2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Loop parallelism: a new skeleton perspective on data parallel patterns

M. Danelutto, M. Torquati
Department of Computer Science, University of Pisa
Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
Email: {marcod,torquati}@di.unipi.it

***Abstract*—Traditionally, skeleton based parallel programming frameworks support data parallelism by providing the programmer with a comprehensive set of data parallel skeletons, based on different variants of map and reduce patterns. On the other side, more conventional parallel programming frameworks provide application programmers with the possibility to introduce parallelism in the execution of loops with a relatively small programming effort. In this work, we discuss a "ParallelFor" skeleton provided within the FastFlow framework and aimed at filling the usability and expressivity gap between the classical data parallel skeleton approach and the loop parallelisation facilities offered by frameworks such as OpenMP and Intel TBB. By exploiting the low run-time overhead of the FastFlow parallel skeletons and the new facilities offered by the C++11 standard, our ParallelFor skeleton succeeds to obtain comparable or better performance than both OpenMP and TBB on the Intel Phi many-core and Intel Nehalem multi-core for a set of benchmarks considered, yet requiring a comparable programming effort.**

Keywords: parallel design patterns, algorithmic skeleton, data parallelism, loop parallelism, multi- and many-core.

I. Introduction

Algorithmic skeletons have been around since the '90s as a viable and effective solution to support parallel application development. An algorithmic skeleton is a general purpose, efficient, reusable, parametric parallelism exploitation pattern [1]. Application programmers may instantiate skeletons (or proper composition of skeletons) to encapsulate and exploit the full parallel structure of their applications. Business code may be passed as a parameter to the generic skeleton, thus turning the generic skeleton into a part of a parallel application. Algorithmic skeletons are usually provided to the application programmers as library entries and therefore a complete *separation of concerns* is achieved: application programmers are in charge of the design of the most convenient parallel application structure through proper selection of the skeletons (skeleton compositions) among those provided by the skeleton framework, while system programmers are in charge of target architecture specific, efficient and scalable implementation of the parallel skeletons.

Overall, the algorithmic skeleton approach guarantees efficiency, scalability and some kind of functional and performance portability across different target architectures (possibly including hardware accelerators and coprocessors) provided that the application programmer succeeds modelling the parallel structure of the application at hand using a proper composition of the available skeletons. Unfortunately, structuring the parallelism of an application with skeletons requires an extra effort by the application programmer. The amount and kind of effort required in the different skeleton programming frameworks often impaired the acceptance of the frameworks despite the encouraging performance values demonstrated. The problem is often exacerbated taking into account that single parallel patterns may be trivially expressed in other state-of-the-art non structured parallel programming frameworks such as OpenMP [2].

Data parallel applications in general, and sequential iterative kernels with independent iterations (parallel loops, from now on) in particular, have been proved to be easily implemented on multi-core platforms using the FastFlow framework by *streamisation of loop iterations* implemented using the *task-farm* parallel pattern [3].

On the one hand, this approach provides the programmer with great flexibility, allowing also to fully customise the scheduling policy and/or to nest multiple level of parallel computations. On the other hand, it requires a significant re-factoring of the original sequential code thus introducing possible new bugs and not preserving sequential equivalence.

Furthermore, when using a skeleton-based parallel approach for a given parallel problem, a proper selection of the appropriate implementation skeleton (typically totally in charge of the application programmer) together with a correct implementation of the sequential wrapper code is of foremost importance for obtaining the best performance.

For all these reasons, we decided to implement a set of parallel patterns on top of the basic FastFlow skeletons to ease the implementation of parallel loops using a skeleton-based run-time. Almost like in OpenMP pragma compiler directives [4], with the new ParallelFor pattern the programmer is only in charge to identify parallel loops, without the need to rewrite or change the loop body. Currently, the new pattern is implemented using C++ macros, thus covering only a (significant) subset of all possible parallel loop cases. An extension of this pattern is currently under development.

The remainder of the paper is structured as follows: Sec. II briefly introduces the background. Sec. III discusses the FastFlow parallel loop implementation details. Sec. IV discusses results obtained by using a set of tests. Finally, Sec. V draws conclusions.

This work has been partially supported by FP7 STREP ParaPhrase (www.paraphrase-ict.eu).

1066-6192/14 $31.00 © 2014 IEEE
DOI 10.1109/PDP.2014.13

II. Background

Loop parallelism is a topic that has been repeatedly investigated over the years using different approaches and techniques for iterations scheduling [5], [6], [7]. In this paper we concentrate for performance comparison on OpenMP [2] and TBB [8], which represent to a major extent, the most widely used and studied frameworks for loop parallelisations.

A. OpenMP

OpenMP uses a directive based approach, where the programmers annotate their programs with pragmas that instruct the compilers about the parallelism to be used in the program. Despite the introduction of task-based parallelism in recent versions of the standard, loop parallelism is the most important part of many OpenMP programs.

In OpenMP, two constructs are used to parallelise a loop: the *parallel* and the *loop* construct. The *parallel* construct, introduced by the `parallel` directive, declares a parallel region which will be executed in parallel by a pool of threads. The *loop* construct, introduced by the `for` directive, is placed within the parallel region to distribute the loop iterations to the threads executing the parallel region (thread team). OpenMP supports several strategies for distributing loop iterations to threads. The strategy may be specified via the `schedule(type[,chunk])` clause, which is appended to the `for` directive. The `type` of scheduling policy can be:

- *static*: loop iterations are divided into as equal as possible pieces of size *chunk* and then statically assigned to threads in a round-robin fashion. Default chunk size is $\#iterations/\#threads$.
- *dynamic*: use the internal work queue to give a chunk-sized block of loop iterations to each thread dynamically. When a thread completes the execution of the chunk, it retrieves the next chunk from the top of the work queue. By default, the chunk size is 1.
- *guided*: similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies the minimum size chunk to use. The size of the initial block is proportional to $\#iterations/\#threads$, subsequent blocks are proportional to $\#iterations\_remaining/\#threads$. The default chunk size is 1.
- *auto*: with this policy the decision regarding scheduling is delegated to the compiler and/or run-time system. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads. The optional chunk cannot be specified with this policy.
- *runtime*: uses the OMP_SCHEDULE environment variable to specify which one of the previous loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

If not otherwise specified, in the tests considered in this paper, we always used the following pragma directive:

```
#pragma omp parallel for schedule(runtime) num_threads(N)
```

B. TBB

Intel Threading Building Blocks (TBBs) [8] is a library that enables support for scalable parallel programming using standard C++. It provides higher-level abstractions using generic programming to exploit task-based parallelism without considering the underlying platform details and threading mechanisms. The tasks generated by the higher-level abstractions are then scheduled using a work-stealing policy [9].

The TBB `parallel_for` and `parallel_foreach` methods implement the *map* pattern so they may be used to parallelise independent invocation of the elemental function body of a for loop whose number of iterations is known in advance. C++11 lambda functions can be used as arguments to these calls so that the loop body function can be described as part of the call rather than being separately declared.

The reduction pattern can be accessed via the `parallel_reduce` construct. It allows the specification of an arbitrary combiner function. However, in order for the result to be computed deterministically the reduction function needs to be associative and commutative.

The `parallel_for` splits the half-open range $[0, niter)$ into sub-ranges and processes each sub-range r as a separate task using a serial for loop in the code. The range and sub-range are implemented as `blocked_range` objects. The function template `parallel_for` maps a functor across range of values. It is provided in several forms, in this paper we used the range-based algorithm version with C++11 lambda expressions. As an example of use, consider the following serial code:

```
for( size_t  i=0; i<N; ++i)   B[i] = f(A[i]);
```

it may be rewritten in TBB using C++11 lambda as follows:

```
parallel_for (blocked_range<size_t>(0,N),[&](blocked_range<size_t>r)
    {for(size_t  i=r.begin(); i!=r.end(); ++i) B[i]=f(A[i]);});
```

C. FastFlow

FastFlow[1] is a C++ based parallel programming framework built on top of POSIX threads aimed at providing the parallel programmer with a set of pre-defined algorithmic skeletons modelling the main stream-parallel patterns [10], [3].

Together with the *sequential* code wrapper, it provides two basic algorithmic skeletons: i) a *farm* skeleton, applying in parallel the function modelled by an inner skeleton composition (the farm worker) to all the items appearing on its input stream, and delivering results to its output stream; and ii) a *pipeline* skeleton, applying in sequence the functions implemented by its inner skeleton compositions (the pipeline stages) to the items appearing on the pipeline input stream, and delivering the results to the pipeline output stream. Both pipelines and farms, when used at the topmost level in the skeleton composition, support a *feedback-channel* providing the programmer with the possibility to move data back from the output stream directly to the input stream.

The farm skeleton can be instantiated in several different forms allowing to fully customise the task scheduling and

[1] FastFlow is an open source project: http://mc-fastflow.sourceforge.net/

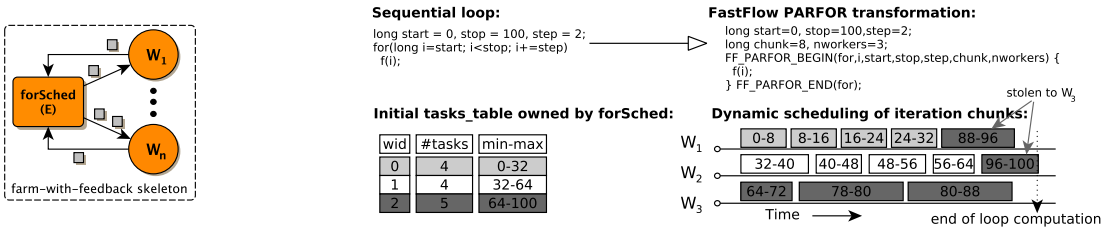

Fig. 1: PARFOR skeleton (left). Example of a parallel loop with 100 iterations, chunk=8 and 3 worker threads (right).

gathering policies. For the implementation of the PARFOR pattern we used the one in which the Emitter thread (the scheduler thread) has been completely programmed to schedule loop iterations and where farm workers are sequential wrappers executing chunks of the loop iterations.

III. FastFlow `PARFOR`

We implemented a set of C++ macros on top of the basic FastFlow skeletons to ease the implementation of parallel loops using a skeleton-based run-time. Currently the following macros have been implemented:

```
/* parallel for */
FF_PARFOR_BEGIN(name,index,start,stop,step,chunk,nworkers)
  /* loop body here */
FF_PARFOR_END(name);

/* parallel reduce */
FF_PARFORREDUCE_BEGIN(name,var,identity,idx,
                      begin,end,step,chunk,nworkers)
  /* loop body here */
FF_PARFORREDUCE_END(name,var,op);
```

which can be used to parallelize a loop having the form `for(idx=start;idx<stop;idx+=step)` and also a reduce loop with one scalar reduction variable (`var`), respectively. `op` may be one of `+,*,-,/,&,^,|`. `chunk` is the grain size; `nworkers` is the number of threads in the team. [2]

The above macros take care of the full set of implementation details related to efficient parallelism exploitation, in particular those needed to instantiate the skeleton, create, initialise and destroy the pool of threads and finally free memory. In order to avoid these overheads in case the ParallelFor is called multiple times (for example within to another loop), the FF_PARFOR/REDUCE_INIT/START/STOP/DONE macros have been also defined:

```
/* initialises the PARFOR name */
FF_PARFORREDUCE_INIT(name,decltype(var),nworkers);
do {
 FF_PARFORREDUCE_START(name,var,identity,idx,
                       begin,end,step,chunksize,nworkers)
    /* loop body here */
 FF_PARFORREDUCE_STOP(name,var,op);
} while (...) ;
FF_PARFORREDUCE_DONE(name);/* terminates the PARFOR name */
```

The macros implementation make widely use of C++11 lambda functions. They are a C++11 feature already supported by several compilers, such as Intel C++ 12.0, GCC 4.5, and Microsoft Visual Studio 2010. They are unnamed closures (i.e. function objects that can be constructed and managed like data) that allow functions to be syntactically defined where and when needed. When lambda functions are built, they can capture the state of non-local variables named in the wrapped code by value or by reference. This implicitly may require the use of dynamic memory allocation.

The skeleton implementing the PARFOR is a *farm-with-feedback* whose structure is sketched in Fig. 1 (left). Arrows denote communication channels implemented using lock-free Single-Producer Single-Consumer queues [11]. The *forSched* thread implements the task scheduling policy. A task is simply a pair of 2 long integers $< start, stop >$ defining a chunk of loop iterations. Worker threads (W_i) receive tasks "upon request", that is, except for the very first(s) task(s), the scheduler sends a new task to the worker only if completion of the previous task assigned to the worker has been sent back through the feedback channel. This simple policy ensures good work load balancing among worker threads without the need of implementing more complex and costly techniques.

The scheduling policy currently implemented, works as follow: loop iterations are divided into equal pieces of size *chunk*; contiguous chunks are logically assigned to the same thread trying to equalise as much as possible the number of chunks for each thread in the pool; the *forSched* sends chunks to workers upon request; during the computation of the for loop, if the *forSched* does not have any chunk of iterations to schedule to the requesting thread, it tries to "steal" a chunk (if available) from another thread. To implement this simple policy, a *task_table* containing the number of currently available tasks and the minimum and maximum iteration indexes yet to be executed is maintained in the *forSched* as a private data structure (see Fig. 1 right-end side).

The main objective of the scheduling policy proposed, is to try to obtain a good trade-off between *workload balancing* and *chunk-to-thread affinity* assignment. Maintaining as much as possible loop iterations affinity is an important performance factor on shared-cache SPMs to increase the probability to find the data needed for the computation in one of the cache level hierarchies, thus reducing somehow the communication overhead incurred by addressing non-local data on SMP NUMA platforms. The chunk-to-thread affinity assignment policy, which implements the idea proposed in the early '90s of *loop affinity scheduling* on shared memory multiprocessors [12], is implemented totally in the *forSched* using a *task_table* having one entry for each worker thread.

In the PARFOR implementation, no shared data structure among threads is maintained in order to avoid locking over-

[2]The PARFORREDUCE can be easily extended to support multiple reduction variables and any associative and commutative user function.

heads. The main drawback of using a centralised thread for iterations scheduling is that it may become a bottleneck when an high number of worker threads is used. Although this is not a big issue in principle because multiple scheduler threads can be used together (for example organised in a tree topology – exchanging latency with bandwidth), we will see in the experimental section that the *forSched* is not a bottleneck for current state-of-the-art multi/many-core.

The pseudo-code executed by the generic worker *W* and by the *forSched* threads may be stated as follows:

```
// worker pseudo-code
void *svc(void *t) { /* t is a task coming from the forSched */
  auto task = (task_t*)t;
  res = F(task->start,task->end, threadid); /* calling lambda */
  return t; /* returns the task to the forSched thread */
}

// forSched pseudo-code
void* svc(void *t) { /* t is the result task coming from workers */
  if (first_time) { /* t=nullPtr at the very beginning */
    for(int i=0;i<nw;++i) { /* nw is the number of workers */
      if (get_task(i, &task)) { /* gets a new task for worker i */
        send_to(i, task); /* sends to worker i the new task */
        update_task_table(i); /* updates i-entry in the table */
      } else
        send_eos_to(i); /* terminates worker i */
    return GO_ON; /* returns and gets a new task from workers */
  }
  if (--totaltasks == 0) return NULL; /* end of computation */
  wid = get_sender(); /* getting the id of the sender worker */
  id = wid;
  for(int cnt=0;cnt<nw;++cnt) { /* trying to get a new task */
    if (get_task(id, &task)) {
      send_to(wid); /* sends the task to the worker with id wid */
      update_task_table(id);
      return GO_ON;
    }
    /* no tasks available, try to steal a task fron another wid */
    id = (id+1) mod nw;
  }
  if (!pending_task[wid]) send_eos_to(wid); /* terminates wid */
  return GO_ON;
}
```

IV. Experiments

To evaluate the FastFlow PARFOR implementation we performed a set of experiments using 2 simple synthetic micro-benchmarks and 3 benchmarks: standard dense matrix multiplication algorithm, inner product computation and the iterative Jacobi computation. We compared our implementation against OpenMP 3.1 [4] (using Intel icc 13.0.1 and gcc 4.8.1) and Intel TBB (from Intel Composer XE 2013 suite). For both compilers we used the optimisation flag -O3. All tests have been repeated 5 times then the the average value considered.

As target architectures we considered the many-core Intel Xeon Phi 5110P (hereinafter Intel Phi) featuring 60 cores running at 1056 Mhz interconnected by a bi-directional ring bus. It is a SMP computing node connected to the host domain through a PCI Express (PCIe) bus. It runs an embedded Linux x86_64 OS that provides basic functionalities such as process/thread creation, scheduling and memory management. Intel Phi cores run independently of each other, having support for 4-way HW multi-threading being able to execute 4 threads by interleaving instructions. Each core has a 32 KB L1 data cache and a 512 KB L2 cache and 512 bit-wide vector unit used to execute single/double precision SIMD instructions. The total amount of memory is 8 GB (GDDR5) comprising 8 memory controllers.

The second architecture we considered is a standard dual-socket NUMA Intel multi-core Xeon E5-2630 Nehalem micro-architecture (hereinafter Intel Nehalem) running at 2.30GHz featuring 12 cores (6+6) each one 2-way Hyperthreading. Each core has 32KB private L1, 257KB private L2 and 16MB shared L3. The operating system is Linux 2.6.32 x86_64 shipped with Red Hat Enterprise 6.3.

On the Intel Phi platform, all OpenMP tests have been executed enabling thread affinity using the environment variable: KMP_AFFINITY="granularity=fine,scatter". In the following we use the notation "*OMP sched*" to denote the OpenMP version in which *sched* is the scheduling policy selected using the OMP_SCHEDULE environment variable. On the Intel Nehalem platform, OMP_PROC_BIND has been set to true for the 2 micro-benchmark and in the inner product example.

For TBB, on both platforms, we found that, for the tests considered, the best result is obtained by using the *affinity_partitioner*, so we used the notation *TBB-ap* for TBB runs.

For the FastFlow PARFOR, we use the notation "*FF chunk*" where *chunk* is the chunk size used for the considered test. All FastFlow tests have been compiled on both platforms with the flag "-DNO_DEFAULT_MAPPING" except for the 2 micro-benchmarks tests on the Intel Nehalem where the default thread pinning policy was enabled.

As a performance metric, together with the overall execution time, we used also the speedup, measured as $S(n, sz) = Seq(sz)/T(n, sz)$ where $Seq(sz)$ is the sequential execution time and $T(n, sz)$ is the parallel execution time using n worker threads and keeping fixed the problem size, sz.

A. Micro-benchmark tests

Micro-benchmarks mimic a real ParallelFor in which a synthetic computation is performed on each loop iteration. The objective is to evaluate the basic run-time overhead associated with iterations scheduling. We devised two simple micro-benchmarks, each one having one single ParallelFor:
parfor1, in which the number of iterations N is kept fixed (N=10×10^5) and during each iteration a nested loop with a variable number of iterations M, uniformly distributed in the half-open range $[0, 1000)$, is executed; *parfor2*, in which the number of iterations per thread is constant (10×10^4 iterations), during each iteration only a simple assignment of the loop index to the corresponding array element is executed. The pseudo-code of the 2 micro-benchmarks is reported in the following:

```
parfor1:
  for(int i=0;i<N;++i)
    for( volatile int j=0;j<M;j++);

parfor2:
  for(int i=0;i<(K*nth);++i)
    V[i]=i;
```

Figure 2 to 4 sketch the execution time varying the number of worker threads, obtained by running the 2 benchmarks on the Intel Phi platform (Fig. 2 is related to the Intel Phi, while in Fig 3 and Fig. 4 are related to the Intel Nehalem using the Intel icc and the GNU g++ compilers, respectively).

For these tests we consider a fixed user-defined grain size equal to 1000 loop iterations, and for OpenMP the scheduling

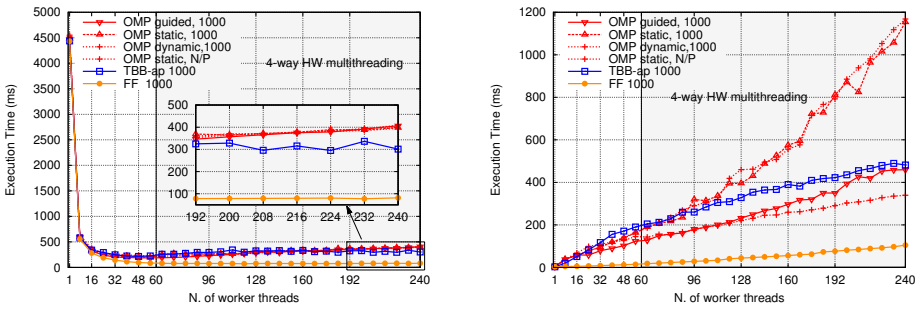

Fig. 2: *parfor*/*parfor2* (left/right) benchmarks execution on Intel Phi many-core.Benchmarks compiled with Intel icc 13.0.1

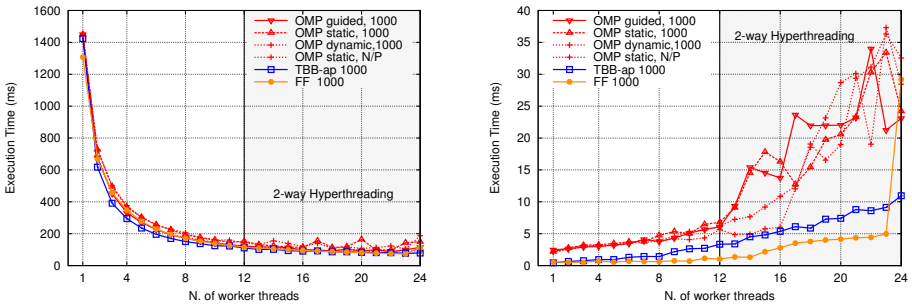

Fig. 3: *parfor*/*parfor2* (left/right) benchmarks execution on Intel Nehalem. Benchmarks compiled with Intel icc 13.0.1.

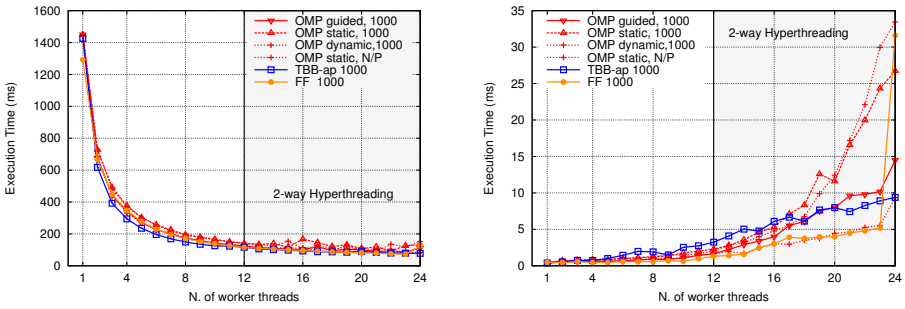

Fig. 4: *parfor*/*parfor2* (left/right) benchmarks execution on Intel Nehalem. Benchmarks compiled with GNU gcc 4.8.1.

policy considered are: *static*, *guided* and *dynamic*. As a reference value, we also tested the OpenMP static scheduling case in which the grain size is $N/\#threads$ (this case is reported in the figures as N/P). In almost all cases, the FastFlow PARFOR shows a lower run-time overhead in particular when an high number of worker threads is used. The execution time increment of the FastFlow PARFOR when all the available cores/contexts are used, more evident in Fig. 3 and in Fig. 4 for the *parfor2* benchmark, is mainly due to the non-blocking run-time support of the framework. In fact, the total amount of threads implementing the PARFOR is $\#worker\_thread + 1$ and when $\#worker\_threads = \#cores$ the extra thread used (which performs busy-waiting during synchronisation), introduces non negligible overhead especially in fine grain computations.

B. Standard matrix multiplication

The first application benchmark is a kernel computing the standard dense matrix multiplication algorithm ($C = A \times B$) for square $N \times N$ matrices storing double precision elements. The pseudo code of the algorithm is the following:

```
void matmul(double *A,double *B, double *C) {
  for(long i=0;i<N;++i)
    for(long j=0;j<N;++j)
      for(long k=0;k<N;++k)
        C[i*N+j] += A[i*N+k]*B[k*N+j];
}
```

For this test, we selected both relatively small matrices $N = 512$, so that the different run-time support overhead becomes more evident due to the finer grain computation, and also bigger matrices $N = 2048$, in order to analyse the speedup on the considered platforms. The Intel icc compiler is able to auto vectorize the innermost loop simply inserting the `#pragma ivdep` just before the for loop.

Fig. 5 and Fig. 6 report the execution time and the speedup obtained on the Intel Phi and Nehalem, respectively.

In the OpenMP tests we used the *auto* scheduling policy because it allows to obtain the best performance (we tried many combinations between *static*, *dynamic*, *guided* and grain

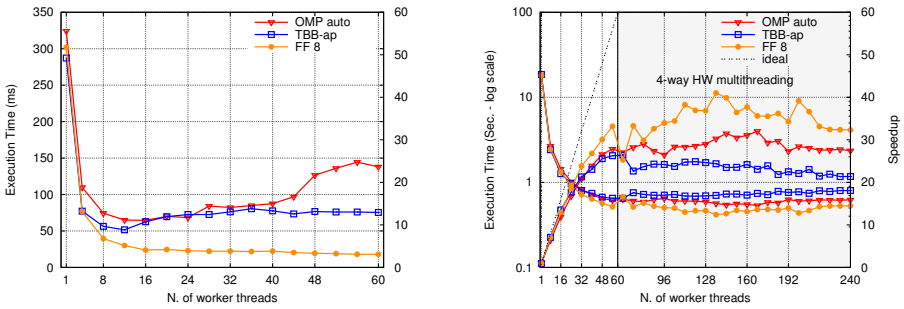

Fig. 5: Matrix Multiplication performance on the Intel Phi. Matrices size: $N = 512$ (left) and $N = 2048$ (right).

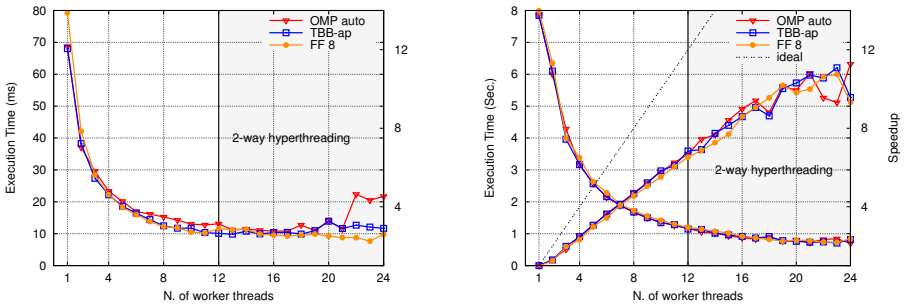

Fig. 6: Matrix Multiplication performance on the Intel Nehalem. Matrices size: $N = 512$ (left) and $N = 2048$ (right).

size). For the FastFlow PARFOR we used a chunk size of 8 for both platforms.

For small matrices the FastFlow PARFOR obtains a significant performance improvement with respect to both OpenMP and TBB on the Intel Phi platform and comparable performance results on the Intel Nehalem multi-core. For the bigger matrix, the maximum speedup on the Intel Phi is 41 using 136 worker threads corresponding to an overall execution time of 417ms. The same test executed on the Intel Nehalem, obtains a minimum execution time of 730ms using 23 threads and a speedup of 10.8.

C. Inner product computation

The inner or "dot" product is a simple example of a map pattern combined with a reduction operation. The map is the initial pair wise multiplication of vector elements, and the reduction is the summation of the results of that multiplication. More formally, given two arrays A and B each with n elements, the dot product $A \times B$ is the resulting scalar given by $\sum_{i=0}^{n-1} A[i] \times B[i]$. In our test, we considered the possibility to repeat a number of times ($NTIMES$) the reduction operation to study the behaviour of such situations, so the pseudo code of the benchmark tested is the following:

```
void dotprod( dounbe *A, double *B)
  init_vector (A,B); /* init data */
  double sum = 5.0;
  for(int z=0;z<NTIMES;++z) {
    for(long i=0;i<N;++i)
      sum += A[i]*B[i];
  }
}
```

In OpenMP the inner for loop has been parallelised using the following pragma:

```
#pragma omp parallel for default(shared) schedule(runtime)
                reduction(+:sum) num_threads(nworkers)
```

The TBB code of the parallel reduce using the standard template library features and the C++11 lambda closure is:

```
sum+=parallel_reduce(blocked_range<long>(0,N),double(0),
    [=] (tbb::blocked_range<long> &r, double in) {
      return inner_product(A+r.begin(),A+r.end(),B+r.begin(),
                           in,plus<double>(), multiplies<double>());
    },plus<double>(), affinity_partitioner ());
```

finally the FastFlow code is:

```
FF_PARFORREDUCE_START(dp,sum,0.0,i,0,N,1, chunksize, nworkers){
  sum += A[i]*B[i];
} FF_PARFORREDUCE_STOP(dp, sum, +);
```

In terms of conciseness and expressivity OpenMP offers the best solution among the three, on the other hand, both in TBB and in FastFlow the parallel code can be written with just few lines of code, so the programming effort required is minimal.

Figure 7 shows the execution time obtained from the three versions of the *dotprod* code on the Intel Phi platform when $NTIMES = 1$ and the chunk size is equal to N/P. For this test, the same scheduling policy and the same chunk size (in case of PARFOR) are used in both loops. We consider both the case in which the internal loop executed by each worker thread is automatically vectorised by the compiler (Fig. 7 left hand side) and the case in which the auto-vectorisation has been manually disabled (Fig. 7 right hand side). In both cases, the best performance is obtained using the FastFlow PARFOR pattern. It is interesting to note that, with auto-vectorisation disabled, the PARFOR is much slower than the OpenMP and TBB version up to 32 worker threads. But thanks to the high

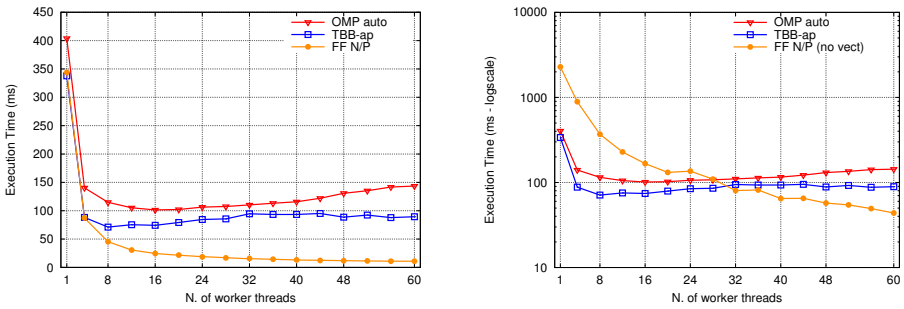

Fig. 7: *dotprod* (left) and *dotprod* without loop vectorization for FF (right). Intel Phi plaform. $NTIMES = 1$, $N = 100 \times 10^6$

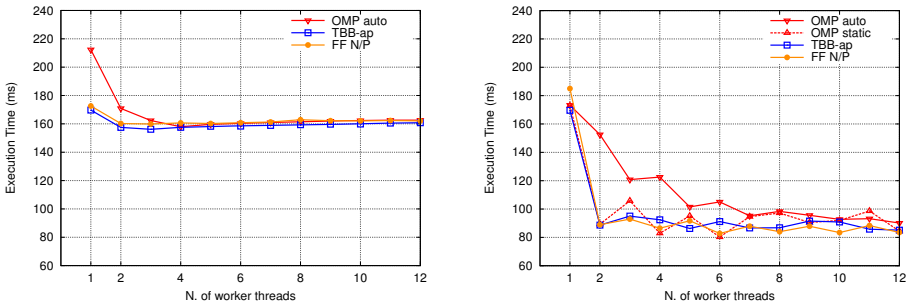

Fig. 8: *dotprod* (left) and *dotprod* with parallel arrays initialisation (right) on Intel Nehalem. $NTIMES = 10$, $N = 10 \times 10^6$

number of cores of the Intel Phi and to the good scalability of the PARFOR run-time, the FastFlow version is able to obtain a lower execution time (by using 60 cores the execution time is almost four times the execution time obtained with the auto-vectorisation enabled: $43.7ms$ vs $10.8ms$).

Figure. 8 sketches the execution time of the the inner product application on the Intel Nehalem platform with $NTIMES = 10$. Here the performance are almost the same. We then measured the impact of parallel initialisation of the A and B vectors on the parallel reduce loop. Initialising the 2 vectors in parallel allows to allocate portions of A and B in the 2 NUMA nodes of the platform, thus allowing the subsequent parallel reduce loop to exploit more memory bandwidth. This is possible because the standard *malloc* call just reserves the virtual address space but does not actually "touch" the physical memory. When different threads running on the cores of the 2 NUMA nodes initialise the two vectors, the memory pages are allocated on the NUMA node on which the thread that first "touched" the memory page resides. The results obtained are shown in Fig. 8 (right) where we can see that all 3 frameworks are able to exploit the "first-touch" NUMA policy' performance boost. However, for these tests, FastFlow and TBB obtain slightly better performance than OpenMP.

D. 2-D Jacobi iterative algorithm

Here we consider the program to solve a finite difference discretisation of Helmholtz equation using the Jacobi iterative method. This is a classical data-parallel 5-points stencil computation on a grid of $m \times n$ points. The source code of this test is part of the OpenMP Source Code Repository[3] (*c_jacobi01.c*). We modified only minor things in order to compile the code with the C++ compiler. The pseudo-code of the algorithm is the following one:

```
void jacobi (...) {
  while(k<=maxit && error > tol) {
    /* 1st par. loop, copy new solution into old array */
    for(int j=0;j<m;j++)
      for(int i=0;i<n;++i) uold[i+m*j]=u[i+m*j];
    /* 2st par. loop+reduction, computes the stencil and residual */
    for(int j=1;j<(m-1);++j)
      for(int i=1;i<(n-1);++i) {
        resid = compute_resid(f,i,j, uold,ax,ay);
        /* updates solution */
        u[i + m*j] = uold[i + m*j] - omega * resid;
        /* accumulates residual error */
        error =error + resid*resid;
      }
    error = sqrt(error) / (n*m); k++;
  }
```

The 2 for-j loops in the pseudo-code above, can be both computed in parallel at each iteration of the external while loop. Between the 2 loops there is an implicit barrier.

Figure 9 shows the performance obtained when a large grid of 10000×10000 double precision elements is considered on the 2 platforms. In this test, the FastFlow PARFOR (using a chunk size of 10 iterations for both loops) is able to obtain better performance than the OpenMP implementation starting from 48 worker threads, reaching a maximum speedup of 46.8 using 158 workers and a minimum execution time of 297.8ms. On the Intel Nehalem the performance obtained by the 3 implementations is almost the same. The minimum completion time for the FastFlow PARFOR in this case is 437ms using 6 worker threads.

Then, we tested the same code using a relatively small grid of 2000×2000 elements. The lower overhead of the FastFlow

[3] http://sourceforge.net/projects/ompscr/

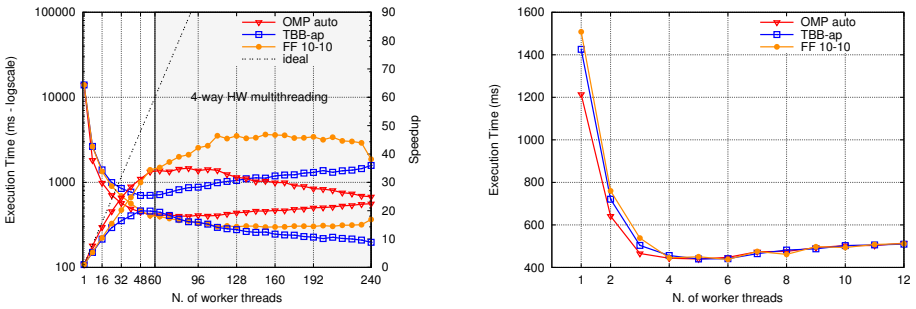

Fig. 9: *jacobi* application performance $m \times n = 10000 \times 10000$. Intel Phi platform (left). Intel Nehalem multi-core (right).

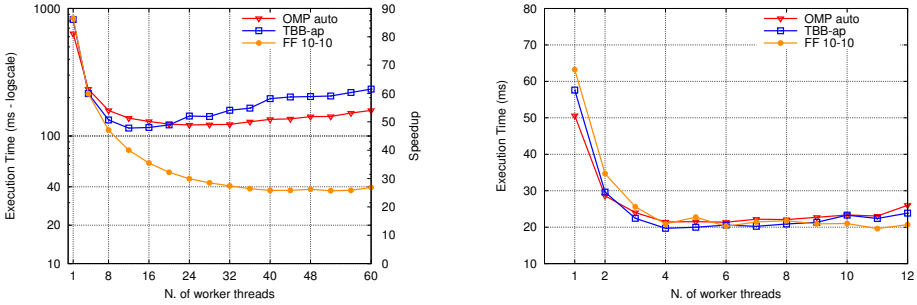

Fig. 10: *jacobi* application performance $m \times n = 2000 \times 2000$.Intel Phi platform (left). Intel Nehalem multi-core (right).

PARFOR pattern allows to obtain the minimum completion time both on the Intel Phi and the Intel Nehalem. This test confirms the results obtained in the *matmul* tests: when fine grain computations are considered the FastFlow PARFOR is able to obtain better performance than OpenMP and TBB versions on the 2 platforms considered, instead for medium/coarse grain computations the three parallel implementations obtain comparable performance results.

V. Conclusions

This paper presents the ParallelFor skeleton implementation provided within the FastFlow parallel framework. The proposed skeleton allows to parallelise independent loops with a moderate programming effort similar to that required by well-known parallel frameworks such as OpenMP and Intel TBB.

A simple yet effective dynamic tasks scheduling policy has been studied and implemented. By using a set of benchmarks, we demonstrated that the ParallelFor implementation proposed is able to obtain comparable or even better performance results with respect to those achieved parallelising the same code using OpenMP and Intel TBB. All tests have been executed on two different platforms: a Intel Xeon Phi many-core and a Intel Nehalem 12-core NUMA multi-core.

As future work, we are currently planning to extend the applicability of the current ParallelFor skeleton implementation and to study dynamic loop iterations offloading to general purpose many-core and GPUs.

References

[1] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Comput.*, vol. 30, no. 3, pp. 389–406, Mar. 2004.

[2] L. Dagum and R. Menon, "OpenMp: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating code on multi-cores with FastFlow," in *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, ser. LNCS, vol. 6853, aug 2011, pp. 170–181.

[4] "OpenMP Architecture Review Board: OpenMP Application Program Interface. Version 3.1," July 2001, http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[5] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic OpenMP loop scheduling: a combined compiler and runtime approach," in *Proc. of the 8th intern. conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. 2012, pp. 88–101.

[6] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *Proc. of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 75–84.

[7] M. Curtis-Maury, X. Ding, C. D. Antonopoulos, and D. S. Nikolopoulos, "An evaluation of OpenMP on current and emerging multithreaded/multicore processors," in *Proc. of the 2005 and 2006 intern. conference on OpenMP shared memory parallel programming*, ser. IWOMP'05/IWOMP'06. 2008, pp. 133–144.

[8] "Intel Threading Building Blocks, project site," 2013, http://threadingbuildingblocks.org.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Comput.*, vol. 37, no. 1, pp. 55–69, 1996.

[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, 2013, ch. 13.

[11] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, ser. LNCS, vol. 7484. aug 2012, pp. 662–673.

[12] E. P. Markatos and T. J. Leblanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 104–113, 1994.