CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Driving asynchronous distributed tasks with events

Citation for published version:
Brown, N, Thomson Brown, O & Bull, J 2018, 'Driving asynchronous distributed tasks with events', Paper presented at 4th Workshop on Open Source Supercomputing, Dallas, United States, 12/11/18 - 12/11/18. <https://sc18.supercomputing.org/proceedings/workshops/workshop_pages/ws_opensu102.html>

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 22. Sep. 2020

Driving asynchronous distributed tasks with events

Nick Brown
EPCC,
The University of Edinburgh
Email: n.brown@epcc.ed.ac.uk

Oliver Thomson Brown
EPCC,
The University of Edinburgh

J. Mark Bull
EPCC,
The University of Edinburgh

***Abstract*—Open-source matters, not just to the current cohort of HPC users but also to potential new HPC communities, such as machine learning, themselves often rooted in open-source. Many of these potential new workloads are, by their very nature, far more asynchronous and unpredictable than traditional HPC codes and open-source solutions must be found to enable new communities of developers to easily take advantage of large scale parallel machines. Task-based models have the potential to help here, but many of these either entirely abstract the user from the distributed nature of their code, placing emphasis on the runtime to make important decisions concerning scheduling and locality, or require the programmer to explicitly combine their task-based code with a distributed memory technology such as MPI, which adds considerable complexity. In this paper we describe a new approach where the programmer still splits their code up into distinct tasks, but is explicitly aware of the distributed nature of the machine and drives interactions between tasks via events. This provides the best of both worlds; the programmer is able to direct important aspects of parallelism whilst still being abstracted from the low level mechanism of how this parallelism is achieved. We demonstrate our approach via two use-cases, the Graph500 BFS benchmark and in-situ data analytics of MONC, an atmospheric model. For both applications we demonstrate considerably improved performance at large core counts and the result of this work is an approach and open-source library which is readily applicable to a wide range of codes.**

I. Introduction

More and more, HPC is demonstrating to new communities the benefits it can provide. From machine learning and data analytics to interactive supercomputing, these represent important new application areas that heavily rely on open-source software and can benefit greatly from the power of supercomputing. Whilst traditional HPC users were often more willing to accept closed source tools such as compilers and libraries, these new generations of application areas have often grown up in the open-source community and as such expect a level of openness that HPC has not required before. But these new use-cases bring with them their own challenges and issues, for instance many of these new are not naturally regular and the traditional approach of writing bulk synchronous parallel codes, that arguably MPI has excelled at over the past 25 years, is no longer applicable. With a focus on using open-source to build a complete system from the hardware upwards, it is not just the applications that are driving future challenges in parallel programming. There are a generation of new open-source HPC hardware technologies, such as the RISC-V [1] architecture, which will likely mean that future open-source systems will expose significant amounts of parallelism that the programmer must exploit. In order to be able to take advantage of these future machines it will be very important for more traditional computationally based applications to scale to many hundreds of thousands, if not millions, of cores, and to do this one needs to break apart the bulk synchronous nature of these codes.

It is our view that programming models which encourage and support the programmer to write highly asynchronous codes will be of great use in the future. We actually see this sort of approach already in other fields, for instance web developers leverage AJAX to make asynchronous calls from a website to a server and *at some point* their callback will be executed with the resulting data. In the mean time their code can be executing other functionality and this ability to work out of lock-step avoids excessive idle time and maintains user interactivity. However, not least because web-browsers tend to only run a single thread, HPC is far more complex than web-browsers and a key question is how best to achieve this whilst abstracting the programmer from all the low level concerns of parallelism which they must currently face with technologies like MPI. It is already fairly common for developers of asynchronous applications to write their own, bespoke, communication layer, but this is not ideal and they are forced to do this due to the inadequacies of the programming technologies. Instead, following the philosophy of the open-source community, this should be abstracted away in a reusable library such that developers are free to concentrate on their application, with the underlying technology efficiently and seamlessly supporting the asynchronous nature of their parallel code.

Task-based models are seen as a possible solution, where the programmer splits up the computation into distinct tasks that are eligible for execution once a number of dependencies have been met. A big benefit of this is that the underlying mechanism of parallelism is abstracted, so that the programmer can concentrate on their application without worrying about the tricky low level details. Task-based models also encourage programmers to break their code into loosely coupled, distinct, components which promotes asynchronicity. However, a major limitation is that these models have traditionally been grounded in a shared memory assumption which significantly restricts the problem sizes that they can address. Whilst there is effort under way to apply task-based models to distributed memory machines, many still maintain this shared memory abstraction, which places great emphasis on the runtime to

make the best decisions in matters impacting parallelism or require the programmer to explicitly mix distributed memory technologies such as MPI.

In this paper we present a new, open-source, approach to task-based models, where the programmer still works with the concept of tasks but is explicitly aware of the distributed nature of their code. It is our belief that, by making explicit the distribution, programmers can gain the best of both worlds - an ability to architect their parallelism based upon a realistic view of how this will execute so as to optimise performance, complete with a task-based model abstracting them from the low level details of how this interaction takes place. It is our hypothesis that this encourages and supports the programmer to write highly asynchronous codes, where the developer can set up interactions but does not need to worry about exactly when these will take place in their code.

This paper is organised as follows; Section II describes our approach in detail and provides a simple code example to further flesh out the concepts. In Section III we discuss related work currently being undertaken by the community, and compare and contrast this with our approach. Section IV introduces more advanced aspects of our approach to ensure generality to a wide range of applications. To give the reader a flavour of the generality of our approach we demonstrate two use-cases in sections V and VI. An EDAT implementation of the BFS kernel from the Graph500 benchmark is studied in Section V and the in-situ data analytics of an atmospheric model, MONC, is explored as a short case-study in Section VI. Lastly, in Section VII we draw conclusions and discuss further work.

II. Event Driven Asynchronous Tasks (EDAT)

Our approach is known as Event Driven Asynchronous Tasks (EDAT) and a library has been developed to implement and test these concepts. The idea is to provide the programmer with a realistic view of the distributed nature of their code but to still abstract them from the low level details of how data is communicated by using a task-based model. The programmer is still working with the view of multiple nodes, each containing a distinct memory space and a number of CPU cores that can execute code. Hence in this paper we will use the terminology of process, a process being a distinct memory space running over one or more cores (such as a NUMA region) and these contain a number of *workers*. There are two key concepts in EDAT:

- **Tasks** are represented by the programmer as a function and are submitted to EDAT for running locally once a number of (event) dependencies have been met. The submission of a task is non-blocking, and once the dependencies have been met the task is eligible to run and moved into a *ready queue* for execution by a *worker* when one becomes available. It is common (and the default behaviour) to map one *worker* per CPU core, but this is configurable by the user. Tasks can be nested, where a task submits other tasks.
- **Events** are the major way in which tasks and processes interact. An event is fired from one process to another (which can be the same process) and are labelled with an event identifier (EID). Events may contain some optional payload data which the task that consumes it is then able to process. The firing of events is non-blocking, and data itself is copied into an internal buffer such that once an event has been fired, the programmer can modify the data later on in their code without any worry of inconsistency.

Whilst events might seem similar to messages, crucially the way in which events are sent and delivered to tasks is entirely abstracted from the programmer. The programmer need not care about the low level details of how events are physically sent, handling events when they arrive, how these are mapped to the dependencies of tasks, or how tasks are queued up. Programmers work with the concept of tasks, but are explicitly aware of the distributed nature of their application and drive interactions through events.

A. Submission of tasks

Tasks are submitted to EDAT for execution by the programmer and this can be done at any point in the code. Listing 1 illustrates the API call, where *task_function_ptr* is a function pointer to the task itself. The *num_event_dependencies* argument is the number of event dependencies that must be met before this task is eligible for execution. After this argument, for each event, the programmer provides the source rank (process ID) of where that event originates from, and a string event identifier which is depicted as *<int event_source, char * event_identifier>* in Listing 1. The programmer can view this as a task consuming events that match the task's dependencies.

```
void edatSubmitTask(task_function_ptr, int
    num_event_dependencies, <int event_source, char *
    event_identifier>...)
```

Listing 1: Submitting a task to EDAT

Listing 2 illustrates the function signature of a task, where both the number of events (*number_of_events*) that a task has been executed with, and an array of structures containing each event, *events*, are passed to the task function as arguments. The *EDAT_Event* structure contains the optional payload data and also metadata about the event, including the source rank, string event identifier, data type of payload data and number of elements in the payload data. The library guarantees that the order in which events are delivered to a task matches the dependency order that the programmer expressed when submitting the task. Based on this the programmer can make assumptions about the order of events in the task's array argument and need not search through event metadata, which could slow down tasks with large numbers of dependencies.

```
void task(EDAT_Event * events, int number_of_events) {
  ....
}
```

Listing 2: Task function

EDAT also provides some inbuilt constants that the programmer can use for an event's source argument. For instance *EDAT_SELF* specifies that the event should originate from the same rank and *EDAT_ANY* is a wild-card where the task dependency will be met based upon an event with a matching string identifier from any source process.

B. Firing of events

Events are fired from a source process to a target process, and this firing of events can occur at any point in an application. Events serve two purposes; firstly they contribute to the activation of tasks by matching dependencies of those tasks; and secondly they provide the ability to communicate data between tasks if required. We promote the notion of *fire and forget* on behalf of the programmer, so the EDAT library makes a copy of any payload data during this call, and the user's code will continue to execute regardless of whether the event has been delivered, or of the state of the corresponding task. The programmer can immediately reuse the data passed to the event, and once fired there is no way for the programmer to detect the status of an event or cancel it. Instead there is simply the guarantee that fired events will, at some point, be delivered to the target rank.

```
void edatFireEvent(void* data, int data_type, int number_elements,
      int target_rank, const char * event_identifier)
```

Listing 3: Firing an event

Listing 3 illustrates the API call for firing an event where *data* is optional payload data to send, *data_type* is the type of this data (EDAT provides in-built types including integers, booleans, floats and doubles), and *number_elements* the number of elements contained in the data. The event will be fired to the *target_rank* (which can be *EDAT_SELF*) and the string *event_identifier* is the identifier used to match up this event with dependencies of outstanding tasks.

The ordering of events and task dependencies is deterministic and EDAT maintains a strict ordering of events fired from a target to source process. This guarantees that events will arrive at a target process in the same order in which they fired from a source process. The order in which tasks consume events is determined by the order in which they are submitted to EDAT; a task submitted before another task on a process has a higher precedence in the consumption of events.

C. Bringing these ideas together: A simple example

In designing EDAT it was our intention to keep the concepts and API as simple as possible, and the calls described so far are sufficient to develop an irregular parallel application. Listing 4 illustrates a simple parallel EDAT code running with (at least) two processes. As can be seen the programmer typically writes their code SPMD style and EDAT provides a *edatGetRank* call to obtain the current rank which can be used to specialise which processes execute what code. In this example, at line 7, Process 0 submits the task, *task1*, which has no event dependencies (the second argument given to *edatSubmitTask* is zero) and so this task will be immediately eligible for execution by an idle worker on that process.

Process 1 submits two tasks, *task2* at line 9 which depends on a single event from Process 0 with event identifier *event1*. *task3* is submitted at line 10 with two event dependencies, an event from Process 0 with identifier *event2* and an event from Process 1 with the identifier *event3*.

As *task1* (lines 16 to 20) executes on Process 0, two events are fired. The first event, fired to Process 1 with the identifier *event1* at line 17 contains no payload data (*EDAT_NONE* as the datatype). This event corresponds to the dependency of *task2* on Process 1 and as such *task2* will be eligible for execution by an idle worker as soon as this event arrives. The second event fired by *task1* to Process 1, line 19, contains a single integer as payload data and this event corresponds to the first dependency of *task3*. The other dependency of *task3* is fired at line 24 by *task2* running on Process 1 to itself (note the *EDAT_SELF* target.) Once both these events have arrived on Process 1 then the dependencies of *task3* have been met and this task will be eligible for execution. The task itself extracts out the payload data from each event, adds these integers together, and prints the result.

```
1  #include "edat.h"
2  #include <stdio.h>
3
4  int main() {
5    edatInit();
6    if (edatGetRank() == 0) {
7      edatSubmitTask(task1, 0);
8    } else if (edatGetRank() == 1) {
9      edatSubmitTask(task2, 1, 0, "event1");
10     edatSubmitTask(task3, 2, 0, "event2", 1, "event3");
11   }
12   edatFinalise();
13   return 0;
14 }
15
16 void task1(EDAT_Event * events, int num_events) {
17   edatFireEvent(NULL, 0, EDAT_NONE, 1, "event1");
18   int d=33;
19   edatFireEvent(&d, 1, EDAT_INT, 1, "event2");
20 }
21
22 void task2(EDAT_Event * events, int num_events) {
23   int d=100;
24   edatFireEvent(&d, 1, EDAT_INT, EDAT_SELF, "event3");
25 }
26
27 void task3(EDAT_Event * events, int num_events) {
28   printf("%d\n", *((int*) events[0].data) + *((int*) events[1].
        data));
29 }
```

Listing 4: Simple EDAT example code

Figure 1 illustrates these same tasks and interactions of the code from Listing 4. The programmer determines what interacts with what and based upon this they don't need to worry about how events are physically delivered or when, just that they will and tasks depending upon these events will execute when they arrive. It can be seen from this example that our

approach encourages the programmer to write asynchronous, loosely coupled, code as there is no explicit synchronisation or assumptions when events will arrive. Instead the programmer is submitting tasks based upon a number of events arriving in the knowledge that, at some point, these tasks will execute but they need not care about exactly when or impose limitations on the progress of other processes.

If one was to try and achieve the same thing in a technology like MPI directly then they would need to address many more low level concerns, such as explicitly testing for messages, carrying around numerous request handles, explicitly mapping functions to threads and repeatedly checking on progress.

Hence in our approach the programmer is much more of an architect of parallelism; they are designing what aspects of their code interact, but not how this actually happens. As many of the performance issues with codes are caused by a poor high level design [2] we believe that this approach of supporting the programmer writing distributed, asynchronous codes in a manner that they can concentrate on the high level design is valuable. Given enough tasks, the overhead of sending many small messages, which can be unavoidable with irregular applications, can also be mitigated with an asynchronous task-based approach such as ours because message latency can be effectively hidden.

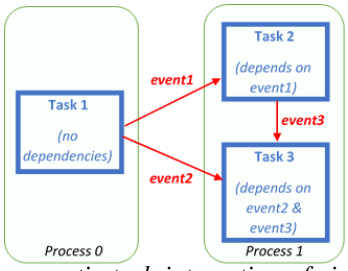

Fig. 1: Diagrammatic task interaction of simple example

D. Collective dependencies and events

The interactions described until now have been point to point, sent from a distinct source process to a distinct target process. EDAT also provides the *EDAT_ALL* constant that can be used for source and/or target ranks. When used as the target rank in firing events it will send that events to all processes (i.e. a broadcast), when used as a dependency in submitting tasks this requires events from all processes in order to meet that dependency. Listing 5 sketches out a naive reduction in EDAT, where a task is submitted to run when an event with identifier *event* is received from all processes (including itself.) Once this dependency is met the task will be mapped to an idle worker for execution and in the example here the task sums up integers from each process (in the payload data of each event.) Whilst in reality the programmer would most likely adopt a more complex collective algorithm, such as a tree-based approach, this would work equally well with EDAT and the simple example here is for illustration.

```
edatSubmitTask(task, 1, EDAT_ALL, "event");

void task(EDAT_Event * events, int num_events) {
  int total=0;
  for (int i=0;i<num_events;i++) {
        total+=*((int*) events[i].data);
  }
}
```

Listing 5: Reduction tasks

By combining collective events with collective task dependencies, one can build a barrier as illustrated in listing 6. Because the submission of tasks and firing of events is non-blocking, crucially this is a non-blocking barrier, and the task will not execute on a worker of any process until all processes have fired their corresponding events and these have been received. Therefore the execution of this task represents that each process has reached some logical point in the code by all processes. One use for this is when interacting with external, blocking collective calls (such as parallel NetCDF) where when individual processes call a function they block until all other processes have issued a corresponding call. This can result in excessive wait time, so instead an asynchronous barrier can be used at this call point and the barrier task is then executed at similar times on each process (assuming there are idle workers) which calls the blocking collective function and effectively reduces the time a process is blocking for other processes to issue the corresponding call.

```
edatSubmitTask(task, 1, EDAT_ALL, "event");
.....
edatFireEvent(NULL, EDAT_NONE, 0, EDAT_ALL, "event");
```

Listing 6: Barriers in EDAT

E. Termination

The *edatFinalise* call of listing 4 will put the main thread to sleep and wait until the code has finished executing before terminating the program. Four conditions must be met on all processes before EDAT terminates:

1) There are no outstanding submitted tasks waiting for their dependencies to be satisfied
2) All tasks eligible for execution have been full executed and all workers are now idle
3) All events have been delivered to their target process
4) All events have been consumed by tasks (i.e. there are no outstanding events waiting to satisfy the dependencies of a task not yet submitted).

Once these four conditions are met on every process then EDAT will terminate. Of course the programmer is entirely abstracted from the mechanism of this, but should be aware that submitted tasks need to run and events need to be consumed before their code can terminate.

F. Implementation

We wrote the EDAT library in C++ and the library itself is fairly small, at around 4000 lines of code. The scheduler follows a First In First Out (FIFO) policy for executing tasks, which are queued up when they are submitted. The underlying transport layer is designed to be pluggable, currently, due to

its ubiquity, there is an MPI based implementation and other mechanisms can be easily added. Progress polling for aspects such as new events and termination can either be carried out by a dedicated thread or an idle worker. In the later case, polling is mapped to an idle worker but this worker prioritises task execution and polling will be *swapped out* in preference to running a task if there are no other idle workers available.

III. Related work

As a community, many of our parallel programming tools, such as MPI, are best suited to applications where the parallelism is predictable and regular. In cases where interactions and application behaviour is itself unpredictable and irregular, then these existing tools start to come unstuck. The asynchronicity of codes that is now required by these developments means that existing paradigms, whilst they have served us well in a regular, synchronous world, are more and more becoming a hindrance to the programmer and are not suited to the communication patterns of a growing number of applications. The reader can view our approach as a combination of the event based co-ordination pattern [2], with concepts from the task-based models paradigm to support the programming of asynchronous, large scale, parallel codes.

A. Task-based models

One possible solution to programming irregular applications are task-based models and typically these models commonly present themselves to the programmer via pragmas or a library API. Technologies such as OpenMP tasks [3] and OmpSs [4] require the programmer to annotate their code with pragmas which are then processed by the compiler. In these approaches the programmer expresses their dependencies based upon input and output variables and the compiler and runtime is able to generate a task dependency graph based upon this information. A task with specific input dependencies can not run until other tasks with these same dependencies as outputs have executed. In this manner the programmer is splitting their code up into these distinct, loosely coupled, work units and directing the order in which they can run based upon their variables (including function arguments.)

Alternatively the model presents itself entirely as an API, whether it be a function library or other abstraction such as C++ templates. Legion [5], High Performance ParalleX (HPX) [6] and the Open Community Runtime (OCR) [7] are examples of this approach. One benefit here is that there is no need for a specialist compiler, but many of these library approaches require the programmer to learn significant new concepts and techniques, before undertaking large scale code re-factoring. Fundamentally this is because with the pragma approach there is a compiler that can extract out sufficient information, whereas with libraries this information must be supplied directly by the programmer at runtime, and technologies such as Legion result in fairly verbose code. Whilst EDAT is a library based approach, we have been careful to limit the number of new concepts and API calls that the programmer need learn to effectively use the technology.

Irrespective of how the task model it is presented, interoperability between shared memory and distributed memory technologies is a challenge when one wishes to scale task-based codes. There is a significant increase in complexity when one addresses distributed memory architecture, including challenges of optimal data placement, efficient communication and task locality. Broadly, there are two common approaches; either to place the burden of these challenges onto the runtime and keep it simple for the programmer with the programming interface largely unchanged (for instance OmpSs has adopted a directory/cache [8]), or place the burden on the programmer and require them to explicitly mix the task-based and distributed memory (such as MPI) technologies.

Whilst the nature of pragma based approaches means that they invariably heavily rely on the concept of variables in their abstraction (which makes moving away from a shared memory abstraction challenging), library based models also commonly hide any distribution. For instance OCR [7], an asynchronous task-based runtime, has the notation of event-driven tasks. However their concept of event driven tasks and our approach is very different. OCR abstracts data migration across distributed memory spaces via data blocks which contain an event that maps to task dependencies. Crucially here, though, the programmer has limited control and visibility over the locality of these data blocks, not least because the placement of OCR tasks, which can only generate one single event, is entirely abstracted by the runtime. This lack of transparency is compounded by the fact that OCR is not intended to be used directly by the application developer but instead as a lower level building block supporting higher level task-based models. This lack of visibility for the programmer, where it is not necessarily obvious how their data is distributed and the cost of communication related to this, limits opportunities for optimisation. Indeed EDAT demonstrated superior performance and scaling in contrast to OCR up to 32768 cores with Intel's stencil 2D benchmark in a earlier work [9].

B. Active messaging

Another attractive approach to abstracting asynchronicity from the programmer is that of active messaging [10], where a sender provides not only the data but also a message handler to be executed by the receiver once the message arrives. In HPC the main success of this approach has been at the lower level transport layer, such as GASNet [11], instead of technologies that are designed to be accessible to the application developer.

A number of active message end user libraries have been developed, for instance AM++ [12] and AMMPI [13], but these have deficiencies which limits their utility for irregular applications and uptake by the community has been poor. For instance AM++ is bulk synchronous in operation and communications take place only in epochs. Neither technologies utilise background threads for monitoring the sending or receiving of messages and the user has to handle this manually (either by calling the poll function directly, creating a thread to do this or in AM++ when the epoch ends.) These active

messaging technologies are not multi-threaded and as such there is no model to execute message handlers concurrently.

To some extent the reader can view our approach as a combination of task-based models with some of the ideas from active messaging. Unlike the active messaging technologies, our approach is more abstract where EDAT events contribute towards the activation of code and can be composed together before a task will run. Charm++ [14] is potentially closest to our approach where the programmer writes their code in C++ and defines parallel objects, called chares. Conceptually these chares are objects and contain a number of entry methods that can be invoked remotely by calling methods on proxy objects on other cores. There are a number of key differences between our approach and Charm++:

- In Charm++ the programmer must write their code Object Oriented (OO) style in C++ with extra boilerplate *proxies*. This often requires completely rethinking and redesigning their application in OO C++. Charm++ is more of an ecosystem, rather than a task-based model, and as such there is no way of gradually adopting Charm++. Instead codes needs to be ported in a *big-bang approach*, this is not the case for EDAT which is language agnostic.
- In following the OO model, Charm++ limits the programmer to sending data from one process to another process when instantiating a remote method (effectively a task). It is possible to get round this with a Charm++ *reduction*, but that sits outside the basic abstraction and adds extra complexity. Instead, in EDAT, events from any number of different processes can contribute to the execution of a task (the task dependencies), it's that simple.
- The Charm++ model provides a default *one-at-a-time* execution model, where only one method of an object can be executing at any one time. It is possible to override this but then the programmer is on their own. In EDAT we take the opposite view, and whilst it is possible to structure the submission of tasks for this *one-at-a-time* approach this is not imposed. Additionally EDAT provides finer grained synchronisation controls, which will be described in section IV-C integrated with EDAT to provide richer control.
- There is no way in Charm++ to drip feed data to remote methods (tasks) and as such all communication must take place before a method is invoked. Instead in EDAT, as we will see in section IV-B, it is possible to start tasks based upon a minimal number of dependencies and then test or wait for the arrival of further events. This provides more flexibility to the programmer in controlling task granularity.
- Whilst the Charm++ source code is available, the licence is much more limiting than those more commonly associated with open-source software. For instance commercial use of Charm++ requires the purchase of a licence, any user modified version must be clearly marked and renamed to notify users that it is not the original software, users can only re-use 10% of the source code without obtaining a further licence and code developed using Charm++ must cite specific papers. In short, whilst the source code of Charm++ is freely available the licence restrictions means that it feels a long way from the freedoms enjoyed by open-source software such as our own EDAT library.

By bringing in the ideas and concepts from the task-based paradigm and driving tasks with events we believe that this achieves the broad aims of active messaging but in a clear and unambiguous manner.

IV. Advanced EDAT concepts

Whilst the techniques described in Section II are sufficient for many applications, a number of more advanced concepts are useful for specific applications.

A. Persistent tasks and events

The tasks and events discussed up until this point are transitory. This means that when a task's dependencies are met it is then removed from the *outstanding* queue and hence the task only executes once. Likewise; when transitory events are consumed, then they are gone. However, it is also possible for both tasks and events to be persistent, where tasks can run any number of times and events consumed any number of times.

```
void edatSubmitPersistentTask(task_function_ptr, int
    num_event_dependencies, <int event_source, char *
    event_identifier>)
```

Listing 7: Submitting persistent tasks to EDAT

Listing 7 illustrates the API call for persistent tasks which contains identical arguments to the standard *edatSubmitTask* for transitory tasks, just the call has a different name. This means the programmer can submit a persistent task which will be executed whenever the event dependencies are met. In the case of multiple task dependencies there can be multiple versions of a persistent task in progress, where some dependencies have been met for a persistent task a number of times and other dependencies are still outstanding for each copy of that persistent task. It is also possible to associate a name with persistent tasks so that programmers can then manage them later on in code, for example to remove them. Unlike transient tasks, persistent tasks are not required to have executed before the code can terminate.

```
void edatFirePersistentEvent(void* data, int data_type, int
    number_elements, int target_rank, const char *
    event_identifier)
```

Listing 8: Firing a persistent event

Listing 8 denotes the API call for persistent events which, as soon as they have been consumed as a task dependency will be fired again. However this re-firing of the event is done locally; even if a persistent event is sent from a remote process then the fact it is persistent it handled by the target.

B. Pausing tasks and checking for events

In an ideal world one would write their tasks in a way that only requires interaction via their input dependencies. However we found that more flexible support was useful. The first of these, a task *wait*, is illustrated in listing 9 where at some point in the task the programmer instructs EDAT to consume events matching the provided dependencies. If these dependencies can not be satisfied then task will be paused, the task switched out and the worker free to execute other pending tasks. At some point, when the event dependencies specified by the programmer arrive, the task will be reactivated and a worker will continue to run the task after this line of code. The events themselves are returned as an *EDAT_Event* structure. Similarly to the submission of tasks, the programmer provides the number of event dependencies and for each of these the source of the event and an event identifier.

```
EDAT_Event* edatWait(int number_event_dependencies, <int
    event_source, char * event_identifier>...)
```

Listing 9: Task wait API call

This is useful as it enables the programmer to start a task with a minimal set of dependencies met, do some work and then wait until other dependencies arrive to do further work with those. The major advantage of this over splitting the task into a number of sub-tasks is that of context, where task's local variables are maintained and can be used as before once the task re-activates.

Of course there is some overhead associated with the thread level context switch required when pausing a task and as such the EDAT API also provides a *edatRetrieveAny* call which is a variation of the task wait. If events corresponding to the dependencies are not available, instead of pausing the task for them to arrive, the task continues executing regardless and the call also provides the number of events returned. This is useful in situations where other work could be done by the task even if the events have not yet arrived and allows the programmer to obtain a subset of events and process these as they are made available.

C. Safely sharing local data

There are most likely multiple workers, such as one per core in a NUMA region, and so multiple tasks per process are likely executing concurrently. A common challenge with this form of concurrency is avoiding conflict, especially if these tasks are working on the same state. There are a number of ways to protect global state and the programmer can use events to enforce ordering and/or mutual exclusion which is shown in Listing 10. In this example a persistent task is submitted by Process 0 (line 2) to update some state whenever values are received from Process 1. Shared data is allocated which is provided as the data payload to the *edatFireEvent* call at line 5, with the type *EDAT_ADDRESS* that specifies the programmer is passing a reference to the data rather than the data itself. When the event *values* arrives from Process 1, the task will run on a worker but, regardless of whether other events with the same *value* identifier also arrive from Process 1 whilst this task is running, another copy of the persistent task will not execute as it also relies on the *data* event. Just before the task completes it fires an event with identifier *data* and the same shared data address to itself (line 10). Effectively this ensures that only one instance of the task is running at any one time and maintains mutual exclusion over the shared state.

```
1  if (edatGetRank() == 0) {
2    edatSubmitPersistentEvent(task, 2, EDAT_SELF, "data", 1, "
         values");
3    int * shared_data=(int*) malloc(sizeof(int) * 10);
4    edatFireEvent(shared_data, EDAT_ADDRESS, 1,
         EDAT_SELF, "data");
5  }
6
7  void task(EDAT_Event * events, int num_events) {
8    ...
9    edatFireEvent(events[0].data, EDAT_ADDRESS, 1,
         EDAT_SELF, "data");
10 }
```

Listing 10: Events to protect local data

Whilst managing shared data safety via events fits with the general ethos of EDAT, it does represent locking at a rather coarse grained level. The programmer can use *edatWait* to make this more fine grained, or alternatively EDAT provides *edatLock*, *edatUnlock* and *edatTestLock* calls for explicitly acquiring locks. These all accept a string lock name as an identifier and, in addition to explicit unlocking by the programmer, when a task finishes all locks that it has acquired will be automatically released. When the programmer pauses a task using the *edatWait* of Section IV-B, all acquired locks will also be released automatically and when the task reactivates all these locks will be reacquired before task execution continues.

V. Graph 500 BFS benchmark

The open-source Graph500 benchmark [15] focuses on data intensive supercomputing and whilst the computation in the benchmark is trivial, the communication is irregular, unpredictable and the most significant performance factor. In the latest version of the reference benchmark the steering committee have developed an active messaging layer on top of MPI which abstracts the specifics of the underling calls. This is a prime example of how MPI is insufficient for these types of codes and application developers forced to develop their own abstractions. Not only does this approach add considerably to the work of the end-developers, but inevitably not all will be experienced HPC programmers and as such might missing opportunities for optimisation and performance. Whilst the Graph 500 is an open-source code, the active messaging layer is highly tied to this application and as such is not easily reusable by other codes.

In this paper we are focusing on the breadth first search (BFS) kernel which, starting from a single source vertex, traverses a randomly generated graph in phases to find and label the neighbours of vertices. The kernel proceeds until all neighbours of all vertices are labelled. The reference

implementation proceeds in a level synchronous approach, where neighbours of all vertices in a given level of the graph are communicated before the processes explicitly synchronise, unpack received information and then proceed with received neighbour information onto the next level in the graph.

Figure 2 illustrates the EDAT task graph for our implementation of the BFS kernel. To compare like with like we implemented a level synchronous approach in EDAT. Even though adding in this extra synchronisation to our EDAT driven BFS kernel is not necessarily the most natural way of using our approach, this avoids revisiting nodes and helps us conduct a fair comparison. A persistent task is scheduled for specific levels in the graph, *n*. This task depends on events with the identifier *visit_n* (where *n* is the level) and for each vertex will store the required parent information and examine each neighbouring vertex, sending an event to the process holding that neighbour vertex with the identifier *visit_n+1* where *n+1* is the next level.

Fig. 2: EDAT BFS task graph

Figure 3 illustrates the performance of of the Graph 500 reference implementation and EDAT version of the BFS kernel (compiled with the GNU 6.3 compiler). This is measured in Traversed Edges Per Second (TEPS) and is for a problem size of scale 29 (536 million verticies) with an edge factor of 16 (8.5 billion edges), run on a Cray XC30. It can be seen that, whilst the performance of the reference version outperforms the EDAT version at lower numbers of nodes, the EDAT version catches up and by 1280 nodes (30720 physical cores) the EDAT version is out performing the reference implementation (2.23e+10 TEPS vs 1.50e+10 TEPS.)

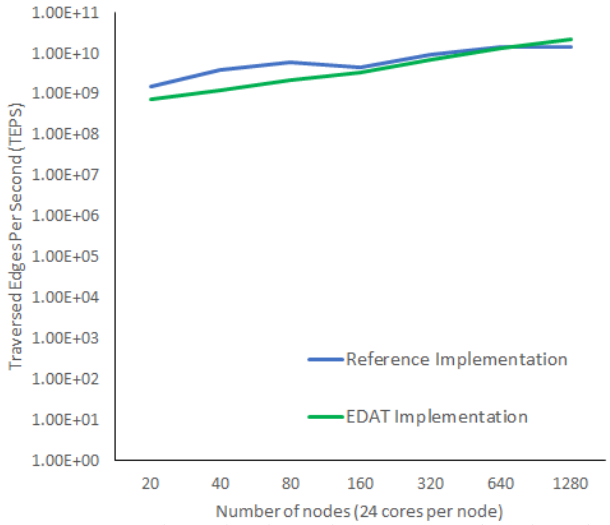

Fig. 3: Graph500 level synchronous BFS benchmark

We found that the best configuration for the EDAT version was to run with one process per physical core, each of these cores containing one worker. The reason behind this is that the visiting task needs to interact with some global state (i.e. whether the vertex has been visited previously and for the scheduling of the next level.) When running with multiple workers then this state needs to be protected via locks (see section IV-C) for every vertex visited. Bearing in mind there are 536 million verticies with this scale problem, continually locking and unlocking adds overhead. In terms of polling for progress we found the optimal setting was to enable the dedicated progress thread and also enable hyperthreading. Effectively each physical core presents itself as two logical cores, one of which was running the progress thread and the other a worker to execute tasks. For a fair comparison with the reference benchmark we also experimented with enabling hyperthreading and the MPI progress thread but this did not significantly improve the performance of that code.

There is some additional overhead associated with the scheduling of tasks and managing of dependencies and this is a major factor behind the lower performance at smaller core counts. However at larger core counts the scalability that EDAT provides becomes more beneficial and, even though we are only running one worker per process and the algorithm is level synchronous, the fact that the progress thread can progress in the background, gather in the events and have these all ready when the task is scheduled is beneficial in contrast to the reference approach. Additionally the reference implementation very much works in steps of communicating neighbours for the next level, gathering verticies for the next level and updating its state. Our implementation, whilst it is still working in a level synchronous manner, combines the update of state and communication of neighbours, effectively running them concurrently, which is of benefit.

This benchmark also illustrates that our approach can be applied incrementally to existing codes rather than requiring an entire rewrite that is often needed by other task-based technologies such as Charm++ and Legion. In this work just the BFS kernel itself was rewritten in EDAT, with other part of the Graph500 benchmark such as graph generation, validation and edge counting remaining in MPI. The EDAT library has been written in a way that supports inter-operability with application level MPI which we feel is important because it enables the programmer to concentrate their effort on porting scalability hotspots in their code without having to rewrite en-mass other, less critical, areas.

VI. In-situ data analytics case-study

The Met Office NERC Cloud model (MONC) [16] is an open-source (BSD licensed) high resolution modelling framework that employs large eddy simulation to study the physics of turbulent flows and further develop and test physical parametrisations and assumptions used in numerical weather and climate prediction. MONC is a major model used by UK atmospheric and climate communities, and modern parallelisation has opened up the possibility of running on tens of

thousands of cores and soon to be hundreds of thousands. However in addition to generating the raw, *prognostic* data, the model also needs to analyse this into higher level *diagnostic* information. It does this via an in-situ approach [17] where cores are split up between computation and data analytics; typically one core in a NUMA region runs the data analytics and *services* the remaining computation cores.

The MONC developers wrote their own be-spoke analytics code where computational processes send data at different times and in different orders to their corresponding in-situ core. Therefore, whilst the in-situ cores often have to communicate, there is no knowing exactly when each analytics core will be ready to issue this communication or the order of communications; for instance one core might issue communications for field A and then B whilst other cores issue communications for B and then A. The nature of communications typically involves very many small messages, often involving all the analytics cores. It is important to embrace this irregularity, not only because global synchronisation protocols would be expensive, but also to keep each analytics core busy processing data to avoid their computational cores being forced to stall.

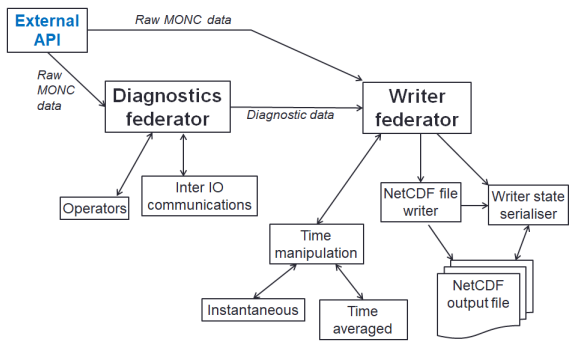

Fig. 4: MONC in-situ data analytics pipeline

The developers of MONC wrote their bespoke communications layer on top of MPI point-to-point for the data analytics to handle the irregular nature of their application. This adds considerable complexity and bloat to the code, is very much tied to MONC and is also a performance bottleneck. Figure 4 illustrates the data analytics pipeline of MONC. In the existing MONC code a thread from a pool was activated to handle requests in the *external API* from a computational core such as raw data message arrival. In this case that same thread is then used to process and write the data. Whenever analytics cores need to communicate with each other (e.g. the *inter IO communications*) data is sent tagged with the timestep and field name and the root communication core will start a new thread to process the resulting messages. This mixing in application code of logic and communications mechanism is messy, and the application programmer needs to mix higher and lower level concerns which is not ideal.

The interactions of figure 4 are really an instance of the event based co-ordination pattern [2] and as such, it was our hypothesis that EDAT would be a natural way of expressing this use-case. We replaced the MONC be-spoke communications code with EDAT instead (which has bindings for Fortran). The *external API* submits a single persistent task on start-up for computational core registration and once this happens, via the firing an event matching the dependency of task registration, then two further tasks are submitted, a de-registration task for that core and a persistent task handling data from that core. Also, on start-up, the writer federator and diagnostics federators each submit a persistent task, within the diagnostic federator EDAT tasks are also used for the *inter IO communications*. This has resulted in a decrease of 9% in the code size of the in-situ data analytics and, most importantly, this code was the tricky, low level logic around communications, rather than the data analytics application-specific code. An additional 2% of the analytics code was modified, with the remaining 89% of the code unchanged. This illustrates how our approach can be retrofitted into existing applications without wide scale code changes.

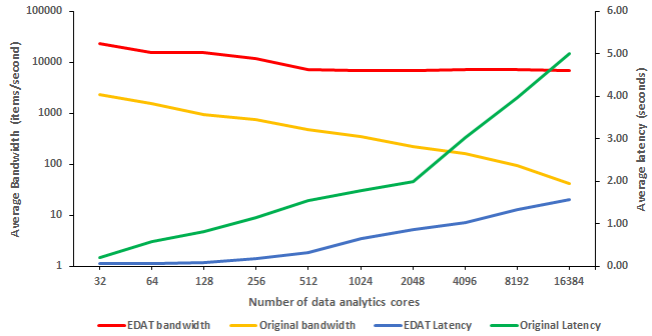

Fig. 5: Bandwidth and latency against number of data analytics cores for EDAT and original MONC in-situ data analytics

Performance tests on a Cray XC30 (using the GNU 6.3 compiler), have been performed and for the purposes of benchmarking there is a 1 to 1 ratio between computational and data analytics cores. This is so we could run with a large number of analytics cores, all of which need to communicate. A standard, stratus cloud, test case has been used with the computational code modified to repeatedly send raw data, saturating its corresponding data analytics core with as much data as possible. In the EDAT implementation we configured 10 workers per analytics process and in the original MONC code have set the same number of threads. The analytics required for each value involves both arithmetic operations and reductions between all analytics cores. Figure 5 illustrates the results of these tests, measuring both the average bandwidth (number of items per second that can be processed, ignoring the file write time), and latency (average amount of time it takes for each raw data item to be processed and produce a resulting value, again ignoring the time writing to disk) against the number of data analytics cores.

From the results of figure 5 it can be seen that the bandwidth afforded by the EDAT approach is an order of magnitude greater than the original, bespoke, MONC code. The average latency is also significantly smaller for the EDAT implementation in comparison to the original code. One of the major reasons for this is because the MONC developers

had not formalised their communications via tasks and there was a complex inter-dependence between threads, and certain situations where progress can not be made without a number of threads running concurrently. Memory cleaning functionality was also required in the original MONC communications code, freeing up the state of previous interactions, which for safety must lock out many other activities running. Instead, by formalising the interaction into tasks the overall architecture is more obvious, poor design choices are avoided, much of the state cleaning is performed by the underlying EDAT library and the diagnostic and writer federators now run concurrently.

Global communication between the analytics cores (a reduction) is required for processing each data item. Hence both the bandwidth decreases and latency increases as the number of analytics cores increases. But also the reduction root is automatically distributed amongst the analytics cores such that activities after communication are more distributed as the number of analytics cores is increased. This explains the levelling off of bandwidth, at around 7000 items per second at larger core counts for the EDAT version. As the original MONC code contains significant overhead it does not benefit from this increased distribution.

VII. Conclusions and further work

Driven by the rise of areas such as data science and machine learning, there are vast numbers of potential HPC users in communities not traditionally associated with supercomputing, which have the potential to transform our field. However many of these communities have strong open-source roots and an eco-system without open-source software and hardware will likely limit uptake. Additionally, the parallel workloads of these new communities is likely to be far more unpredictable and irregular, along with a large number of programmers who are potentially already familiar with writing asynchronous codes in other domains such as web-development. Current ubiquitous parallel programming technologies are not well suited to these new types of application, and whilst task-based models look like a promising approach, the current generation all exhibit disadvantages for large scale codes.

In this paper we have described a new, open-source, approach to task-based models where the programmer is explicitly aware of the distributed nature of their application but still abstracted from the low level details of how components interact. Splitting problems up into distinct tasks and driving the interaction of these via events, provides a high level view of parallelism. This means programmers can focus on important areas such the design of parallelism, and optimisations including locality.

The general idea of our approach is simple and few new concepts need to be mastered. It is our opinion that tasks and events maps well to how people should be thinking about parallelism; their parallel tasks, where these are located and how they interact. We have demonstrated good performance and scaling of EDAT with both the Graph500 BFS benchmark on up to 1280 nodes (30720 cores) and the real world MONC model with 16384 data analytics cores. Not only does this illustrates the ability of our approach to scale well to high core counts, but additionally provides an important step change in capability, especially for in-situ data analytics, that these sorts of applications can now take advantage of.

So far we have concentrated on events explicitly fired by user code, but they could represent other behaviours. For example an event could be generated based on some resource becoming available, the activation of a timer, or a hardware event. There could be any number of machine generated events that then allow the programmer to submit tasks that interact with different states of the machine itself in a unified manner. Our approach is also applicable to heterogeneous architectures, for instance EDAT could fire events, potentially directly from the GPU to the NIC without having to go via the CPU and this is also an avenue for further work.

The EDAT library implements the concepts described in this paper and has been used for the experiments conducted. EDAT is open-source, BSD licensed and available at https://github.com/EPCCed/edat which also includes documentation, examples and further benchmarks. We are very keen for more contributors and users to become involved and, whilst the library is perfectly usable for real world codes, inevitably there are further library level optimisations that can be made in the future. In addition to being directly usable from C and C++, Fortran and Python bindings are also provided.

References

[1] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The risc-v instruction set manual," *Volume I: User-Level ISA', version*, vol. 2, 2014.

[2] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Pearson Education, 2004.

[3] E. Ayguadé *et al.*, "The design of openmp tasks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 404–418, 2009.

[4] A. Fernández *et al.*, "Task-based programming with ompss and its application," in *European Conference on Parallel Processing*. Springer, 2014, pp. 601–612.

[5] E. Slaughter and et al, "Legion programming system (feb 2017), version 16.10. 0."

[6] H. Group. (2018). [Online]. Available: http://stellar.cct.lsu.edu/

[7] T. G. Mattson *et al.*, "The open community runtime: A runtime system for extreme scale computing." in *HPEC*, 2016, pp. 1–7.

[8] J. Bueno Hedo, "Run-time support for multi-level disjoint memory address spaces," 2015.

[9] N. Brown, O. Brown, and D. Holmes, "Event driven asynchronous tasks," *Poster at ISC2018*, 2018.

[10] T. Von Eicken *et al.*, *Active messages: a mechanism for integrated communication and computation*. ACM, 1992, vol. 20, no. 2.

[11] D. Bonachea, "Gasnet specification, v1. 1," 2002.

[12] J. J. Willcock *et al.*, "Am++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 401–410.

[13] Ammpi: Active messages over mpi - quick overview. [Online]. Available: http://www.cs.berkeley.edu/âĹijbonachea/ammpi/

[14] L. V. Kale *et al.*, "Charm++: Parallel programming with message-driven objects," *Parallel programming using C+*, pp. 175–213, 1996.

[15] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray UserâĂŹs Group (CUG)*, vol. 19, pp. 45–74, 2010.

[16] N. Brown *et al.*, "A highly scalable met office nerc cloud model," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*. University of Edinburgh, 2015, pp. 132–137.

[17] N. Brown, M. Weiland, A. Hill, and B. Shipway, "In situ data analytics for highly scalable cloud modelling on cray machines," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, p. e4331, 2018.