The pSather Fat Parallel Virtual Machine (PFPVM) Interface

Evolving Draft Document

Revision: 1.15
October 4, 1994

Jeff Bilmes
<bilmes@icsi.berkeley.edu>

International Computer Science Institute
1947 Center St, Berkeley CA, 94704

Abstract

A critical part of the plan for portable pSather is the definition of a run-time interface that can be efficiently implemented on the wide range of platforms targeted by the project. This document describes in detail the runtime interface assumed to exist by the pSather compiler. Implementation details are not described herein.

1 Introduction

This document is intended for pSather compiler writers or runtime interface implementors. It is intended to describe the the pSather runtime interface; it acts as a set of extensions to the serial Sather runtime system. Particular implementation details are not described herein, yet language semantics, where appropriate, are included. It is assumed that the reader is familiar with Sather and pSather. For semantic descriptions of these languages, see the Sather [1] and pSather [2] language specifications.

We choose the fat C approach: the runtime’s parallel functionality is implemented in C on the native machine (thus, fat). We found this approach to have several advantages.

1. It is more efficient.
   
   • Compilers of today are poor relative to hand optimized code. Good efficiency can often be achieved, however, if code is hand-tuned and unrelated to the compilation process. This results in the compiler doing relatively little, and the runtime doing relatively much.
   • The runtime system should not depend on Sather data structures (e.g., Queues, hash tables, etc.) because they will not be optimized for a particular architecture. The runtimes should be free to optimally implement all of its quirky and specialized data structures however it chooses. It should use whatever the hardware provides without regard to how the user level Sather data structures are implemented. Although this makes the runtimes inherently non-portable, the efficiency gained is probably more important.

2. It is cleaner and simpler. The runtime code is located in one module set and in one language. There is no complicated Sather/C split. The runtime interface is clearly defined by an abstraction layer. The compiler emits code that makes calls to the runtime routines directly.

3. It makes the compiler’s job easier and the compiler more portable, although the runtimes themselves (which are already non-portable) become less portable.

1 The latest version of this document may be obtained from http://www.icsi.berkeley.edu/Sather/ps/pfpvm.ps
Through out this document, the symbols described might be macros, typedefs, functions, or variables as deemed appropriate by the runtime system implementor. The compiler must not assume any particular definition of a feature.

Furthermore, certain features may or may not be implemented exactly as suggested by the runtime interface. For example, the implementation of the wait counter mechanism (see below) might use the normal memory access routines. Of course, this will depend on the current machine architecture: the compiler may safely assume that any feature is implemented as efficiently as possible on each architecture.

The key point is that the runtime interface provides the illusion of a parallel machine with a certain set of features. The compiler may safely assume this illusion is real. Whether the illusion is real on a particular machine, however, depends on the features of the underlying hardware architecture and operating system software.

2 Abstract Machine

The following definitions define each processor’s cluster id, and the total number of clusters.

- `PS_HERE` -- the current cluster id, type INT
- `PS_CLUSTERS` -- the number of clusters, type INT

3 Far Pointers

A far pointer is a runtime object that can reference any location within the abstract machine. What follows are declarations related to far pointers.

Types:

- `PS_FAROB` -- a C pSather far object pointer.
- `OB` -- a near object pointer (same as a Sather object pointer).

Definitions:

- `PS_FO_VOID` -- the far object void value. type PS_FAROB
- `PS_FO_CID(fo)` -- the cluster ID of a far pointer. type INT
- `PS_FO_OBS(fo)` -- the pointer portion of a far pointer. type OB
- `PS_FO_OBS_CHK(fo)` -- same as above but check for nearness.
- `PS_FO_ISVOID(fo)` -- returns true if 'fo' is a void far pointer.
- `PS_FO_NEAR(fo)` -- returns true only if 'fo' is not void and points to a location on the callee's cluster.
- `PS_FO_FAR(fo)` -- returns true only if 'fo' is not void and points to a location off the callee's cluster.
- `PS_FO_WHERE(fo)` -- returns the cluster ID of the far pointer.
- `PS_FO_ID(fo)` -- an ID for this far pointer. returns an INT

Conversion routines:

- `PS_FAR_TO_NEAR(fo)` -- returns the near pointer portion of a far object.
- `PS_NEAR_TO_FAR(OB o, int cluster)` -- returns a far object constructed from 'o' and 'cluster'
4 Memory access for Far Pointers

All runtime memory access routines use a split-phase mechanism. Therefore, all reading and writing of variables located on remote processors are not guaranteed to complete until an explicit wait is performed. The type of wait depends on the type of access, as described below.

The following type is defined:

\[
\text{PS\_WAITCTR} \quad \text{-- wait counter type}
\]

Also, in routines that take a hint actual argument, the following values are defined:

\[
\begin{align*}
\text{PS\_LONG\_WAIT} & \quad \text{-- The memory access will take a long time.} \\
& \quad \text{It's probably more efficient to context switch.}
\end{align*}
\]

\[
\begin{align*}
\text{PS\_AVG\_WAIT} & \quad \text{-- The memory access may or may not take a long time.} \\
& \quad \text{The runtime will probably spin for a while} \\
& \quad \text{and then context switch after a timeout.}
\end{align*}
\]

\[
\begin{align*}
\text{PS\_SHORT\_WAIT} & \quad \text{-- The memory access will not take a long time.} \\
& \quad \text{It's probably wasteful to context switch. The} \\
& \quad \text{runtime system will probably spin-wait.}
\end{align*}
\]

\[
\text{PS\_HINT\_TYPE} \quad \text{-- The type of the above.}
\]

4.1 Reading

The following definitions read a value of type \(<\text{type}>\) from the location pointed to by \(\text{fo}\).\(^2\) The result is placed in the local location pointed to by \(\text{local\_result\_pointer}\). The second version has an argument giving the word-offset that should be added to \(\text{fo}\).\(^3\) Thus, \(\text{fo}\) plus this offset points to the location on the remote node from which to fetch the value.

\[
\begin{align*}
\text{PS\_READFAR\_<type>}(\text{PS\_FAROB p,OB local\_result\_pointer}); \\
\text{PS\_READFAR\_A\_<type>}(\text{PS\_FAROB fo,int offset,OB local\_result\_pointer}); \\
& \quad \text{where } \text{<type>} \text{ can be:} \\
& \quad \text{CHAR,INT,BOOL,INT,FLT,OB,FLT,OB,FLT,OB,FLT} \\
& \quad \text{-- Wait for all reads to complete} \\
\text{PS\_RWAIT(PS\_HINT\_TYPE hint)}
\end{align*}
\]

The following definitions are similar to that of above, but they utilize a \(\text{wait counter}\), a mechanism by which the compiler can be assured completion of individual read requests. Certain machine architectures support binary wait counters (e.g., Cray T3D), but on an architecture that doesn’t, the runtime might implement this as it pleases.

The \(\text{wait counter}\) mechanism allows the computer to place multiple memory accesses into the same set, and then wait for the completion of all the memory accesses at once. At one extreme, each outstanding memory access may be contained in its own set, and each may be waited for individually. At the other extreme, all memory accesses may be placed in the same set, and then all outstanding memory accesses must be waited for simultaneously. In between these two extremes, we have a general mechanism that allows the compiler to optimize memory accesses and implement

\(^2\)If \(\text{fo}\) is near, this will be a more expensive local access.
\(^3\)This version of the routine is needed by the debugger/tracer to identify object locations.
an arbitrary consistency protocol. Once again, if appropriate for a given machine, the runtime is free to implement a stronger consistency protocol than what is implied by the method the compiler chooses.

```c
PS_READFAR_<type>_W(PS_FAROB fo,OB local_result_pointer,
   PS_WAITCTR wc);
PS_READFAR_A_<type>_W(PS_FAROB fo,int offset,OB local_result_pointer,
   PS_WAITCTR wc);
where <type> can be:
   CHAR,INT16,B00L,INT,FLT,OB,FLTD,FAROB,FLTX,FLTDX
-- Wait on particular wait counter
PS_IWAIT(PS_WAITCTR wc,int hint)
```

The following definitions are for reading variables through far pointers assumed to be near. There are both checking and non-checking versions. There are no waiting operations for near pointer access since all such operations are assumed to complete immediately.

```c
PS_READNEAR_<type>(PS_FAROB fo,OB local_result_pointer);
PS_READNEAR_A_<type>(PS_FAROB fo,int offset,OB local_result_pointer);
where <type> can be:
   CHAR,INT16,B00L,INT,FLT,OB,FLTD,FAROB,FLTX,FLTDX
```

These next versions check the FO_NEAR(fo) condition before access.

```c
PS_READNEAR_<type>_CHK(PS_FAROB fo,OB local_result_pointer);
PS_READNEAR_A_<type>_CHK(PS_FAROB fo,int offset,
    OB local_result_pointer);
where <type> can be:
   CHAR,INT16,B00L,INT,FLT,OB,FLTD,FAROB,FLTX,FLTDX
```

### 4.2 Writing

The writing routines are dual of the reading ones. The differences should be clear.

```c
PS_WRITEFAR_<type>(PS_FAROB fo,<type> value);
PS_WRITEFAR_A_<type>(PS_FAROB fo,int offset,<type> value);
where <type> can be:
   CHAR,INT16,B00L,INT,FLT,OB,FLTD,FAROB,FLTX,FLTDX
PS_WWAIT(int hint) -- Wait for all writes to complete
```

The following are wait-counter versions of far writes.

```c
PS_WRITEFAR_<type>_W(PS_FAROB fo,<type> value,PS_WAITCTR wc);
PS_WRITEFAR_A_<type>_W(PS_FAROB fo,int offset,<type> value,
   PS_WAITCTR wc);
where <type> can be:
   CHAR,INT16,B00L,INT,FLT,OB,FLTD,FAROB,FLTX,FLTDX
-- Wait on particular wait counter
PS_IWAIT(PS_WAITCTR wc,int hint)
```
The following are near access versions.

```c
PS_WRITENEAR_<type>(PS_FAROB fo,<type> value);
PS_WRITENEAR_A_<type>(PS_FAROB fo,int offset,
            <type> value);
```

These next versions check the FO_NEAR(fo) condition before access.

```c
PS_WRITENEAR_<type>_CHK(PS_FAROB fo,  
            <type> value);
PS_WRITENEAR_A_<type>_CHK(PS_FAROB fo,int offset,  
            <type> value);
```

where <type> (once again) can be:
CHAR, INT16, BOOL, INT, FLT, OB, FLTDX, FAROB, FLT, FLTX, FLTDX

## 5 Bulk Memory Transfer

The following definitions provide for bulk memory transfer between clusters.

**Reading:**

```c
PS_RBULK(PS_FAROB far_src,OB near_dest,int size_in_bytes)
PS_RWAIT(int hint) -- Wait for all reads to complete
```

**Writing:**

```c
PS_WBULK(OB near_src,PS_FAROB far_dest, int size_in_bytes)
PS_WWAIT(int hint) -- Wait for all writes to complete
```

**Reading with a wait counter:**

```c
PS_WBULK_W(OB near_src,PS_FAROB far_dest, int size_in_bytes,
            PS_WAITCTR wc)
```

**Writing with a wait counter:**

```c
PS_RBULK_W(PS_FAROB far_src,OB near_dest,int size_in_bytes,
            PS_WAITCTR wc)
PS_IWAIT(PS_WAITCTR wc,int hint) -- Wait on particular wait counter
```

## 6 par Statement

The following statements supporting the par construct:

```c
PS_PARBEGIN -- corresponds to the 'par' construct
PS_PAREND -- the end of the 'par' block.
PS_COHORT -- the corresponding 'par' GATE
```

Multiple PS_PARBEGINs may be nested as needed.
7 Threads

Although pSather has minimal communication with its threads, the following definitions are defined:

- **PS_CLEARED** -- Returns true if the current thread has been cleared.
- **PS_TRAP_CLEAR** -- Returns the value of the current thread’s trap_clear flag.
- **PS_SET_TRAP_CLEAR(BOOL b)** -- Sets the current thread’s trap_clear flag to 'b'.
- **PS_DEFER** -- yields to the scheduler. i.e., this is a good time to context switch.
- **PS_POLL** -- poll for incoming messages, non-blocking routines, etc. A context switch might result from calling PS_POLL.
- **PS_POLL_NB** -- same as PS_POLL but guaranteed not to context switch. This is used as an optimization in routines the compiler has dubbed {em non-blocking} and that can borrow the stack from some other thread.

8 Locking

There are several types of locks in pSather. The following definitions will create a variable that can hold a pointer to a particular lock.

- **PS_MUTEX** -- MUTEX variable
- **PS_GATE** -- local value-less GATE
- **PS_GATET** -- local GATE{T} where T is a far pointer
- **PS_GATEV** -- local gate{T} where T is a value type.
- **PS_RW_LOCK** -- a reader-writer lock
- **PS_WR_LOCK** -- a writer-reader lock
- **PS_FRW_LOCK** -- a fair reader-writer lock.

A **PS_GATET** is optimized for passing around objects that are of size less than or equal to that of a PS_FARPTR. Therefore, if a particular value-type has size less than or equal to that of a far pointer, it would be more efficient simply to use that.

The following are creation routines. They return a lock of the appropriate type.

- **PS_MUTEX_CREATE** -- Returns new PS_MUTEX
- **PS_GATE_CREATE** -- Returns new PS_GATE
- **PS_GATET_CREATE** -- Returns new PS_GATET
- **PS_GATEV_CREATE(int size_in_bytes)** -- Returns new PS_GATEV
- **PS_RW_LOCK_CREATE** -- Returns new PS_RW_LOCK
- **PS_WR_LOCK_CREATE** -- Returns new PS_WR_LOCK
- **PS_FRW_LOCK_CREATE** -- Returns new PS_FRW_LOCK
The following are the Locking and unlocking routines. All routines take one argument: the lock they should operate on. We distinguish between the attempt of a lock, which attempts the lock once returning immediately if the attempt failed, and the acquiring of a lock, which will block the calling thread until the lock has been acquired.

Mutex routines:

PS_MUTEX ATT (PS_MUTEX mu) -- attempt/acquire
PS_MUTEX_REL (PS_MUTEX mu) -- release

GATE routines:

PS_GATE ATT (PS_GATE g) -- attempt/acquire
PS_GATE EMPTY ATT (PS_GATE g) -- attempt/acquire when empty
PS_GATE NOTEMPTY ATT (PS_GATE g) -- attempt/acquire when not empty
PS_GATE HAS THREAD ATT (PS_GATE g) -- attempt/acquire when thread
PS_GATE NOT THREADS ATT (PS_GATE g) -- attempt/acquire when no thread
PS_GATE REL (PS_GATE g) -- release

GATEET routines where T is a pointer:

PS_GATEET ATT (PS_GATEET g) -- attempt/acquire
PS_GATEET EMPTY ATT (PS_GATEET g) -- attempt/acquire when empty
PS_GATEET NOTEMPTY ATT (PS_GATEET g) -- attempt/acquire when not empty
PS_GATEET HAS THREAD ATT (PS_GATEET g) -- attempt/acquire when thread
PS_GATEET NOT THREADS ATT (PS_GATEET g) -- attempt/acquire when no thread
PS_GATEET REL (PS_GATEET g) -- release

GATEET routines where T is a value-type:

PS_GATEV ATT (PS_GATEV g) -- attempt/acquire
PS_GATEV EMPTY ATT (PS_GATEV g) -- attempt/acquire when empty
PS_GATEV NOTEMPTY ATT (PS_GATEV g) -- attempt/acquire when not empty
PS_GATEV HAS THREAD ATT (PS_GATEV g) -- attempt/acquire when thread
PS_GATEV NOT THREADS ATT (PS_GATEV g) -- attempt/acquire when no thread
PS_GATEV REL (PS_GATEV g) -- release

PS_RW_LOCK (reader/writer lock) routines:

PS_RW_LOCK ATT (PS_RW_LOCK l) -- attempt/acquire
PS_RW_LOCK REL (PS_RW_LOCK l) -- release

PS_WR_LOCK (writer/reader lock) routines:

PS_WR_LOCK ATT (PS_WR_LOCK l) -- attempt/acquire
PS_WR_LOCK REL (PS_WR_LOCK l) -- release

PS_FRW_LOCK routines:

PS_FRW_LOCK ATT (PS_FRW_LOCK l) -- acquire
PS_FRW_LOCK REL (PS_FRW_LOCK l) -- release
8.1 pSather lock/try Statements

First, we define the following tuples which group together the interface of a particular lockable object.

- **PS_MUTEX_TUPLE**
- **PS_GATE_\{EMPTY, NOTEMPTY, HASTHREAD, NOTHREADS\}_TUPLE**
- **PS_GATESET_\{EMPTY, NOTEMPTY, HASTHREAD, NOTHREADS\}_TUPLE**
- **PS_GATEEV_\{EMPTY, NOTEMPTY, HASTHREAD, NOTHREADS\}_TUPLE**
- **PS_RW_LOCK_\{READER, WRITER\}_TUPLE**
- **PS_WR_LOCK_\{READER, WRITER\}_TUPLE**
- **PS_FRW_LOCK_\{READER, WRITER\}_TUPLE**

The following definition support the pSather lock and try statements. Each is a function taking a variable number of arguments. The first argument specifies either the disposition, either **lock** or **try**. The second argument is the number of locks given in subsequent arguments. The remaining arguments are a lock and its corresponding lock-tuple.

```c
-- atomically acquire all locks.
PS_LOCK(int disposition, -- either PS_LACQUIRE or PS_LATTEMPT
    int num_tries, -- number of tries before suspending.
    int num_triples, -- number of lock triples
    lock1, lock_tuple1, -- first lock
    lock2, lock_tuple2, -- second lock
    ...);
```

It is certainly possible that, in a particular program, the set of locks are never in a state such that all predicates in a lock statement hold. We therefore must formalize the semantics of a lock statement. The **PS_LOCK** function guarantees the following: if the locks and their corresponding predicates hold unboundedly often during a period which a **PS_LOCK** function might be called, then that function call instance will eventually be succeed. This provides a fairness (i.e., starvation free) guarantee for lock statements conditioned on the existence of certain properties of the user program.

It is the compilers job ensure a lock release function within a lock statement unlocks a lock that was locked in a syntactically enclosing lock statement. That is, all the release functions defined above will unlock the corresponding lock without regard to any enclosing scope.

8.2 Special Gate Operations

There are a set of routines that support special behavior of **GATE** objects.

**GATE: Value-less gates.**
- **PS_GATE_SET(PS_GATE g)**
- **PS_GATE_GET(PS_GATE g)**
- **PS_GATE_ENQUEUE(PS_GATE g)**
- **PS_GATE_DEQUEUE(PS_GATE g)**
- **PS_GATE_ISEMPTY(PS_GATE g)** -- Returns BOOL
- **PS_GATE_HAS_THREAD(PS_GATE g)** -- Returns BOOL
- **PS_GATE_CLEAR(PS_GATE g)**
- **PS_GATE_SYNC(PS_GATE g)**
GATE\{T\} where T is a far pointer.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATE{T} where T is a far pointer.</td>
<td></td>
</tr>
<tr>
<td>PS_GATET_SET(PS_GATET (g),PS_FAROB (fo))</td>
<td>-- Returns a PS_FAROB</td>
</tr>
<tr>
<td>PS_GATET_GET(PS_GATET (gt))</td>
<td>-- Returns a PS_FAROB</td>
</tr>
<tr>
<td>PS_GATET_ENQUEUE(PS_GATET (g),PS_FAROB (fo))</td>
<td>-- Returns a PS_FAROB</td>
</tr>
<tr>
<td>PS_GATET_DEQUEUE(PS_GATET (g))</td>
<td>-- Returns BOOL</td>
</tr>
<tr>
<td>PS_GATET_IS_EMPTY(PS_GATET (g))</td>
<td>-- Returns BOOL</td>
</tr>
<tr>
<td>PS_GATET_HAS_THREAD(PS_GATET (g))</td>
<td>-- Returns BOOL</td>
</tr>
<tr>
<td>PS_GATET_CLEAR(PS_GATET (g))</td>
<td></td>
</tr>
<tr>
<td>PS_GATET_SYNC(PS_GATET (g))</td>
<td></td>
</tr>
</tbody>
</table>

9 Thread Forking Operations

The following definitions list the thread forking operations supported by the runtime system. The routines distinguish between routines that might block (blocking routines), and those that will not block (non-blocking routines). There are different forking routines for each category. Non-blocking routines might run by borrowing some other threads stack — because they are guaranteed not to block, deadlock will not happen in this case.

Normal RPC style call. Example: \g\.foo \@ far;

PS\_CALL(function func, -- the C function to call
      int cid, -- the cluster id to run it on
      OB loc, -- location to place return value (if any).
      This may be PS\_VOID if g.\foo has no
      return value.
      int num, -- number of arguments
      <ARGTYPE> type1, -- first arguments type
      arg1, -- first argument
      ...) |

Same as above but func is a non-blocking routine.

PS\_CALL\_NB(<same arguments as PS\_CALL>)

Fork. Example: :- g.\foo \@ far.

PS\_FORK(function func, -- the C function to call
      int cid, -- the cluster id to run it on
      ...) |
int num, \quad -- \text{number of arguments}  
<\text{ARGTYPE}> type1, \quad -- \text{first arguments type}  
arg1, \quad \quad -- \text{first argument}  
...

Same as above but \texttt{func} is a non-blocking routine.

\texttt{PS\_FORK\_NB(<same arguments as PS\_FORK>)}

Normal call with gate attached. Example \texttt{g := f.\text{foo} @ far;}

\texttt{PS\_CALL\_GATE(PS\_GATE g,}  
\hspace{1cm} \text{function func,}  
\hspace{1cm} \text{int cid,} \quad -- \text{the cluster id to run it on}  
\hspace{1cm} \text{int num,} \quad -- \text{number of arguments}  
\hspace{1cm} <\text{ARGTYPE}> type1, \quad -- \text{first arguments type}  
\hspace{1cm} arg1, \quad \quad -- \text{first argument}  
\hspace{1cm} ...)

Same as above but \texttt{func} is a non-blocking routine.

\texttt{PS\_CALL\_GATE\_NB(<same arguments as PS\_CALL\_GATE>)}

Fork with gate attached: \texttt{g := f.\text{foo} @ far;}

\texttt{PS\_FORK\_GATE(PS\_GATE g,}  
\hspace{1cm} \text{function func,}  
\hspace{1cm} \text{int cid,} \quad -- \text{the cluster id to run it on}  
\hspace{1cm} \text{int num,} \quad -- \text{number of arguments}  
\hspace{1cm} <\text{ARGTYPE}> type1, \quad -- \text{first arguments type}  
\hspace{1cm} arg1, \quad \quad -- \text{first argument}  
\hspace{1cm} ...)

Same as above but \texttt{func} is a non-blocking routine.

\texttt{PS\_FORK\_GATE\_NB(<same arguments as PS\_CALL\_GATE>)}

Normal call with attached. Pointer version. Example \texttt{g := f.\text{foo} \& \& \& far;}

\texttt{PS\_CALL\_GATET(PS\_GATET g,}  
\hspace{1cm} \text{function func,}  
\hspace{1cm} \text{int cid,} \quad -- \text{the cluster id to run it on}  
\hspace{1cm} \text{int num,} \quad -- \text{number of arguments}  
\hspace{1cm} <\text{ARGTYPE}> type1, \quad -- \text{first arguments type}  
\hspace{1cm} arg1, \quad \quad -- \text{first argument}  
\hspace{1cm} ...)

Same as above but \texttt{func} is a non-blocking routine.

\texttt{PS\_CALL\_GATET\_NB(<same arguments as PS\_CALL\_GATET>)}
Fork with gate attached. Pointer version. $g := f.foo \& far$

$$\text{PS\_FORK\_GATET}(\text{PS\_GATET \ g},$$

\begin{verbatim}
    function func,
    int cid, -- the cluster id to run it on
    int num, -- number of arguments
    <ARGTYPE> type1, -- first arguments type
    arg1, -- first argument
    ...
\end{verbatim}

Same as above but \texttt{func} is a non-blocking routine.

$$\text{PS\_FORK\_GATET\_NB}(<\text{same arguments as PS\_CALL\_GATET}>)$$

Normal call with attached. Value version. Example $g := f.foo \& far$

$$\text{PS\_CALL\_GATEV}(\text{PS\_GATEV \ g},$$

\begin{verbatim}
    function func,
    int cid, -- the cluster id to run it on
    int num, -- number of arguments
    <ARGTYPE> type1, -- first arguments type
    arg1, -- first argument
    ...
\end{verbatim}

Same as above but \texttt{func} is a non-blocking routine.

$$\text{PS\_CALL\_GATEV\_NB}(<\text{same arguments as PS\_CALL\_GATEV}>)$$

Fork with gate attached. Pointer version. $g := f.foo \& far$

$$\text{PS\_FORK\_GATEV}(\text{PS\_GATEV \ g},$$

\begin{verbatim}
    function func,
    int cid, -- the cluster id to run it on
    int num, -- number of arguments
    <ARGTYPE> type1, -- first arguments type
    arg1, -- first argument
    ...
\end{verbatim}

Same as above but \texttt{func} is a non-blocking routine.

$$\text{PS\_FORK\_GATEV\_NB}(<\text{same arguments as PS\_CALL\_GATEV}>)$$

In the preceding routines, the string \texttt{<ARGTYPE>} corresponds to a Sather type, and refers to one of the following:

- \texttt{TYPECHAR}
- \texttt{TYPEINT}
- \texttt{TYPEBOOL}
- \texttt{TYPEINT}
- \texttt{TYPEFLT}
- \texttt{TYPEOB}
- \texttt{TYPEFLTD}
- \texttt{TYPEFAROB}
- \texttt{TYPEFLTX}
- \texttt{TYPEFLTDX}
10  Broad-Fork Operation

The following routine will fork a copy of func on all clusters simultaneously. All arguments will be copied on each instance of each routine.

PS_BROADFORK(function func, -- the C function to call
int num, -- number of arguments
<ARGTYPE> type1, -- first arguments type
arg1, -- first argument
...
)

11  Replicated Heap Allocation

The following routine will allocate memory on the replicated heap. It returns a near pointer (an OB) to the caller’s portion of the replicated heap.

OB PS_REP_HEAP_ALLOC(int number_bytes);

References

    http://www.icsi.berkeley.edu/Sather/ps/manual.ps

    http://www.icsi.berkeley.edu/Sather/ps/psather.ps