existing `p2.cpp` file.

This is worth noting because it means that the CnC related files that the programmer creates can remain as separate as possible from the code that 'solves the problem'. In this parameter study example, the programmer already has working code that solves their problem, CnC is just a tool to run it with varying parameters as efficiently as possible in parallel. Therefore, limiting the need to know the intricities of the computational program allows for the code to be parallelized easily by people who might not know the details of how the code works (see Section 6).

First we give 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 we gave the function, showing that our 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, we ran 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 noticable 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.

The tests above used 8 values of a as a small example with clear structure to show that actual run time screen output can confirm that CnC works. The value of parallel

Table 5.1: Poisson runs for increasing values of M where M is number of runs. The mean time is the average time it took for each run. The estimated time is the expected time if perfectly parallel.

M	num threads	max time	worst case	mean time	best time	actual time
8	8	2.97	2.97	2.30	2.30	2.98
64	8	4.63	31.75	2.54	20.33	21.19
256	8	4.98	137.82	2.71	86.84	88.44
1024	8	5.03	556.18	2.75	351.91	352.63

computing lies in applying CnC to much larger numbers of a values, of course. Therefore, we ran the code with M random a values (selected uniformly between 0 and 1000) on 8 threads. Table 5.1 shows the results of such a study, using 8, 64, 256, and 1024 for levels of M. The column "max time" represents the wall time of the longest of the M cases, and "mean time" represents the average time among the M cases. The column "worst time" = $M \times$ "max time", which gives some idea of the worst case run time for the scenario. The column "best time" = $M \times$ "mean time", which gives an idea of the average case run time for the scenario. Finally, the column "actual time" gives the observed time to run the scenario.

In theory, the total run time should be the average run time of each Poisson call, multiplied by the number of Poisson calls (M), divided by the number of threads (eight). Notice for the case of $M = 1024$ that the average run time was 2.75 seconds. The perfectly parallelized run-time would be 351.91 seconds and our actual total time was 352.63 seconds. This clearly shows how readily capable CnC is at running parameter studies for heavy computation problems.

We noted in the tests with 8 values of a on 1 and 8 threads on a $N \times N$ mesh with $N = 512$ that there was a signficant overhead for each case associated with using multiple threads. To analyze this further, we repeat these studies with $N = 1024$ and $N = 2048$, which increases the wall clock time for each a value significantly. Here is the screen output using mesh size $N = 1024$ and a single thread:

```
  a          error          iter       time
486.90   3.061169e-07     735        14.74
135.44   6.426513e-07    1204        18.23
274.75   3.722823e-07     974        14.82
916.46   1.327366e-07     591         9.12
561.38   2.321447e-07     732        11.18
700.98   1.620323e-07     683        10.46
840.19   1.330215e-07     618         9.47
840.19   1.330215e-07     618         9.50
Total time =                        97.52
```

Next with eight threads:

```
916.46   1.327366e-07    591      14.46
561.38   2.321447e-07    732      17.77
840.19   1.330215e-07    618      18.15
700.98   1.620323e-07    683      19.59
486.90   3.061169e-07    735      20.67
274.75   3.722823e-07    974      24.36
840.19   1.330215e-07    618      25.34
135.44   6.426513e-07   1204      26.19
Total time =                      26.20
```

And finally, using mesh size $N = 2048$. First with one thread:

```
486.90   1.386914e-07   1464      97.27
135.44   1.599581e-07   2443     149.05
274.75   9.086264e-08   1991     121.64
916.46   7.561528e-08   1214      74.54
561.38   8.433994e-08   1461      89.63
700.98   1.296888e-07   1414      86.66
840.19   7.705277e-08   1271      77.96
840.19   7.705277e-08   1271      77.93
Total time =                     774.67
```

Next with eight threads:

```
840.19   7.705277e-08   1271     121.31
916.46   7.561528e-08   1214     179.06
840.19   7.705277e-08   1271     180.02
700.98   1.296888e-07   1414     193.27
486.90   1.386914e-07   1464     196.49
135.44   1.599581e-07   2443     214.99
561.38   8.433994e-08   1461     220.41
274.75   9.086264e-08   1991     229.87
Total time =                     229.87
```

We find that parallelizing from 1 thread to 8 creates overhead that causes the individual Poisson calculations on 8 threads to take a noticable amount of time longer than the Poisson calculations on 1 thread. This overhead is not constant and scales up with each increase in the mesh size.

6 DNA

This CnC test applies the same tools and style of code as the previous Poisson problem on a current biological problem. At the REU Site: Interdisciplinary Program in High Performance Computing (`www.umbc.edu/hpcreu`) where this research took place, another team attempted to create an algorithm to solve a DNA related problem. The full report on their problem can be found here [1]. Their research produced serial code written in C that required a parameter study with n different values. To parallelize their parameter study, we changed the file type from C (`.c`) to C++ (`.cpp`), created a CnC file for the item and tag collections, and created a main program, similar to the previous Poisson example. While the significance of this code is explained in their report in terms of this research, we emphasize here how simple it is to use CnC to parallelize pre-existing serial code for a parameter study. We were able to parallelize this DNA problem parameter study in less than 1 hour by just editing the main and `.cnc` files from our Poisson problem. The only knowledge we had about the project was the parameters and returns of their function should be set to and which parameters needed parallelization. From there, we changed our tag and item collections to reflect this and edited a few parts of the compute step. In MPI, parallelization would require more lines of code and knowledge of the given problem, making CnC a better choice for this type of task. Table 6.1 shows that CnC parallelizes the code and makes parameter studies even for the largest required value of n feasible within reasonable amount of time.

Table 6.1: Wall clock time in seconds for calculating DNA entropy for all sample sizes from 1 to n using 1, 2, 4, and 8 threads.

num threads	$n = 32$	$n = 64$	$n = 128$	$n = 256$	$n = 512$
1	0.075	0.910	13.704	219.137	3531.563
2	0.044	0.506	7.530	118.186	1847.807
4	0.029	0.292	4.201	68.140	1094.142
8	0.026	0.228	3.252	48.577	771.015

7 Conclusions

CnC is definitely a viable option for parallel computing. Constraints on what can be parallelized in the code is explicitly stated, but the actual distribution of the parallel code is done automatically at runtime. In this way, the computer can distribute parallel tasks as efficiently as possible, as demonstrated by our Poisson and DNA problems. The nature of CnC's parallelization makes operations that require accessing parallel elements in order counter productive and time costly, shown in our Pi example. Also, based on the amount of computation work a problem contains, overhead may be created that effects runtime. The type of parallel problem where we found that CnC excels at are parameter studies where multiple runs of a method each may vary in memory and run-time in unknown ways. Both

the Poisson and the DNA problems showed how useful both techniques from parallel computing, and using an intuitive tool, such as CnC, can be for helping application researchers in their research.

Acknowledgments

This research was conducted during Summer 2011 in the REU Site: Interdisciplinary Program in High Performance Computing (`www.umbc.edu/hpcreu`) in the UMBC Department of Mathematics and Statistics. This program is also supported by UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). The co-authors Adjogah, Mckissack, and Sibeudu were supported, in part, by a grant to UMBC from the National Security Agency (NSA). The computational hardware in HPCF (`www.umbc.edu/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] A. Coates, A. Ilchenko, M. K. Gobbert, N. K. Neerchal, P. O'Neill, and I. Erill, *Optimization of computations used in information theory applied to base pair analysis*, Tech. Rep. HPCF–2011–13, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2011.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, second ed., 1988.

[3] K. Knobe, M. Blower, C.-P. Chen, L. Treggiari, S. Rose, and R. Newton, *Intel Concurrent Collections for C++ 0.6 for Windows and Linux*. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc, accessed July 11, 2011.

[4] R. McGregor, *Using C++*, Que Corporation, 1999.

[5] P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.

[6] A. M. Raim and M. K. Gobbert, *Parallel performance studies for an elliptic test problem on the cluster tara*, Tech. Rep. HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[7] W. Savitch, *Absolute C++*, Pearson Education, 2002.