TTL_PAPERS

TTL implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization

AFAPI

Aggregate Function Application Program Interface

Version 960801

C Reference Card

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

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

NOTE: TTL_PAPERS communication complexity using *n* PEs is indicated for each function by:
▫ constant
□ often fewer than *n* transmissions
☐ at least *n*-1 transmissions

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

`afapi.h`
C header file for general use

`inline.h`
Definitions of inline primitives (`p__` and `s__`)

Data Types

GCC Declaration	Type Suffix
`int` (barrier group bit mask)	**`bar`**
`int` (boolean)	**`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`**
(byte block, compressed)	**`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

AFAPI Aggregate Functions

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

`p_`α **`p_bcastGet`**α**`(void)`**
▫ Broadcast get value from sending PE

`void p_bcastPut`α**`(p_`**α **`d)`**
▫ Broadcast put **`d`** from this PE

`void p_bcastGetz(void* a, int n)`
▫ Broadcast get **`n`** bytes into address **`a`**

`void p_bcastPutz(void* a, int n)`
▫ Broadcast put **`n`** bytes from address **`a`**

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

`void p_gather`α**`(p_`**α **`*p, p_`**α **`d)`**
☐ Gather an array **`p[`***i*`]`** = **`d`** from PE *i*

`p_`α **`p_scatPut`**α**`(p_`**α **`*p)`**
☐ Scatter put so PE *i* gets **`p[`***i*`]`**

`p_`α **`p_scatGet`**α**`(void)`**
☐ Scatter get

Associative Reductions

`p_`β **`p_reduceAnd`**β**`(p_`**β **`d)`**
▫ Bitwise AND of **`d`** from each PE

`p_`β **`p_reduceOr`**β**`(p_`**β **`d)`**
▫ Bitwise OR of **`d`** from each PE

`p_`β **`p_reduceXor`**β**`(p_`**β **`d)`**
□ Bitwise XOR of **`d`** from each PE

`p_`α **`p_reduceMin`**α**`(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

Scans (Parallel Prefix)

p_β p_scanAndβ(p_β d)
- □ Bitwise AND of **d** up to this PE

p_β p_scanOrβ(p_β d)
- □ Bitwise OR of **d** up to this PE

p_β p_scanXorβ(p_β d)
- ☐ Bitwise XOR of **d** up to this PE

p_α p_scanMinα(p_α d)
- □ Minimum of **d** up to this PE

p_α p_scanMaxα(p_α d)
- □ Maximum of **d** up to this PE

p_γ p_scanAddγ(p_γ d)
- ☐ Sum of **d** up to this PE

p_γ p_scanMulγ(p_γ d)
- ☐ Product of **d** up to this PE

Voting and 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 (>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

AFAPI Signals

The AFAPI aggregate signal system provides replicated shared memory and allows arbitrary user-installed handlers. The sender of a signal is blocked until all processors respond, either by sending a signal or **s_poll()**. Similar concurrent signals can be processed as a group, identified to the signal handler by the **p_bar** argument.

Aggregate Signal Control

void s_ignore(p_bar m)
- Default (null) signal handler

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

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

void s_exit(void)
- Flush and disable signal system

Aggregate Signal Primitives

void s_signal(int s)
- Broadcast signal **s**

void s_poll(void)
- Poll for pending signal(s)

void s_wait(void)
- Barrier synchronization after flushing signals

Replicated Shared Memory

void s_update(void *a, int n)
- Shared memory update (write) **n**-byte object at address **a** to all PEs; result is fully coherent RSM

void s_initLock(s_lock *l)
- Initialize shared memory lock **l** to **S_INITLOCK**

void s_acquire(s_lock *l)
- Acquire shared memory lock **l**

void s_release(s_lock *l)
- Release shared memory lock **l**

AFAPI 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.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);
}
```

AFAPI Compile and Run

gcc *source*.**c -o** *exec* -O afapi.a
- C compile command

afrun *exec args*
- Run the executable *exec* using **rsh**-based script