Creating Correct and Scalable Parallel Applications Using Intel Developer Tools

Ekaterina Antakova

Intel Corporation, Nizhny Novgorod, Russia

kate@antakova.ru

Abstract. *Implementing parallelism is known to be a nontrivial programming task which becomes harder when newly developed parallel code is not correct or does not provide expected speedups. The approach to parallelizing applications suggested in this paper involves Intel(R) developer tools for checking correctness and modeling application parallel behavior while working with its serial code. This allows easy experiments with different parallel solutions for getting projections on performance and confirmation of correctness of parallel code before putting big efforts into its implementation.*

Keywords

Application correctness, memory analysis, threading analysis, scalability, performance, parallel programming, modeling, developer tools.

1 Introduction

In this paper we propose an approach for parallelizing a serial application using Intel developer tools for checking code correctness and modeling parallel behavior. This approach starts by making sure the code is stable enough to add parallelism using Intel(R) Inspector XE, then doing safe and *relatively* quick experiments on different code pieces as a means of introducing parallelism with different tasks granularity using Intel(R) Advisor XE. Using Intel Advisor XE enables checking how scalable a parallel version will be and predicting data sharing issues in future parallel code. After achieving necessary characteristics of projected performance gain and fixing potential threading issues, a parallel version of an application is implemented.

2 Making an Application Parallel in 7 Steps

Design of parallel applications and parallelization of serial programs is a complex task encountered by many modern software engineers. Most of the common issues found by software engineers when writing parallel applications are ensuring correctness after introducing parallelism (no data races corrupting the data and no crashes while accessing shared data), ensuring there are no deadlocks leading to an application hang, and gaining the necessary performance improvement from parallel execution which also scales with larger number of cores/processors.

This article provides an example of how to use developer tools from Intel(R) Parallel Studio XE 2013 and Intel(R) Cluster Studio XE 2013 suites to solve these common problems. It gives a step-by-step approach for tackling these problems and illustrates these steps on a sample application which finds duplicates in source files.

Software engineers would like to achieve the following goals when working on introducing parallelism to an application: (1) ensure application correctness throughout the cycle of experiments and in the final parallel version, (2) get the best performance and scalability gains from parallelism and (3) spend less developer time on parallelizing by avoiding debugging complex parallel issues when you are not sure that the current approach will yield enough benefits in the end.

Intel Parallel Studio XE 2013 and Intel Cluster Studio XE 2013 tools help achieve these three goals in the following way: Intel Inspector XE helps the user address complex correctness issues in an application (both serial and parallel) while Intel Advisor XE provides insight on parallelism potential while maintaining all the benefits of the serial application, e. g. all existing test cases still work and no instabilities caused by parallel execution.

With the above goals in mind let’s consider a step-by-step approach for working towards parallelism. We assume that the original serial application is quite stable, runs without any crashes or hangs, produces acceptable results, and there are some test cases we can rely on to ensure the correct application output. This means the application is ready for parallelizing.

The following steps are proposed for designing a parallel version of an application:

Step 1. Run Intel Inspector XE memory analysis to determine if the application leaks memory or has any incorrect memory accesses. Fixing memory issues early in the development cycle helps us to be sure that application won’t leak more memory in the parallel version and won’t suffer from crashes and incorrect memory accesses when processing more data.

Step 2. After all correctness issues are reviewed and fixed in the serial code, collect application performance metrics using Intel Advisor XE Survey tool (in Release build configuration). Review the most time consuming application functions and loops and choose some of them to start modeling parallelism there.

Step 3. Insert Intel Advisor XE annotations in the selected loops and functions source code, rebuild the application and run Suitability analysis to see the projection of future parallel behavior. During this step the application is executed serially, there is no real parallelism added yet. Repeat steps 2 and 3 with different loops and functions to see what performance gain they bring if parallelized and to check different task granularities. Finally, use this information to choose the list of functions and loops where parallelization seems valuable.

Step 4. Run correctness analysis with Intel Advisor XE for the annotated application in Debug mode to identify potential threading issues like deadlocks or data races in the code where parallelism has been modeled. For the sake of tool performance, it is recommended to reduce the number of iterations in loops and limit input data for Correctness analysis. Review the issues found by Correctness analysis and fix them using lock annotations or code reorganization like replacing shared variables with private variables.

Step 5. Rerun Suitability analysis to determine if the updated version meets your desired performance gains and scalability. If the projected gains are not acceptable, one conclusion can be that effective parallelization is impossible for this particular part of the algorithm with this parallel task division because of hard data dependencies that cannot be effectively resolved. However, you may try rerunning steps 3 and 4 choosing different loops, functions and shared data protection approaches until getting the desired performance result or deciding to tweak the algorithm.

Step 6. Change source code annotations to use real parallel framework calls. It is worth checking to be sure that the application works as expected on existing test cases after adding parallel code.

Step 7. Run Intel Inspector XE threading analysis on the final parallel application to make sure parallel version has no threading issues like data races or deadlocks. It is likely that you won’t get any threading issues at this point as the potential issues should have been fixed after running Correctness check on step 4. Note: it’s also worth re-running Intel Inspector XE memory analysis to ensure that changed code did not get any regressions in terms of memory usage.

Figure 1 represents proposed steps for modeling and implementing parallelism in a serial application.

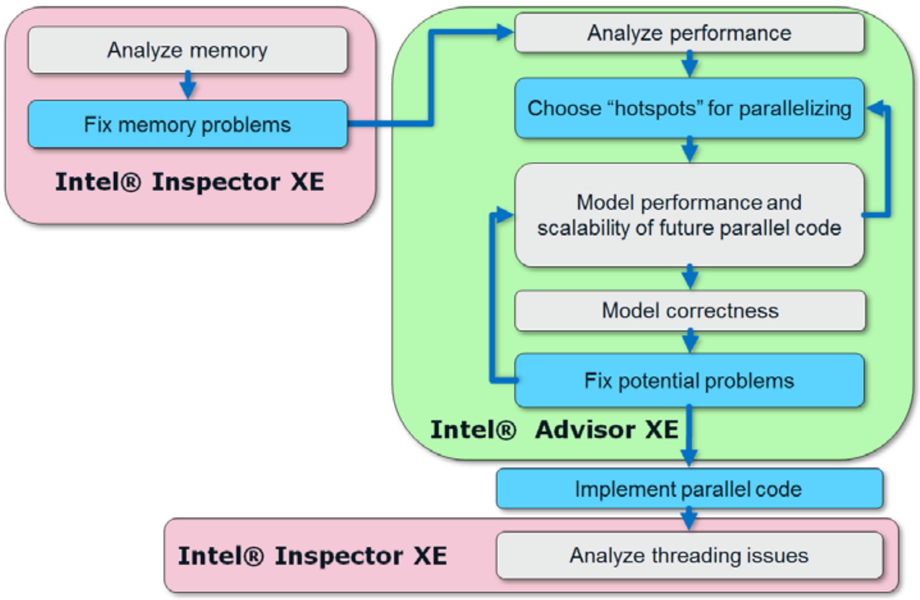


Fig 1. Proposed steps for designing and implementing parallelism

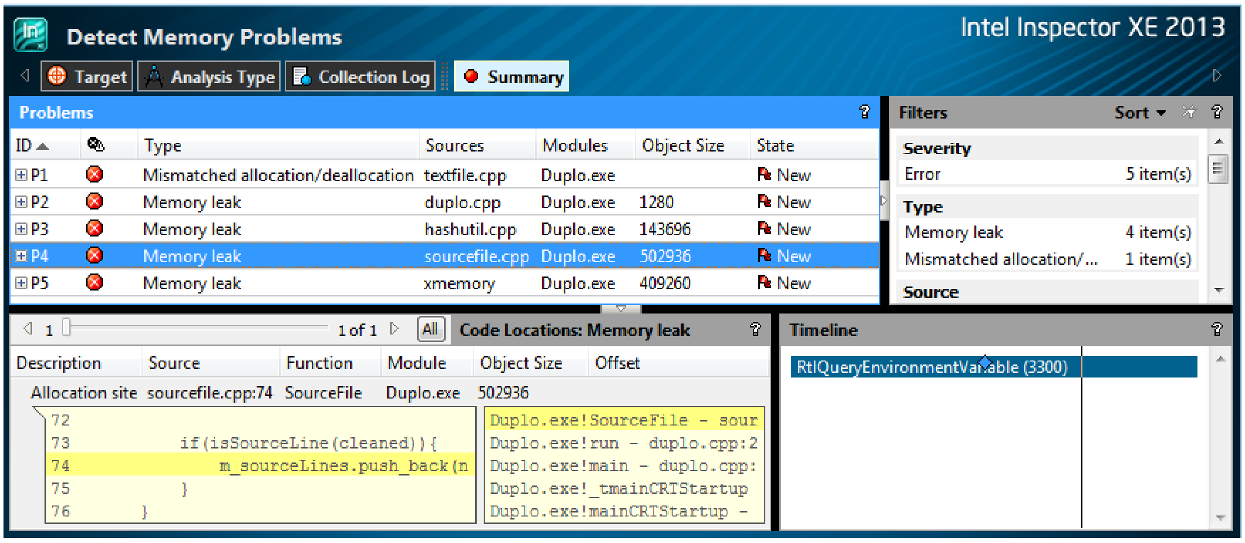
3 Sample Application

To illustrate the proposed approach for parallelism, let's apply it to an open-source application *duplo* (http://duplo.sourceforge.net/), a program for searching for duplicate lines in a source code base. The application should have parallelism potential as checking different pairs of source files for duplicates can be done independently - in parallel. We will be experimenting with the 32-bit version of the *duplo* application on a Microsoft* Windows 7 machine with 4 cores using Intel(R) Composer XE as a compiler.

4 Applying the 7 Steps to the Application

Step 1. Collect Inspector XE memory errors for the application built in Debug mode.

Memory problems summary in Inspector XE tool for *duplo* application is shown on figure 2.



ID		Type	Sources	Modules	Object Size	State
P1	●	Mismatched allocation/deallocation	textfile.cpp	Duplo.exe		New
P2	●	Memory leak	duplo.cpp	Duplo.exe	1280	New
P3	●	Memory leak	hashutil.cpp	Duplo.exe	143696	New
P4	●	Memory leak	sourcefile.cpp	Duplo.exe	502936	New
P5	●	Memory leak	xmemory	Duplo.exe	409260	New

Description	Source	Function	Module	Object Size	Offset
Allocation site	sourcefile.cpp:74	SourceFile	Duplo.exe	502936	

```

        if(isSourceLine(cleaned)){
            m_sourceLines.push_back(n
        }
    }
```

```
Duplo.exe!SourceFile - sour
Duplo.exe!run - duplo.cpp:2
Duplo.exe!main - duplo.cpp:
Duplo.exe!_tmainCRTStartup
Duplo.exe!mainCRTStartup -
```

Fig 2. Memory problems reported by Intel Inspector XE in *duplo* application

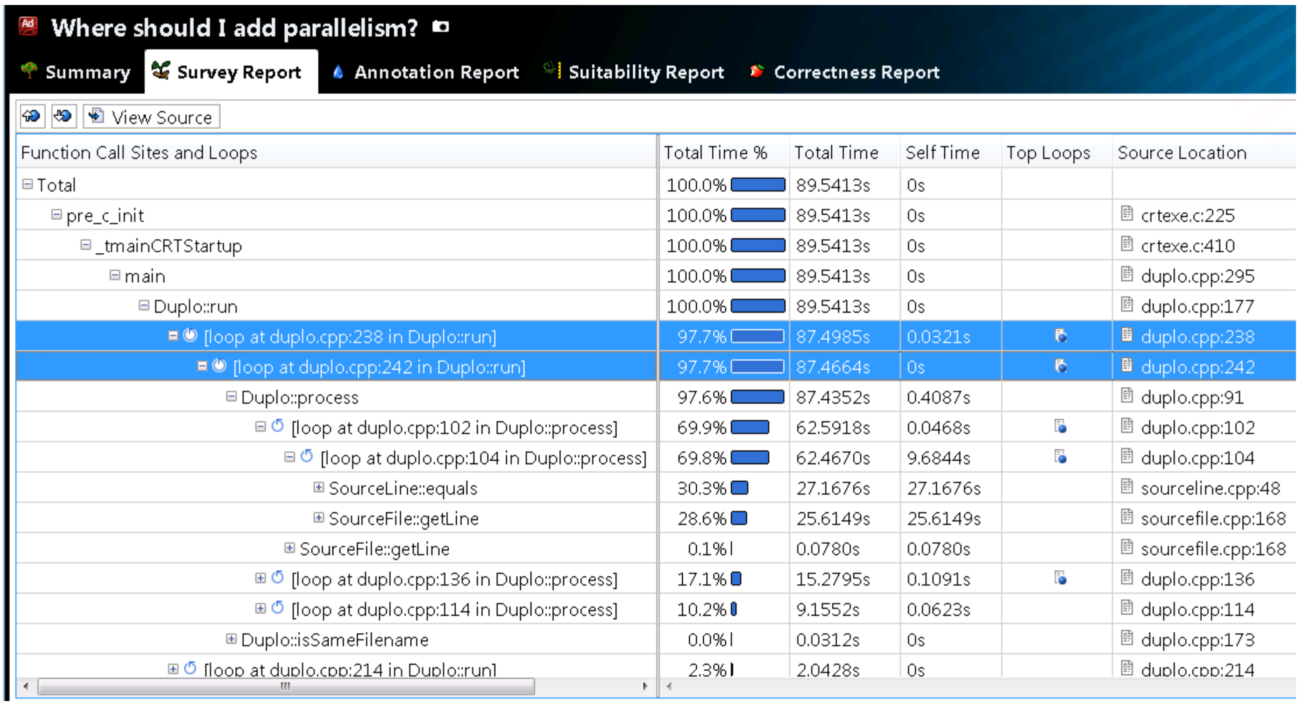
The summary of problems here shows that the application has one incorrect memory allocation/deallocation call pair. In this particular case it is using `operator delete` incorrectly for one array that `operator new[]` was called to create. This issue is worth fixing as incorrect usage of the `delete` operator may lead to undefined behavior with different compilers. There are also a couple of memory leaks in the code. Memory leaks should be just cleared out to use memory more efficiently and avoid any issues that might be triggered if this code is reused and run multiple times in any bigger software systems. As Intel Inspector XE shows the total leaked amount of memory in bytes in Object Size column, we can see that the application leaks about a megabyte of memory for a relatively small input dataset consisting of 30 files.

These memory leaks are appearing because many objects such as SourceFile and SourceLine are being created in the heap using `operator new` and there is no explicit deleting of these objects. We fix the issue by adding appropriate deallocations for these objects. The mismatched allocation/deallocation issue also gets fixed by adding `delete[]` operator instead of the simple object deletion which was used in the original *duplo* application.

We make sure all the problems are fixed in a subsequent Intel Inspector XE run that lists 0 problems in Summary window and continue to the next step for designing parallel code.

Step 2. Collect Intel Advisor XE Survey analysis for the application in Release configuration.

Using Intel Advisor XE modeling tool we're going to identify the functions and loops of the *duplo* application that have the longest execution time. The summary of application execution time can be seen on figure 3.



Function Call Sites and Loops	Total Time %	Total Time	Self Time	Top Loops	Source Location
⊟ Total	100.0%	89.5413s	0s		
⊟ pre_c_init	100.0%	89.5413s	0s		crtexe.c:225
⊟ _tmainCRTStartup	100.0%	89.5413s	0s		crtexe.c:410
⊟ main	100.0%	89.5413s	0s		duplo.cpp:295
⊟ Duplo::run	100.0%	89.5413s	0s		duplo.cpp:177
⊟ [loop at duplo.cpp:238 in Duplo::run]	97.7%	87.4985s	0.0321s	●	duplo.cpp:238
⊟ [loop at duplo.cpp:242 in Duplo::run]	97.7%	87.4664s	0s	●	duplo.cpp:242
⊟ Duplo::process	97.6%	87.4352s	0.4087s		duplo.cpp:91
⊟ [loop at duplo.cpp:102 in Duplo::process]	69.9%	62.5918s	0.0468s	●	duplo.cpp:102
⊟ [loop at duplo.cpp:104 in Duplo::process]	69.8%	62.4670s	9.6844s	●	duplo.cpp:104
⊞ SourceLine::equals	30.3%	27.1676s	27.1676s		sourceline.cpp:48
⊞ SourceFile::getLine	28.6%	25.6149s	25.6149s		sourcefile.cpp:168
⊞ SourceFile::getLine	0.1%	0.0780s	0.0780s		sourcefile.cpp:168
⊞ [loop at duplo.cpp:136 in Duplo::process]	17.1%	15.2795s	0.1091s	●	duplo.cpp:136
⊞ [loop at duplo.cpp:114 in Duplo::process]	10.2%	9.1552s	0.0623s		duplo.cpp:114
⊞ Duplo::isSameFilename	0.0%	0.0312s	0s		duplo.cpp:173
⊞ [loop at duplo.cpp:214 in Duplo::run]	2.3%	2.0428s	0s		duplo.cpp:214

Fig 3. Intel Advisor XE in *duplo* application

Here the total execution time is displayed in a calls hierarchy starting from program main function. As we can see, the program spends all its time inside the `Duplo::run()` method, and in that method the most time consuming loop is a loop at `duplo.cpp` line 238. This loop takes 97.7% of program execution time. Also, this loop has another loop inside starting at line 242 of the `duplo.cpp` source file. The inner loop calls the `Duplo::process()` method which also has a couple of loops inside with different workloads in them.

Intel Advisor XE gives suggestions on which loops to parallelize, indicating them with an icon in the Top Loops column. In this paper we're going to concentrate on two the topmost application loops, highlighted in blue on figure 3. While the tool gives some suggestions on where to start parallelizing an application, the developer is free to experiment with any approach and model the program's future parallel behavior whether it was recommended as "top loop" or not.

Step 3. Insert Intel Advisor XE annotations to the selected loops and check the predicted parallel performance.

To utilize Intel Advisor XE modeling capabilities let's choose an outer loop for modeling parallelism and insert special annotation macros into the source code that will serve as markers showing which parts of the code are going to be parallelized.

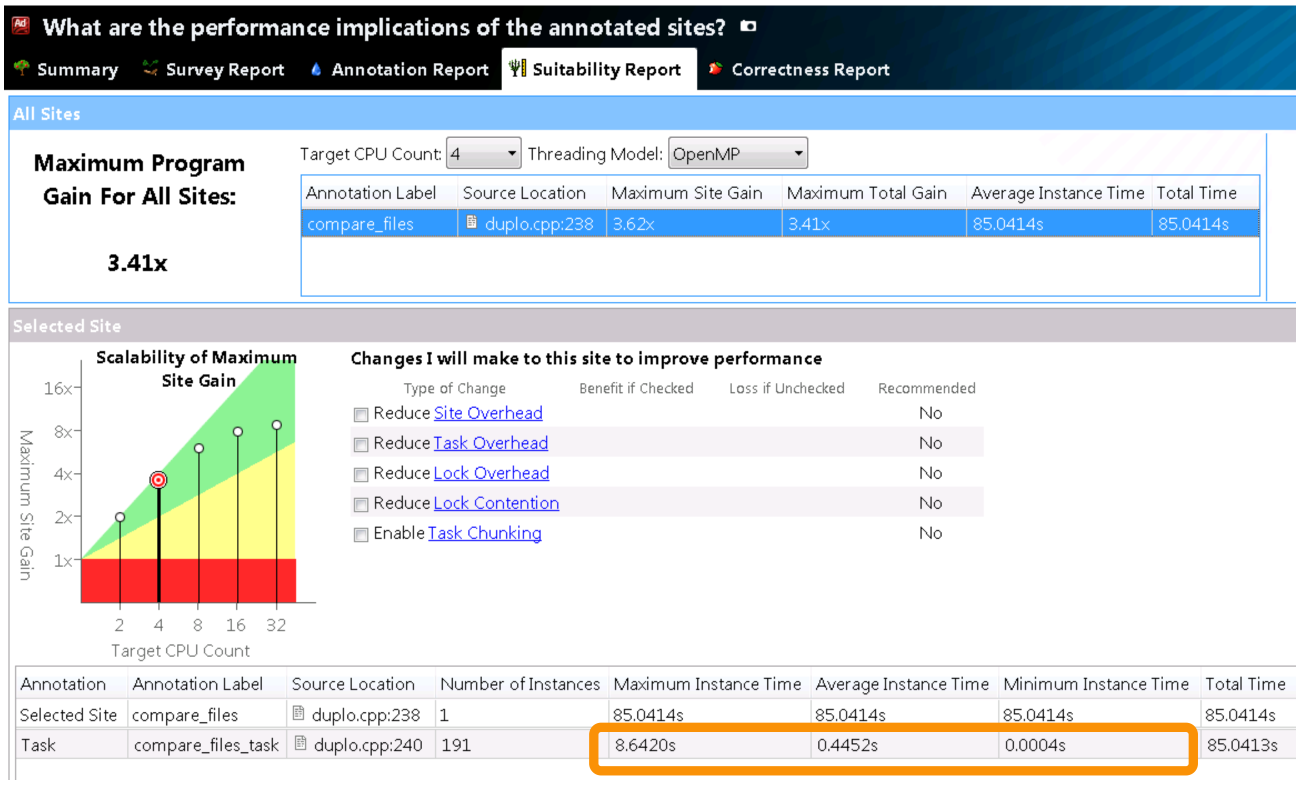
The source code with annotations continues to run in serial as annotations do not change any program logic. This gives developers some advantages. For example, all existing tests will still pass for this version of application. Also, there are no complex multi-threading issues introduced by annotations, and the application development may continue safely while modeling is being done with Intel Advisor XE annotations.

Figure 4 shows source code with `ANNOTATE_SITE_BEGIN`, `ANNOTATE_SITE_END` and `ANNOTATE_ITERATION_TASK` annotations in the outer loop of files processing.

```
ANNOTATE_SITE_BEGIN( compare_files );
for(int i=0;i<(int)sourceFiles.size();i++){
      ANNOTATE_ITERATION_TASK(compare_files_task);
      outfile << sourceFiles[i]->getFilename();
      int blocks = 0;

      for(int j=0;j<(int)sourceFiles.size();j++){
            if(i > j && !isSameFilename(sourceFiles[i]->getFilename(), sourceFiles[j]->getFilename())){
                  blocks+=process(sourceFiles[i], sourceFiles[j], outfile);

            }
      }

      if(blocks > 0){
            outfile << " found " << blocks << " block(s)" << std::endl;
      } else {
            outfile << " nothing found" << std::endl;
      }

      blocksTotal+=blocks;
}
ANNOTATE_SITE_END(compare_files);
```

Fig 4. Annotations in outer loop

Let’s run Intel Advisor XE Suitability analysis to see predicted performance implications of parallelizing this file processing loop.



Maximum Program Gain For All Sites: 3.41x

Target CPU Count: 4   Threading Model: OpenMP

Annotation Label	Source Location	Maximum Site Gain	Maximum Total Gain	Average Instance Time	Total Time
compare_files	duplo.cpp:238	3.62x	3.41x	85.0414s	85.0414s

Changes I will make to this site to improve performance

Type of Change	Benefit if Checked	Loss if Unchecked	Recommended
Reduce Site Overhead			No
Reduce Task Overhead			No
Reduce Lock Overhead			No
Reduce Lock Contention			No
Enable Task Chunking			No

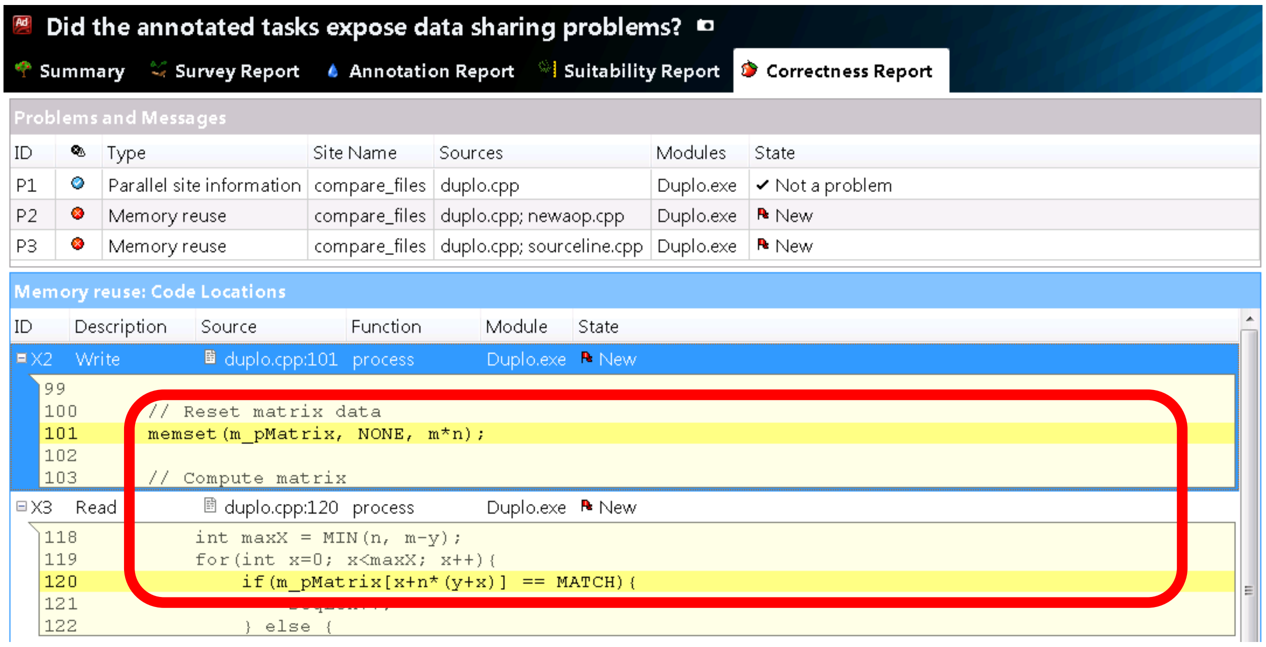
Annotation	Annotation Label	Source Location	Number of Instances	Maximum Instance Time	Average Instance Time	Minimum Instance Time	Total Time
Selected Site	compare_files	duplo.cpp:238	1	85.0414s	85.0414s	85.0414s	85.0414s
Task	compare_files_task	duplo.cpp:240	191	8.6420s	0.4452s	0.0004s	85.0413s

Fig. 5. Modeling of performance implications of parallelizing outer loop

The suitability report for modeling parallelism in outer program loop (in figure 5) shows that parallel iterations of the loop were executed 191 times. Maximum time of one iteration is 8.642 seconds and minimum iteration time was very small – 0.0004s. This represents a big imbalance between parallel tasks in this loop and this input data workload. However, even with this imbalance Intel Advisor XE suggests that parallelizing this loop will bring 3.41X speedup for the whole program if you use OpenMP parallelism. Scalability diagram also shows projected speedups for different number of CPUs.

Step 4. Run Intel Advisor XE Correctness analysis on the selected loop where parallelism is modeled.

Parallel execution of loop iterations can insert correctness issues into the application like data races on shared memory structures. Intel Advisor XE Correctness analysis shows these potential issues before parallel code is even implemented and gives the developer a chance to fix them in the modeling stage.



Did the annotated tasks expose data sharing problems?

Summary | Survey Report | Annotation Report | Suitability Report | Correctness Report

Problems and Messages

ID		Type	Site Name	Sources	Modules	State
P1	●	Parallel site information	compare_files	duplo.cpp	Duplo.exe	✓ Not a problem
P2	●	Memory reuse	compare_files	duplo.cpp; newaop.cpp	Duplo.exe	New
P3	●	Memory reuse	compare_files	duplo.cpp; sourceline.cpp	Duplo.exe	New

Memory reuse: Code Locations

ID	Description	Source	Function	Module	State
X2	Write	duplo.cpp:101	process	Duplo.exe	New

```

// Reset matrix data
memset(m_pMatrix, NONE, m*n);

// Compute matrix
```

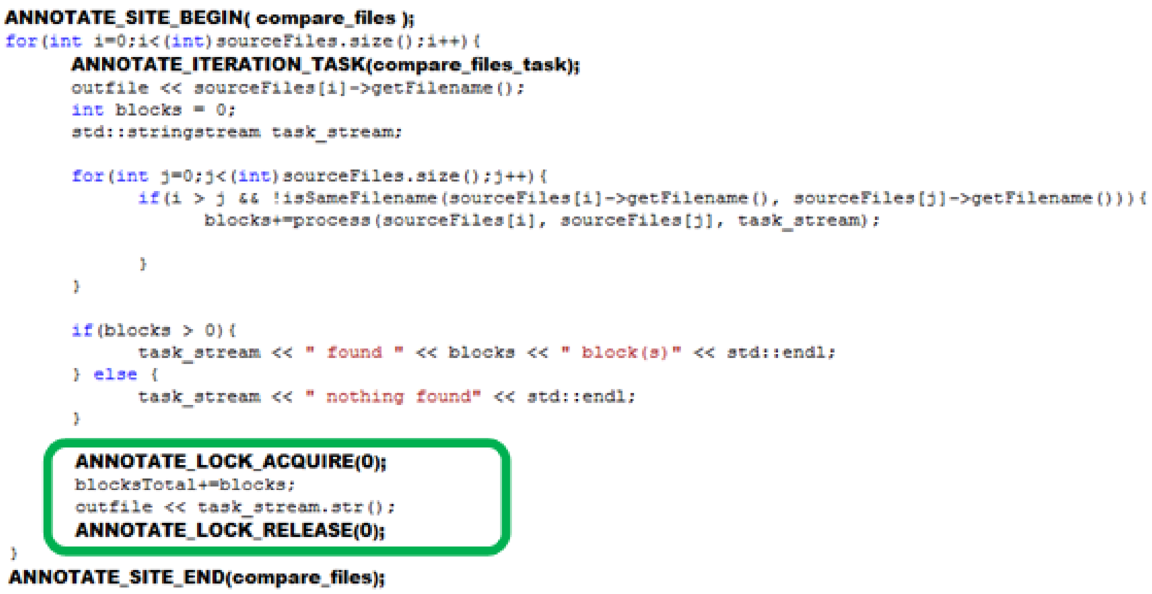
ID	Description	Source	Function	Module	State
X3	Read	duplo.cpp:120	process	Duplo.exe	New

```
    int maxX = MIN(n, m-y);
    for(int x=0; x<maxX; x++){
        if(m_pMatrix[x+n*(y+x)] == MATCH){
            sequent++;
        } else {
```

Fig 6. Modeling potential correctness issues of annotated outer loop

Correctness analysis results in figure 6 shows that future parallel loop execution will reuse the same memory for the `m_pMatrix` variable. Another “Memory Reuse” problem shows that the same output file stream is used in parallel iterations non-exclusively. One possible way to fix these errors would be by using separate matrixes in loop iterations to eliminate data sharing and adding a lock for writing to file operation. We’re going to model lock behavior with Intel Advisor XE lock annotations `ANNOTATE_LOCK_ACQUIRE` and `ANNOTATE_LOCK_RELEASE`.

Figure 7 shows what the code with lock annotations looks like.



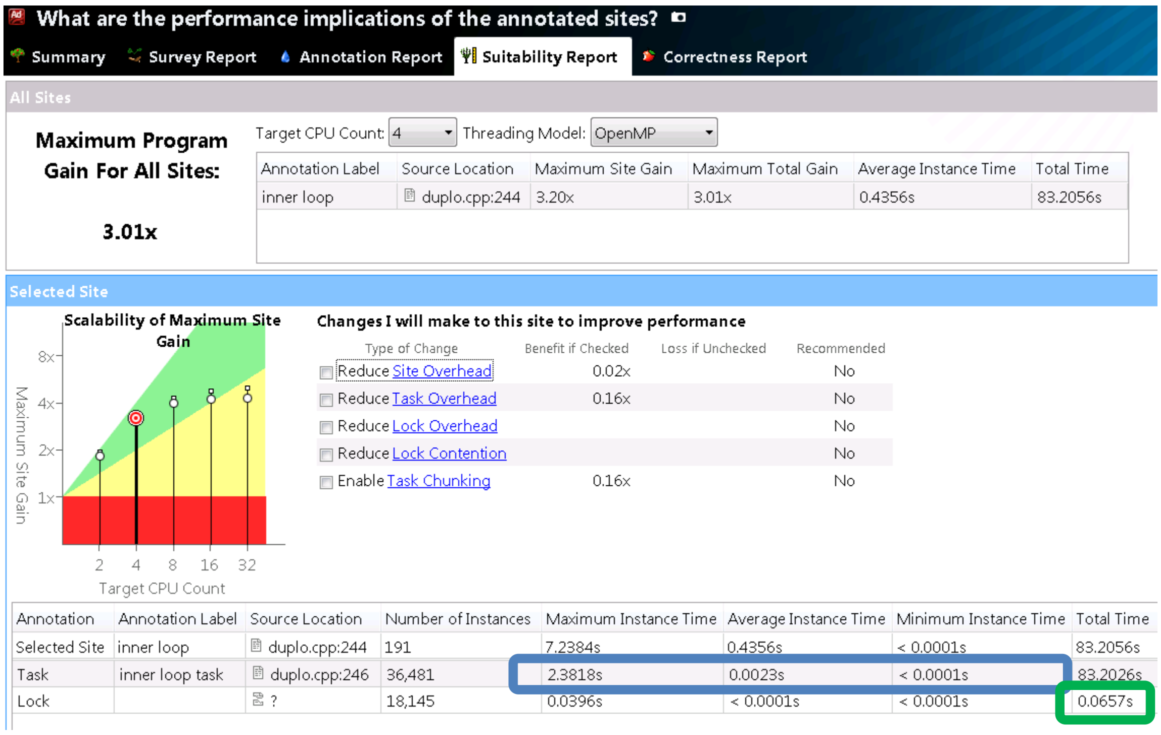
```
ANNOTATE_SITE_BEGIN( compare_files );
for(int i=0;i<(int)sourceFiles.size();i++){
        ANNOTATE_ITERATION_TASK(compare_files_task);
        outfile << sourceFiles[i]->getFilename();
        int blocks = 0;
        std::stringstream task_stream;

        for(int j=0;j<(int)sourceFiles.size();j++){
                if(i > j && !isSameFilename(sourceFiles[i]->getFilename(), sourceFiles[j]->getFilename())){
                        blocks+=process(sourceFiles[i], sourceFiles[j], task_stream);

                }
        }

        if(blocks > 0){
                task_stream << " found " << blocks << " block(s)" << std::endl;
        } else {
                task_stream << " nothing found" << std::endl;
        }
        ANNOTATE_LOCK_ACQUIRE(0);
        blocksTotal+=blocks;
        outfile << task_stream.str();
        ANNOTATE_LOCK_RELEASE(0);
}
ANNOTATE_SITE_END(compare_files);
```

Fig 7. Outer loop with lock annotations added

After ensuring there are no correctness issues left in the current version of modeled application, we rerun Intel Advisor XE Suitability analysis to see how adding the lock impacted the predicted performance. The updated Suitability result gives us a slightly different number of overall predicted speedup – 3.40X comparing to 3.41X without the lock. The lock is not making a big difference for this outer loop, as it takes little total time and does not downgrade the predicted execution time significantly.

Now that we have predicted the performance improvement of this parallel version (**Step 5**) it is useful to perform experiments on other loops and hot functions of the application.

The inner loop of file processing was also analyzed using steps 4 and 5. Annotations were added to the loop source code and Suitability metrics were collected for it. Figure 8 shows the results of Suitability modeling of inner loop.


What are the performance implications of the annotated sites?

Summary | Survey Report | Annotation Report | Suitability Report | Correctness Report

All Sites

Maximum Program Gain For All Sites: 3.01x

Target CPU Count: 4 Threading Model: OpenMP

Annotation Label	Source Location	Maximum Site Gain	Maximum Total Gain	Average Instance Time	Total Time
inner loop	duplo.cpp:244	3.20x	3.01x	0.4356s	83.2056s

Selected Site

Changes I will make to this site to improve performance

Type of Change	Benefit if Checked	Loss if Unchecked	Recommended
Reduce Site Overhead	0.02x		No
Reduce Task Overhead	0.16x		No
Reduce Lock Overhead			No
Reduce Lock Contention			No
Enable Task Chunking	0.16x		No

Annotation	Annotation Label	Source Location	Number of Instances	Maximum Instance Time	Average Instance Time	Minimum Instance Time	Total Time
Selected Site	inner loop	duplo.cpp:244	191	7.2384s	0.4356s	< 0.0001s	83.2056s
Task	inner loop task	duplo.cpp:246	36,481	2.3818s	0.0023s	< 0.0001s	83.2026s
Lock		?	18,145	0.0396s	< 0.0001s	< 0.0001s	0.0657s

Fig. 8. Modeling of performance implications of parallelizing inner loop

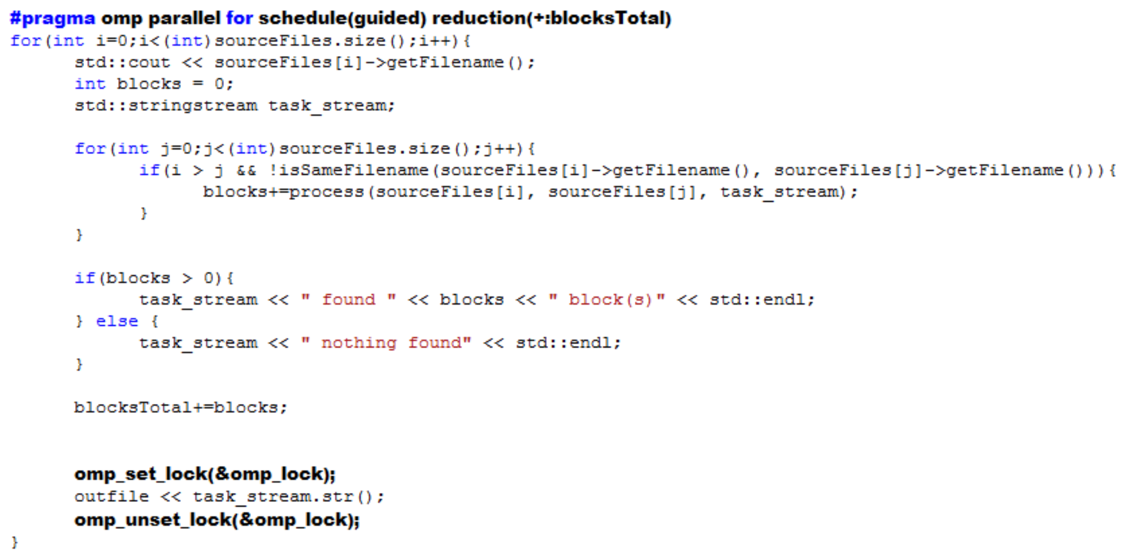
The overall program speedup of the parallel version is predicted to be 3.01X in this case. As we can see from the inner loop task metrics, this loop has also some imbalance in iterations time: from 2.3818 seconds to 0.0001 seconds. This makes parallelization harder for the loop. We should also note that total lock time for this loop is not that big compared to total application time, which means that a lock does not limit performance and scalability in this case. However, Task Overhead time for this loop in Intel Advisor XE is shown as an important metric contributing to overall application performance projection. This means that tasks are quite small in this parallel site and creating such tasks may bring noticeable overhead to the whole program.

Comparing the performance predictions for parallel versions of the two observed loops, let’s implement parallel version of the first outer loop which projects better performance and scalability improvement– **3.41X** for the whole program compared to 3.01X for the inner loop of files processing.

Step 6. Implementing the parallel code.

In this paper, an OpenMP implementation was used for parallelizing the outer loop of files processing in *duplo* application. Annotations were replaced with a `parallel for` construct having reduction on a sum variable `blocksTotal` and with lock operation on writing the output to file.

Figure 9 shows OpenMP parallel implementation of outer loop source code.



```
#pragma omp parallel for schedule(guided) reduction(+:blocksTotal)
for(int i=0;i<(int)sourceFiles.size();i++){
      std::cout << sourceFiles[i]->getFilename();
      int blocks = 0;
      std::stringstream task_stream;

      for(int j=0;j<(int)sourceFiles.size();j++){
            if(i > j && !isSameFilename(sourceFiles[i]->getFilename(), sourceFiles[j]->getFilename())){
                  blocks+=process(sourceFiles[i], sourceFiles[j], task_stream);
            }
      }

      if(blocks > 0){
            task_stream << " found " << blocks << " block(s)" << std::endl;
      } else {
            task_stream << " nothing found" << std::endl;
      }

      blocksTotal+=blocks;


      omp_set_lock(&omp_lock);
      outfile << task_stream.str();
      omp_unset_lock(&omp_lock);
}
```

Fig. 9. Parallel code implementation using OpenMP

Threading errors analysis for the resulting parallel application with Intel Inspector XE (**Step 7**) showed no threading issues in this parallel code, which correlates with Intel Advisor XE Correctness analysis results that showed no potential threading issues for the serial annotated version of the application with the new locks.

It is interesting to compare predicted (theoretical) performance improvement given by Intel Advisor XE tool and real performance improvement achieved by parallelization. In this case, a speedup of **3.15X** was achieved in OpenMP-based parallel version of the application on a 4-cores processor while Intel Advisor XE suggested 3.40X improvement.

5 Conclusion

Designing scalable and correct parallel application involves experimenting with different approaches to parallelism. Intel Advisor XE models performance implications and potential correctness issues before parallel code is implemented. This kind of modeling allows determining the correct approaches with the best predicted performance results before heavy investments are made in parallel implementation.

Memory and threading errors are often unstable to reproduce and hard to debug. The Intel Inspector XE correctness checking tool helps in finding these errors in both serial and parallel code making development and debugging faster.

This paper demonstrates an effective usage model of Intel Parallel Studio XE tools for designing and implementing parallel applications. The proposed 7 steps approach helps in creating stable, highly-performing and scalable parallel applications while investing reasonable efforts.

6 Acknowledgments

Author expresses her sincere gratitude to Trevor D'Arcy-Evans at http://duplo.sourceforge.net/ for providing *duplo* application under GNU General Public License version 2.0 (GPLv2).