Blip: an integrated parallel API for the D1 language

F. Mohamed
Humboldt Universität zu Berlin

November 15, 2010

Abstract

A task based API to describe the parallelization at the shared memory level, based on independent recursive tasks is described. It offers a simple conceptual model, and integrated with a way to delay and resubmit an arbitrary task can be used to efficiently implement several parallelization approaches. It is argued that such an API integrated with MPI like parallelization and a remote procedure call layer represents a meaningful level of increasing parallelization and program complexity. An effort is done to allow parallel programs to run with a small overhead on single CPUs. The API presented has been realized in a first little optimized version, that already shows good performance, and is available as part of the open source blip library.

1 Introduction

Parallelization is an important issue, and in the High Performance Computing (HPC) field parallel computers have a long history, but parallelization has recently gained relevance also in desktop computer, as it was found that the larger and larger amounts of transistors available can be used more efficiently adding multiple cores, rather than make a single core go faster. Indeed all the newer processors announced have multiple cores. Writing parallel programs is notoriously more difficult than writing sequential programs, because it is more difficult for the programmer to keep track of the possible execution sequences. Thus having a conceptual model of the program that the programmer can understand is as important as having a model that can be efficiently executed.

There are several approaches to this complex problem, too many to describe them all, Dongarra et al. (2008) gives an overview of some of the methods and tools that can be used.

An important distinction is how memory is handled. In the shared memory approach, which is the model used by multicore designs, a unique address space is shared between all processors. This makes it easier to adapt sequential programs, but is more difficult to reason about the properties of the programs

unless one restricts himself to simple access patterns. In distributed memory approaches each processor has its own memory space. In general it is more complex to program for it, because the programmer has to explicitly think about the data distribution and moving it around. A cluster of computers can be described by this model.

Normally the shared memory model does not scale so well because it hides potentially costly operations from the programmer and enforces some coherency between distant parts. Numa (non uniform memory architecture), tries to give to shared memory model some awareness of the non coherency but often programs written with the distributed model scale better exactly because the programmer has thought about the costly operations.

The shared memory approach normally used in the HPC field is the OpenMP standardARB (1997, 2008). Languages that support a partitioned global address space (PGAS) like Allen et al. (2008); Chamberlain et al. (2007); Saraswat et al. (2010); El-Ghazawi et al. (2005); Numrich and Reid (1998) allow one to have a global view, while still having local storages, each process can access both his local and remote just indexing. The conceptual advantage of this is that one can easily migrate to it starting from a local view, and as at least in some cases it is possible to optimize the layout and patterns used to access the memory independently from the algorithm used. Locks and more recently atomic operations are supported by most OS and are used in conventional system programmingStevens and Rago (2005). CilkBlumofe et al. (1995) offers yet another approach to the parallelization that uses work stealing Burton and Sleep (1981).

There are several distributed memory approaches, in the HPC field MPIForum (1995, 2009) is probably the most used method. Unix offers several interprocess communication methods (pipes, memory sharing,...)Stevens and Rago (2005). Communicating sequential processes Hoare (1978) inspired several parallelization approaches, of which erlang's actor based parallelism Armstrong et al. (1996) is probably the most well known implementation. Remote procedure calls are a very useful method to communicate between separated processes, and has been implemented in varios way using various protocols from nfs to web services. Recently also Peer to Peer (and grid computing) like `http://boinc.berkeley.edu` gained relevance.

Both the shared memory and the distributed memory approaches have their merit, and it seems that to take advantage of current HPC computers both have to be used. Typically openMP and MPI are used in the HPC field, and more recently an hybrid approach is required also by GPU processing interfaces like Cuda and OpenCL.

What we present here is not a completely revolutionary method, but rather a refinement and combination of several ideas that were already present, and that demonstrated their usefulness. We take a more humble approach that what was attempted in other high productivity languages Allen et al. (2008); Chamberlain et al. (2007); Saraswat et al. (2010), we just try to give the tools to implement different paradigms reasonably efficiently. Offering these in a uniform framework makes a difference both from a pragmatic point of view

(making them easier to use) and also from the conceptual point of view, as having a clear conceptual model helps in avoiding bugs. The parallelization has been developed as a library, because the language used was flexible enough to permit it. This allows one to potentially expose the whole machinery, so that if a user really wants to do something special that is not allowed by the high level API, it can still do it.

The parallelization strategy has 3 tiers a symmetric multiprocessing level (SMP, i.e. shared memory), one that is basically MPI, and and rpc level.

2 SMP Parallelization

The SMP level wants to take advantage of the increasing presence of multiple cores and hyperthreading, and uses a shared memory approach. Threads are an obvious way to try to use these resources, but take advantage optimally of the computational resources when their number is equal to the number of executing units. With hyperthreading, or when another resource (as memory access) is constrained the optimal number might be smaller.

Some languages try to automatically parallelize, and subdivide the calculation in blocks that are then executed in parallel. This is a difficult problem, we aim at handling the simpler problem of optimally scheduling tasks on parallel processors. There are several libraries that aim at a similar purpose: Intel Threading Building Blocks intel (2010), Microsofts Task Parallel Library (TPL)[1], libdispatch[2] (the core of Apple's Grand Central Dispatch), XWS Cong et al. (2008) (the X10 Work Stealing framework), PFunc Kambadur et al. (2009) (a generic C++ task library). In this work we introduce an approach that for the shared memory part merges the conventional threads and Cilk like approaches by considering independent recursive tasks. Coping well with both kinds of parallelism is important, so that the programmer can implement several parallelization schemes within the library, and avoid oversubscribing the processors.

For a single recursive task subdividing the work in n_cpu chunks is a natural choice. Since the first versions openmp has made it simple to do for large loops. Such an approach minimizes the switching overhead, but is not so easy to generalize, already nested loops are difficult to distribute correctly. Furthermore this approach is not robust with respect to having other work progress concurrently (both by your own program and by the OS). If same task A might be executed both alone and concurrently with task B, automatically finding the correct number of threads to assign to task A is not straightforward.

Assuming that task management and switching costs x it is much better to try to subdivide the problem in as many tasks as possible, but making sure that each task on average still needs y¿¿x time to execute (for example y¿50x) independently of the number of processors. In this case a scheduler can keep processors busy by distributing tasks around, and if load on one processor changes

[1] `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`
[2] `http://libdispatch.macosforge.org/`

the work can be automatically rescheduled. This means that the unit of computation one has to think about is not the thread, but one of these smaller units, that we call tasks.

Having relatively many small tasks also works well as latency hiding mechanism, many tasks can do operations that can stall the CPU: I/O, network and disk operations, or waiting for a GPU computation. To avoid wasting CPU cycles one should switch to another task as soon one of these operations is started, and switch back only when more work can be performed on them.

Creating as many tasks as possible has a drawback: task handling and management uses resources, and some problems would create lot of tasks. The solution to this is not to create too many tasks at once but create them lazily. This must be done in concert with the evaluation strategy.

To find a way to strike a balance between having enough tasks to keep all processors busy, and minimizing their number to consume little resources, we will consider a recursive or better tree like computation. These represent an important class of computations, all divide and conquer approaches can be cast in this form, thus having a good solution for them will likely cover a broad range of applications. A recursive function calls itself possibly several times, spawning several tasks which will spawn other tasks. To evaluate this on a single processor there is a well known efficient strategy: eager evaluation, evaluate first the subtasks before finishing the main task. This means a depth first evaluation of the tasks. In general we will not get rid of the task switching overhead (this is safely possible only in few instances), but we will avoid creating too many task at the same time.

Now we have the optimal strategy for a single processor, but what we are interested in, is to distribute the work between several processors. To simplify things we will assume that it is never worthwhile to keep a processor idle. Thus we take the view of the computational scientist: one that knows that the problem is big enough to use the whole machine, and is interested in maximum throughput. Deciding to use only part of the available resources, or putting an upper bound on the latency of the tasks is a refinement that will not be considered here.

What should a processor that has no work do? It should actively go looking for more work asking around. Each core (or optionally each logical core) has a queue that keeps the tasks that should be executed. An idle processor tries to steal some work from the neighboring schedulers in the order given by the level of memory hierarchy sharing.

Ideally when stealing one should try to equalize the amount of work between the two queues. As the execution time of a task is unknown this is impossible to do exactly, but in general one should try to steal large tasks, which means root tasks (that will spawn other tasks) of recursive tasks. Thus the strategy is to steal root tasks from other processors, but execute children tasks first on each processor. With this strategy the amount of communication and suspended tasks should be minimized. Applying this strategy strictly one can expect to have at most the depth of the recursive tree call times the number of queues suspended tasks. In practice one has a little bit more because to reduce the

switches a bit of slackness is introduced before switching to subtasks, overproducing subtasks.

If one is using a kind of "divide & conquer" algorithm this scheduling has the very nice property of being compatible with cache hierarchy. As each processor tries to execute independently a whole subtree it is very likely that the memory accesses are also structured like a tree and moving a supertask will move all accesses to a large region of memory to one processor, while subtasks will partition the accessed memory in smaller and smaller regions. This is similar to cache oblivious algorithms Acar et al. (2002); Frigo and Strumpen (2006), even if due to the relatively large coarseness of tasks it will probably not hold for the smallest caches (unless the task itself is cache oblivious or cache aware). As the task stealing procedure keeps into account the different cost of stealing by trying to steal from queues that share part of the cache hierarchy first, this property is preserved as much as possible.

The sketched procedure is an efficient parallel evaluation strategy for recursive tasks. CilkBlumofe et al. (1995) execution strategy is very similar to this, in fact if tasks don't stall the queue of Cilk will perform a depth first execution and steal the root tasks. This only breaks down and becomes less efficient if tasks have dependencies or one allows several threads to work on the same queue (by default our parallelization scheme shares a queue between all hyperthreads of the same core). But the simplified queue structure of Cilk can be realized with a locking protocol that can be more efficient, and reduces the scheduling overhead in the normal case. Future versions might well try to use some of the tricks used by Cilk. As Intel bought Cilk++ unsurprisingly the same technique is used also by the task scheduler used by intel threading blocksintel (2010).

OpenMp in the version 3.0 has introduced the concept of task and tasksets which theoretically allow one to perform a parallelization as described, but in practice there are two problems. One is that most implementations don't handle tasks very efficiently, and how well it has to work is not defined, so that it is difficult to rely on this parallelization especially if one wants to be portable and still retain a good efficiency. The other issue is more subtle, OpenMp has been developed as a way to annotate code, so that without annotations one has valid serial code. This works beautifully for loops, but with tasks the difference between the serial and parallel version increases. In this case having just one code to think about (always having tasks) is simpler.

The recursive evaluation approach is not always correct: independent tasks, as for example different requests in a server should be scheduled on an equal footing, independently on how deep each one is. If one has a request that does a computationally intensive task, and immediately after one that just looks up a value, always prioritizing deep task will increase the latency between request and answer very much.

Threads cope well with this situation, as they have an implicit "fairness" in them that means that all threads have more or less the same chance of executing (up to their priority), and should not be ignored for long times. This is correct for external tasks, but applying it to recursive tasks is wasteful, as it forces tasks up in the hierarchy to be executed, giving a breath first execution that

will allocate many tasks.

Now the conceptual model for the SMP approach we propose here is clear: we schedule independent recursive tasks. Independent tasks should be scheduled fairly, whereas within a single recursive tasks subtasks will be executed first. Idle processors steal tasks trying to take root tasks.

This is different from the intel thread blocks that offer only a shared queue to which tasks can be enqueued to guarantee some fairness. The better support for fairness clearly might increase context switches and lead to a worse cache usage, but it was deemed important to be able to use the generic scheduler for remote requests ensuring that the latency does not explode. The current implementation is still quite simple, but normally executes at most a single task for each independent task before executing again a task from the first independent task.

The execution characteristics within a recursive task or between different tasks is very different, and it is the programmer job to choose between them, Recursive tasks *might* be parallel and should be executed in parallel if free hardware resources are available, whereas independent tasks *should* be executed in parallel with some kind of fairness. Recursive tasks cope well with data parallel workloads, thread-like parallelization copes well with webserver-like load. Making some automatic scheduler choose between them is not going to work well and will either give high latency or waste resources. Thus we leave this choice to the programmer.

Some languages try have special syntax to expose parallel operations, but normally the program does not have so many of them to make any special syntax really worth it, it should be clean, but method calling works just well enough. The main API that a user sees is

```
/// Action is the type of an action (similar to a closure or a function)
alias void delegate() Action;
/// the interface of tasks
interface TaskI{
  /// yields (temporarily stops) the execution of the current task
  void yield();
  /// maybe yields the current task (use this to avoid creating too many tasks,
  /// while reducing context switches)
  void maybeYield();
  /// yields until all the subtasks of this task have finished
  void finishSubtasks(){
  /// starts this task as a subtask of superTask which defaults to the current task
  void spawn(TaskI superTask=null);
  /// spawns and yield
  void spawnYield(TaskI superTask=null){ spawn(superTask); maybeYield(); }
  /// adds an action to perform after the task has finished
  void appendOnFinish(Action action);
  /// waits for the completion of this task
  void wait();
  /// delays the current task. opStart is executed after the task has been flagged
  /// as delayed, but before stopping the current execution. Use opStart to start
```

```
  /// the operation that will resume the task
  void delay(Action opStart=null);
  /// resubmit a delayed task (the task has to be delayed just once)
  void resubmitDelayedSingle();
  /// starts this task and waits for its completion, i.e. submit and wait
  void executeNow(TaskI superTask)
  /// retain/release/autorelease (to allow immediate reuse of the task object)
}
/// the normal kind of tasks
class Task:TaskI{
  /// constructor that creates a new task called name and executing action
  static Task opCall(char[] name, Action action){ /*...*/ }
}
```

Normally one does something like

```
Task t=Task("myTask",&obj.myAction).appendOnFinish(&obj.cleanup);
t.autorelease.submitYield();
```

The first statement creates a task that executes obj.myAction() and after it is finished calls obj.cleanup(). The second one starts it as subtask of the current task. As we are not interested in the task object we allow immediate reuse of the task object by calling autorelease. By giving a special superTask as argument to submit and submitYield one can change the way the task is scheduled, for example independent tasks can be started passing $defaultTask$ as argument. There are also SequentialTasks which enforce a sequential execution on their subtasks and can be passed as arguments to submit.

Sometime one wants to suspend a task waiting for an event. If one blocks the executing thread, then CPU time is wasted and one might even deadlock (for example if it is waiting for another task that cannot be executed as the executing thread is waiting). It is much better to delay the task, and go on executing other tasks while ensuring that when the event happens the task is resubmitted. To achieve this one should

```
TaskI tAtt=taskAtt.val; // get the current task
tAtt.delay(delegate void(){
    waitForSomeEvent(tAtt);
});
```

waitForSomeEvent should call tAtt.resubmitDelayedSingle() when the event takes place.

That event based programming can lead to better performance has been realized since some time, and blip uses libev `http://software.schmorp.de/pkg/libev` so that i/o, periodic events,... can be implemented efficiently. The ability to delay a task is also important when working with GPUs so that the main CPU can work on other things while waiting for the GPU.

In our implementation tasks are always explicit, and no attempt is made to flatten the computation, either fully (as in NESL Blelloch et al. (1995)), or partially (as in Cilk, where some spawn are replaced with normal C calls). Our aim is to provide a good scheduling framework for more coarse grained

tasks explicitly under the control of the program. This does not mean that the programmer should always think and program at the task level, indeed using the API described it is possible to define many useful abstractions. For example one often wants to parallelize loops on arrays or iterators and for that there are helpers that allow one to write

```
foreach(i, ref el;pLoopArray(myBeautifulArray,30)){
// ...
}
```

to loop in parallel on the array myBeautifulArray trying to have blocks of 30 elements, giving the index i and a modifiable reference to the current element called el.

Another useful concept to build parallel programs are dataflow variablesRoy and Haridi (2004) which are related to futuresJr and Hewitt (1977). Dataflow variables can be set several times but always to the same value. Reading a dataflow variable blocks until a value is set. The nice thing about dataflow variables is that they do not introduce race conditions: a program using them either always blocks or never does. blip implements dataflow variables, just like several kinds of semaphores, conditional waits and locks can be implemented on the top of the API sketched here. Also communicating serial processes (CSP) Hoare (1978) and similar can be implemented using it.

This shows that tasks with the interface presented here are a good choice as basic API.

3 MPI

The message passing interface is a widely used distributed memory model in HPC. It does not only offer point to point messages, its distinguishing feature are advanced and efficiently implemented collective communication primitives. It simplifies the full distributed model by assuming that the participating processes are reliable, so that the can be identified with numbers 0..rank-1, or even with higher dimensional indexes, and subdivided is several ways. MPI 2 allows the dynamic addition and removal of processors, but reliability is still assumed: there are no partial failures, something that simplifies the programming greatly. Most collective communication operations introduces a synchronization that avoids most race possibilities. If one wants to use all available resources to solve a single problem the MPI model can be better suited than any actor based model where one doesn't know exactly which computational resources are available.

Seen the success of this model it was simply adopted as it is using only a very simple interface (so that one can potentially have several MPI like interfaces), and adding support for easy serialization of arbitrary objects.

4 Rpc

MPI is a powerful model, and successfully hides several of the complexities of distributed computing, while giving lot of its power, but has a big drawback: processes cannot be easily added or removed. Sometime more flexibility would be welcome.

Remote procedure calls (rpc) is an old approach to this, in its more modern variant in which objects are published, and then called through a local proxy (as in Distributed Objects, CORBA and similar) it represents a very convenient communication API. Web services also use a similar approach to communicate and use urls to identify the various resources. This is a quite natural and useful approach so we also follow it.

Several protocols and transport mechanisms can be supported, for example both MPI or tcp/ip protocols can be used. The most useful protocol defined uses tcp/ip sockets and a simple binary serialization of its arguments, and is called stcp (simple tcp). Its urls have the form stcp://*host*:*port.group*/obj/*objectName* where the .group part might be dropped for the default group. Method calls simply add the method name to the url. Overloading is not supported, all names have to be unique.

Objects are publisher through a vendor (that normally is a separated object). All their remote evaluations happen in a task returned by the vendor. So for example to enforce CSP the vendor object has simply to return a sequential task, and all remote calls will be serialized. Each vended object has a corresponding proxy object that helps making remote calls look like local calls.

The proxy object and vendor objects take care of serializing/unserializing (or mashalling/unmarshalling) the arguments and result so that they can be sent to the remote party. A separate object called ProtocolManager actually sends and receives the data. The serialization has been realized through a generic serialization protocol, so that binary serialization might be particularly efficient, but how to serialize the basic types remains up to the serializer. One can support several serializers implementing a single interface, and adding several serializers does not add extra code to the objects to be serialized (for example through template instantiations). Indeed there are two serializers in the library, one that generates json code, and another that writes compact binary code with separated description of the types (to reduce the overhead when serializing arrays of objects).

Some methods might be declared as *oneway* which means that the system will not wait for their completion. With them one can have asynchronous notifications, but the failure of a oneway method can be silent.

The generation of vendor objects and proxy objects for a given class can be automatized, and thanks to a feature of the language used (string mixins) one can simply give a base name, the list of methods to be exposed, and possibly interfaces to generate vendor and proxy. The vendor and proxy objects are quite general and can be used for several protocols, in fact the proxy gives its serialized arguments, and a method to unserialize the result to its protocolHandler together with its first argument.

P2P (peer to peer) protocols like distribued hashtables, use an argument (key) to know how to send/address the request. If care is taken to use the first argument as key, then also P2P protocols can use the proxy and vendors generated automatically. In that case a single proxy will cover all servers, whereas in the normal case of stcp one has one proxy for each server/connection.

A local proxy object that avoids the serialization is also generated, so that local communication can use this optimized proxy. One could use the local object directly, but to get exactly the same behavior with respect to the parallelization the use of a wrapper can be safer. The decision of whether to use the serializing proxy, or the local proxy for local objects depends on the protocol. Stcp uses local proxies when possible.

The use of Rpc make it simpler to create weakly coupled parallel systems, potentially using several different protocols. The technical part of serializing and communicating the data is simplified. The use of urls to identify the resources helps making the program less dependent on exactly which protocol is used, as one can obtain a proxy with ProtocolHandler.proxyForUrl(url) without caring about the actual protocol used. Published objects can easily be used as actors or as communicating serial processes.

One still has to carefully think about which objects/interfaces to expose, and which action to take when a connection fail to create a robust system. If the failure happens when sending or receiving then an exception is raised, and the caller should cope with it, but a partial failure is also possible where messages are lost due to the crash of the remote party without any notification. There is a facility (FailureManager) where call backs and url can be registered, and if the system detects the failure of a remote host it will trigger appropriate actions, but in any case one should to design the protocol in such a way that it can handle such unexpected failures.

Dchem uses the stcp rpc protocol to create a client/server setup, where clients get work to do from a server, and one can connect to the server during the computation, query about the current status of the computation and possibly change the kind of explorations performed.

5 Current implementation

5.1 Technical aspects

The primary goal of this paper has been to discuss an API that helps the programmer, enabling him or her to express his algorithm as clearly as possible, while being flexible enough to handle all common cases. A secondary goal is also to give an open source implementation of this API that can be used now.

Technically all the interfaces presented here have been implemented in blip library `http://dsource.org/projects/blip` released under the Apache 2.0 license and written in the D programming language version 1.0 `http://www.digitalmars.com/d/1.0/index.html`. D builds upon C like C++, but does not try to be fully backward compatible. D has templates, classes, interfaces

and modules (no .h files), and is garbage collected but also support pointers and assembler statements.

There are several opensource D 1.0 compilers: dmd, the original D compiler; gdc, which uses the gcc backend; and ldc, which uses the llvm backend. Using the ldc compiler on linux the support for x86_64 architecture is very good, other architectures can also be used, but will probably need more effort.

As standard library tango `http://dsource.org/projects/tango` is used. For the development of the library it was crucial to have access to the source of the runtime of D, to be able to remove some bugs related to threading, fiber handling or GC interaction, some of which were located in the D runtime. The library was used both on mac with the dmd compiler and on linux using the ldc compiler.

5.2 SMP parallelization implementation

The whole implementation is in userspace, and assumes that the application has almost exclusive use of some processors (by default all processors usable by the application). For each of these cores a deque based queue is lazily allocated when needed, along with worker threads that if possible are pinned to that core. Each element of the deque is a priority queue that represents a recursive task. There is an object (StarvingManager) that keeps a list of the starving locked threads and redirects tasks to them if needed, and coordinates the stealing of tasks. Thanks to that threads that cannot steal any task wait on a mutex, using no CPU.

Though the StarvingManager one can directly access the queues of the various processors, the locality information and if needed add tasks by hand. Stealing uses locality information provided by hwloc `http://www.open-mpi.org/projects/hwloc/` to build a structure in which it is possible to loop on all neighbors of a given level in random order. This is used to try to steal from neighbors first and to redirect tasks to the closest idle neighbor. A thread local variable is used to keep the current task, and another to keep the current cache. the cache is a numa node shared object that can be used to keep cached value and free pools. The use of the cache allows one to avoid the expensive allocations, and still guarantee the use of memory in the current numa node.

By default all tasks have their own fiber (java green thread), which means that the have their own stack in an heap allocated region. To support all kind of programs the stack has to be large enough (unless one uses a segmented stack as the go language does), the actual size depends on the application used, the current default is a very generous 1MB. This obviously limits the number of executing tasks that one can have. Another drawback of always using a separated fiber for each task is that the memory is less local. To reduce these problems the current implementation is very aggressive in reusing fibers. A big advantage of using dedicated fibers is that one can suspend and resume any task independently, thus if one never locks a thread delaying the tasks that have to wait then the executing threads will never stall, and one can have any kind of dependencies between the tasks. Obviously a circular dependency would mean

that those tasks will never be resumed.

This approach works well most of the time. Most of the time is not always, sometime one needs to ensure that a task stays on one processor, or one might want to start with a specific task distribution that was calculated using information of the numa topology. The advantage of realizing everything as a library and with normal objects is that this is easily possible. For example it is possible to insert tasks by hand in the various queues, to better take advantage of memory affinity of a data set (the normal scheduling algorithm only takes into account a good cache hierarchy performance).

At least for the HPC field, where an application can safely assume that she is the sole user of all the processors, the current implementation works well, and avoids excessive polling when idle.

5.3 Performance

A very important aspect of a parallelization scheme that aims at being used in practice is its performance. A nice and conceptually clean approach will not be used if its performance is not reasonable. In general newer systems are more challenging, because the gap between CPU and memory tends to widen.

We performed our benchmaks on a two socket system using Intel Xeon X5570 @ 2.93GHz system with 141GB of RAM. The system has two times 4 cores, each having hyperthreading, giving a total of 16 logical processing units.

5.3.1 SMP

To benchmark the SMP parallelization, to do it we calculate a gaussian function $exp(-\alpha_i(x^2+y^2+z^2))$ using arrays containing random numbers between 0 and 1 for α, and -1 and 1 for x, y and z We calculated 10'000'000 gaussians with tasks of 16'000 or 4'000 gaussians, using the standard parallel loops on arrays of the blip library (pLoopArray). The code of the test is distributed together with the library in the tests directory, and was compiled with the ldc compiler using the -inline -release -O2 flags (having a good scaling is more difficult with optimized code). This benchmark uses quite a bit of memory bandwidth, because the amount of computation for each element is still relatively small, so the scaling will be limited if the system has insufficient memory bandwidth.

All measurement were repeated 6 times, and the average of the best two times is reported. These numbers were well reproducible. The memory was initialized on purpose by a single thread, thus numa locality is not explicitly managed, as is expected to be the case in most situations, and the test stresses the whole memory hierarchy. When reducing the number of processors, processors as separated as possible were used.

As can be seen from the plot1 the scaling is fairly good up to 8 processors, but hyperthreading does not really help much for this kind of floating point and memory intensive code, because the single core does not have the resources to work on two threads in parallel. Using 4 times smaller tasks gives almost

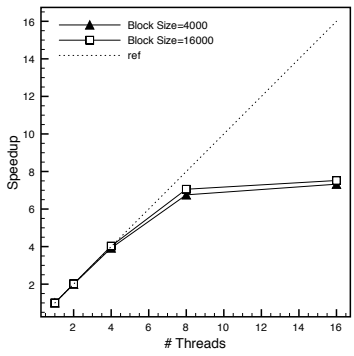

Figure 1: Speedup (refTime/time) of working with of blocks of 4000 Gaussians (triangles) or 16000 Gaussians (squares) using the time with 16000 Gaussians on 1 thread as refTime. The dotted line is the ideal scaling. As can be seen hyperthreading (8→16)does not scale, but the rest scales fairly well. The scaling with blocks of 16'000 is close to the maximum scaling that can be acheived, with 4'000 the scaling begins to degrade

exactly the same results, both in absolute times and speedup even if calculating 4'000 gaussians takes only $\sim 160\mu s$.

This example shows that for tasks that take a few hundreds of μs, are independent and reasonably memory friendly the scheduling overhead is small. To quantify this overhead we look at the parallel calculation of the fibonacci sequence using the basic iterative procedure for $n < 2$: $fib(n) = fib(n-1) + fib(n-2)$, spawning a new task for both $fib(n-1)$ and $fib(n-2)$. This generates lot of tasks, and as the operation is so simple almost all the time is spent in task handling.

fib(22), that spwans 28'657 tasks, takes $0.189s$ on two processors, which means $\sim 7\mu s$ per task. Increasing the number of processors increases the likelihood of contention, and with 16 processors one has in average a 2.5 times more overhead. With respect to tasks of hundreds of μs, which we don't expect to be difficult to generate in typical HPC programs, this overhead is still small and perfectly acceptable.

The nice scaling shown for the calculation of the gaussians does not hold if the tasks have to contend for a single lock. Condition locks and semaphores aware of the SMP scheduling, an event based approach, and dataflow variables as offered by the blip library can help avoiding these problems.

The good performance shown does not mean that it is not possible to improve it further, this is the first version of the library, focused on correctness, not on speed. Indeed the basic fiber switch time is one order of magnitude smaller that the scheduling overhead, showing that there is still optimization potential.

Another place where the current implementation can be improved is memory

allocation. The current D implementation uses a global lock for allocation and garbage collection. Due to this for example the tests of the multidimensional array NArray that allocate several arrays for each test, and where test itself is very often very quick (a vector addition for example) are dominated by the allocation, and as that is serial the NArray tests do not scale with the number of processors, they even get a bit worse due to the contention of the GC lock. There is work to improve the GC and allocation runtime of D, but fortunately typical hpc tasks are already organized in such a way that they do most of the allocation outside the computational intensive part, or can use the numa aware cache pools.

5.3.2 Rpc

As the MPI part is just a wrapper around an MPI implementation it does not make much sense to benchmark it. The remote procedure call part on the other hand warrants some benchmarks. As rpc uses the normal tcp protocol as soon as the computers are separated the network latency will dominate all results, and the timings will quantify the performance of the network and the tpc/ip implementation rather than our implementation. To give an idea of the different costs involved several simple functions taking or returning few integers or doubles are called in the testRpc program. Almost all the functions return, only 1 in few hundreds of function calls is a oneway function.

There are different possible distances, the first is a simple local call. Non oneway methods calling the same function trough a local proxy cost 1.6 times as much as a local call whereas if the arguments and the result are serialized the costs grows to 2.6 times as the local call. Oneway methods are around 500 times as costly using a proxy, because they spawn a separate task. Finally a call using a socket, and libev to send and receive the serialized arguments on the same computer (using the loopback interface) costs 10'000 times a local call.

The very large latency of even the loopback interface is a good reminder of the fact that a program has to be thought in a way that takes in account the latency. An automatic parallelization at this level cannot be expected to perform well, unless the tasks have very little dependency. The network connection to a separate host, even using a fast interconnect has an even larger latency, 200 times the loopback interface on our machine, being around $4.5ms$.

6 Conclusions

We have presented an integrated approach to parallelization that is a refinement of several used paradigms. At SMP level it can handle both recursive or data parallel tasks optimizing the throughput and independent requests keeping the latency within bounds. This API can be used well in event based programming, is simple to reason about, so that a programmer can reason about the expected efficiency, while still allowing an efficient implementation. It is argued that this approach is a good basis for several parallelization models.

The value of the MPI model is discussed, and that part is adopted basically unchanged.

To address some inflexibilities of MPI, especially with respect to adding and removing processes a final rpc parallelization level, where resources are identified with urls, is described. The rpc level can use SMP parallelization optimally using independent recursive tasks for each request. Rpc can use either local function calls, the MPI protocol or the stcp protocol (based on tcp/ip sockets), and potentially other protocols. Using the stcp protocol one can add and remove processes to a computation, but it is up to the programmer handle the complexity of what to do when this happens.

It might seem unsatisfying to have these different levels, but as argued in Kendall et al. (1994) parallel computing has inherently more complex than normal programming because it has to keep into account other problems, not just latency and memory access, but also partial failure and concurrency. The presented levels mirror and increase of complexity:

- the SMP level exposes a conceptually simple shared memory multithreading model that can be efficiently evaluated and on the top of which most SMP models can be realized
- the MPI level introduces distributed memory, but keeps the reliability and global failure characteristics of the single process model, and most collective communication modes avoid race conditions
- rpc has all the complexity of a parallel program, and while it can lift from the programmer the task of writing serialization, connection creation and low level protocol, it remains up to the programmer to choose how to cope with partial failure and concurrency.

To keep the complexity of the rpc level (which one can use like the actor model) manageable, one can restrict himself to several pattern: client/server, map/reduce, p2p. In some cases dedicated software (like grid computing packages, or google map/reduce) might be a better choice, but the discussion of these approaches goes beyond the scope of this paper.

Clearly programming at the rpc level is more complex, but it also gives more power, one should not have to write all his application at that level, but *if* one has made the effort to write part of a program at a given level it should pe possible to use it with less distribution without too much effort, and overhead. So the MPI wrapping also have a local class that implements the full MPI interface locally for a single process MPI without resorting to any MPI library call. Likewise the rpc url and calls also work locally. This way a fully parallel program can be used also on a single machine, simplifying the work of the programmer.

The whole implementation is available as part of the blip library which is released under the liberal open-source Apache 2.0 license. This will hopefully encourage its adoption, and improvement by others. The author wishes to thank the Alexander von Humboldt foundation, the Hlrn computing facilities, and Prof J. Sauer for supporting part of this work.

References

Acar, U., G. Blelloch, and R. Blumofe (2002). The data locality of work stealing. *Theory of Computing Systems 35*(3), 321–347.

Allen, E., D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. S. Jr., and S. Tobin-Hochstadt (2008). *The Fortress Language Specification*. Sun Microsystems.

ARB (1997). *OpenMP Fortran Application Program Interface version 1.0*.

ARB (2008). *OpenMP Application Program Interface Version 3.0*. OpenMP Architecture Review Board.

Armstrong, J., R. Virding, C. Wikström, and M. Williams (1996). *Concurrent Programming in Erlang* (2. edition ed.). Prentice Hall.

Blelloch, G., J. Sipelstein, J. Hardwick, and M. Zagha (1995). Nesl user's manual (for nesl version 3.1). *oai.dtic.mil*.

Blumofe, R., C. Joerg, and B. Kuszmaul (1995). Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 207–216.

Burton, F. and M. Sleep (1981). Executing functional programs on a virtual tree of processors. *FPCA '81 Proceedings*.

Chamberlain, B., D. Callahan, and H. Zima (2007). Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications 21*(3), 291.

Cong, G., S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen (2008). Solving large, irregular graph problems using adaptive work-stealing. *Proceedings of the 37th International Conference on Parallel Processing*.

Dongarra, J., R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. Mcmahon, A. Snavely, J. Vetter, and K. Yelick (2008). Darpa's hpcs program: History, models, tools, languages. *Advances in Computers 72*, 1–100.

El-Ghazawi, T., W. Carlson, and J. Draper (2005). Upc language specifications v1. 2. *UPC Consortium*.

Forum, M. P. I. (1995). *MPI: A Message-Passing Interface Standard 1.1*. University of Tennessee.

Forum, M. P. I. (2009). *MPI-2.2*. University of Tennessee.

Frigo, M. and V. Strumpen (2006). The cache complexity of multithreaded cache oblivious algorithms. pp. 271–280.

Hoare, C. (1978). Communicating sequential processes. *Communications of the ACM*.

intel (2010). *Intel(R) Threading Building Blocks, Reference Manual*. 1.20.

Jr, H. B. and C. Hewitt (1977). The incremental garbage collection of processes. *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, 55–59.

Kambadur, P., A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine (2009). Pfunc: modern task parallelism for modern high performance computing. *Proceedings of SC'09*.

Kendall, S., J. Waldo, A. Wollrath, and G. Wyant (1994). A note on distributed computing. *Sun Microsystems, Inc. Mountain View, CA, USA*.

Numrich, R. and J. Reid (1998). Co-array fortran for parallel programming. *ACM Sigplan Fortran Forum 17*(2), 1–31.

Roy, P. V. and S. Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press.

Saraswat, V., B. Bloom, I. Peshansky, O. Tardieu, and D. Grove (2010). Report on the programming language x10. *http://x10.codehaus.org/*, 1–231.

Stevens, W. R. and S. A. Rago (2005). *Advanced programming in the Unix environment*.