Athapascan-0b for PVM: Adding threads to PVM

Michel Christaller, APACHE project, IMAG Institute, Grenoble, France.
(Michel.Christaller@imag.fr)

Need for threads

PVM proved to be a useful tool for parallel programming. It is easy to use, efficient and portable. For example, one can make a parallel program for large matrices multiplication, by splitting matrices in chunks and passing them in messages between the involved processors. This will probably be an efficient parallel implementation of the operation « matrix multiplication », and a truly portable one, under PVM. However, the operands of the operation may be results of other independent computations, which could also be done in parallel. For example, in the expression $A^2 \times B^{-1}$, A^2 and B^{-1} could be computed in parallel. This is not always easy under PVM, as matrix square and matrix inversion are themselves parallel operations involving several processors. One way to do this is to compute each parallel operation sequentially (e.g. A^2, then B^{-1}, then $A^2 \times B^{-1}$). But, if one operation does not need all the available processors, resources are wasted. Also, when a processor waits for a message, it does nothing else.

There is no easy way to proceed parallel operations in parallel under PVM, unless to introduce the multiplexing capabilities of a lightweight threads package. Then, each parallel operation involves only a thread of execution on each task which participates to the computation. Several parallel operations can be conducted in parallel, as threads have distinct contexts. As the available processors can be shared by different parallel operations, resource utilization is increased; and even static or dynamic load balancing can be performed to maximize it. Finally, when a thread waits for a message to arrive, the processor can switch to another thread of execution to compute something else. Threads permit thus to achieve better resource utilization.

How to combine PVM and threads ?

Some of the best qualities of PVM are its efficiency and portability. Adding multithreading to PVM must keep them. Other aspects are ease of use and versatility as a foundation for future parallel programming tools.

There are some constraints: PVM is rapidly evolving; tracking changes will be a nightmare as patches or even major releases arrives. And vendor specific versions of PVM are completely different, although compatible.

So the best way is to put threads aside PVM, not into it (at least to someone not implied in PVM development or definition). We will see that it does not decrease performance, given a predefined behavior for the application.

On the other hand, threads packages are numerous. Some offer only non-preemptive threads with no support for Unix features (signals etc..), others are preemptive and prioritized, with thread-safe system libraries...

In order to be portable, a multithreading PVM could only rely on minimum thread support from a threads package: non preemptive threads, no priorities, only thread fork, suspend, resume and kill. Extensions can be added for enhanced threads packages, but the base is here.

Another way to combine a communication package and a threads package is to add functionnalities to local threads to operate them remotely (e.g. fork thread, suspend and resume it, etc.. remotely). This leads to what can be called Asynchronous Remote Procedure Calls (ARPC). One can fork a thread remotely, and let it compute some function (it is a RPC), but the caller is not blocked; he can compute something before waiting for the child thread results (thus it is an asynchronous call). These two views (expand a thread package with communication primitives, or expand PVM to multithreading capabilities) are dual.

Athapascan-0b for PVM

Athapascan-0b for PVM is a package which encapsulates PVM and put aside threads kernels. It is an evolution of Athapascan-0a, which attempted to define it's own parallel language. Athapascan-0b is only, as PVM, a library of primitives. Currently, it is only available under the C language. It was implemented and ported on five systems: Sun Solaris, Sun Os 4.0, PCs under Linux (Rex library), IBM Aix (dcepthreads), Data General DGUX (J.Briat's threads).

Since we do not use a preemptive threads kernel by default, our threads will relinquish processor control only when they are waiting for communication or synchronization. Otherwise, they compute as far as they can. This behavior is compatible with most scientific applications; only those which strongly need time sharing between concurrent threads, or a strict implementation of priorities, would not run without preemption.

The key features of a communication extended thread-package or of a multithreading extended communication package are:

how do we create remote threads ?
how do threads communicate with each other ?

thread creation

Remotely forking a thread involves sending one or more messages. One must create a header, which will record the caller's id (at least if an answer is needed), then the called function's id. Then, some or all parameters must be packed. The message is sent. When the thread creation request is handled on the callee site, a new thread is created, which will execute the given function. Unpacking parameters can be done before the function is called (but this needs compiler support to determine parameters types), or after. In the latter case, the function receives as argument the message id from which true parameters can be unpacked. When the thread dies, if an answer is needed, it packs results in another message, then sends it to the caller. At some time, the caller will block waiting for the answer, and eventually get the reply message, and unpack results.

There is two possibilities for handling thread creation requests:

- the current running thread may be interrupted by the arriving message; but this will need the communication package to be able to produce interrupts on message arrival. As PVM user side cannot (without modification), we have not chosen these scheme.

- the other possibility is to poll for message arrival. One can poll at predefined intervals, or at some events. Polling too often will slow computation. According to the non-preemption scheme (where threads run as far as they can), an appropriate polling method is to poll only when no thread is ready to run. This reduces polling cost to the minimum - poll only when absolutely needed - probably less than the cost of a message interrupt, since this polling can be done when threads relinquish processor control, so without interrupt or thread switch overhead. This scheme as the disadvantage to buffer thread creation requests until they can be served. But PVM deamon does this well. On the other hand, a message request is small in front of a thread structure - mainly the thread stack. When a program forks a lot of threads, this will reduce memory consumption.

Results messages can be taken into account in the same way - only when no thread is ready to run. This delays reception of results, but since the processor is doing useful work in the meantime, there is no direct loss in efficiency. However, this has an important effect on thread scheduling. Note it would be hard to implement a global thread scheduling at low cost, since we are in a distributed environment. The only reasonable scheduling (partial) order that can be assured, is that, if processor A launches two threads on processor B, they will be scheduled in the order they were launched - since the underlying message passing system offers « processor consistency ». In this scheme, there is no global scheduling. Perhaps scientific computations can be handled without any stronger scheduling than a processor consistent one. This has to be validated by practice. Since results messages only awake already existing threads, it is possible to poll for results before polling for thread creation requests. This can help to keep the number of threads low.

communication between threads

In a shared memory system, threads communicate easily by sharing memory. In a message passing system, thread communication can be done in three main ways:

- Threads are able to send and receive messages directly to and from other threads. It is a pure extension of message passing from tasks to threads.

- Threads communicate in a connection-oriented mode: between the caller and the callee, a bi-directional « pipe » is established. A restricted pipe can be formed of one request sent, then one answer sent back: this corresponds to a procedure call with normal parameters and results passing.

- Threads do not communicate remotely. They use local shared memory. For one thread to communicate with another thread on another processor, it does an RPC on the other processor, thus creating a third thread, which can communicate by shared memory with the target thread.

These three ways seem legitimate:

Message passing is the usual way to program parallel operations. For example, a common parallel matrix multiplication operation involves shifting groups of rows of one matrix over the processors. This collective shift is usually done by point to point message passing. Thus it seems useful to let groups of threads communicate via point to point or collective message passing.

Asynchronous remote procedure call is the base mechanism to introduce parallelism in a procedural language. This can be extended to a connection-oriented mode, where « pipes » are built. Pipes can be a powerful tool to organize communication. For example, to obtain a parallel sort of an array, it is possible to split the array in chunks, then sort each chunk onto different processors, then merge all sorted chunks via a merge-sort algorithm. Since the algorithm used to sort each chunk can be a decreasing one (like the bubble sort, which determines the major element, then the major of the remaining...), it is possible to begin to merge sorted chunks as soon as the first major element of each chunk is determined. Thus via pipes, one can sort each chunk and merge the results in parallel.

The usual way for threads to communicate is by shared memory. For example, one can fork remote threads, which will remotely access memory (read and write). This can lead to build a distributed memory. In order to avoid to create a temporary thread, only for communicating, one can imagine an ADA-like accept instruction, which will trap some thread creation requests into the current thread. Shared memory regions can be implemented in the previous way, a thread controlling accesses of other threads to a shared region. This can also be used to establish pipes between sibling threads.

At this time, only the normal (no pipes) asynchronous remote procedure call was implemented in Athapascan-0b for PVM. The others will be experimented soon.

An example: parallel sort

Let us examine a parallel sort algorithm as an example. An array of n integers, T, will be sorted using p processors (we suppose for simplification that $n = k \bullet p$). On each processor, a thread will be created, which will execute the « sort » function, with as parameters a chunk of T. The thread will give as result this chunk sorted.

The sort function can be written in Athapascan-0b for PVM in the following way:

```
void sort(ath_t_handle hdl) {          - PVM receive buffer is automatically set to the incoming message,
int S[k];                              - and PVM send buffer is automatically initiated for response
    pvm_upkint(&S, k, 1);
    sort S using some sequential algorithm...
    pvm_pkint(&S, k, 1);
}                                      - response is automatically sent back
```

The parameter *hdl* of the sort function means the « handle » used to receive arguments and send results back. Here, it is not used since pvm buffers are automatically set before the function entry, and the response buffer sent after function exit.

The calling program can be written as:

```
main(int argc, char ** argv) {
int T[n], R[p][k];                      - array R will contain sorted chunks
ath_t_handle hdl[p];                    - define the handles to do ARPCs
int task[p];                            - PVM taskid of involved tasks

    for (i=0; i<p; i++) {
        hdl[i]=ath_call(task[i], ep_sort);   - prepare call of « ep_sort » entry point onto task[i]
        pvm_pkint(&(T[i*k]), k, 1);          - pack arguments
        ath_flush(hdl[i]);                   - send request
    }

    for (i=0; i<p; i++) {
        ath_wait(hdl[i]);                    - wait for results
        pvm_upkint(&(R[i]), k, 1);           - unpack results
        ath_close(hdl[i]);                   - free internal structures
    }
    merge sort all R[i]
}
```

The *ath_wait* primitive can lead the running thread to yield the processor, if results have not already arrived. Thus, other parallel operations than this sorting of T, can be done in parallel.

Each slave task executes the following code:

```
main(int argc, char ** argv) {

    ath_init();                         - initialize the Athapascan-0b runtime
    ath_newep(ep_sort, sort, null);     - attach the « ep_sort » entry point to the sort function
    ath_serve();                        - begin to serve entry points
}
```

The third parameter of *ath_newep* can be used to specify a pointer to a private storage for all threads that execute this entry point. It is set to null since it is not used.

After the main thread executes the *ath_serve* primitive, it waits for the task to terminate. To be short, we will not show here how to load or terminate the slave tasks. However, after the *ath_serve*, the task is ready to get a request for the *ep_sort* entry point, and to fork a thread which will execute the sort function.

Availability

The Athapascan-0b for PVM will be available under the general GNU license. It will be possible to download a prototype by anonymous ftp on the server ftp.imag.fr (directory /pub/APACHE), on the 1st of november. Please mail a note to Michel.Christaller@imag.fr when downloading it. We encourage you to test it, and report any comment.