

Functional Programming Interface for Parallel and Distributed Computing

Ivan Petriakov and Ivan Gankevich(✉)

Saint Petersburg State University, 7-9 Universitetskaya Emb.,
St. Petersburg 199034, Russia
st049350@student.spbu.ru, i.gankevich@spbu.ru
https://spbu.ru/

Abstract. There are many programming frameworks and languages for parallel and distributed computing which are successful both in industry and academia, however, all of them are isolated and self-contained. We believe that the main reason that there is no common denominator between them is that there is no intermediate representation for distributed computations. For sequential computations we have bytecode that is used as an intermediate, portable and universal representation of a programme written in any language, but bytecode lacks an ability to describe process communication. If we add this feature, we get low-level representation on top of which all the frameworks and languages for parallel and distributed computations can be built. In this paper we explore how such intermediate representation can be made, how it can reduce programming effort and how it may simplify internal structure of existing frameworks. We also demonstrate how high-level interface can be build for a functional language that completely hides all the difficulties that a programmer encounters when he or she works with distributed systems.

Keywords: API · Intermediate representation · C++ · Guile

1 Introduction

There are many programming frameworks and languages for parallel and distributed computing [11,12,15–17] which are successful both in industry and academia, however, all of them are isolated and self-contained. We believe that the main reason that there is no common denominator between these frameworks and languages is that there is no common protocol or low-level representation for distributed computations. For sequential computations we have bytecode (e.g. LLVM [9], Java bytecode, Guile bytecode) that is used as an intermediate, portable and universal representation of a programme written in any language; also we have assembler which is non-portable, but still popular intermediate representation. One important feature, that bytecode and assembler lack, is an

Supported by Council for grants of the President of the Russian Federation (grant no. MK-383.2020.9).

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12956, pp. 496–510, 2021.
https://doi.org/10.1007/978-3-030-87010-2_37

ability to communicate between parallel processes. This communication is the common denominator on top of which all the frameworks and languages for parallel and distributed computations can be built, however, there is no universal low-level representation that describes communication.

Why common low-level representation exists for sequential computations, but does not exist for parallel and distributed ones? One of the reasons, which applies to both distributed and parallel computations, is the fact that people still think about programmes as sequences of steps — the same way as people themselves perform complex tasks. Imperative languages, in which programmes are written as series of steps, are still prevalent in industry and academia; this is in contrast to unpopular functional languages in which programmes are written as compositions of functions with no implied order of computation. Another reason which applies to distributed computations is the fact that these computations are inherently unreliable and there is no universal approach for handling cluster node failures. While imperative languages produce more efficient programmes, they do not provide safety from deadlocks and fault tolerance guarantees. Also, they are much more difficult to write, as a human have to work with mutable state (local and global variables, objects etc.) and it is difficult to keep this state in mind while writing the code. Functional languages minimise the usage of mutable state, provide partial safety from deadlocks (under the assumption that a programmer does not use locks manually) and can be modified to provide fault tolerance. From the authors' perspective people understand the potential of functional languages, but have not yet realised this potential to get all their advantages; people realised the full potential of imperative languages, but do not know how to get rid of their disadvantages.

In this paper we describe low-level representation based on *kernels* which is suitable for distributed and parallel computations. Kernels provide automatic fault tolerance and can be used to exchange the data between programmes written in different languages. We implement kernels in C++ and build a reference cluster scheduler that uses kernels as the protocol to run applications that span multiple cluster nodes. Then we use kernels as an intermediate representation for Guile programming language, run benchmarks using the scheduler and compare the performance of different implementations of the same programme.

To prevent ambiguity we use the term *parallel* to describe computations that use several processor cores of single cluster node, the term *distributed* to describe computations that use several cluster nodes and any number of cores on each node, and term *cluster* to describe anything that refers to local cluster (as opposed to geographically distributed clusters which are not studied in this paper). *Intermediate representation* in our paper is a particular form of abstract syntax tree, e.g. in functional languages *continuation passing style* is popular intermediate representation of the code.

2 Methods

2.1 Parallel and Distributed Computing Technologies as Components of Unified System

In order to write parallel and distributed programmes the same way as we write sequential programmes, we need the following components.

- Portable low-level representation of the code and the data and includes means of decomposition of the code and the data into parts that can be computed in parallel. The closest sequential counterpart is LLVM.
- Cluster scheduler that executes parallel and distributed applications and uses the low-level representation to implement communication between these applications running on different cluster nodes. The closest single-node counterpart is operating system kernel that executes user processes.
- High-level interface that wraps the low-level representation for existing popular programming languages in a form of a framework or a library. This interface uses cluster scheduler, if it is available and node parallelism is needed by the application, otherwise the code is executed on the local node and parallelism is limited to the parallelism of the node. The closest single-node counterpart is C library that provides an interface to system calls of the operating system kernel.

These three components are built on top of each other as in classical object-oriented programming approach, and all the complexity is pushed down to the lowest layer: low-level representation is responsible for providing parallelism and fault tolerance to the applications, cluster scheduler uses these facilities to provide transparent execution of the applications on multiple cluster nodes, and high-level interface maps the underlying system to the target language to simplify the work for application programmers.

High-performance computing technologies have the same three-component structure: message passing library (MPI) is widely considered a low-level language of parallel computing, batch job schedulers are used to allocate resources and high-level interface is some library that is built on top of MPI; however, the responsibilities of the components are not clearly separated and the hierarchical structure is not maintained. MPI provides means of communication between parallel processes, but does not provide data decomposition facilities and fault tolerance guarantees: data decomposition is done either in high-level language or manually, and fault tolerance is provided by batch job scheduler. Batch jobs schedulers provide means to allocate resources (cluster nodes, processor cores, memory etc.) and launch parallel MPI processes, but do not have control over messages that are sent between these processes and do not control the actual number of resources used by the programme (all resources are owned exclusively by the programme, and the programme decides how to use them), i.e. cluster schedulers and MPI programmes do not talk to each other after the parallel processes were launched. Consequently, high-level interface is also separated from the scheduler. Although, high-level interface is built on top of the

low-level interface, batch job scheduler is fully integrated with neither of them: the cluster-wide counterpart of operating system kernel does not have control over communication of the applications that are run on the cluster, but is used as resource allocator instead.

The situation in newer big data technologies is different: there are the same three components with hierarchical structure, but the low-level representation is integrated in the scheduler. There is YARN cluster scheduler [14] with API that is used as a low-level representation for parallel and distributed computing, and there are many high-level libraries that are built on top of YARN [1,2,8,17]. The scheduler has more control over job execution as jobs are decomposed into tasks and execution of tasks is controlled by the scheduler. Unfortunately, the lack of common low-level representation made all high-level frameworks that are built on top of YARN API use their own custom protocol for communication, shift responsibility of providing fault tolerance to the scheduler and shift responsibility of data decomposition to higher level frameworks.

To summarise, the current state-of-the-art technologies for parallel and distributed computing can be divided into three classes: low-level representations, cluster schedulers and high-level interfaces; however, responsibilities of each class are not clearly separated by the developers of these technologies. Although, the structure of the components resembles the operating system kernel and its application interface, the components sometimes are not built on top of each other, but integrated horizontally, and as a result the complexity of the parallel and distributed computations is sometimes visible on the highest levels of abstraction.

Our proposal is to design a low-level representation that provides fault tolerance, means of data and code decomposition and means of communication for parallel and distributed applications. Having such a representation at your disposal makes it easy to build higher level components, because the complexity of cluster systems is hidden from the programmer, the duplicated effort of implementing the same facilities in higher level interfaces is reduced, and cluster scheduler has full control of the programme execution as it speaks the same protocol and uses the same low-level representation internally: the representation is general enough to describe any distributed programme including the scheduler itself.

2.2 Kernels as Objects that Control the Programme Flow

In order to create low-level representation for parallel and distributed computing we borrow familiar features from sequential low-level representations and augment them with asynchronous function calls and an ability to read and write call stack frames.

In assembler and LLVM the programme is written in imperative style as a series of processor instructions. The variables are stored either on the stack (a special area of the computer's main memory) or in processor registers. Logical parts of the programme are represented by functions. A call to a function places

all function arguments on the stack and then jumps to the address of the function. When the function returns, the result of the computation is written to the processor register and control flow is returned to the calling function. When the main function returns, the programme terminates.

There are two problems with the assembler that need to be solved in order for it to be useful in parallel and distributed computing. First, the contents of the stack can not be copied between cluster nodes or saved to and read from the file, because they often contain pointers to memory blocks that may be invalid on another cluster node or in the process that reads the stack from the file. Second, there is no natural way to express parallelism in this language: all function calls are synchronous and all instructions are executed in the specified order. In order to solve these problems we use object-oriented techniques.

We represent each stack frame with an object: local variables become object fields, and each function call is decomposed into the code that goes before function call, the code that performs function call, and the code that goes after. The code that goes before the call is placed into `act` method of the object and after this code the new object is created to call the function asynchronously. The code that goes after the call is placed into `react` method of the object, and this code is called asynchronously when the function call returns (this method takes the corresponding object as an argument). The object also has `read` and `write` methods that are used to read and write its fields to and from file or to copy the object to another cluster node. In this model each object contains enough information to perform the corresponding function call, and we can make these calls in any order we like. Also, the object is self-contained, and we can ask another cluster node to perform the function call or save the object to disk to perform the call when the user wants to resume the computation (e.g. after the computer is upgraded and rebooted).

The function calls are made asynchronous with help of thread pool. Each thread pool consists of an object queue and an array of threads. When the object is placed in the queue, one of the threads extracts it and calls `act` or `react` method depending on the state of the object. There are two states that are controlled by the programmer: when the state is *upstream* `act` method is called, when the state is *downstream* `react` method of the parent object is called with the current object as the argument. When the state is *downstream* and there is no parent, the programme terminates.

We call these objects *control flow objects* or *kernels* for short. These objects contain the input data in object fields, the code that processes this data in object methods and the output data (the result of the computation) also in object fields. The programmer decides which data is input and output. To reduce network usage the programmer may delete input data when the kernel enters *downstream* state: that way only output data is copied back to the parent kernel over the network. The example programme written using kernels and using regular function call stack is shown in Table 1.

Table 1. The same programme written using regular function call stack (left) and kernels (right). Here `async_call` performs asynchronous function call by pushing the child kernel to the queue, `async_return` performs asynchronous return from the function call by pushing the current kernel to the queue.

```
int nested(int a) {
  return 123 + a;
}

void main() {
  // code before
  int result = nested();
  // code after
  print(result);
}
```

```
struct Nested: public Kernel {
  int result;
  int a;
  Nested(int a): a(a) {}
  void act() override {
    result = a + 123;
    async_return();
  }
};

struct Main: public Kernel {
  void act() override {
    // code before
    async_call(new Nested);
  }
  void react(Kernel* child) override {
    int result = ((Nested*)child)->result;
    // code after
    print(result);
    async_return();
  }
};

void main() {
  async_call(new Main);
  wait();
}
```

This low-level representation can be seen as an adaptation of classic function call stack, but with asynchronous function calls and an ability to read and write stack frames. These differences give kernels the following advantages.

- Kernels define dependencies between function calls, but do not define the order of the computation. This gives natural way of expressing parallel computations on the lowest possible level.
- Kernels can be written to and read from any medium: files, network connections, serial connections etc. This allows to implement fault tolerance efficiently using any existing methods: in order to implement checkpoints a programmer no longer need to save memory contents of each parallel process, only the fields of the main kernel are needed to restart the programme from the last sequential step. However, with kernels checkpoints can be replaced with simple restart: when the node to which the child kernel was sent fails, the copy of this kernel can be sent to another node without stopping the programme and no additional configuration from the programmer.

– Finally, kernels are simple enough to be used as an intermediate representation for high-level languages: either via a compiler modification, or via wrapper library that calls the low-level implementation directly. Kernels can not replace LLVM or assembler, because their level of abstraction is higher, therefore, compiler modification is possible only for languages that use high-level intermediate representation (e.g. LISP-like languages and purely functional languages that have natural way of expressing parallelism by computing arguments of functions in parallel).

2.3 Reference Cluster Scheduler Based on Kernels

Kernels are general enough to write any programme, and the first programme that we wrote using them was cluster scheduler that uses kernels to implement its internal logic and to run applications spanning multiple cluster nodes. Single-node version of the scheduler is as simple as thread pool attached to kernel queue described in Sect. 2.2. The programme starts with pushing the first (or *main*) kernel to the queue and ends when the main kernel changes its state to *downstream* and pushes itself to the queue. The number of threads in the pool equals the number of processor cores, but can be set manually to limit the amount of parallelism. Cluster version of the scheduler is more involved and uses kernels to implement its logic.

Cluster scheduler runs in a separate daemon process on each cluster node, and processes communicate with each other using kernels: process on node A writes some kernel to network connection with node B and process on node B reads the kernel and performs useful operation with it. Here kernels are used like messages rather than stack frames: kernel that always resides in node A creates child message kernel and sends it to the kernel that always resides in node B. In order to implement this logic we added *point-to-point* state and a field that specifies the identifier of the target kernel. In addition to that we added source and destination address fields to be able to route the kernel to the target cluster node and return it back to the source node: (parent-kernel, source-address) tuple uniquely identifies the location of the parent kernel, and (target-kernel, destination-address) tuple uniquely identifies the location of the target kernel. The first tuple is also used by *downstream* kernels that return back to their parents, but the second tuple is used only by *point-to-point* kernels.

There are several responsibilities of cluster scheduler:

– run applications in child processes,
– route kernels between application processes running on different cluster nodes,
– maintain a list of available cluster nodes.

In order to implement them we created a kernel queue and a thread pool for each concern that the scheduler has to deal with (see Fig. 1): we have

- timer queue for scheduled and periodic tasks,
- network queue for sending kernels to and receiving from other cluster nodes,
- process queue for creating child processes and sending kernels to and receiving from them, and
- the main processor queue for processing kernels in parallel using multiple processor cores.

This separation of concerns allows us to overlap data transfer and data processing: while the main queue processes kernels in parallel, process and network queues send and receive other kernels. This approach leads to small amount of oversubscription as separate threads are used to send and receive kernels, but benchmarks showed that this is not a big problem as most of the time these threads wait for the operating system kernel to transfer the data.

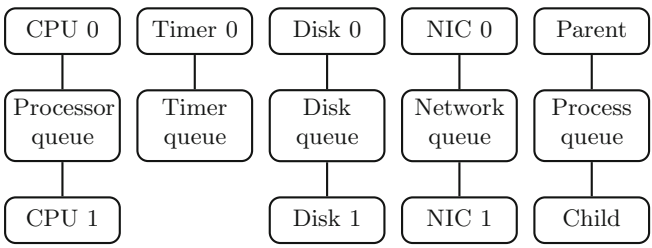

Fig. 1. Default kernel queues for each cluster scheduler concern.

Cluster scheduler runs applications in child processes; this approach is natural for UNIX-like operating systems as the parent process has full control of its children: the amount of resources can be limited (the number of processor cores, the amount of memory etc.) and the process can be terminated at any time. After introducing child processes into the scheduler we added cluster-wide source (target) application identifier field that uniquely identifies the source (the target) application from which the kernel originated (to which the kernel was sent). Also each kernel carries an application descriptor that specifies how to run the application (command line arguments, environment variables etc.) and if the corresponding process is not running, it is launched automatically by the scheduler. Child processes are needed only as means of controlling resources usage: a process is a scheduling unit for operating system kernel, but in cluster scheduler a child process performs something useful only when the kernel (which is a unit of scheduling in our scheduler) is sent to the corresponding application and launched automatically if there is no such application. Application spans multiple cluster nodes and may have any number of child processes (but no more than one process per node). These processes are launched on-demand and do nothing until the kernel is received. This behaviour allows us to implement dynamic

parallelism: we do not need to specify the number of parallel processes on application launch, the scheduler will automatically create them. To reduce memory consumption stale processes, that have not received any kernel for a long period of time, may be terminated (new processes will be created automatically, when the kernel arrives anyway). Kernels can be sent from one application to another by specifying different application descriptor.

Child process communicates with its parent using optimised child process queue. If the parent process does not want to communicate, the child process continues execution on the local node: the applications written using cluster scheduler interface work correctly even when the scheduler is not available, but use local node instead of the cluster.

Since the node may have multiple child processes, we may have a situation when all of them try to use all processor cores, which will lead to oversubscription and suboptimal performance. In order to solve this problem, we introduce weight field which tells how many threads will be used by the kernel. The default is one thread for ordinary kernels and nought threads for cluster scheduler kernels. Process queue tracks the total weight of the kernels that were sent to child processes and queues incoming kernels if the weight reaches the number of processor cores. Also, each cluster node reports this information to other nodes for better load balancing decisions.

The scheduler acts as a router for the kernels: when the kernel is received from the application, the scheduler analyses its fields and decides to which cluster node it can be sent. If the kernel has *downstream* or *point-to-point* state, the kernel is sent to the node where the target kernel resides; if the kernel has *upstream* state, load balancing algorithm decides which node to send the kernel to. Load balancing algorithm tracks the total weight of the kernels that were sent to the specified node and also receives the same information from the node (in case other nodes also send there their kernels), then it chooses the node with the lowest weight and sends the kernel to this node. If all nodes are full, the kernel is retained in the queue until the enough processor cores become available. The algorithm is very conservative and does not use work-stealing for improved performance, however, the fault tolerance is easier to implement [4,5] when the target and the source node fields do not change during kernel lifetime which is not the case for work-stealing scenario.

The last but not the least responsibility of the scheduler is to discover and maintain a list of cluster nodes and establish persistent network connections to neighbours. Cluster scheduler does this automatically by scanning the network using efficient algorithm: the nodes in the network are organised in artificial tree topology with the specified fan-out value and each node tries to communicate with the nodes which are closer to the root of the tree. This approach significantly reduces the number of data that needs to be sent over the network to find all cluster nodes: in ideal case only one kernel is sent to and received from the parent node. The algorithm is described in [6]. After the connections are established, all the *upstream* kernels that are received from the applications' child processes are routed to neighbour nodes in the tree topology (both parent

and child nodes). This creates a problem because the number of nodes "behind" the parent node is generally different than the number of nodes behind the child nodes. In order to solve this problem we track not only the total weight of all kernels of the neighbour node, but the total weight of each node in the cluster and sum the weight of all nodes that are behind the node A to compute the total weight of node A for load balancing. Also, we apply load balancing recursively: when the kernel arrives at the node, load balancing algorithm is executed once more to decide whether the kernel can be sent locally or to another cluster node (except the source node). This approach solves the problem, and now applications can be launched not only on the root node, but on any node without load balancing problems. This approach adds small overhead, as the kernel goes through intermediate node, but if the overhead is undesirable, the application can be launched on the root node. Node discovery and node state updates are implemented using *point-to-point* kernels.

To summarise, cluster scheduler uses kernels as unit of scheduling and as communication protocol between its daemon processes running on different cluster nodes. Daemon process acts as an intermediary between application processes running on different cluster nodes, and all application kernels are sent through this process to other cluster nodes. Kernels that are sent through the scheduler are heavy-weight: they have more fields than local kernels and the routing through the scheduler introduces multiple overheads compared to direct communication. However, using cluster scheduler hugely simplifies application development, as application developer does not need to worry about networking, fault tolerance, load balancing and "how many parallel processes are enough for my application": this is now handled by the scheduler. For maximum efficiency and embedded applications the application can be linked directly to the scheduler to be able to run in the same daemon process, that way application kernels are no longer sent though daemon process and the overhead of the scheduler is minimal.

2.4 Parallel and Distributed Evaluation of Guile Expressions Using Kernels

Kernels low-level interface and cluster scheduler are written in C++ language. From the authors' perspective C is too low-level and Java has too much overhead for cluster computing, whereas C++ is the middleground choice. The implementation is the direct mapping of the ideas discussed in previous sections on C++ abstractions: kernel is a base class (see listing 1.1) for all control flow objects with common fields (`parent`, `target` and all others) and `act`, `react`, `read`, `write` virtual functions that are overridden in subclasses. This direct mapping is natural for a mixed-paradigm language like C++, but functional languages may benefit from implementing the same ideas in the compiler or interpreter.

```
enum class states {upstream, downstream, point_to_point};

class kernel {
public:
  virtual void act();
  virtual void react(kernel* child);
  virtual void write(buffer& out) const;
  virtual void read(buffer& in);
  kernel* parent = nullptr;
  kernel* target = nullptr;
  states state = states::upstream;
};

class queue {
public:
  void push(kernel* k);
};
```

Listing 1.1. Public interface of the kernel and the queue classes in C++ (simplified for clarity).

We made a reference implementation of kernels for Guile language [3]. Guile is a dialect of Scheme [13] which in turn is a dialect of LISP [10]. The distinct feature of LISP-like languages is homoiconicity, i.e. the code and the data is represented by tree-like structure (lists that contain atoms or other lists as elements). This feature makes it possible to express parallelism directly in the language: every list element can be computed independently and it can be sent to other cluster nodes for parallel computation. To implement parallelism we created a Guile interpreter that evaluates every list element in parallel using kernels. In practice this means that every argument of a procedure call (a procedure call is also a list with the first element being the procedure name) is computed in parallel. This interpreter is able to run any existing Guile programme (provided that it does not use threads, locks and semaphores explicitly) and the output will be the same as with the original interpreter, the programme will automatically use cluster nodes for parallel computations, and fault tolerance will be automatically provided by our cluster scheduler. From the authors' perspective this approach is the most transparent and safe way of writing parallel and distributed programmes with clear separation of concerns: the programmer takes care of the application logic, and cluster scheduler takes care of the parallelism, load balancing and fault tolerance.

Our interpreter consists of standard *read-eval-print* loop out of which only *eval* step uses kernels for parallel and distributed computations. Inside *eval* we use hybrid approach for parallelism: we use kernels to evaluate arguments of procedure calls and arguments of `cons` primitive asynchronously only if these arguments contain other procedure calls. This means that all simple arguments (variables, symbols, other primitives etc.) are computed sequentially without creating child kernels.

Evaluating procedure calls and `cons` using kernels is enough to make `map` form parallel, but we had to rewrite `fold` form to make it parallel. Our parallelism is based on the fact that procedure arguments can be evaluated in parallel without affecting the correctness of the procedure, however, evaluating arguments in parallel in `fold` does not give speedup because of the nested `fold` and `proc` calls: the next recursive call to `fold` waits until the call to `proc` completes. Alternative procedure `fold-pairwise` does not have this deficiency, but is only correct for `proc` that does not care about the order of the arguments (`+`, `*` operators etc.). In this procedure we apply `proc` to successive pairs of elements from the initial list, after that we recursively call `fold-pairwise` for the resulting list. The iteration is continued until only one element is left in the list, then we return this element as the result of the procedure call. This new procedure is also iterative, but parallel inside each iteration. We choose `map` and `fold` forms to illustrate automatic parallelism because many other forms are based on them [7]. Our implementation is shown in listing 1.2.

```
(define (map proc lst) "Parallel map."
  (if (null? lst) lst
    (cons (proc (car lst)) (map proc (cdr lst)))))
(define (fold proc init lst) "Sequential fold."
  (if (null? lst) init
    (fold proc (proc (car lst) init) (cdr lst))))
(define (do-fold-pairwise proc lst)
  (if (null? lst) '()
    (if (null? (cdr lst)) lst
      (do-fold-pairwise proc
        (cons (proc (car lst) (car (cdr lst)))
              (do-fold-pairwise proc (cdr (cdr lst))))))))
(define (fold-pairwise proc lst) "Parallel pairwise fold."
  (car (do-fold-pairwise proc lst)))
```

Listing 1.2. Parallel `map` and `fold` forms in Guile.

3 Results

We tested performance of our interpreter using the forms in listing 1.2. For each form we applied synthetic procedure that sleeps 200 ms to the list with 96 elements. Then we ran the resulting script using native Guile interpreter and our interpreter and measured total running time for different number of threads. For native Guile interpreter the running time of all forms is the same for any number of threads. For our interpreter `map` and `fold-pairwise` forms run time decreases with the number of threads and for `fold` form run time stays the same (Fig. 2).

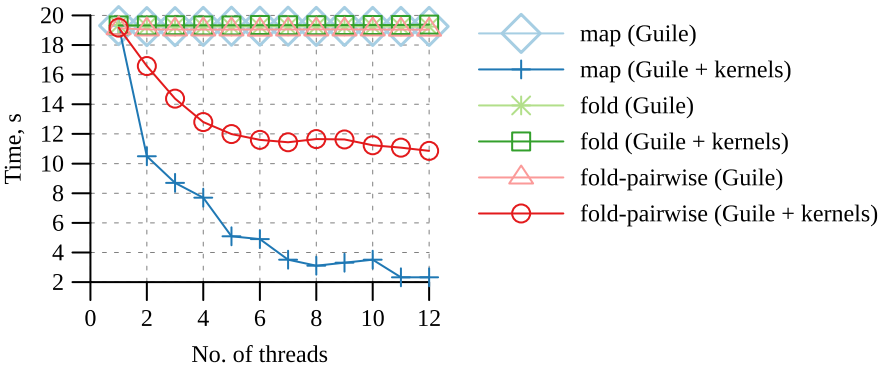

Fig. 2. The run time of the forms from listing 1.2 for different number of parallel threads and different interpreters.

4 Discussion

Computing procedure arguments in parallel is a natural way of expressing parallelism in functional language, and in our tests the performance of the programme is close to the one with manual parallelism. Lower performance is explained by the fact that we introduce more overhead by using asynchronous kernels to compute procedure arguments where this does not give large performance gains (even with ideal parallelism with no overhead). If we remove these overheads we will get the same time as the original programme with manual parallelism. This is explained by the fact that the main loop of the programme is written as an application of `map` form and our interpreter makes it parallel. Executing this loop in parallel gives the largest performance gains compared to other parts of the programme. We expect that the difference between automatic and manual parallelism to be more visible in larger and more complex programmes, and in future plan to benchmark more algorithms with known parallel implementations.

5 Conclusion

Using procedure arguments to define parallel programme parts gives new perspective on writing parallel programmes. In imperative languages programmers are used to rearranging loops to optimise memory access patterns and help the compiler vectorise the code, but with parallel-arguments approach in functional languages they can rearrange the forms to help the interpreter to extract more parallelism. This parallelism is automatic and does not affect the correctness of the programme (of course, you need to serialise access and modification of the global variables). With help of kernels these parallel computations are extended to distributed computations. Kernels provide standard way of expressing parallel and distributed programme parts, automatic fault tolerance for master and worker nodes and automatic load balancing via cluster scheduler. Together kernels and arguments-based parallelism provide low- and high-level programming

interface for clusters and single nodes that conveniently hide the shortcomings of parallel and distributed computations allowing the programmer to focus on the actual problem being solved rather than fixing bugs in his or her parallel and distributed code.

Future work is to re-implement LISP language features that are relevant for parallelism in a form of C++ library and use this library for parallelism, but implement the actual computations in C++. This would allow to improve performance of purely functional programmes by using the functional language for parallelism and imperative language for performance-critical code.

Acknowledgements. Research work is supported by Council for grants of the President of the Russian Federation (grant no. MK-383.2020.9).

References

1. Apache Software Foundation: Hadoop. https://hadoop.apache.org
2. Apache Software Foundation: Storm. https://storm.apache.org
3. Galassi, M., Blandy, J., Houston, G., Pierce, T., Jerram, N., Grabmueller, M.: Guile Reference Manual (2002)
4. Gankevich, I., Tipikin, Y., Korkhov, V.: Subordination: providing resilience to simultaneous failure of multiple cluster nodes. In: Proceedings of International Conference on High Performance Computing Simulation (HPCS 2017), pp. 832–838. Institute of Electrical and Electronics Engineers (IEEE), NJ, USA, July 2017. https://doi.org/10.1109/HPCS.2017.126
5. Gankevich, I., Tipikin, Y., Korkhov, V., Gaiduchok, V.: Factory: non-stop batch jobs without checkpointing. In: International Conference on High Performance Computing Simulation (HPCS 2016), pp. 979–984, July 2016. https://doi.org/10.1109/HPCSim.2016.7568441
6. Gankevich, I., Tipikin, Y., Gaiduchok, V.: Subordination: cluster management without distributed consensus. In: International Conference on High Performance Computing Simulation (HPCS), pp. 639–642 (2015). https://doi.org/10.1109/HPCSim.2015.7237106
7. Hutton, G.: A tutorial on the universality and expressiveness of fold. J. Funct. Programm. **9**(4), 355–372 (1999). https://doi.org/10.1017/S0956796899003500
8. Islam, M., et al.: Oozie: towards a scalable workflow management system for Hadoop. In: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies. SWEET 2012. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2443416.2443420
9. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2004, USA, p. 75. IEEE Computer Society (2004)
10. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4), 184–195 (1960). https://doi.org/10.1145/367177.367199

11. Pinho, E.G., de Carvalho, F.H.: An object-oriented parallel programming language for distributed-memory parallel computing platforms. Sci. Comput. Programm. **80**, 65–90 (2014). https://doi.org/10.1016/j.scico.2013.03.014. Special section on foundations of coordination languages and software architectures (selected papers from FOCLASA11)
12. Stewart, R., Maier, P., Trinder, P.: Transparent fault tolerance for scalable functional computation. J. Funct. Programm. **26**, e5 (2016). https://doi.org/10.1017/S095679681600006X
13. Sussman, G.J., Steele, G.L.: The first report on scheme revisited. High. Order Symb. Comput. **11**, 399–404 (1998). https://doi.org/10.1023/A:1010079421970
14. Vavilapalli, V.K., et al.: Apache Hadoop YARN: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC 2013, pp. 1–16. ACM, New York (2013). https://doi.org/10.1145/2523616.2523633
15. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: a language for distributed parallel scripting. Parallel Comput. **37**(9), 633–652 (2011)
16. Yu, Y., et al.: DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. Proc. LSDS-IR **8** (2009)
17. Zaharia, M., et al.: Apache Spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664