Wlodzimierz BIELECKI, **Marek PALKOWSKI**
FACULTY OF COMPUTER SCIENCE, WEST POMERANIAN UNIVERSITY OF TECHNOLOGY,
ul. Żołnierska 49, 71-210 Szczecin

# Programming synchronization-free parallelism using Intel Threading Building Blocks

**Prof. dr hab. inż. Wlodzimierz BIELECKI**

Is head of the Software Technology Department of the West Pomeranian University of Technology, Szczecin. His research interest includes parallel and distributed computing, optimizing compilers, extracting both fine- and coarse grained parallelism available in program loops.

*e-mail: wbielecki@wi.zut.edu.pl*

**Dr inż. Marek PAŁKOWSKI**

Has graduated and obtained his PhD degree in Computer Science from the Technical University of Szczecin, Poland. The main goal of his research is extracting parallelism from program loops and developing Iteration Space Slicing Framework.

*e-mail: mpalkowski@wi.zut.edu.pl*

**Abstract**

Extracting synchronization-free parallelism by means of the Iteration Space Slicing Framework results in parallel pseudo-code that is independent on a parallel computer architecture and API/library, hence it cannot be directly compiled. For producing parallel programs for shared memory multiprocessors, Threading Building Blocks (TBB) can be applied that is a library supporting scalable parallel programming based on the standard C++ language. In this paper, we present how to benefit from TBB in practice on the basis of pseudo-code representing synchronization-free slices produced by a tool using the Omega Library. Results of experiments with the NAS benchmarks suite are presented.

**Keywords**: synchronization-free slices, parallel computing, tasking, Intel Threading Building Blocks.

## Programowanie równoległości wolnej od synchronizacji przy użyciu Intel TBB

**Streszczenie**

Zastosowanie techniki opartej na ekstrakcji równoległości pozbawionej synchronizacji w pętlach programowych pozwala na wygenerowanie pseudokodu, który jest niezależny od architektury komputera oraz języka lub biblioteki programowania. Taki kod nie może być wprost kompilowany. Jest wymagane przekształcenie takiego pseudokodu na rzeczywisty kod równoległy. W tym celu może być zastosowane narzędzie Intel Threading Building Blocks, które jest biblioteką wspierająca skalowalne programowanie równoległe w standardzie C++. Nie wymaga specjalnego języka programowania i specjalnych kompilatorów. Zaletą biblioteki Threading Building Blocks jest możliwość uruchomienia w dowolnym środowisku programowo-sprzętowym i systemie operacyjnym. W artykule przedstawiono korzyści wynikające z tworzenia aplikacji równoległych za pomocą TBB. Wyjaśniono sposób poszukiwania instancji instrukcji fragmentów kodu przy użyciu biblioteki Omega i tworzenie najpierw równoległego pseudo-kodu, a dalej transformacja pseudokodu na kod równoległy z wykorzystaniem TBB. Proponowane podejście zostało zweryfikowane za pomocą zbioru pętli testowych z benchmarku NAS. Zbadano przyspieszenie i efektywność kodu równoległego oraz skalowalność w aspekcie do zmiennego rozmiaru obliczeń badanych pętli.

**Słowa kluczowe**: fragmenty kodu pozbawione synchronizacji, równoległość, zadaniowość, Intel Threading Building Blocks.

## 1. Introduction

Multi-core processors have made parallel programming a topic of interest for every programmer. Computer systems without multiple processor cores have become relatively rare. In our recent work [1] we proposed several algorithms to extract coarse-grained parallelism represented with synchronization-free slices consisting of the loop statement instances by means of the Iteration Space Slicing Framework (ISSF). Parallelism is represented by pseudo-code, i.e., such a code represents all extracted parallelism but cannot be directly compiled. There is the need to convert it to a real code being dependent on an computer architecture and API/Language.

In the current paper, we focus on generating the parallel Intel Threading Building Blocks (TBB) code [2], [3] on the basis of a pseudocode representing synchronization-free slices. Intel TBB is a C++ template library for developing parallel programs; it extends C++ by abstracting away thread management and allowing straightforward parallel programming. The parallel code consists of tasks, not threads. Moreover, the library maps tasks onto threads in an efficient manner [3].

## 2. Iteration Space Slicing Framework

The approach to extract synchronization-free parallelism in program loops by means of the Iteration Space Slicing Framework requires an exact representation of dependences. Two statement instances $s_1(I)$ and $s_2(J)$ are *dependent* if both access the same memory location and if at least one access is a write. $s_1(I)$ and $s_2(J)$ are called the source and destination of a dependence, respectively, provided that $s1(I)$ is lexicographically smaller than $s_2(J)$ ($s_1(I) \prec s_2(J)$, i.e., $s_1(I)$ is always executed before $s_2(J)$).

To describe and implement our algorithms, we choose the dependence analysis proposed by Pugh and Wonnacott [4] where dependences are represented by dependence relations whose constraints are described in the Presburger arithmetic (built of affine equalities and inequalities, logical and existential operators); the Omega library is used for computations over such relations [5].

A dependence relation is a tuple relation of the form

$$\{[\textit{input list}] \rightarrow [\textit{output list}]: \textit{constraints}\}; \qquad (1)$$

where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples and *constraints* is a Presburger formula describing constraints imposed upon *input list* and *output list*.

We use standard operations on relations and sets, such as intersection ($\cap$), union ($\cup$), difference (-), domain of relation (domain($R$)), range of relation (range($R$)), relation application (given a relation $R$ and set $S$, $R(S) = \{[e']: \exists e \in S, e \rightarrow e' \in R\}$), positive transitive closure (given a relation $R$, $R^+ = \{[e] \rightarrow [e']: e \rightarrow e' \in R \,||\, \exists e'' \text{ s.t. } e \rightarrow e'' \in R \,\&\, e'' \rightarrow e' \in R^+\}$), transitive closure ($R^* = R^+ \cup I$, where $I$ is the identity relation). These operations are described in detail in [5].

Iteration Space Slicing [1] takes dependence information as the input to find all statement instances that must be executed to produce the correct values for the specified array elements.

**Definition 1.** Given a dependence graph, *D*, defined by a set of dependence relations, *S*, a *slice* is a weakly connected component of graph *D*, i.e., a maximal subgraph of *D* such that for each pair of vertices in the subgraph there exists a directed or undirected path.

If there exist two or more slices in *D*, then taking into account the above definition, we may conclude that all slices are synchronization-free, i.e., there is no dependence between them.

**Definition 2.** An *ultimate dependence source (destination)* is a source (destination) that is not the destination (source) of another dependence. Ultimate dependence sources and destinations represented by relation *R* can be found by means of the following calculations: (domain(*R*) - range(*R*)) and (range(*R*) - domain(*R*)), respectively.

**Definition 3.** A *source(s) of a slice* is an ultimate dependence source(s) that this slice contains.

**Definition 4.** A *representative* loop statement instance of a slice is its lexicographically minimal source.

Further on in this paper, we refer to representative loop statement instances as to representatives.

The approach to extract synchronization-free slices [1] relies on the transitive closure of an affine dependence relation describing all dependences in a loop and consists of two steps. First, representatives of slices are found in such a manner that each slice is represented with its lexicographically minimal statement instance. Next, slices are reconstructed from their representatives and code scanning these slices is generated. Details of this technique are presented in paper [1].

In the second step, we can use the Gen affine algorithm [1] in order to reconstruct slices. The algorithm allows us to generate code scanning synchronization-free slices of an arbitrary topology of the dependence graph when the transitive closure of a dependence relation representing all the dependences available in a given program loop. This algorithm uses well-known code generation techniques to scan elements of affine sets representing synchronization-free slices. Slices reconstruction can be realized also by means of other algorithms presented in papers [6], [8]. The output of these algorithms is pseudo-code representing all extracted synchronization-free slices. Converting pseudo-code into a parallel program can be implemented using various multi-threading tools, for example: OpenMP, NVIDIA CUDA, MPI, or Intel TBB.

## 3. Intel Threading Building Blocks

Intel Threading Building Blocks offers a rich and complete approach to expressing parallelism in a C++ program. TBB is not just a threads-replacement library; it represents a higher-level, task-based parallelism that abstracts platform details and threading mechanisms for performance and scalability[2].

There is a variety of approaches to parallel programming, ranging from the use of platform-dependent threading primitives to exotic new languages. The advantage of Threading Building Blocks is that it works at a higher level than raw threads. The library can be used with any compiler supporting ISO C++ and it does not require special languages or compilers. [3].

TBB differs from typical threading packages in the following ways [2]: enables programmer to specify tasks instead of threads; is compatible with other threading packages; emphasizes scalable, data-parallel programming and targets threading for performance; relies on generic programming - the library can be integrated with the C++ Standard Template Library (STL).

Intel Threading Building Blocks enables the programmer to write parallel programs when parallelism is already extracted. Scalable parallelism can be represented by a loop of iterations running simultaneously without interfering with each other. In order to implement TBB loops including synchronization-free slices, the directive *tbb:parallel_for* can be used.

## 4. Example

Let us illustrate how parallel TBB loops can be formed by means of the following *FT_auxfnct_2* NAS loop [9].

```
for(i=1; i<=N1; i++)
  for(k=1; k<=N2; k++)
    for(j=1; j<=N3; j++){
      y[j][k][i]=y[j][k][i]*twiddle[j][k][i];
      x[j][k][i]=y[j][k][i];
    }
```

Dependences available in this loop are represented by the dependence relation returned by Petit [5]:

```
R = {[i,k,j,22] -> [i,k,j,26] : 1 <= i <= N1 && 1
<= k <= N2 && 1 <= j <= N3}.
```

Following the Gen affine algorithm [1] and using the Omega calculator, we extract representatives of slices, represented by the following set:

```
SOUR = {[i,k,j,22]: 1 <= i <= N1 && 1 <= k <= N2
&& 1 <= j <= N3}.
```

Next, we reconstruct synchronization-free slices and generate parallel pseudo-code:

```
if (N3 >= 1 && N2 >= 1) {
 par for(t1 = 1; t1 <= N1; t1++) {
  for(t2 = 1; t2 <= N2; t2++) {
   for(t3 = 1; t3 <= N3; t3++) {
  y[t3][t2][t1]=y[t3][t2][t1]*twiddle[t3][t2][t1];
    if (t3 >= 1 && N3 >= t3 && t2 >= 1 && N2 >= t2
&& t1 >= 1 && N1 >= t1) {
     x[t3][t2][t1]=y[t3][t2][t1];
   }
}}}}
```

It represents all extracted slices.

Then, we manually transform the above code to the parallel Intel TBB code:

```
using namespace tbb;
...
class FT_Aux {

float ***x;
float ***y;
float ***twiddle;

public:
void operator()( const blocked_range<size_t>& r )
const {

int t2, t3;
if (N3 >= 1 && N2 >= 1) {
// parallel loop
for( size_t t1=r.begin(); t1 != r.end(); ++t1 ){
 for(t2 = 1; t2 <= N2; t2++) {
  for(t3 = 1; t3 <= N3; t3++) {
  y[t3][t2][t1]=y[t3][t2][t1]*twiddle[t3][t2][t1];
  if (t3 >= 1 && N3 >= t3 && t2 >= 1 && N2 >= t2
&& t1 >= 1 && N1 >= t1) {
    x[t3][t2][t1]=y[t3][t2][t1];
   }
  }
}}}}

//constructor
FT_Aux( float ***_x, float ***_y, float
***_twiddle)
  {x = _x; y = _y; twiddle = _twiddle;}

};
```

```
int main(int argc, char **argv){

...
// parallelizing loops with tasking
parallel_for(blocked_range<size_t>(1,N1+1),
FT_Aux(x, y, twiddle));
...
}
```

The `using` directive in the example enables the programmer to use the library identifiers without having to write out the namespace prefix `tbb` before each identifier.

An instance of the `FT_Aux` class needs member fields that remember all the local variables that were defined outside the original loop but used inside it. Usually, the constructor for the body object will initialize these fields, though `parallel_for` does not care how the body object is created.

The iteration space here is of type `size_t`, and goes from 1 to N1. The template function `tbb::parallel_for` breaks this iteration space into chunks, and runs each chunk on a separate thread. The first step in parallelizing this loop is to convert the loop body into a form that operates on a chunk. The form is an STL-style function object, called the body object, in which `operator()` processes a chunk. The object is an important function of the FT_Aux class and includes the body of the parallel loop.

Note the argument to `operator()`. A `blocked_range<T>` is a template class provided by the library. It describes a one-dimensional iteration space over type T. This class is the first argument of `parallel_for`. The second argument is the constructor of the FT_Aux class with pointers to shared data. `The Parallel_for` function is called in the main program.

Since version 2.2, Intel TBB chooses chunk sizes automatically, depending upon load balancing needs. The heuristic attempts to limit overheads while still providing ample opportunities for load balancing. The default automatic chunking is recommended for most uses. However, it can be controlled manually as the third argument of `blocked_range<T>` describing size of grain [3].

## 5. Experiments

We have examined parallel codes produced on the basis of the Iteration Space Slicing Framework for the following three NAS Parallel Benchmark loops [9]: *FT_auxnfct_2*, *UA_diffuse_5*, *UA_setup_16* presented in table 1.

We have studied the speedup and efficiency of TBB parallel codes on the multiprocessor machine: 8x Intel Quad Core, 1.6 GHz, 3 GB RAM, Fedora Linux. The results of experiments: time, sped-up S, and efficiency E for 1, 2, 8, and 32 cores are presented in Table 2. The produced parallel code is characterized by positive speed-up (S>1). The efficiency of parallel code increases as the volume of computations executed by this code increases.

Analyzing the data in this table, we can conclude that the presented approach can be applied to producing efficient parallel code for real-life loops using the Iteration Space Slicing Framework and Intel Threading Building Blocks.

Tab. 1. Examined loops from NAS benchmark
Tab. 1. Badane pętle programowe z zestawu NAS

FT_auxfnct_2

```
for(i = 1; i <= N1; i++){
 for(k = 1; k <= N2; k++){
  for(j = 1; j <= N3; j++){
     y[j][k][i]=y[j][k][i]*twiddle[j][k][i];
     x[j][k][i]=y[j][k][i];
}} }
```

UA_diffuse_5

```
for(k = 1; k <= N1; k++)
 for(iz = 1; iz <= N2; iz++)
  for(j = 1; j <= N3; j++)
    for(i = 1; i <= N4; i++)

r[i][j][iz]=r[i][j][iz]+u[i][j][k]*wdtdr[k][iz];
```

UA_setup_16

```
for(i=1; i<=N1; i++)
 for(j=1; j<=N2; j++)
  for(ip=1; ip<=N3; ip++)
     wdtdr[i][j] = wdtdr[i][j] +
wxm1[ip]*dxm1[ip][i]*dxm1[ip][j];
```

Tab. 2. Time of calculation, speed-up, efficiency of parallelized loops executed by different number of CPU processors
Tab. 2. Czas wykonania, przyspieszenie i efektywność obliczeń zrównoleglonych pętli równoległych wykonanych za pomocą różnej liczby procesorów

| Loop | Parameters | 1 CPU | 2 CPUs | | | 8 CPUs | | | 32 CPUs | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Time | Time | S | E | Time | S | E | Time | S | E |
| FT_auxnfct_2 | N1,N2,N3=100 | 0,387 | 0,231 | 1,675 | 0,838 | 0,081 | 4,778 | 0,597 | 0,046 | 8,413 | 0,263 |
| | N1,N2,N3=150 | 1,931 | 1,065 | 1,813 | 0,907 | 0,301 | 6,415 | 0,802 | 0,130 | 14,854 | 0,464 |
| | N1,N2,N3=200 | 4,859 | 2,431 | 1,999 | 0,999 | 0,661 | 7,351 | 0,919 | 0,322 | 15,090 | 0,472 |
| UA_diffuse_5 | N1,N2=64, N3=N4=32 | 0,031 | 0,023 | 1,348 | 0,674 | 0,008 | 3,875 | 0,484 | 0,006 | 5,167 | 0,161 |
| | N1, N2=64, N3=N4=128 | 0,155 | 0,097 | 1,598 | 0,799 | 0,031 | 5,000 | 0,625 | 0,015 | 10,333 | 0,323 |
| | N1,N2=64, N3=N4=256 | 0,767 | 0,422 | 1,818 | 0,909 | 0,121 | 6,339 | 0,792 | 0,050 | 15,340 | 0,479 |
| UA_setup_16 | N1,N2=500 | 0,453 | 0,300 | 1,510 | 0,755 | 0,076 | 5,961 | 0,745 | 0,032 | 14,156 | 0,442 |
| | N1,N2=1000 | 2,153 | 1,376 | 1,565 | 0,782 | 0,322 | 6,686 | 0,836 | 0,125 | 17,224 | 0,538 |
| | N1,N2=2000 | 6,541 | 4,065 | 1,609 | 0,805 | 1,007 | 6,496 | 0,812 | 0,329 | 19,881 | 0,621 |

## 6. Related work

Along with Intel Threading Building Blocks, another promising abstraction for C++ programmers is OpenMP [2], [7]. OpenMP is a language extension consisting of pragmas, routines, and environment variables. OpenMP has been referred to as "convenient for Fortran-style code written in C". However, OpenMP contains nothing special for language C++ [2]. The loop structures are the same loop nests that were developed for vector supercomputers—an earlier generation of parallel processors that performed tremendous amounts of computational work in very tight nests of loops and were programmed largely in Fortran. Transforming those loop nests into parallel code could be very rewarding in terms of results.

Intel Threading Building Blocks can be joined with other threading libraries. Below is an example that parallelizes an outer loop with OpenMP and an inner loop with Intel Threading Building Blocks [3].

```
void TBB_NestedInOpenMP() {
#pragma omp parallel
{
   #pragma omp for
   for( int i=0; i<M; ++j ) {
        paralel_for(blocked_range<size_t>(1,N1+1),
        FT_Aux(x, y, twiddle));
        }
   }
}
```

Supercomputer users, with their thousands of processors, do not generally have the luxury of shared memory, so they use message passing, most often through the popular Message Passing Interface (MPI) standard [10]. The tool exposes the control of parallelism at its lowest level. As such, they offer maximum flexibility, but at a high cost in terms of programmer effort, debugging time and maintenance costs. Using Threading Building Blocks to express parallelism with tasks allows developers to express more concurrency and finer-grained concurrency than would be possible with threads, leading to increased scalability [2].

## 7. Conclusion and future work

There has been presented a way how to form representative loop statement instances of synchronization-free slices in practice by means of Intel Threading Building Blocks. Experiments with NAS loops were carried out for speed-up and efficiency analysis using machine containing 32 processing units.

Threading Building Blocks is designed for C++, and thus to provide the simplest possible solutions for the types of programs written in C++. Hence, Threading Building Blocks is not limited to statically scoped loop nests.

In our future work, we plan to design a tool for the automatic translation of ISSF pseudo-code to a TBB application.

## 8. References

[1] Beletska A., Bielecki W., Cohen A., Palkowski M., Siedlecki K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel Computing, 37, s. 479–497, 2011.

[2] Reinders J.: Intel Threading Building Blocks, Outfitting C++ for Multi-core Processor Parallelism, Print ISBN:978-0-596-51480-8, O'Reilly Media, 2007.

[3] Intel Threading Building Blocks – Tutorial, Document Number 319872-006US, www.intel.com, 2011.

[4] Pugh W. and Wonnacott D.: An exact method for analysis of value-based array data dependences. In In Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Springer-Verlag, 1993.

[5] Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T. and Wonnacott D.: The omega library interface guide. Technical report, USA, 1995.

[6] Bielecki W., Palkowski M.: Using message passing for developing coarse-grained applications in openmp. In: J. Cordeiro, B. Shishkov, A. Ranchordas and M. Helfert, Editors, ICSOFT (PL/DPS/KE), INSTICC Press (2008), pp. 145–152.

[7] OpenMP API, www.openmp.org.

[8] Bielecki W., Beletska A., Palkowski M. and San Pietro P.: Finding synchronization-free parallelism represented with trees of dependent operations. In ICA3PP, volume 5022 of LNCS, pp.185-195. Springer, 2008.

[9] NAS benchmark suite. http://www.nas.nasa.gov.

[10] The Message Passing Interface (MPI) standard, http://www.mcs.anl.gov/research/projects/mpi/.

[11] Pugh W. and Rosser E.: Iteration space slicing and its application to communication optimization. In International Conference on Supercomputing, pp. 221-228, 1997.

*otrzymano / received: 03.09.2011*
*przyjęto do druku / accepted: 03.10.2011* *artykuł recenzowany*

## INFORMACJE