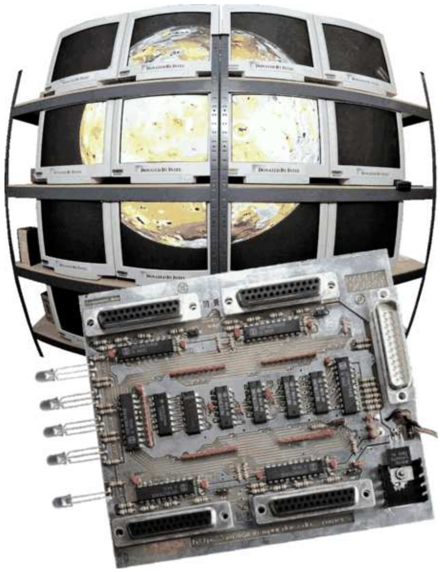


AFAPI

Aggregate Function
Application Program Interface

Versions from 980801

Generic C reference for AFAPI:

- *SHMAPERS generic unix uniprocessor*
- *SHMAPERS generic unix SMP*
- *CAPERS Linux clusters*
- *WAPERS Linux clusters*
- *TTL_PAPERS Linux clusters*

`http://garage.ecn.purdue.edu/~papers/`

Prof. Hank Dietz & “The Aggregate”
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

Average *n*-PE communication complexity for SHMAPERS is *O(n)*. CAPERS works only with 2 PEs. For WAPERS or TTL_PAPERS, each function is marked: ▫ is *O(1)*, □ is <*O(n)*, and ☐ is >=*O(n)*.

All processors act synchronously to perform each `p_` prefix operation. Any processor can independently call an asynchronous signal routine (`s_` prefix). Simultaneous use of `p_` and `s_` routines is illegal.

Include files

`#include <AFAPI/afapi.h>`
C header file for general use

`#include <AFAPI/inline.h>`
Definitions of inline primitives (`p__` and `s__`)

Data Types

GCC Declaration	Type Suffix
int (barrier group bit mask)	**bar**
int (boolean, returned as 0 or 1)	**1u**
char	**8**
unsigned char	**8u**
short	**16**
unsigned short	**16u**
int	**32**
unsigned int	**32u**
long long	**64**
unsigned long long	**64u**
float	**32f**
double	**64f**
void *	**ptr**
(**void *** with **int** byte count)	**z**

$\delta \in \{$**1u**, **8**, **8u**, **16**, **16u**, **32**, **32u**, **64**, **64u**$\}$

$\gamma \in (\delta \cup \{$**32f**, **64f**$\})$

$\beta \in (\delta \cup \{$**bar**, **ptr**$\})$

$\alpha \in (\delta \cup \{$**bar**, **32f**, **64f**, **ptr**$\})$

p_α
AFAPI portable type name; e.g., **unsigned int** is **p_32u**, **double** is **p_64f**, **char** is **p_8**

Constants and Special Variables

(const int) NPROC
Number of PEs; compile-time constant

(const int) IPROC
PE number, 0..**NPROC**-1

(const int) CPROC
PE number of console/control PE

Control

int p_init(void)
☐ Check-in with AFAPI; return 0 for no error

void p_exit(void)
▫ Check-out with AFAPI

int p_confirm(void)
☐ Confirm that AFAPI is functioning correctly; return 0 for no error

Primitives

void p_wait(void)
▫ Barrier synchronization

p_bar p_waitbar(p_1u f)
□ Bit mask with 2^k set *iff* PE*k* **f**!=0

p_1u p_any(p_1u f)
▫ Was **f**!=0 on any enabled PE?

p_1u p_all(p_1u f)
▫ Was **f**!=0 on all enabled PEs?

Communications

*(**n** must have the same value on all PEs; PEs where **a**==0 do not store results)*

p_α **p_bcastPut**α**(p_**α **d)**
with **p_**α **p_bcastGet**α**(void)**
▫ Broadcast **d** from one PE to all

void p_bcastPutz(void *b, int n)
with **void p_bcastGetz(void *a, int n)**
▫ Broadcast **n** bytes at ***b** to ***a**

p_α **p_putGet**α**(p_**α **d, int s)**
□ Put **d**, get (return) **d** from PE **s**

void p_putGetz(void*a, void*b, int n, int s)
□ Put **n** bytes from ***b**, get to ***a**

void p_gatherα**(p_**α ***a, p_**α **d)**
☐ Gather an array **a[***i***]** = **d** from PE *i*

void p_gatherz(void *a, void *b, int n)
☐ Gather **n**-byte objects at ***b** into ***a**

p_α **p_scatPut**α**(p_**α ***a)**
with **p_**α **p_scatGet**α**(void)**
☐ Scatter returns **a[IPROC]**

void p_scatPutz(void *a, void *b, int n)
with **void p_scatGetz(void *a, int n)**
☐ Scatter **n**-byte objects at ***b** into ***a**

p_α **p_all2all**α**(p_**α ***a, p_**α ***b)**
☐ All-to-all **a[***i***]** = **b[IPROC]** from PE *i*

void p_all2allz(void *a, void *b, int n)
☐ All-to-all **n**-byte objects at ***b** into ***a**

Associative Reductions & Scans

(Parallel prefix uses **scan** *rather than* **reduce***; complexities are given as* **reduce** */* **scan***)*

p_β p_reduceAndβ(p_β d)
▫ / □ Bitwise AND of **d** from each PE

p_β p_reduceOrβ(p_β d)
▫ / □ Bitwise OR of **d** from each PE

p_β p_reduceXorβ(p_β d)
□ / ☐ Bitwise XOR of **d** from each PE

p_α p_reduceMinα(p_α d)
▫ / □ Minimum of **d** from each PE

p_α p_reduceMaxα(p_α d)
▫ / □ Maximum of **d** from each PE

p_γ p_reduceAddγ(p_γ d)
☐ / ☐ Sum of **d** from each PE

p_γ p_reduceMulγ(p_γ d)
☐ / ☐ Product of **d** from each PE

Voting & Scheduling

int p_first(p_1u f)
□ Minimum **IPROC** of PE with **f**!=0; **NPROC** if no PE had **f**!=0

int p_count(p_1u f)
□ Count number of PEs with **f**!=0

int p_quantify(p_1u f)
□ Quantify number of PEs with **f**!=0; return is 0, 1, 2 (meaning >1), or **NPROC**

int p_voteCount(int v)
☐ How many PEs voted for our **IPROC**?

p_bar p_vote(int v)
☐ Bit vector with 2^k set *iff* PE*k* **v==IPROC**

int p_matchCountβ(p_β v)
☐ How many PEs had our **v** value?

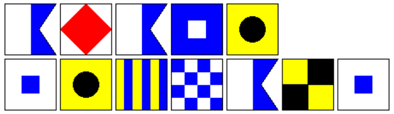
p_bar p_matchβ(p_β v)
☐ Bit vector with 2^k set *iff* PE*k* had our **v** value

int p_rankα(p_α d)
☐ Position of **d** in sorted order, smallest is 0

int p_rankedα(p_α d, int r)
☐ **IPROC** of PE where **d** has rank **r**

Partitioning

AFAPI soon will support structured dynamic partitioning so that barrier groups can be subdivided and recombined. Without syntax changes, the calls listed here will operate on the *current barrier group*.



Signals

Signals must be installed by all PEs prior to their use (**s_install()** can be called before **s_init()**). The sender of a signal is blocked until all processors respond, either by **s_signal()** or **s_poll()**. When all processors enter a signal handler, the argument value is a **p_bar** mask identifying all concurrent senders of that signal. An independent data structure, and/or **p_** communications, must be used to pass additional arguments to a signal handler.

int s_install(int s, void(*f)(p_bar m))
▫ Install signal handler **f()**; **s** can be **S_SIGANY** or a previously assigned signal number; return is signal number or -1 for failure

void s_ignore(p_bar m)
▫ Default (null) signal handler; install this over a user handler to disable the handler

int s_init(void)
▫ Enable signal system, return 0 for no error

void s_exit(void)
▫ Flush and disable signal system

void s_signal(int s)
▫ Send signal **s** to all PEs; **s_signal(S_SIGANY)** waits for a signal

p_1u s_pending(void)
▫ return 1 if signal(s) are pending, else 0

void s_poll(void)
▫ Poll for & service pending signal(s)

void s_wait(void)
▫ Barrier synchronization after flushing signals

Libraries

AFAPI libraries for common math functions (**AFMATH**), driving video walls (**AFVWALL** and **AFXWALL**), implementing replicated shared memory (**AFRSM**), and even implementing the Message Passing Interface 2.0 standard (**AFMPI**), have been built. Any of these may be installed and used simultaneously, provided that the AFAPI was able to **s_install()** all the signal handlers they use.

Sample Program

Compute approximate value of π; algorithm from M. J. Quinn, *Parallel Computing Theory And Practice*, McGraw Hill, 1994.

```
#include <stdio.h>
#include <AFAPI/afapi.h>

main(int argc, char **argv)
{
  p_32f width, sum;
  p_32 intervals, i;

  /* check-in with AFAPI */
  if (p_init()) exit(1);

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the local computations */
  sum = 0;
  for (i=IPROC; i<intervals; i+=NPROC) {
    p_32f x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }

  /* sum local results & scale */
  sum = p_reduceAdd32f(sum) * width;

  /* only the console PE prints */
  if (IPROC == CPROC) {
    printf("Pi is %14.12f\n", sum);
  }

  /* check-out */
  p_exit();
  exit(0);
}
```

Compile & Run Commands

Various environment variables are used to determine the AFAPI version and PE numbering; use a **make** file to compile. Compiled SHMAPERS AFAPI executable files can be run directly, if environment variables have been set. Other versions may require "helper" programs **afrun**, **afcleanup**, etc. See the **README** for your particular AFAPI version.