Parallelizing N-Queens with Intel® Parallel Composer

Sample Code Guide

Document Number: 320506-001US

Revision: 1.2

World Wide Web: http://www.intel.com/

Disclaimer and Legal Information

Contents

1 Introduction

This sample code guide will explore new and already existing parallel programming language extensions, directives, and libraries available in Intel® Parallel Composer.

The basic idea is to present the parallelism alternatives provided by the Intel® Parallel Composer and show how to apply them to a given algorithm. The key questions we are addressing are how much coding is necessary to parallelize a given serial application and which mechanisms are provided to benefit of threading with higher level constructs?

These new parallel methods we are showing in this sample are:

- OpenMP* 3.0 directives
- New parallel extensions: `__taskcomplete`, `__task`, `__finish`, `__par`
- Intel® Threading Building Blocks (Intel® TBB) along with the "lambda" functions language extension

Each of these new methods provide the C++ developer with a new range of scalability, performance, and usability for developing parallel applications not previously available in Intel® C++ Compiler products.

To illustrate the use and performance of these new features, this sample code guide demonstrates how to use each method to implement parallel solutions to the N-Queens problem, which is a well-known programming problem that has definite solutions. This guide describes using each of these parallel techniques, shows the code for each implementation, and compares and contrasts the utility and performance of each implementation.

All code blocks and examples shown in this guide are taken from the actual code samples included with the Intel® Parallel Composer.

1.1 N-Queens Samples

The N-Queens samples are a collection of Microsoft Visual Studio* project, solution, and C and C++ source files. The algorithm can be tackled by different approaches; for example, one can choose to optimize for speed or memory or both; therefore, there

are several different samples available. The samples illustrate different approaches and algorithms to solve the N-Queens problem.

If you are not familiar with the N-Queens problem, you can easily find information on the history of the problem, and other possible methods for solving it, on the worldwide web.

NOTE: The N-Queens problem was first published in 1848 in a German chess magazine by Max Bezzel. The correct answer for the problem was found in 1850 by Dr. Franz Nauck. The first solution for the general N-Queens problem was published in 1969. [1]

1.1.1 Locating the Samples

After installing Intel® Parallel Composer, the N-Queens samples are located in the `<install-directory>\Samples\en_US\C++` directory as a ZIP archive named `NQueens.zip`. Unpack this archive to a writable directory of your choice. A Microsoft Visual Studio* solution is provided containing several projects.

1.2 The Problem

The ultimate goal of the N-Queens problem is to find the total number of distinct solutions to place a given number of chess queens on a quadratic chess board with a specific number of squares on one side of a board (N) in a way that no two queens would be able to attack each other.

Two queens can attack each other when they are placed on the same horizontal row, vertical column or in one of (2*(2*n - 1)) possible diagonals.

For example, on an eight-by-eight board, there are 92 distinct solutions. The difference between a unique and distinct solution is that distinct solutions allow for symmetrical operations, like rotations and reflections of the board, to be counted as one; so an eight-by-eight board has 12 unique solutions.

Another interesting fact is that the complexity of the N-Queens problem increases exponentially. The world record at the moment is held for the solution on a 25 x 25 board. It was solved on a heterogeneous cluster around the world similar to the more popular Seti@home project and the cumulated computing time was over 50 years!

```
The number of solutions is: 2.207.893.435.808.352
Start date: October 8th 2004
End time: June 11th 2005
```

```
Computation duration time = 4444h 54m 52s 854 i.e. 185 days 4 hours 54
minutes 52 seconds 854
```

The results are from (http://proactive.inria.fr/index.php?page=nqueens25).

1.2.1 Problem Analysis

The most common approach is to use the brute-force approach, which is to simply test each position for each board size.

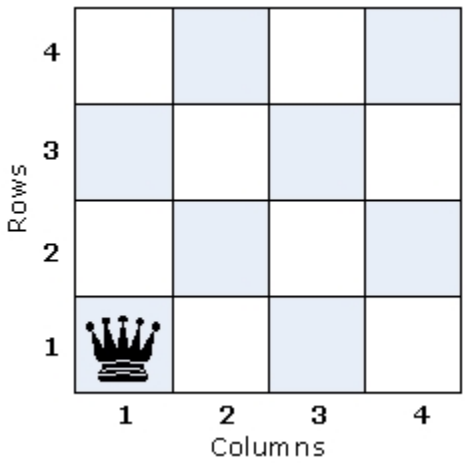
Unfortunately, trying a brute-force algorithm by testing if every position on the board is valid will inevitably lead to a huge number of positions to validate $(n^2)!/((n^2)-n)!$; even for a four-by-four board the possible number of positions is 16!/12! = 43680.

Applying a heuristic approach by only allowing one queen per row reduces the problem size to a vector of n elements, where each element holds the position in the row and the number of possible solutions reduces to nn.

This approach yields 256 positions a four-by-four board. One can reduce the number further by allowing for only one queen per column, which will result in a tuple of queens per column $(Q_1...Q_n)$ which reduces the number of solutions to n!: only 24 for our four-by-four board.

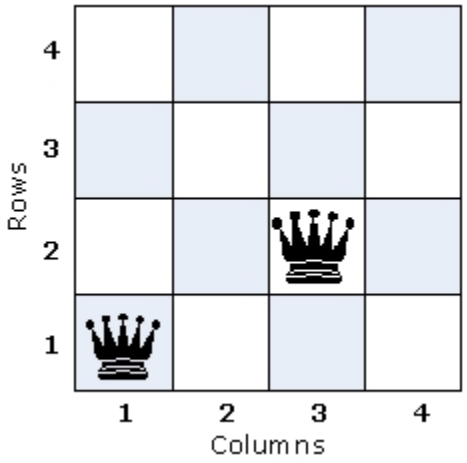
One can review the problem of the four-by-four board to illustrate one possible solution. We start by placing the first queen in row 1, column 1, as shown in Figure 1.

Figure 1



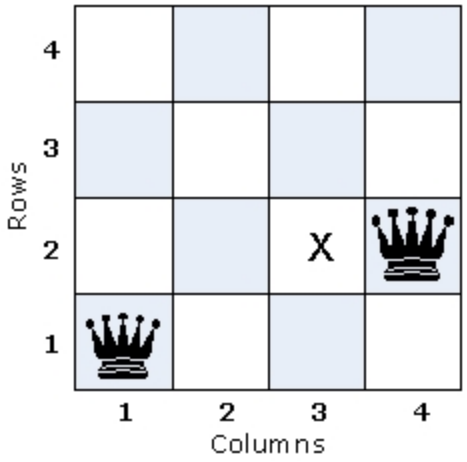
Since there is no chance to place another queen on column 1, the next queen must be placed in row 2, with the first possible placement being row 2, column 3 (Figure 2).

Figure 2



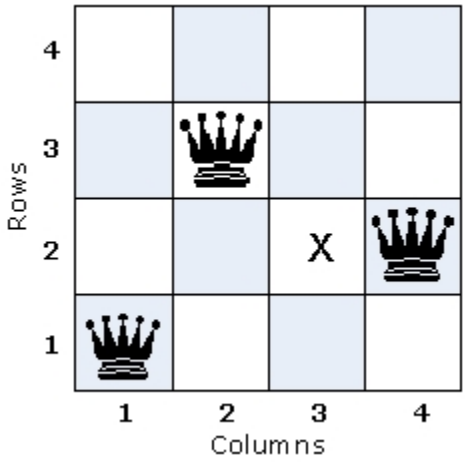
Unfortunately, there is no way to place a queen on the third row so this placement is not a viable solution; therefore, one must try another column on row 2, in this case column 4, as shown in Figure 3.

Figure 3



The new position seems valid; the next placement might be row 3, column 2 (Figure 4).

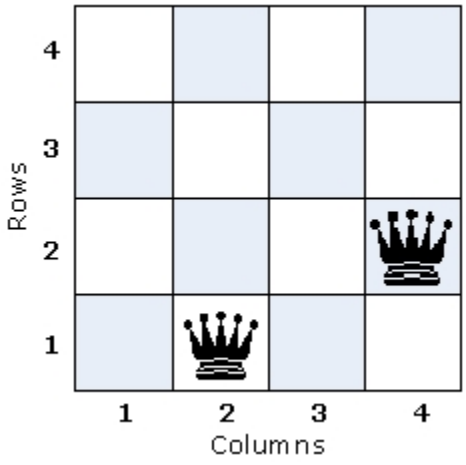
Figure 4



Unfortunately, the new placement leaves no options for row 4.

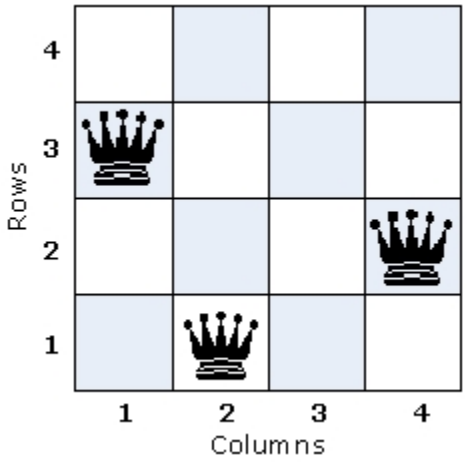
Starting over, one can place a queen at row 1, column 2 (Figure 5). Continuing the original strategy, place the next queen at row 2, column 4.

Figure 5



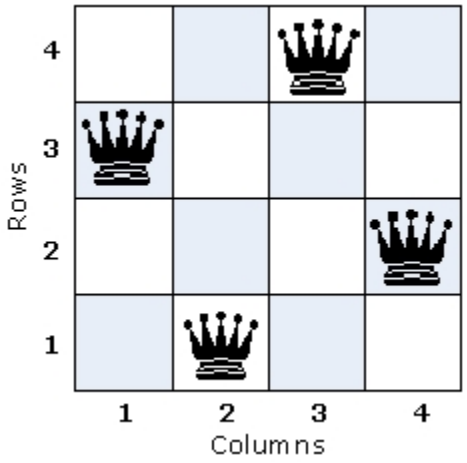
The new placement allows one to add a new queen in row 3, column 1 (Figure 6).

Figure 6



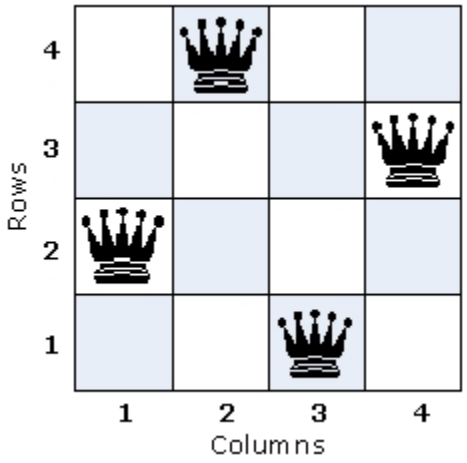
Finally, there is a possible solution because the new layout allows for another queen to be placed in row 4, column 3 (Figure 7).

Figure 7



There might be other solutions. Starting in row 1, column 3 and stepping through the placements again, one finds another solution that mirrors the first (Figure 8).

Figure 8

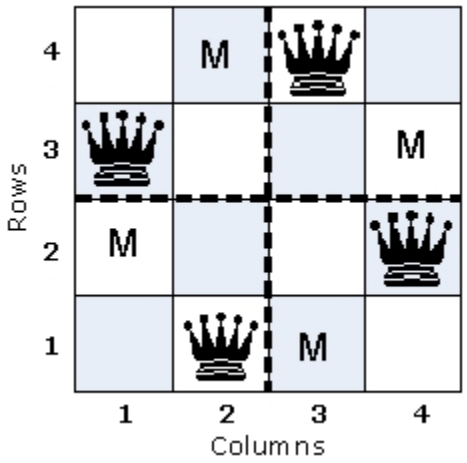


A check of other possible solutions should convince you that none exist, so there are two solutions on the four-by-four board. These are the only distinct solutions; how many unique solutions exist?

To answer that question, one must look for symmetry. If you section the board using dotted lines, and create mirror images reflected across the lines you might see another solution (Figure 9).

There is only one unique solution as we can get the other one by reflection.

Figure 9



We could present the solution as a vector (v) of (n) values (for the rows). Every value contains the number of the column where the queen can be placed.

For the original sixteen squares (four-by-four) we have found: v[4] = [2 4 1 3] and [3 1 4 2]

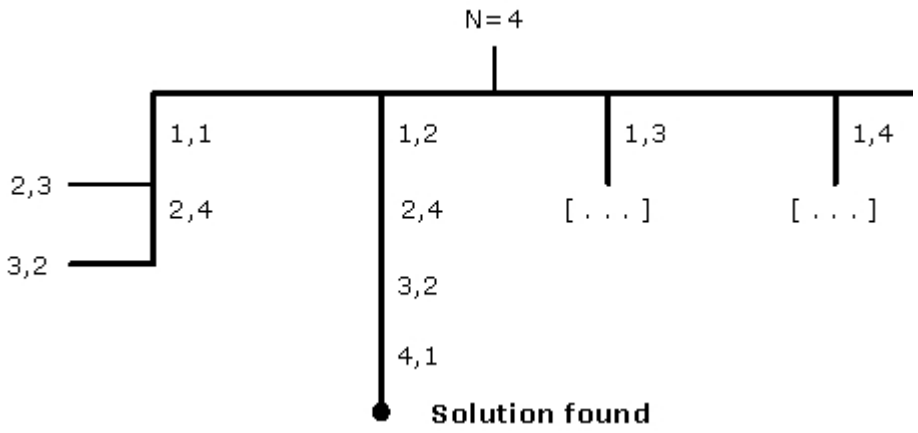
1.2.2 Algorithm

To summarize, the most obvious approach to solving the problems was to add pieces recursively to the board trying to find all possible solutions.

It is relatively easy to determine that we can add one queen in each column and look for solutions found based on the original placement of the first queen. The different solutions are independent from the solutions from the queens in the other columns. This is an example of a tree search, because the algorithm iterates down to the bottom of the search tree once a queen is placed on the board.

The search tree of the above example looks like this:

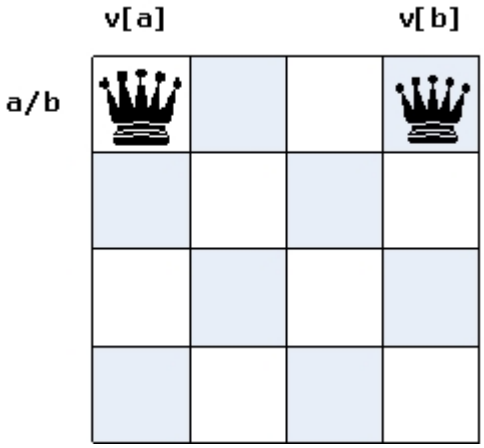
Figure 10



To code our approach, one must define conditions when queen (a) can be attacked by queen (b).

First, one must deal with the cases where queens are in the same row or column, v[a] = v[b] here [4 x x 4].

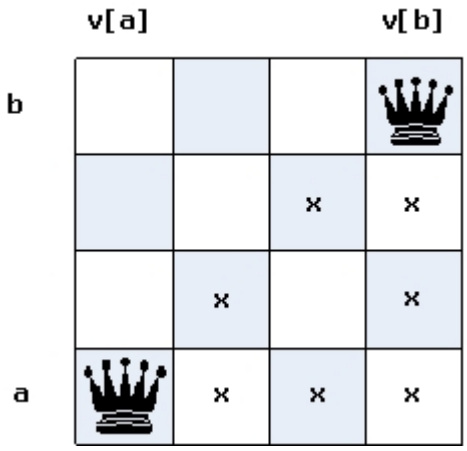
Figure 11



Second, one must define the condition where the queen can be attacked diagonally. The difference of absolute row count and column count is equal, for a < b.

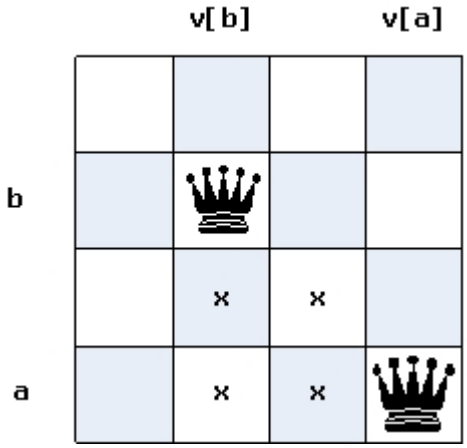
Both queens can attack each other if they stay on the same diagonal. There are two possibilities: b-a = v[b] - v[a]

Figure 12



or, b-a = v[a] - v[b]

Figure 13



We can combine both conditions to b-a = absolute value | v[a] -v[b] |. The resulting algorithm might look like the following in pseudo code:

```
Read board size: n

Set row  to 1
Set v[1] to 0
while ( column > 0) do
[
 Find out a safe place in column where to safely place a queen
 if there is a location
   put v[row] = location
   if [row = n]
        [Write solution vector v[1 ... n]]
   end
   else
   if [row < n]
     go forward one line : row = row + 1
     set v[row] to 0
   end
   else do
   one column back : column = column - 1
   end
 end
]
```

Backtracking improves upon a brute-force search by building solutions step-by-step, and being able to discard a partial solution along with all solutions that would be built from it.

This works for the eight-queen problem; for example, if one starts from an empty board and adds queens one-by-one, the moment we add a queen there is a conflict, and one can see that adding more queens will not remove the conflict; therefore, all solutions derived from that queen can be discarded without testing them.

2 Parallelizing the Solution

This section describes the main "solver" routine and describes how to go about parallelizing it. We'll look at some basic parallel analysis and decomposition of the algorithm so that we know what can be parallelized. Then, we'll describe each of the parallel solutions.

At the end of each solution, we'll describe what we think are the tradeoffs to both usability and performance we see in the methods.

First of all we need to find out where most of the time is spent in our program. This will directly lead us to the point where we have to start with the parallelization. We will use a profiler to find the hotspot in our program. In our case, this is the Intel Performance Tuning (PTU) from the Whatif.intel.com web site (http://whatif.intel.com/); however, you could some other profiler, such as the Intel® VTune™ Performance Analyzer.

2.1 The Serial "Solver" Function

We have implemented a couple of serial versions. The first version is almost an inefficient direct translation of the pseudo code to C++, where we used a Standard Template Library (STL) vector to keep the information of the solution vector. This version is kind of inefficient as it resizes the STL vector any time a backtracking solution is discarded. The next version replaces the variable vector by a fixed array to get rid of the performance penalty. Of course this could be also handled differently. The major change is that we use a more modular approach where we encapsulate the solver and the placing algorithm in separate functions.

The straightforward algorithm is of course not the fastest one. Martin Richards the father of BCPL presented in 2005 a solution for the Queens problem using a recursive bit pattern algorithm. Our C++ version is a direct translation of the presented MCPL version. Some additional notes on the different algorithms: First of all there are a lot of clever possibilities how to improve the serial code. The second implementation is the easiest one to use in a threaded playground to see how the different paradigms can be implemented, so this is the basis for the parallel algorithms. Please keep in

mind that the purpose of this sample code guide is NOT to teach the fastest possible algorithm but to show different implementations of parallel paradigms.

The third algorithm is only included to show that the use of the fastest algorithm should be done before starting with any parallelization concept. On the other hand it's much more complicated to understand.

NOTE: The back track tree version is based on the paper from Martin Richards. The original paper can be found on: http://www.cl.cam.ac.uk/~mr10/ - Martin Richards "Backtracking Algorithms in MCPL using Bit Patterns and Recursion"

2.1.1 Standard Template Library (STL) Solution

The sample code can be found in: `..\NQueens\nq-stl\`

This sample represents a simple transcription of the above algorithm to C++ using an STL vector object to represent the solution.

The vector methods pop_back/push_back expand and collapse the solution vector. Accessing the STL vector in this manner is so poor that we replaced the vector by an integer array in the next implementation.

2.1.2 Serial Solution

The sample code can be found in: `..\NQueens\nq-serial\`

This sample represents a more modularized version of the N-Queens algorithm. Everything is wrapped into the solve function, where the solution vector is replaced by a simple integer array.

Inside the solve function we call `setQueen()` which loops over the different columns in the first row. The advantage of this approach is that the different computations are independent which makes it easier to parallelize later. The core loop of the nqueens solver function is shown below.

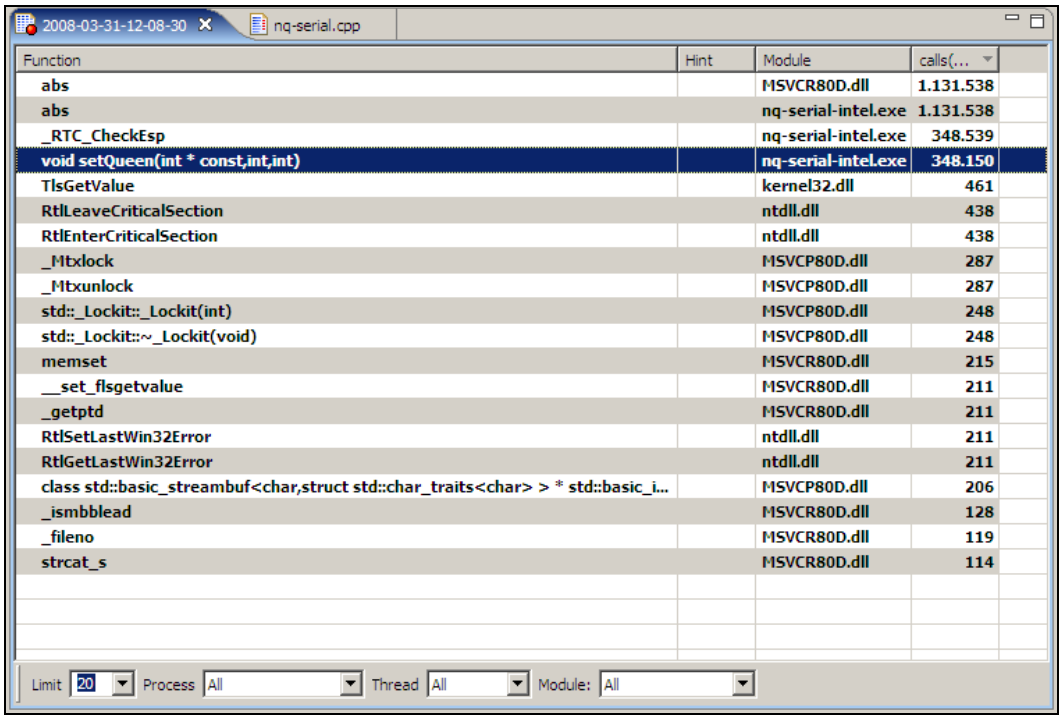
```
void setQueen(int queens[], int row, int col) {
  for(int i=0; i<row; i++) {
     if (queens[i]==col) // vertical attacks
        return;
        if (abs(queens[i]-col) == (row-i) )  // diagonal attacks
                        return;
        }
```

```
        queens[row]=col; // column is ok, set the queen
        if(row==size-1) {
                nrOfSolutions++;
        }
   else {
        for(int i=0; i<size; i++) { // try to fill next row
                setQueen(queens, row+1, i);
  } [...]
```

For each column (i) we start our recursive backtracking algorithm from our driver function **`solve()`**.

```
void solve(int queens[]) {
        for(int i=0; i<size; i++) {
        // try all positions in first row
        // create separate array for each recursion
                setQueen(queens, 0, i);
        }
}
```

Using a profiler allows you to find out where most of the time is spent in the program.



It should not be surprising that most time is spent in the **`abs()`** function, which we use to calculate if two queens are on the same diagonal (see algorithm description).

Therefore **abs()** is the hottest function in **setQueen(),** which itself is called 348150 times by our core solve function from above.

2.2 Parallelizing the "Solver"

This example cannot replace studying in depth the different parallel paradigms we offer, but it will give you enough information to get you started. If you have worked through the serial examples, you should be able to identify hotspots, which follows nicely the Pareto principle which applies to most applications that 80% of the performance is spent in 20% of the code. This consideration, in general, limits the overall parallelization efforts. Next consider that one should start with parallelization on the highest possible abstraction level to make the solution as general as possible. This method should help make the approach reusable and refactorable for later use.

We have three modules to consider: `solve()`, `setQueens()`, and `abs()`. Of course the `solve()` function is the one with the highest abstraction count and the `abs()` function with the lowest. So we use the `solve()` function. By looking at our decision tree we see that the different branches are independent from each other. This is an important and necessary condition for the parallelization strategy.

2.2.1 Windows* 32 Threading API

The sample code can be found in: `..\NQueens\nq-win32api-intel\`

A "classic" parallel implementation of the Nqueens algorithm would make use of a native threading like the Windows* 32 API. The main advantage of the API approach is that the user has more control and power over threading with C++ than with any other languages. But the amount of code it takes to implement a given solution is of course much higher as the programmer has to implement all the thread handling which in the other cases are handled by the runtime. Therefore this approach is sometimes referred to as the assembly language for threading. You are more flexible in assembly language, but you could get what you want much faster in a high level language with results that were as good or at least good enough. Also the number of cores which influences the number of created threads has to be determined. This is not so easy if you have a platform- independent solution in mind. This sample code guide is too short to teach the threading API so we provide only a very rough idea of the underlying method. First of all we have to create a thread local storage to store the thread specific information like which part of the loop to work on. This is done in **`struct _thr_params`**:

```
// Thread local data structure that will
//contain iteration range and message to print out
```

```
struct _thr_params {
        size_t start;
        size_t end;
        int *queens;
        std::string msg;
};
typedef struct _thr_params thr_params;
```

The next thing is to identify the portions of code to thread. This is of course still the module setQueens(). So we have to implement a driver function to coordinate the work of the multiple threads. This is done in run_threaded_loop().

```
void run_threaded_loop (int num_thr, size_t size, int _queens[]) {
        HANDLE* threads = new HANDLE[num_thr];
        thr_params* params = new thr_params[num_thr];

        for (int i = 0; i < num_thr; ++i) {
                // Give each thread equal number of rows
                params[i].start = i * (size/num_thr);
                params[i].end   = params[i].start + (size/num_thr);
                params[i].queens = _queens;

                // Pass argument-pointer to a different
                // memory for each thread's parameter to avoid data
races
                threads[i] = CreateThread (NULL, 0, run_solve,
                static_cast<void *> (&params[i]), 0, NULL);
        }

        // Join threads: wait until all threads are done
        WaitForMultipleObjects (num_thr, threads, true, INFINITE);

        // Free memory
        delete[] params;
        delete[] threads;
}
```

The next part is to encapsulate the code we like to run in parallel into a new function which we **`call run_solve()`**. Every instance of run_solve has its own local data for the start and end point of the loop indices.

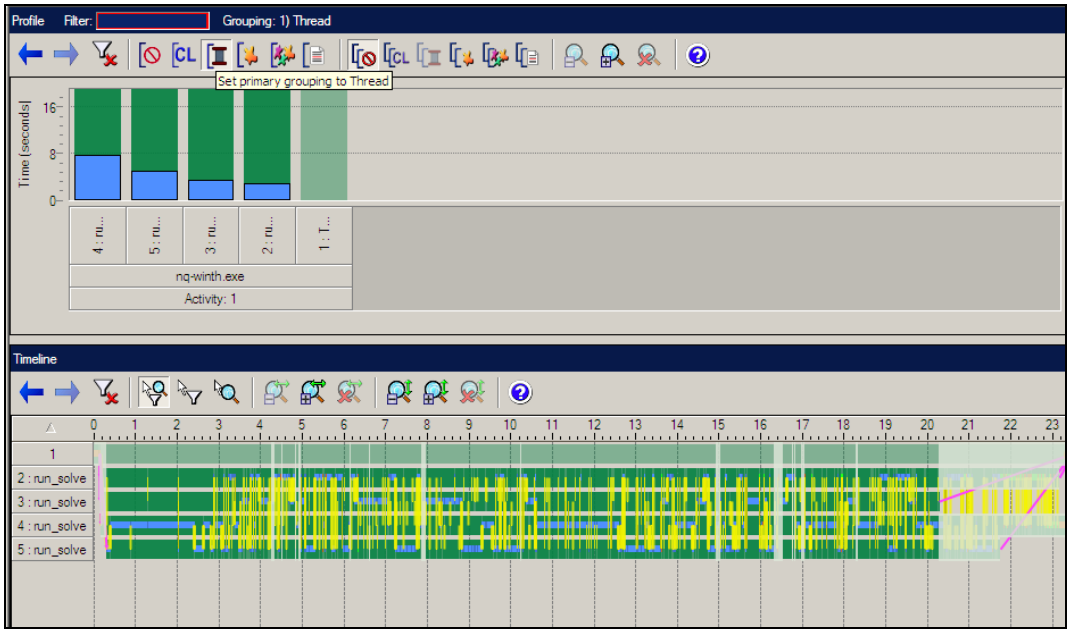
```
// Worker thread function (to be executed by each thread in parallel)
DWORD WINAPI run_solve (void* param)
{
        // Retrieve arguments passed to this thread: param is a pointer
to
        // void, static_cast converts this to a pointer to thr_params
```

```
        thr_params* params = static_cast<thr_params*> (param);

        // Print messages
        for (size_t i = params->start; i < params->end; ++i) {
                  // create separate array for each recursion

                  setQueen(new int[size], 0, i);

  }
        return 0;
}
```

Of course we also have to enable correct programming structures for avoiding race conditions which look very similar to the OpenMP* and Intel® TBB approach:

```
CRITICAL_SECTION lock;
CRITICAL_SECTION plock;
```

By comparing the different solutions it's obvious by just looking at the pure number of source lines that the threading API approach is the most complex method with more pitfalls and opportunities to make an error. But it is also of course the approach with the most freedom and control over the implementation. It would also be nice to implement a clever way to automatically determine the number of cores to choose the right number of threads. In Intel® TBB and OpenMP* this is done automatically. So the threading API approach which we use can easily lead to an over-subscription (more threads than cores) or under-subscription (less threads than cores) when moving the executable to a different machine. Using more threads than cores would look like this in the Intel® Thread Profiler:



The blue bar means oversubscription and leads directly to a performance degradation. This is also left as an exercise for the user together with the task to get rid of the yellow synchronization bars.

2.2.2 Parallelizing with __task and __taskcomplete

The sample code can be found in: `..\NQueens\nq-parexp-intel\`

The Intel® Compiler uses newly added C/C++ language extensions to make parallel programming easier. There are four keywords introduced within this version of the compiler: `__taskcomplete`, `__task`, `__par`, and `__critical`. These keywords are used as statement prefixes. In order for the application to benefit from the parallelism afforded by these keywords, you must use the /Qopenmp compiler option during compilation. The compiler will link in the appropriate runtime support libraries, and the runtime system will manage the actual degree of parallelism. The parallel extensions utilize the OpenMP* 3.0 runtime library, but abstract out the use of the OpenMP* pragmas and directives, keeping the code more naturally written in C or C++.

We can parallelize the function, `solve()`, using `__par`. This prefix allows us to modify the function for parallel processing. Assuming that there is no overlap among the arguments, the `solve()` function is modified with the addition of the `__par` keyword. With no change to the way the function is called, the computation is parallelized. This will look like this:

```
void solve() {
    __par for(int i=0; i<size; i++) {
        // try all positions in first row
        // create separate array for each recursion
        // started here
        setQueen(new int[size], 0, i);
        }
}
```

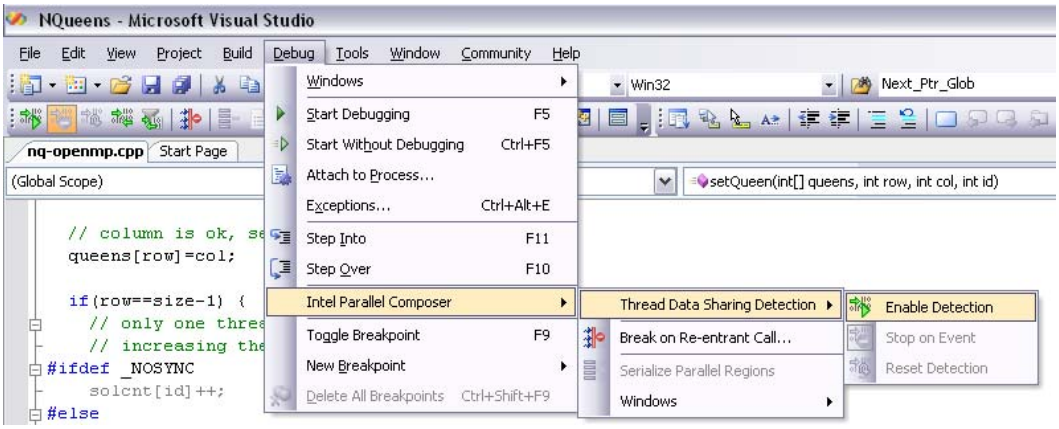
The sample uses the `__par` directive to parallelize the loop by simply putting the macro in front of the loop. If we run our program several times we see that the results are different and it seems we have introduced some kind of error in our program by using the `__par`. This is one of the most common problems in parallel programming and is called a race condition. In this circumstance, multiple threads will each operate on a particular block of code, but the overall outcome of the execution is dependent upon which thread reaches the block of code first. In our example, two processes each increment the value of `nrOfSolutions` by 1 if they found a valid solution.

This consists of three distinct operations:

- Load the value of the `nrOfSolutions` into a register.
- Add 1 to the value in the register.
- Write the contents of the register back into the variable `nrOfSolutions`.

It is clear that if the two processes each operate one after the other, the final value of nrOfSolutions after both processes complete will be equal to the initial value plus 2. Another possible outcome, however, is that the first process could complete the first two steps outlined above, and then that process could be preempted by the second process. The second process would read the value of `nrOfSolutions` as for instance 6, increment it to 7, and then write it back to the variable `nrOfSolutions`. Once the first process resumes, it would continue where it left off, writing 6 back as the value of `nrOfSolutions`. Thus, in the first case, the code generates a final value of 7 for `nrOfSolutions`, and in the second case, the same code generates a final value of `nrOfSolutions` for X. Which result would be generated by any particular execution of the code could be unpredictable. This effect comes into play as we move from a serial program flow to a parallel program flow and is caused by the fact that all threading approaches communicate via the shared memory.

Since detection of race conditions by code inspection is not feasible for non-trivial code, there are tools to help with this task: One way is to instrument the code, execute it with test data and log possible race conditions, for example with the Intel® Thread Checker. Another approach is to flag such conditions during debugging of a desired code path. Intel® Parallel Composer includes the Intel® Parallel Debugger Extension that extends the Microsoft® Visual Studio Debugger with capabilities that facilitate tracking down problems typical for threaded code.



A solution to this problem is to intercept a race condition introduced by the concurrent behavior of the algorithm by protecting the counter for the solutions by a mutual exclusion construct like **_critical**. This will of course slow down the execution a little bit as we serialize the program flow but we will get the right solution. The final code would look like this:

```
__critical
{
        // increasing the solution counter is not atomic
        nrOfSolutions++;
}
```

2.2.3 Starting with OpenMP* 3.0

The sample code can be found in: `..\NQueens\nq-openmp-intel\`

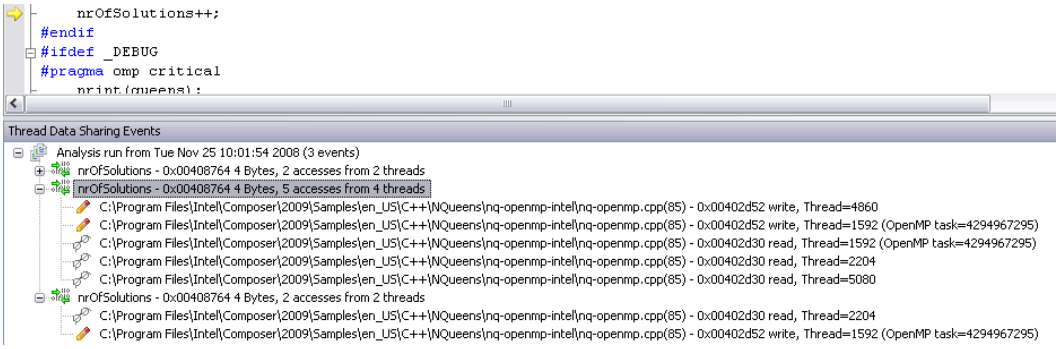
OpenMP* is an industry standard for portable multi-threaded application development. This approach is effective at fine-grain (loop-level) and large-grain (function-level) threading. OpenMP* directives provide an easy and powerful way to convert serial applications into parallel applications, enabling potentially big performance gains from parallel execution on multi-core and symmetric multiprocessor systems. The original source is compiled unmodified. Although directives are inserted into the code, when no action is taken on them (for example, the application is not running in shared memory parallel mode) they do not change the program. For shared memory parallel computers, this allows for simple comparisons between serial and parallel runs. Because only directives are inserted into the code, it is possible to make incremental code changes. The ability to make incremental code changes helps programmers maintain serial consistency. When the code is run on one processor, it gives the same result as the unmodified source code. OpenMP* is a single source code solution that supports multiple platforms and operating systems. There is also no need to

determine the number of cores as the OpenMP* runtime chooses the right number for you.

The latest OpenMP*, version 3.0, contains a new task-level parallelism construct that simplifies its use for parallelizing functions, in addition to the loop-level parallelism for which OpenMP* is most commonly used. In our example we use exactly the same approach as in the parallel exploration version. The only difference is that the **__par** macro is replaced by an OpenMP* pragma with exactly the same functionality.

```
void solve() {
        #pragma omp parallel for
        for(int i=0; i<size; i++) {
        // try all positions in first row
        // create separate array for each recursion
        // started here
        setQueen(new int[size], 0, i);
        }
}
```

The important function in our program is the solve function. Solve is easy to parallelize as the solutions are independent (review the search tree). The sample uses the OpenMP* **#pragma parallel for** to parallelize the important loop. By using the Intel® Parallel Debugger Extension [2] we directly see that the same race condition is happening here.



We can avoid this shared access by protecting the counter for the solutions with a mutual exclusion construct like `#pragma omp critical` or `#pragma omp atomic`:

```
#pragma omp atomic
      nrOfSolutions++; // increasing the solution counter is not atomic
```

There is one distinct advantage of OpenMP* over other typical Windows threading approaches such as Win32 threads. The pragma based technique allows us to use an incremental approach to parallelism this means one decides which parts and hotspot to parallelize and than one can go stepwise through your code and thread wherever it

is necessary. Another advantage is that the serial code is don't destroyed and stays intact. Furthermore one doesn't have to care about advanced thread handling as the OpenMP* runtime automatically does thread pooling whenever possible. Since typical Windows threading does not use thread pooling, each thread encounters the thread startup overhead. This can be prohibitive for many applications. In summary OpenMP* is a flexible and simple set of pragmas, function calls, and environment variables that explicitly instruct the compiler how and where to thread your application. By taking advantage of OpenMP* functionality, threading programming is not that much harder than single-threaded programming.

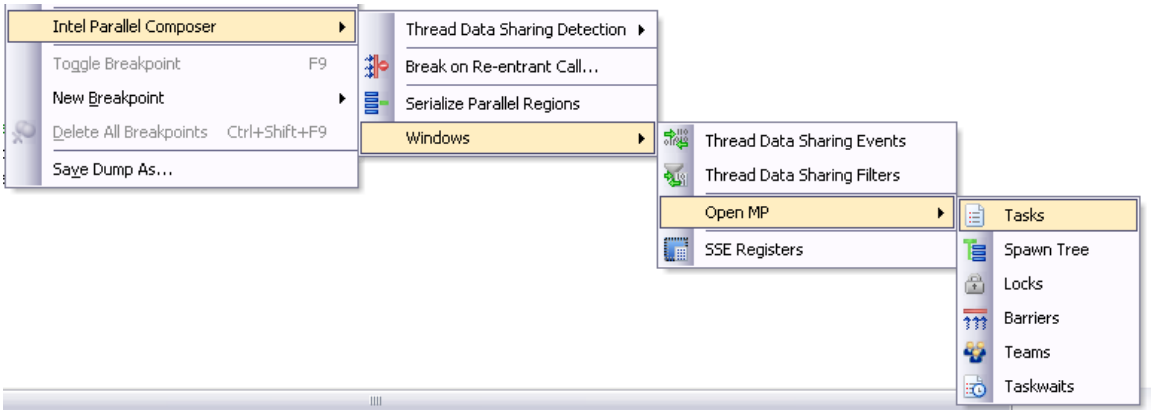
2.2.4 OpenMP* 3.0 Task queuing

The sample code can be found in: …\NQueens\nq-openmp-taskq\

Sometimes programs with irregular patterns of dynamic data structures or with complicated control structures like recursion are hard to parallelize efficiently. The work queuing model allows the user to exploit irregular parallelism, beyond that possible with OpenMP* 2.0 or 2.5. The task pragma specifies the environment within which the enclosed units of work (tasks) are to be executed. When a task pragma is encountered lexically within a task block, the code inside the task block is conceptually queued into the queue associated with the task. To preserve sequential semantics, there is an implicit barrier at the completion of the task. The user is responsible for ensuring that no dependencies exist or that dependencies are appropriately synchronized, either between the task blocks, or between code in a task block and code in the task block outside of the task blocks. For our example, this code would look similar to the following example:

```
#pragma omp parallel
#pragma omp single
{
   for(int i=0; i<size; i++) {
        // try all positions in first row
        // create separate array for each recursion
        // started here
#pragma omp task
          setQueen(new int[size], 0, i);
   }
}
```

In our example we only need one task queue therefore we need to set up the queue only by one thread (omp single). The setQueens calls are independent of each others and therefore they fit nicely into the task concept. With the Intel® Parallel Debugger Extension in Visual Studio, you can easily inspect the state of tasks, teams, locks, barriers, or taskwaits in your OpenMP* program in dedicated windows:



If you want to rule out problems that may have been introduced by threading your code you can also serialize the execution of the parallel regions without recompilation by selecting "Serialize parallel regions" from the Intel-specific debug menu.

As a result the subsequent parallel regions in your program will be executed with only one thread, regardless of the settings of *num_threads* in your program or in the environment. For more information about the Intel® Parallel Debugger Extension [2] please refer to the online help available in Visual Studio*.

2.2.5 Parallelizing with Intel® Threading Building Blocks (Intel® TBB)

The sample code can be found in: `..\NQueens\nq-tbb-intel\`

Intel® Threading Building Blocks (Intel® TBB) offers a rich methodology to express parallelism in a C++ program. It is a library that helps you take advantage of multi-core processor performance. It represents a higher-level, task-based parallelism that abstracts platform details and threading mechanism for performance and scalability. It nicely fits into the object oriented and generic framework of C++. Intel® TBB uses a runtime based programming model and provides the user with generic parallel algorithms based on a template library similar to the standard template library (STL). The Intel® TBB task scheduler does the load balancing for you. With thread-based programming, you are often stuck dealing with load-balancing yourself, which can be tricky to get right. By breaking your program into many small tasks, the Intel® TBB scheduler assigns tasks to threads in a way that spreads out the work evenly.

For the programmers not used to work with templates here is a very brief introduction to get you started with Intel® TBB. Templates are programming constructs, which allow a more generic and data type independent programming. This generic approach can be combined with overloading and inheritance of operators to reach a very high level of abstraction and therefore allow powerful parallel design concepts. Templates are construction skeletons with one or more parameterized types. Templates can be

used to create functions (function templates), classes (class templates) and templates itself (template templates). The big advantage of templates in relation to macros is the type security, which is examined during compile time and thus helps avoids run time errors. Here is an example of a simple function template to compute the maximum of two elements:

```
template <typename T>
T max(T x, T y)
{
  if (x < y)
    return y;
  else
    return x;
}
```

This template is called like a normal function: for example, max(3, 7). The compiler determines the type by looking at the parameters. So this would not only work for integers but for floating point numbers, strings and even classes. The only necessary conditions is that compare operator and the copy constructor are defined for the used data type. The only other thing one needs to know to use Intel® TBB is how functors work. Functors in C++ are used to work on the elements of a given data structure. A functor is a class which is used as a function. The class is defined by overloading the function call operator therefore an `operator()` member function is defined. A simple functor would look like this:

```
#include <list>
#include <algorithm>
#include <iostream>

struct Sum {
   double sum;
   Sum():sum(0.0) {}
   void operator()(const double & d) { sum += d; }
};

std::list<double> myList;
…
Sum mySum = std::for_each(myList.begin(), myList.end(), Sum());
std::cout << "Summe " << mySum.sum << std::endl;
```

The class sum is doing the summation. The constructor is initialized with zero. The `operator()` is defined to add another value. The class is used together with the for_each statement to call the `operator()` for every element in `MyList`. The result is returned by for_each and the result is placed in `mySum.sum`. This methodology already used in C++ is extended by Intel® TBB to realize simple parallelization concepts. Intel® TBB provides a couple of functions templates like `parallel_for`,

`parallel_while`, `parallel_reduce`, `pipeline`, `parallel_sort`, and `parallel_scan` along with some concurrent containers. In our example we make use of the parallel_for template. This functor is used to divide the recursive distribution of the data for the parallel tasks up to an arbitrary selectable grain size. For the processing interval the Intel® TBB class `tbb::block_range` is used which can be instantiated by numbers, pointers and random-access iterators. The functor has to work on the whole range instead on a single element. For the solve loop, the sample implements the parallel_for template function. The first parameter of the call is a blocked_range object that describes the iteration space. The constructor takes three parameters: The lower and upper bound of the range and the <grainsize>. The `parallel_for` subdivides the range into sub-ranges that have approximately <grainsize> elements. The second parameter to the parallel_for function is the function object to be applied to each subrange. So this would look like:

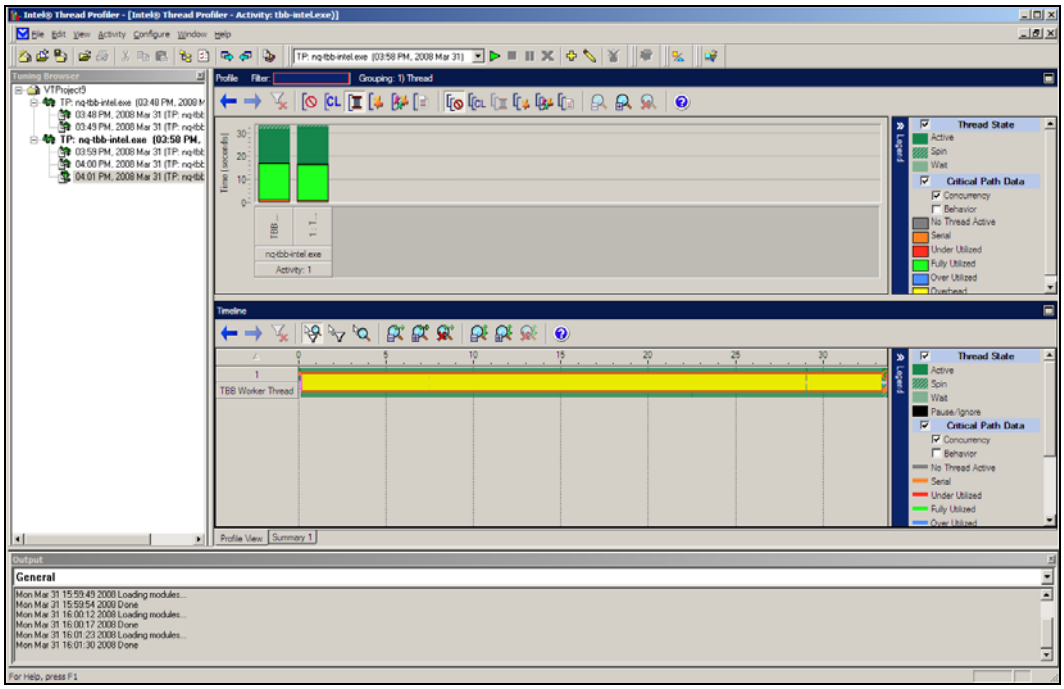
```
class SetQueens {
public:
void operator()( const blocked_range<size_t>& r ) const {
        for( size_t i=r.begin(); i!=r.end(); ++i ) {
        // try all positions in first row
        // create separate array for each recursion
        // started here
        setQueen(new int[size], 0, (int)i);
        }
}

void solve() {
        parallel_for(blocked_range<size_t>(0, size, 1), SetQueens() );
}
```

As in the OpenMP* sample, this sample attempts to protect the solution counter by the mutual exclusion construct **`NrOfSolutionsMutexType`**:

```
if(row==size-1) {
        {
        // increment is not atomic, so setting a lock is required here
        NrOfSolutionsMutexType::scoped_lock mylock(NrOfSolutionsMutex);
        nrOfSolutions++;
        }
// closing block invokes scoped_lock destructor which releases the lock
}
```

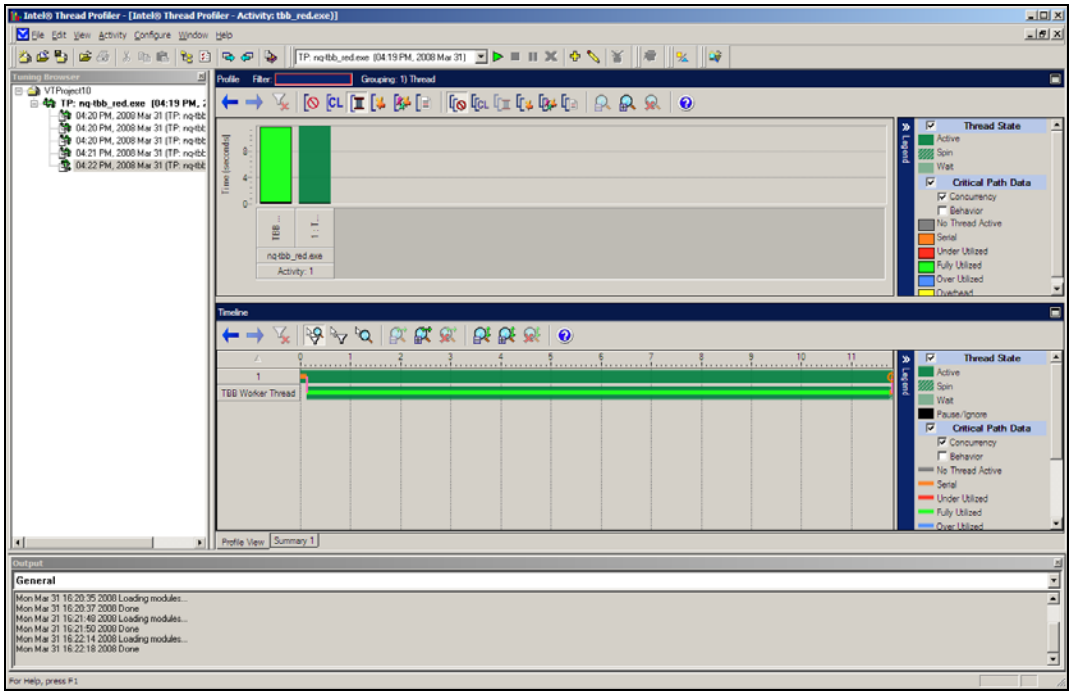
There are of course more and clever ways to implement this in Intel® TBB. For instance a reduce operation can be used but this is left to the reader as an additional exercise. An interesting question is if the parallelization really is efficient and scalable. One method is to look at the runtime and divide it by the number of threads to calculate the speedup in comparison to the serial version.

The disadvantage of this approach is that one gets no insight into the parallel problem. The Intel® Thread Profiler allows interesting insight into the real parallel execution of the concurrent flow graph. It works for all featured approaches. Here is the view of the Intel® TBB runtime.



The two green bars show that our solution is well balanced but with a lot of synchronization overhead which is caused by synchronizing the access to the global variable which counts the number of solutions. It is crucial to all parallelization approaches to validate and verify that the used approach is providing the maximal performance by avoiding slowdowns caused by serialization. (See Amdahl's law on how serial parts of a program can affect the parallel performance.) In our example it is pretty straightforward to get rid of this unnecessary synchronization.

Every time we add a number to the solution counter we have to synchronize this access, as it is a global variable for all threads. To avoid this synchronization we just have to implement a local solution variable for each thread. At the end of the program we simply sum up the local solution variables to the global one. The resulting code run should look similar to the results shown below:



2.2.6 Using *lambda* Functions with Intel® TBB

The sample code can be found in: `..\NQueens\nq-tbb-lambda\`

Lambda functions have been included in the working draft of the next C++ standard C++0x. This addition makes STL/Intel® TBB algorithms an order of magnitude more usable then in the past. The Intel® Compiler is the first C++ compiler which implements lambda function. A lambda abstraction in general defines an unnamed function, so it's a bunch of code without a name. Lambda functions together with closures are a powerful concept because they combine code with a scope. The characteristic of this construct is that it ties a scope to a code block which than can be passed around and the scope doesn't change for the particular block it is tied to. The term used in the standard is "capture" (of an auto), and an auto can be captured by value or reference. A closure is a function that can refer to and alter the values of bindings established by binding forms that textually include the function definition. A lambda construct is almost the same as a function object in C++ or a function pointer in C. A closure then can be seen as a pointer to a function and a pointer to a set of name/variable bindings. Finally a lambda function together with a closure can be seen as a lot of syntactic sugar around function objects and function pointers. This syntactic sugar is intended to be a convenient way to write function objects or lambdas right at the point of use.

This section explains in some detail about using of C++ Lambda expressions. In order to use Intel's implementation of lambda expressions, you need to request C++0X with the command-line option /Qstd that is: /Qstd=c++0X. The compiler creates an anonymous function object upon evaluating a lambda expression. This function object, created by a lambda expression, may live longer than the block in which it is created. You must ensure that it does not use variables that were destroyed before the function object is used.

The following example shows how a function object created by a lambda expression will look like in the context of the Nqueens example. Tighter C++ and Intel® TBB integration allows the simplification of the functor operator() concept by using lambda functions and closures to pass code as parameters:

```
void solve() {
    parallel_for(blocked_range<size_t>(0, size, 1),
     [](const blocked_range<int> &r){
      for (int i = r.begin(); i != r.end(); ++i) {
      setQueen(new int[size], 0, (int)i);
         };
        });
}
```

3 Comparing the Different Methods

Let's finally summarize the parallelization approaches of the different methods like API-based threading methods (the Win32 multithreading API on Windows* OS and the Pthreads library on Linux* OS), OpenMP* and Intel® TBB. We can characterize them by different means. One is the mean of abstraction another one is a mean of control and the last one is a mean of simplicity. We have shown in our example that in exchange for giving up a little flexibility, you get parallelism into your application faster with less impact on your code. This makes you more productive. The Windows* 32 API is the assembly language of parallelism, whereas OpenMP and TBB are the high level languages of parallelism. Depending on your preferences you should now able to decide which method is best for your application. Here are the key characteristics for the various methods.

The API based models are very general; they require the programmer to manually map concurrent tasks to threads. There is no explicit parent-child relationship between the threads; all threads are peers. These models give the programmer control over all low-level aspects of thread creation, management, and synchronization. This flexibility is the key advantage of library-based threading methods. The price of this flexibility is significant code modifications and obviously a lot more coding. Concurrent tasks must

be encapsulated in functions that can be mapped to threads. The other important thing is that most threading API's use arcane calling convention and only accepts one argument so it is often necessary to modify function prototypes and data structures, this may break the abstraction of the program design and fits better in a C than an Object-oriented C++ approach.

OpenMP*, a compiler-based threading method, provides a high-level interface to the underlying thread libraries. With OpenMP*, the programmer uses pragmas (or directives in the case of Fortran) to describe parallelism to the compiler. This removes much of the complexity of explicit threading methods because the compiler handles the details. Due to the incremental approach to parallelism where the serial structure of the application stays intact there are no significant source code modifications necessary. A non OpenMP* compiler simply ignores the pragmas, leaving the underlying serial code intact. However, much of the fine control over threads is lost. Among other things OpenMP* does provide the programmer with a way to set thread priorities or perform event-based or inter-process synchronization. OpenMP* is a fork-join threading model with an explicit master-worker relationship among threads. This narrows the range of problems for which OpenMP* is suited. In general, OpenMP* is best suited to expressing data parallelism while explicit threading API methods are best suited to functional decomposition. OpenMP* is well known to work perfect with loop structures and C code but does not have specific support for C++. OpenMP* 3.0 which includes the task construct, extends OpenMP* by adding support for irregular constructs such as while loops and recursive structures, and function-level parallelism, which is a big improvement.

The Intel® Threading Building Blocks library supports generic scalable parallel programming using standard C++ code like the Standard Template Library (STL). It does not require special languages or compilers. If one needs a flexible high level parallelization approach which fits nicely in an abstract and even generic object oriented approach then Intel® TBB is the right and only choice. Intel® TBB uses templates for common parallel iteration patterns. Intel® TBB supports scalable data parallel programming with nested parallelism. In comparison to the API approach one specifies tasks, not threads, and let the library map tasks onto threads in an efficient way using the Intel® TBB runtime. The Intel® TBB scheduler favors a single, automatic, divide-and-conquer approach to scheduling. It implements task stealing which moves tasks from loaded core to idle ones. In comparison to OpenMP* the generic approach implemented in Intel® TBB allows the user to work with user defined parallelism structures which are not limited to built-in types. Finally combining TBB with lambda functions will reduce the amount of code rearrangement required and will add another level of abstraction to your parallel approach.

The `__task`/`__taskcomplete` extensions can be used in C and C++, even for novice users that want to do some rapid prototyping to get a start on threading. OpenMP* has more features than this simple tasking, and is oriented towards compute-intensive algorithms. These new tasking features are a quick and easy entry into asynchronous programming.

4 Conclusion

Intel® Parallel Composer provides some unique new methods to express parallelism in C++ applications. Each has unique benefits for different use-cases. As you have seen in this sample code guide, some simple modifications to implement OpenMP* or parallel spawn/finish, or Intel® TBB can lead to some impressive performance and scaling gains through multi-threading.

Many algorithms contain optimizations that benefit from serial execution but introduce dependencies that inhibit parallelism. It is often possible to remove such dependencies through simple transformations to make use of any of the above mentioned approaches. It is important to choose an appropriate number of threads to minimize overhead due to thread creation. Creating too many threads hurts performance for many reasons, including increased system overhead, decreased granularity, increased lock contention. In order to avoid race conditions during the execution of a threaded application, mutual exclusion to shared resources is required to allow a single thread to access and change the state of shared resources. The shared resource can be a data structure, or memory in the address space. Minimizing synchronization overheads is a critical to application performance.

Experiment with the sample code provided with the Intel® Parallel Composer. Refer to the product documentation or product website for more information: http://www.intel.com/software/products/.

4.1 References

[1] Hoffman, E.J., Loessi, J.C. and Moore, R.C. (1969): Constructions for the Solution of the m Queens Problem, Mathematics Magazine, p. 66-72.
[2] Robert Mueller-Albrecht (2008): Intel® Parallel Debugger Extension http://software.intel.com/en-us/articles/parallel-debugger-extension.