CHAPTER 13

FastFlow: HIGH-LEVEL AND EFFICIENT STREAMING ON MULTI-CORE

M. Aldinucci[1], M. Danelutto[2], P. Kilpatrick[3], M. Torquati[2]

[1]Dept. of Computer Science, University of Torino, [2]Dept. of Computer Science, University of Pisa, [3]Dept. of Computer Science, Queen's University of Belfast

Computer hardware manufacturers have moved decisively to multi-core and are currently experimenting with increasingly advanced many-core architectures. In the long term, writing efficient, portable and correct parallel programs targeting multi- and many-core architectures must become no more challenging than writing the same programs for sequential computers. To date, however, most applications running on multi-core machines do not exploit fully the potential of these architectures.

This situation is in part due to the scarcity of good high-level programming tools suitable for multi/many-core architectures, and in part to the fact that multi-core programming is still viewed as a kind of exotic branch of high-performance computing (HPC) rather than being perceived as the *de facto* standard programming practice for the masses.

Some efforts have been made to provide programmers with tools suitable for mapping data parallel computations onto both multi-cores and GPUs–the

Programming Multi-core and Many-core Computing Systems, edition. By author

most popular many-core currently available. Tools have also been developed to support stream parallel computations [34, 31] as stream parallelism *de facto* represents a pattern characteristic of a large class of (potentially) parallel applications. Two major issues with these programming environments and tools relate to programmability and efficiency. Programmability is often impaired by the modest level of abstraction provided to the programmer. Efficiency more generally suffers from the peculiarities related to effective exploitation of the memory hierarchy.

As a consequence, two distinct but synergistic needs exist: on the one hand, increasingly efficient mechanisms supporting correct concurrent access to shared memory data structures are needed; on the other hand there is a need for higher level programming environments capable of hiding the difficulties related to the correct and efficient use of shared memory objects by raising the level of abstraction provided to application programmers.

To address these needs we introduce and discuss FastFlow, a programming framework specifically targeting cache-coherent shared-memory multi-cores. FastFlow is implemented as a stack of C++ template libraries[1]. The lowest layer of FastFlow provides very efficient lock-free (and memory fence free) synchronization base mechanisms. The middle layer provides distinctive communication mechanisms supporting both single producer-multiple consumer and multiple producer-single consumer communications. These mechanisms support the implementation of graphs modelling various kinds of parallel/concurrent activities. Finally, the top layer provides, as programming primitives, typical streaming patterns exploiting the fast communication/synchronizations provided by the lower layers and supporting efficient implementation of a variety of parallel applications, including but not limited to classical streaming applications.

In our opinion the programming abstractions provided by the top layer of FastFlow represent a suitable programming model for application programmers. The efficient implementation of these programming abstractions in terms of the lower layers of the FastFlow stack also guarantees efficiency. Moreover, the possibility of accessing and programming directly the lower layers of the FastFlow stack to implement and support those applications not directly supported by the FastFlow high-level abstractions provides all the processing power needed to efficiently implement most existing parallel applications.

In this Chapter we adopt a tutorial style: first we outline FastFlow design and then show sample use of the FastFlow programming environment together with performance results achieved on various *state-of-the-art* multi-core architectures. Finally, a related work section concludes the Chapter.

[1] FastFlow is distributed under LGPLv3. It can be downloaded from SourceForge at `http://sourceforge.net/projects/mc-fastflow/`.

13.1 FastFlow PRINCIPLES

The FastFlow framework has been designed according to four foundational principles: *layered design* (to support incremental design and local optimizations); *efficiency in base mechanisms* (as a base for efficiency of the whole framework); support for *stream parallelism* (intended as a viable solution for implementing classical stream parallel applications and also data parallel, recursive and Divide&Conquer applications); and a programming model based on *design pattern/algorithmic skeleton* concepts (to improve the abstraction level provided to the programmer).

Layered design. FastFlow is conceptually designed as a stack of layers that progressively abstract the shared memory parallelism at the level of cores up to the definition of useful programming constructs supporting structured parallel programming on cache-coherent shared memory multi- and many-core architectures (see Fig. 13.1). These architectures include commodity, homogeneous, multi-core systems such as Intel core, AMD K10, etc. The core of the FastFlow framework (i.e. *run-time support* tier) provides an efficient implementation of Single-Producer-Single-Consumer (SPSC) FIFO queues. The next tier up extends one-to-one queues (SPSC) to one-to-many (SPMC), many-to-one (MPSC), and many-to-many (MPMC) synchronizations and data flows, which are implemented using only SPSC queues and arbiter threads, thus providing lock-free and wait-free arbitrary data-flow graphs (*arbitrary streaming networks*). These networks exhibit very low synchronization overhead because they require few or no memory barriers, and thus few cache invalidations. The upper layer, i.e. *high-level programming*, provides a programming framework based on parallel patterns (see Sec. 13.1). The FastFlow pattern set can be further extended by building new C++ templates. Programs written using the abstractions provided by the FastFlow layered design may be seamlessly ported across the full range of architectures supported. The runtime tier has specific conditional compilation parts targeting the different shared memory and cache architectures in the various target architectures. Extra fine-tuning possibilities will be provided in future FastFlow releases, in particular to allow users to allocate memory in one of the "banks" sported by the target architecture. This, along with the possibility offered to pin a thread to a specific core will provide the user full locality control.

Efficiency of base mechanisms. FastFlow SPSC queues represent the base mechanisms in the FastFlow framework. Their implementations are lock-free and wait-free [18]. They do not use interlocked operations [15]. Also, they do not make use of any memory barrier for Total Store Order processors (e.g. Intel core) and use a single memory write barrier (in the push operation) for processors supporting weaker memory consistency models (full details on FastFlow SPSC can be found in [35]). The SPSC queue is primarily used as a synchronization mechanism for memory pointers in a consumer-producer

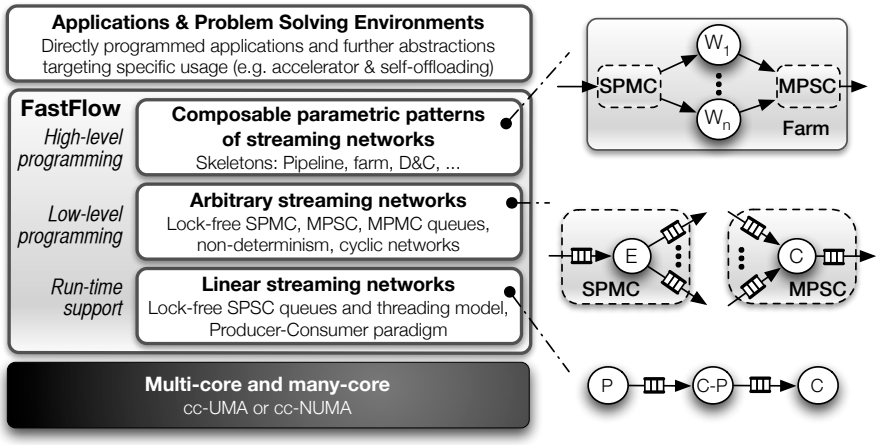


Figure 13.1 FastFlow layered architecture with abstraction examples at the different layers of the stack.

fashion. SPSC FIFO queues can be effectively used to build networks of communicating threads which behave in a dataflow fashion. The formal underpinning of these networks dates back to Kahn Process Networks (KPNs) [20] and Dataflow Process Networks [21]. Table 13.1 shows the average latencies involved in the use of the SPSC queues on different configurations of producers and consumers on state-of-the-art Intel and AMD multi-core architectures.

Stream parallelism. We chose to support only stream parallelism in our library for two basic reasons: i) supporting just one kind of parallelism keeps the FastFlow implementation simple and maintainable, and ii) stream parallel patterns, as designed in FastFlow, allow different other parallelism forms to be implemented (see below), including simple data parallelism, parallelism in recursive calls and Divide&Conquer. *Stream parallelism* is a programming paradigm supporting the parallel execution of a stream of tasks by using a series of *sequential* or *parallel* stages. A stream program can be naturally represented as a graph of independent *stages* (kernels or filters) that communicate explicitly over data channels. Conceptually, a streaming computation represents a sequence of transformations on the data streams in the program. Each stage of the graph reads one or more tasks from the input stream, applies some computation, and writes one or more output tasks to the output stream. Parallelism is achieved by running each stage of the graph simultaneously on *subsequent* or *independent* data elements. Local state may be either maintained in each stage or distributed (replicated or scattered) along streams.

Streams processed in a stream parallel application are usually generated (input streams) and consumed (output streams) externally to the application itself. However, streams to be processed may also be generated *internally* to the application. For example, an embarrassingly data parallel application may be implemented as a pipeline with three stages: the first generates a stream of tasks, each representing one of the data parallel tasks that can be independently processed to compute a subset of the application results; the second processes in parallel this input stream of tasks, producing an output stream of results; and the last stage processes the output stream to rebuild the final (non-stream) result. We refer to the first kind of streams–those produced/consumed outside the application–as *exo-streams* and the second–those produced/consumed internally–as *endo-streams*.

Parallel design patterns (Algorithmic skeletons). Attempts to reduce the programming effort by raising the level of abstraction date back at least three decades. Notable results have been achieved by the *skeletal* approach [12, 13] (a.k.a. *pattern-based* parallel programming). This approach appears to be becoming increasingly popular after being revamped by several successful parallel programming frameworks [36, 14, 34, 8].

Algorithmic skeletons capture common parallel programming paradigms (e.g. MapReduce, ForAll, Divide&Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined functional and extra-functional semantics [2]. Some of these skeleton frameworks explicitly include stream parallelism as a major source of concurrency exploitation [36, 2, 34, 19]: rather than allowing programmers to connect stages into arbitrary graphs, basic forms of stream parallelism are provided to the programmer in high-level constructs such as *pipeline* (modeling computations in stages), *farm* (modeling parallel computation of independent data tasks), and *loop* (supporting generation of cycles in a stream graph and typically used in combination with a farm body to model Divide&Conquer computations). More recently, approaches such as those followed in algorithmic skeletons but based on *parallel design patterns* have been claimed to be suitable to support multi- and many-core programming [8, 23]. Differences between algorithmic skeletons and parallel design patterns lie mainly in the motivations leading to these two apparently distinct concepts and in the research environments where they were developed: the parallel programming community for algorithmic skeletons and the software engineering community for parallel design patterns.

In FastFlow we chose to adopt an algorithmic skeleton/parallel design pattern based approach to address the problems outlined in the introduction, and we restricted the kind and the number of skeletons implemented to keep the size of the implementation manageable while providing a useful skeleton set. This choice allows us to provide full support for an important class of applications, namely *streaming applications* [7, 31].

13.2 FastFlow μ-TUTORIAL

The FastFlow parallel programming framework may be used in at least two distinct ways. A first, classic usage scenario is that related to development "from scratch" of brand new parallel applications. In this case, the application programmer logically follows a workflow containing the following steps:

STEP 1 choose the most suitable skeleton nesting that models the parallelism paradigm that can be exploited in the given application;

STEP 2 provide the parameters needed to correctly instantiate the skeleton(s), including the sequential portions of code modeling the sequential workers/stages of the skeleton(s); and

STEP 3 compile and run the resulting application code, consider the results and possibly go back to step 1 to refine the skeleton structure if it becomes apparent that there is a better combination of skeletons modeling the parallelism exploitation paradigm in mind.

The workflow just mentioned can be used also to parallelize existing applications. In this case, rather than choosing the most suitable skeleton nesting for the *whole* application, the programmer will analyze the application, determine which kernels are worth parallelizing and finally enter the three step process above, with step one being performed only on targeted portions of the code. As result, the sequential flow of control of a given kernel will be substituted by the parallel flow of control expressed by the skeleton nesting.

A second scenario, relating to the use of *software accelerators*, is particularly targeted to low-effort parallelization of existing applications. In this case programmers identify independent tasks within the application. Then they choose a representation for the single task, declare a FastFlow accelerator–e.g. a farm accepting a stream of tasks to be computed–and use the accelerator to offload the computation of these tasks, much in the sense of OpenMP tasks being executed by the thread pool allocated with the scoped `#pragma parallel` directive. This scenario is distinct from the first in that the application programmer using FastFlow in this way does not necessarily need any knowledge of the skeleton framework implemented in FastFlow. Tasks to be computed are simply sent to a generic "parallel engine" computing some user-supplied code. Once the tasks have been submitted, the program can wait for completion of their computation, while possibly performing other different tasks needed to complete application execution.

13.2.1 Writing parallel applications "from scratch"

When designing and implementing new parallel applications using FastFlow, programmers instantiate patterns provided by FastFlow to adapt them to the specific needs of the application at hand. In this section, we demonstrate how the principal FastFlow patterns may be used in a parallel application.

```
#include <ff/pipeline.hpp>
using namespace ff;
int main() {
    ff_pipeline pipe;
    for(int i=0;i<nStages;++i)
        pipe.add_stage(new Stage);
    if (pipe.run_and_wait_end()<0)
        return -1;
    return 0;
}
class Stage: public ff_node {
    int svc_init () {
        printf("Stage %d\n",get_my_id());
        return 0;
    }
    void * svc(void * task) {
        if (ff_node :: get_my_id()==0)
            for(long i=0;i<ntasks;++i)
                ff_send_out(i);
        else printf("Task=%d\n",(long)task);
        return task;
    }
};
```

Figure 13.2 Hello world pipeline.

13.2.1.1 Pipeline A very simple FastFlow pipeline code is sketched in Fig. 13.2. A FastFlow pipeline object is declared in line 4. In line 5 and 6 `nStages` objects of type `Stage` are added to the pipeline. The order of the stages in the pipeline chain is given by the insertion order in the `pipe` object (line 6). The generic `Stage` is defined from line 11 to line 23. The `Stage` class is derived from the `ff_node` base class, which defines 3 basic methods, two optional, `svc_init` and `svc_end` and one mandatory `svc` (pure virtual method). The `svn_init` method is called once at node initialization, while the `svn_end` method is called once when the end-of-stream (EOS) is received in input or when the `svc` method returns `NULL`. The `svc` method is called each time an input task is ready to be processed. In the example, the `svc_init` method just prints a welcome message and returns. The `svc` method is called as soon as an input task is present and prints a message and returns the task which will be sent out by the FastFlow run time to the next stage (if present). For the first stage of the pipeline the `svc` method is called by the FastFlow runtime with a `NULL` task parameter. The first node (the one with id 0 in line 17) generates the stream sending out each task (in this simple case just one `long`) by using FastFlow's runtime method `ff_send_out` (line 19). The `ff_send_out` allows for queueing tasks without returning from the `svc` method. The pipeline can be started synchronously as in the example (line 7) or asynchronously by invoking the method `run` without waiting for the completion, thus allowing overlap with other work. It is worth noting that the `ff_pipeline` class type is a base class of `ff_node` type, so a pipeline object can be used where an `ff_node` object has to be used.

13.2.1.2 Farm A farm paradigm can be seen as a two or three stage pipeline, the stages being a `ff_node` called the *emitter*, a pool of `ff_node`s called *workers* and–optionally–a `ff_node` called the *collector*. A FastFlow farm pattern can be declared using the `ff_farm<>` template class type as in Fig. 13.3 line 4. In line 6 and 7 a vector of `nWorkers` objects of type `Worker` is created and added to the farm object (line 8). The *emitter* node, added in line 9, is used in the example code to generate the stream of tasks for the pool of *workers*. The `svc` method is called by the FastFlow runtime with a NULL task parameter (since, in this case, the *emitter* does not have any input stream)

```
#include <ff/farm.hpp>
using namespace ff;
int main() {
  ff_farm<> farm;
  std::vector<ff_node*> workers;
  for(int i=0;i<nWorkers;++i)
    workers.push_back(new Worker);
  farm.add_workers(workers);
  farm.add_emitter(new Emitter(nTasks));
  farm.add_collector(new Collector);
  if (farm.run_and_wait_end()<0)
    return -1;
  return 0;
}
struct Emitter: public ff_node {
  Emitter(int ntask):ntask(ntask){}
  int svc_init () {
    printf("Work Start\n");
  }
  void * svc(void *) {
    long task = new task_t(ntask--);
    return (void*)task;
  }
  long ntask;
};
struct Worker: public ff_node {
  void * svc(void * task) {
    // do something useful with the task
    return task;
  }
};
struct Collector: public ff_node {
  void * svc(void * task) {
    printf("Task=%d\n",(long)task);
    delete task;
    return GO_ON;
  }
  void svc_end() { printf("Done!\n");}
};
```

Figure 13.3 Hello world farm.

each time a new task has been sent out and until a `NULL` value is returned from the method. Another way to produce the stream without entering and exiting from the `svc` method each time would be to use the `ff_send_out` to generate all the tasks.

The *emitter* can also be used as sequential preprocessor if the stream is coming from outside the farm, as is the case when the stream is coming from a previous node of a pipeline chain or from an external device.

The farm skeleton must have the *emitter* node defined: if the user does not add it to the farm, the run-time support adds a default *emitter* which acts as a stream filter and schedules tasks in a round-robin fashion toward the *workers*. In contrast, the *collector* is optional. In our simple example, the *collector*, added at line 10, gathers the tasks coming from the *workers*, writes a message, and deletes the input task allocated in the *emitter*. Each time the `svc` method is called and the work completed, the *collector*, being the last stage of a three stage pipeline, returns the tag `GO_ON` task which tells the run-time support that further tasks must be awaited from the input channel and that the computation is not finished. The `GO_ON` tag can be used in any `ff_node` class. Finally, as for the pipeline, the farm base class is `ff_node`.

13.2.1.3 Farm and pipeline with feedback In the farm paradigm the *collector* can be connected with a feedback channel to the *emitter*. It is also possible to omit the *collector* by having, for each worker thread, a feedback channel toward the *emitter*. For the pipeline paradigm it is possible to link the last stage of the chain with the first one in a ring fashion. In general, several combinations and nestingnestings of farm, pipeline and feedback channels are possible without any particular limitations to build complex streaming networks. For example, it is possible to have a farm skeleton whose *workers*

are pipelines, or a farm skeleton whose *workers* are other farms, each with a feedback channel.

When a feedback channel is present in the farm paradigm, the performance may strongly depend on the scheduling policies of tasks. FastFlow offers two predefined scheduling policies: dynamic round-robin (DRR), and on-demand (OD). The DRR policy schedules a task to a worker in a round-robin fashion, skipping workers with full input queue. The OD policy is a fully dynamic scheduling, i.e., a DDR policy where each worker has an input queue of a predefined small size (typically 1 or 2 slots). Furthermore, in the farm skeleton, the *emitter* may also be used to implement user-defined scheduling policies, i.e, it is possible to add new scheduling policies tailored to the application behavior by subclassing the `ff_loadbalancer` and redefining the method `selectworker`. The new scheduling class type should be passed as template parameter to the farm object. In this way it is possible, for example, to define weighted scheduling policies by assigning weights to tasks and to schedule the tasks directly to the worker that has the lowest weight at scheduling decision time (i.e. `ff_send_out`).

13.2.2 Using FastFlow as an accelerator

FastFlow can also be used as a *software accelerator* to accelerate existing sequential code. An accelerator is defined by a skeletal composition augmented with an input and an output stream that can be, respectively, pushed and popped directly from the sequential code. Accelerating sequential code differs slightly from plain parallelization of existing code such as that sketched at the end of Sec. 13.2.1. In that case, more extensive application knowledge is needed in order to choose the most suitable parallel pattern composition for the whole application. Instead, when the accelerating methodology is used, programmers have to identify potentially concurrent *tasks* and request their execution (by explicit task offloading) onto the FastFlow skeletal composition in order to let those tasks be computed in parallel. As a consequence, the programmer has only to identify the concurrent tasks in the code and provide a suitable representation of those tasks to be submitted through the accelerator input stream. A detailed description of the FastFlow software accelerator and its usage can be found in [5].

13.3 PERFORMANCE

The FastFlow framework has been validated using a set of very simple benchmarks, starting from low-level basic mechanisms up to the simplest FastFlow patterns: farm and pipeline. Furthermore, a brief description of some significant real-world applications is reported pointing out, for each application, the kind of parallelization used.

	Same core & different contexts		Same CPU & different cores		Different CPUs	
	8-core	48-core	8-core	48-core	8-core	48-core
Average	14.29	-	11.23	19.73	9.6	20.21
Std. Deviation	2.63	-	0.29	2.23	0.1	1.9

Table 13.1 Average latency time and standard deviation (in nanoseconds) of a push/pop operation on a SPSC queue with 1024 slots for 1M insert/extractions, on the Intel 8-core 16-context and on AMD 48-core.

Two platforms are used in the evaluation: *8-core*) Intel workstation with 2 x quad-core Xeon E5520 Nehalem (16 HyperThreads) @2.26GHz; *48-core*) AMD Magny-Cours 4 x twelve-core Opteron 6174 @2.2GHz. Both run Linux x86_64.

13.3.1 Base mechanism latencies

Table 13.1 shows the results obtained when running a synthetic micro-benchmark consisting in a simple 2-stage pipeline in which the first stage pushes 1 million tasks into a FastFlow SPSC queue (of size 1024 slots) and the second stage pops tasks from the queue and checks for correct values. In the table are reported 3 distinct cases obtained by changing the physical mapping of the 2 threads corresponding to the 2 stages of the pipeline: 1) the first and second stage of the pipeline are pinned on the same physical core but on different HW contexts (only for the Intel 8-core architecture); 2) are pinned on the same CPU but on different physical cores (for the AMD 48-core architecture we pinned the two threads on 2 cores of the same die); and 3) are pinned on two cores of two distinct CPUs. On the 8-core box (Intel), FastFlow's SPSC queue takes on average 9.6-11.23 ns per queue operation with standard deviations of less than 1 nS when the threads are on distinct physical cores. Since threads mapped on different contexts of the same core share the same ALUs, the performances are a little bit worse in this case. On the 48-core box (AMD), FastFlow's SPSC queue takes on average 19.7–20.2 ns per queue operation with standard deviations around 2 ns.

It is well known that dynamic memory allocation and deallocation can be very expensive, especially for the management of very small objects. The FastFlow framework offers a lock-free dynamic memory allocator specifically optimized for the allocation/deallocation of small data structures. It is used to allocate FastFlow's tasks (which are usually small) flowing through the FastFlow network, which are frequently allocated and deallocated by different nodes. Figure 13.4 reports the execution time, on both architectures, of a very simple farm application where the *emitter* allocates 10 million tasks each of size 32bytes (4 long) and the generic worker deallocates them after a synthetic computation of 1μs. We compare the standard libc-6 allocator (glibc-2.5-

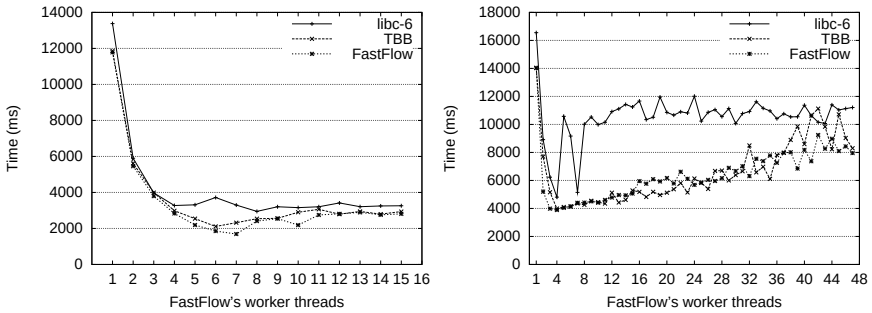


Figure 13.4 Execution time of the farm pattern with different allocators: libc vs. TBB vs. FastFlow allocator on 8-core Intel (Left) and 48-core AMD (Right) using a computational grain of 1μs.

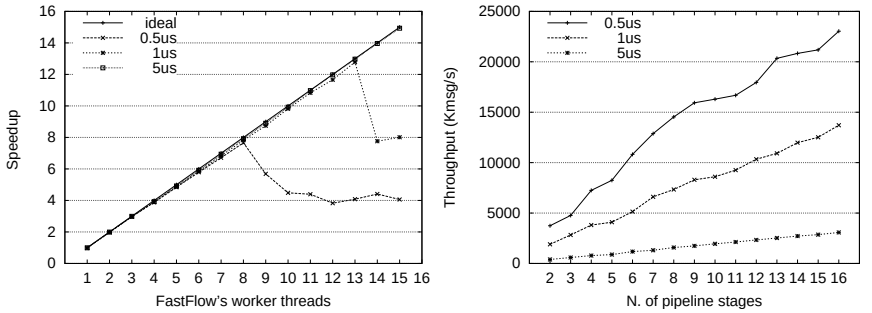


Figure 13.5 Speedup of the farm (Left) and throughput in thousands of messages per second (Kmsg/s) of the pipeline (Right) paradigms for several computational grains.

42), TBB's scalable allocator v.3.0 (30_20101215) and FastFlow's allocator. FastFlow's allocator achieves the best performance for all numbers of threads on the 8-core box (Intel), whereas on the 48-core machine, FastFlow's allocator and TBB's allocator achieve almost the same performance, much better than the standard libc-6 allocator which performs poorly on this architecture.

13.3.2 Efficiency of high-level streaming abstractions

To evaluate the overhead of the communication infrastructure for the FastFlow farm and pipeline paradigms, we developed two simple synthetic micro-benchmark tests. In the micro-benchmarks neither dynamic memory allocation nor access to global data is performed.

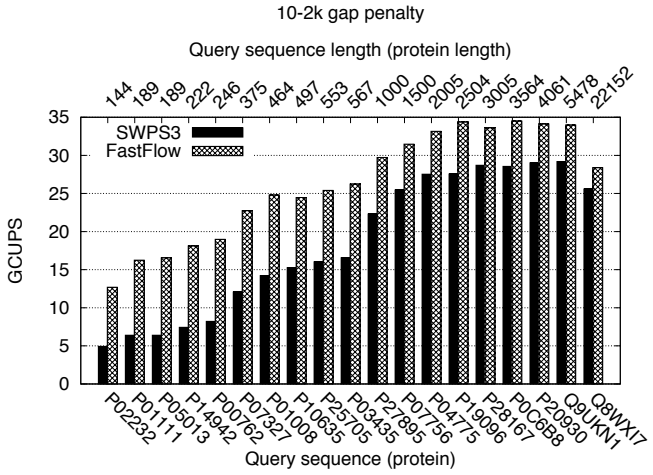


Figure 13.6 SWPS3 vs SWPS3-FF performance for $10-2k$ gap penalties evaluated on Release 57.5 of UniProtKB/Swiss-Prot.

For the farm paradigm, the stream is composed of a sequence of 10 million tasks which have a synthetic computational load associated. By varying this load it is possible to evaluate the speedup of the paradigm for different computation grains. Figure 13.5, left, shows the results obtained for three distinct computation grains: 0.5μs, 1μs and 5μs. The speedup is quite good in all cases. We have almost linear speedup starting from a computation grain of about 1μs. Larger computation grains give better results, as expected.

For the pipeline paradigm, the test consists in a set of N stages where the last stage is connected to the first, forming a ring. The first stage produces 1 million tasks in batch mode, that is, apart from the first 1024 tasks sent out at starting, for each input task received from the last stage, it produces a batch of 256 new tasks. Each task has a synthetic computational load associated, so that the throughput expressed in thousands of messages per second (Kmsg/s) for the entire pipeline can be evaluated using different computational grains. The results obtained are sketched in Fig. 13.5, right.

13.3.3 Real world applications

Several real world applications have been developed using the FastFlow framework. Here we report some of them: the Smith-Waterman biosequence alignment, the parallelization of the YaDT decision tree builder, the parallelization of a stochastic simulator, an edge-preserving denoiser filter, and an extension of the pbzip2 parallel compressor. Table 13.2 summarizes the types of FastFlow patterns and streams used in these applications.

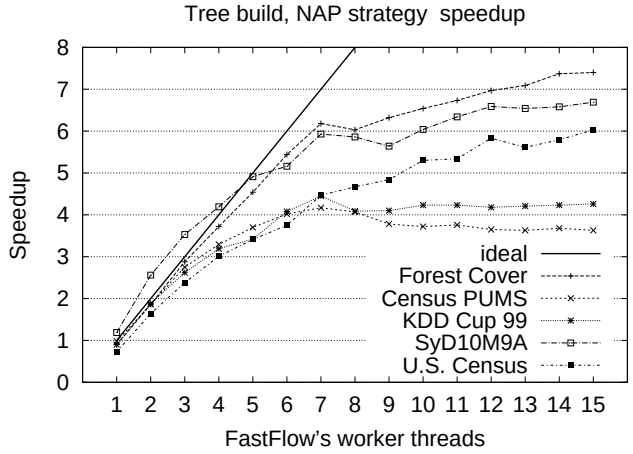
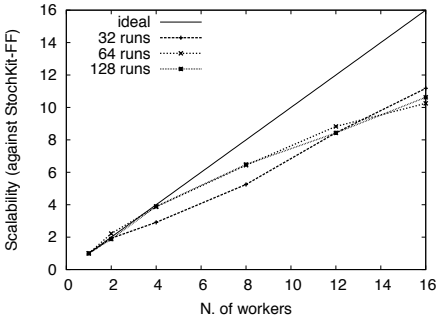


Figure 13.7 YaDT-FF speedup using several standard training sets. Superlinear speedup is due to the fact the farm emitter performs a minimal amount of work contributing to the final result, but the parallelism degree is given related to the worker threads only.

Biosequence alignment: SWPS3-FF. Biosequence similarities can be determined by computing their optimal local alignments using the Smith-Waterman algorithm [29]. SWPS3 [32] is a fast implementation of the Striped Smith-Waterman algorithm extensively optimized for Cell/BE and x86/64 CPUs with SSE2 instructions. SWPS3-FF is a porting of the original SWPS3 implementation to the FastFlow framework [4]. The pattern used in the implementation is a simple farm skeleton without the *collector* thread. The *emitter* thread reads the sequence database and produces a stream of pairs: ⟨ query-sequence, subject sequence ⟩. The query-sequence remains the same for all the subject sequences contained in the database. The generic *worker* thread computes the Smith-Waterman algorithm on the input pairs using the SSE2 instruction set in the same way as the original code and produces the resulting score. Figure 13.6 reports the performance comparison between SWPS3 and the FastFlow version of the algorithm for x86/SSE2 executed on the Intel test platform. The scoring matrix used is BLOSUM50 with 10-2k gap penalty.

As can be seen, the FastFlow implementation outperforms the original SWPS3 x86/SSE2 version for all the sequences tested. [2]

[2]The GCUPS (Giga-Cell-Updates-Per-Second) is a commonly used performance measure in bioinformatics and is calculated by multiplying the length of the query sequence by the length of the database divided by the total elapsed time.



	bzip2	pbzip2		pbzip2_ff	
	T (S)	T (S)	Sp	T (S)	Sp
Comp.	538	97	5.5	72	7.5
Dec.	126	33	3.8	21	6.0

Figure 13.8 Left: Scalability of StockKit-FF(n) against StockKit(1), where n is the number of worker threads. Right: Execution time (T) and speedup (Sp) over bzip2 in the case of a stream of 1078 files: 86% small (0 - 1 MBytes), 9% medium (1 - 10 MBytes), 4% large (10 - 50 MBytes), and 1% very large (50 - 100 MBytes). pbzip2 uses 16 threads. pbzip2_ff uses 16 threads for each accelerator. *Comp* stands for compression and *Dec* for decompression.

Decision tree builder: YaDT-FF. YaDT (Yet another Decision Tree builder) implementation [27], is a heavily optimized, efficient C++ version of Quinlan's C4.5 entropy-based algorithm [26]. It is the result of several optimizations and algorithm redesigns with respect to the original C4.5 induction algorithm, and represents an example of extreme sequential algorithmic optimization. YaDT-FF is the porting of YaDT onto general purpose multi-core architectures.

The decision tree growing strategy is a top-down breadth-first tree visit algorithm. The porting consists in the parallelization of the decision tree visit by exploiting stream parallelism, where each decision node is considered a task of the stream that generates a set of sub-tasks corresponding to the child nodes. In order to obtain additional parallelism the computation of the information gain of attributes associated with each node has also been parallelized. The overall parallelization strategy is described in detail in [6].

The pattern used is a *farm-with-feedback* skeleton which implements the D&C paradigm. Initially the computation is started by offloading the tree root node task so that the stream can be initiated. The *emitter* gets as input the root node task, produces as output the sub-tasks corresponding to the children of the node, scheduling those tasks to a number of worker threads using an application-tailored scheduling policy. The *workers* process the tasks independently and in parallel, and eventually return the resulting tasks to the *emitter*.

The speedup of YaDT-FF is shown in Fig. 13.7 for a set of reference datasets that are publicly available from the UCI KDD archive, apart from *SyD10M9A* which is synthetically generated using function 5 of the QUEST data generator.

Stochastic Simulator: StochKit-FF. StockKit [25] is an extensible stochastic simulation framework developed in the C++ language. It aims at making stochastic simulation accessible to practicing biologists and chemists, while remaining open to extension via new stochastic and multi-scale algorithms. It implements the popular Gillespie algorithm, explicit and implicit tau-leaping, and trapezoidal tau-leaping methods.

StockKit-FF extends StockKit version 1 with two main features: support for the parallel run of multiple simulations on multi-cores; and support for the on-line parallel *reduction* of simulation results, which can be performed according to one or more user-defined associative and commutative functions. StockKit-FF exploits the FastFlow basic *farm* pattern. Each farm worker receives a set of simulations and produces a stream of results that is gathered by the farm *collector* thread and reduced into a single output stream. Overall, the parallel *reduction* happens in a systolic (tree) fashion via the so-called *selective memory* data structure, i.e. a data structure supporting the on-line *reduction* of time-aligned trajectory data by way of user-defined associative functions. More details about StockKit-FF and the *selective memory* data structure can be found in [1].

As shown in Fig. 13.8, StockKit-FF exhibits good scalability when compared with the sequential (one-thread) version of StockKit.

Stream File compressor: pbzip2-FF. This application is an extension of an already parallel application: *pbzip2* [16], i.e. a parallel version of the widely used sequential *bzip2* block-sorting file compressor. It uses pthreads and achieves very good speedup on SMP machines for large files. Small files (less then 1MB) are sequentially compressed. We extend it to manage streams of small files, which can be compressed in parallel. The original pbzip2 application is structured as a farm: the generic input file is read and split into independent parts (blocks) by a splitter thread; then each block is sent to a pool of worker threads which compress the blocks. The farm is hand-coded using pthread synchronizations and is extensively hand-tuned.

The FastFlow port of pbzip2 (*pbzip2_ff*) was developed by taking the original code of the *workers* and including it in a FastFlow farm pattern. Then, a second FastFlow farm, whose *workers* execute the file compression sequentially, was added. The two farms are run as two accelerators and fed by the main thread which selectively dispatches files to the two accelerators depending on the file size. Logically the application is organized as two 2-stage pipelines.

The table in Fig. 13.8 compares the execution times of sequential bzip2, the original pbzip2 and pbzip2_ff on files of various sizes showing the improved speedup of pbzip2_ff against pbzip2. Compression and decompression performance for a single large file shows no performance penalty for pbzip2_ff against hand-tuned pbzip2.

Edge-Preserving denoiser. We also implemented an edge-preserving denoiser, a two-step filter for removing salt-and-pepper noise, which achieved good

	BioAlign	DecTreeBuild	StocSimul	Denoiser	File Compressor
Pattern(s)	Farm	Farm + Loop	Farm	Farm	Pipe(Farm)×2
Stream(s)	exo	endo	endo	endo	exo+endo
Tasks from	DB	recursive calls	sim no.	DP tasks	shell+file chunks

Table 13.2 Patterns and streams used in real world applications

performance on the 48-core platform (AMD). Sequential processing for the test images grows linearly with noise ratio: from 9 to 180 seconds with 10% to 90% noise ratio. The parallel version speeds them up to a range of 0.4 - 4 seconds (further details may be found in [5]).

13.4 RELATED WORK

Structured parallel programming models have been discussed in Sec. 13.1. FastFlow high-level patterns appear in various algorithmic skeleton frameworks, including Skandium [22], Muesli [10] and Muskel [3]. The parallel design patterns presented in [23] also include equivalents of the FastFlow high-level patterns. These programming frameworks, however, do not specifically address stream programming and so FastFlow outdoes them in terms of efficiency. Also, most of the algorithmic skeleton frameworks mentioned above and in Sec. 13.1, with the exception of Skandium, were designed originally to target cluster and networks of workstations, and multi-core support has been–in some cases, e.g. in Muesli and Muskel–only a later addition.

Stream processing is extensively discussed in the literature. Stream languages are often motivated by the application style used in image processing, networking, and other media processing domains.

StreamIt [34] is an explicitly parallel programming language based on the Synchronous Data Flow model. A program is represented as a set of filters, i.e. autonomous actors (containing Java-like code) that communicate through first-in first-out (FIFO) data channels. Filters can be assembled as a *pipeline*, possibly with a *FeedbackLoop*, or according to a *SplitJoin* data-parallel schema. S-Net [28] is a coordination language to describe the communications of asynchronous sequential components (a.k.a. boxes) written in a sequential language (e.g. C, C++, Java) through typed streams. The overall design of S-Net is geared towards facilitating the composition of components developed in isolation. Streaming applications are also targeted by TBB [19] through the *pipeline* construct. However, TBB does not support any kind of non-linear streaming network, which therefore has to be embedded in a pipeline with significant programming and performance drawbacks.

OpenMP is a very popular thread-based framework for multi-core architectures. It chiefly targets data-parallel programming and provides means

to easily incorporate threads into sequential applications at a relatively high level. In an OpenMP program data needs to be labeled as shared or private, and compiler directives have to be used to annotate the code. Both OpenMP and TBB can be used to accelerate serial C/C++ programs in specific portions of code, even if they do not natively include farm skeletons, which are instead realized by using lower-level features such as the *task* annotation in OpenMP and the *parallel_for* construct in TBB. OpenMP does not require restructuring of the sequential program, while with TBB, which provides thread-safe containers and some parallel algorithms, it is not always possible to accelerate the program without some refactoring of the sequential code.

FastFlow falls between the easy programming of OpenMP and the powerful mechanisms provided by TBB. The FastFlow accelerator allows one to speed-up execution of a wide class of existing C/C++ serial programs with just minor modifications to the code.

The use of the lock-free approach in stream processing is becoming increasingly popular for multi-core platforms. The FastForward framework [15] implements a lock- and wait-free SPSC queue that can be used to build simple pipelines of threads that are directly programmed at low-level; arbitrary graphs of threads are not directly supported. The Erbium [24] framework also supports the streaming execution model with lock- and fence-free synchronizations. Among cited works, Erbium is the only framework also supporting the MPMC model. In contrast to FastFlow, scheduling of tasks within MPMC queues are statically arranged via a compilation infrastructure. The trade-off between overhead and flexibility of scheduling is as yet unclear. González and Fraguela recently proposed a general schema (i.e. a skeleton) for Divide&Conquer implemented via C++ templates and using as synchronization library the Intel TBB framework [17].

13.5 FUTURE WORK AND CONCLUSIONS

The FastFlow project is currently being extended in several different directions. FastFlow currently supports cyclic graphs, in addition to standard, non-cyclic streaming graphs. We are using a formal methods approach to demonstrate that the supported cyclic graphs are deadlock-free, exploiting the fact that each time a loop is present, unbounded queues are used to implement point-to-point channels. A version of FastFlow running on standard Windows framework is being finalized.

We are currently planning further developments of FastFlow: i) to increase memory-to-core affinity during the scheduling of tasks in order to be able to optimize consumer-producer data locality on forthcoming many-core architectures with complex memory hierarchy; ii) to provide programmers with more parallel patterns, including data-parallel patterns, possibly implemented in terms of the stream parallel patterns already included and optimized; and iii) to provide simpler and more user-friendly accelerator interfaces.

The emergence of the so-called power wall phenomenon has ensured that any future improvements in computer performance, whether in the HPC centre or on the desktop, must perforce be achieved via multi-processor systems rather than decreased cycle time. This means that parallel programming, previously a specialized area of computing, must become the mainstream programming paradigm. While in the past its niche status meant that ease of development was a secondary consideration for those engaged in parallel programming, this situation is changing quickly: programmability is becoming as important as performance. Application programmers must be provided with easy access to parallelism with little or no loss of efficiency. Traditionally, for the most part, abstraction has been bought at the cost of efficiency, and vice-versa. In this work we have introduced FastFlow, a framework delivering both programmability and efficiency in the area of stream parallelism.

FastFlow may be viewed as a stack of layers: the lower layers provide efficiency via lock-free/fence-free producer-consumer implementations; the upper layers deliver programmability by providing the application programmer with high-level programming constructs in the shape of skeletons/parallel patterns. FastFlow is applicable not only to classical streaming applications, such as video processing, in which the stream of images flows from the environment, but also applications in which streams are generated internally – covering areas such as Divide&Conquer, data parallel execution, etc. While FastFlow has been created primarily to target developments from scratch, provision has also been included for a migrating existing code to multi-core platforms by parallelizing program hot-spots via self-offloading using the FastFlow accelerator. The applicability of FastFlow has been illustrated by a number studies in differing application domains including image processing, file compression and stochastic simulation.

REFERENCES

1. M. Aldinucci, A. Bracciali, P. Liò, A. Sorathiya, and M. Torquati. StochKit-FF: Efficient systems biology on multicore architectures. In *Euro-Par '10 Workshops, Proc. of the 1st Workshop on High Performance Bioinformatics and Biomedicine (HiBB)*, volume 6586 of *LNCS*, pages 167–175, Ischia, Italy, Sept. 2010. Springer.
2. M. Aldinucci and M. Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, Oct. 2007.
3. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Skeletons for multi/many-core systems. In *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 265–272, Sept. 2009. IOS press.
4. M. Aldinucci, M. Danelutto, M. Meneghin, P. Kilpatrick, and M. Torquati. Efficient streaming applications on multi-core with FastFlow: the biosequence

alignment test-bed. In *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 09, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 273–280, Sept. 2009. IOS press.

5. M. Aldinucci, M. Danelutto, M. Meneghin, P. Kilpatrick, M. Torquati. Accelerating code on multi-cores with FastFlow. In *Proc. of 17th Intl. Euro-Par '11 Parallel Processing*, *LNCS*, Bordeaux, France, Aug. 2011. Springer. To appear.
6. M. Aldinucci, S. Ruggieri, and M. Torquati. Porting decision tree algorithms to multicore using FastFlow. In *Proc. of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 6321 of *LNCS*, pages 7–23, Barcelona, Spain, Sept. 2010. Springer.
7. S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies. Language and compiler design for streaming applications. *Int. J. Parallel Program.*, 33(2):261–278, 2005.
8. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *CACM*, 52(10):56–67, 2009.
9. R. Chan, C.-W. Ho, and M. Nikolova. Salt and pepper noise removal by median type noise detectors and detail-preserving regularization. *IEEE Transactions on Image processing*, 14(10):1479–1485, Oct. 2005.
10. P. Ciechanowicz and H. Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-Core Computer Architectures. In *Proc. of the 10th Intl. Conference on High Performance Computing and Communications (HPCC)*, pages 108–113, Los Alamitos, CA, USA, Sept. 2010. IEEE.
11. P. Ciechanowicz, M. Poldner, and H. Kuchen. The Munster skeleton library Muesli – a comprehensive overview. In *ERCIS Working paper*, number 7. ERCIS – European Research Center for Information Systems, 2009.
12. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.
13. M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 3(30): 389–406, 2004.
14. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Usenix OSDI '04*, pages 137–150, Dec. 2004.
15. J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, pages 43–52, New York, NY, USA, 2008.
16. J. Gilchrist. Parallel data compression with bzip2. In *Proc. of IASTED Intl. Conf. on Par. and Distrib. Computing and Sys.*, pages 559–564, 2004.
17. C. H. Gonzalez and B. B. Fraguela. A generic algorithm template for divide-and-conquer in multicore systems. In *Proc. of the 10th Intl. Conference on High Performance Computing and Communications (HPCC)*, pages 79–88, Los Alamitos, CA, USA, Sept. 2010. IEEE.
18. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

19. Intel *Threading Building Blocks*, 2011. `http://www.threadingbuildingblocks.org/`.

20. G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

21. E. Lee and T. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, May 1995.

22. M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 289–296, Pisa, Italy, Feb. 2010. IEEE.

23. T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

24. C. Miranda, P. Dumont, A. Cohen, M. Duranton, and A. Pop. Erbium: A deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proc. of the 2010 Intl. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 11–20, Scottsdale, AZ, USA, Oct. 2010. ACM.

25. L. Petzold. *StochKit: stochastic simulation kit web page*, 2009. `http://www.engineering.ucsb.edu/~cse/StochKit/index.html`.

26. J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kauffman, 1993.

27. S. Ruggieri. YaDT: Yet another Decision tree Builder. In *16th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI 2004)*, pages 260–265, 2004. IEEE.

28. A. Shafarenko, C. Grelck, and S.-B. Scholz. Semantics and type theory of S-Net. In *Proc. of the 18th Intl. Symposium on Implementation and Application of Functional Languages (IFL'06)*, TR 2006-S01, pages 146–166. Eötvös Loránd Univ., Faculty of Informatics, Budapest, Hungary, 2006.

29. T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, Mar. 1981.

30. Sourceforge. *FastFlow project*, 2009. `http://mc-fastflow.sourceforge.net/`.

31. R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, July 1997.

32. A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz. *SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2*, 2008. `http://www.scientificcommons.org/39542148`.

33. H. Tanno and H. Iwasaki. Parallel skeletons for variable-length lists in sketo skeleton library. In *Proc. of 15th Intl. Euro-Par '09 Parallel Processing*, volume 5704 of *LNCS*, pages 666–677, Delft, The Netherlands, Aug. 2009. Springer.

34. W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 11th Intl. Conference on Compiler Construction (CC)*, pages 179–196, London, UK, 2002.

35. M. Torquati. Single-Producer/Single-Consumer Queues on Shared Cache Multi-Core Systems. Technical Report TR-10-20, Dept. Comp. Science, Univ. of Pisa, Nov. 2010. `http://compass2.di.unipi.it/TR/Files/TR-10-20.pdf.gz`.

36. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.