HCLIB: A TASK-BASED PARALLEL PROGRAMMING MODEL

Vivek Kumar (IIIT, New Delhi)
Vivek Sarkar (Georgia Tech, Atlanta)

Acknowledgements

- Alina Sbîrlea
- Deepak Majeti
- Jonathan Sharman
- Max Grossman
- Nick Vrvilo
- Saĝnak Taşırlar
- Sanjay Chatterjee
- Vincent Cavé
- Vivek Kumar
- Vivek Sarkar
- Yonghong Yan
- Zoran Budimlić

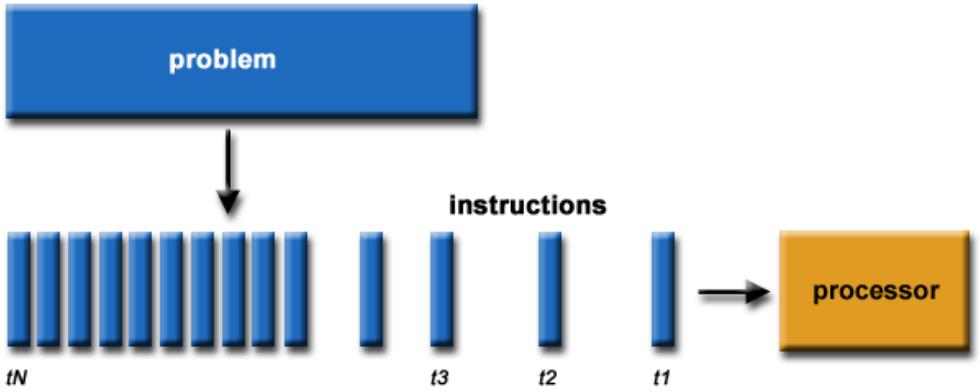
Prerequisites

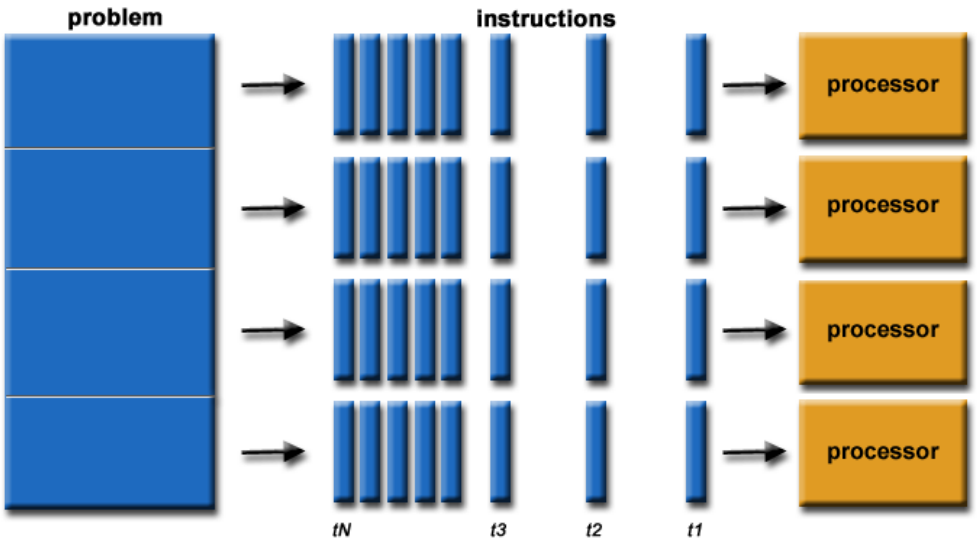
- Mandatory
  - Proficient in C/C++ programming
- Optional
  - Exposure to any parallel programming models such as OpenMP, MPI, Cilk, TBB, etc.

Tutorial Outline

- **Parallel Programming**
  - **What, Why, How?**
- Parallel Programming using Habanero C/C++ library (HClib)
  - Task parallelism (async, finish)
  - Loop parallelism (forasync)
  - Functional parallelism (futures, promises)
  - Dataflow parallelism (async-await)
- Advanced topics: programming for processor affinities, heterogeneous parallelism, cluster parallelism
- Appendix: HClib installation and usage information

What is Parallel Programming?





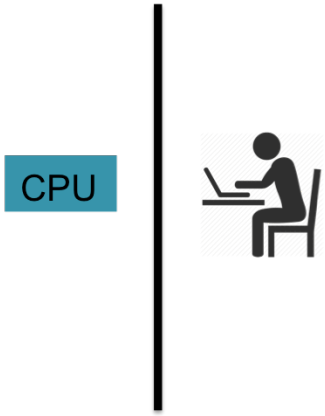
Source: https://computing.llnl.gov/tutorials/parallel_comp/

- Serial
  - One instruction at a time
- Parallel
  - Multiple instructions in parallel

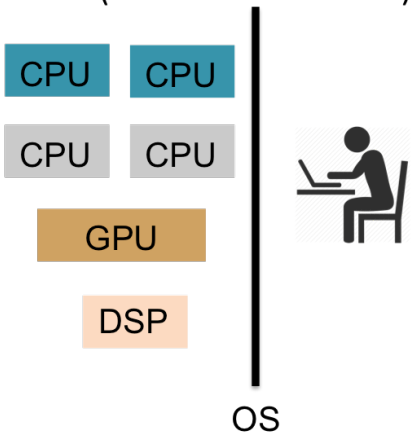
Why Parallel Programming?

Technology Push

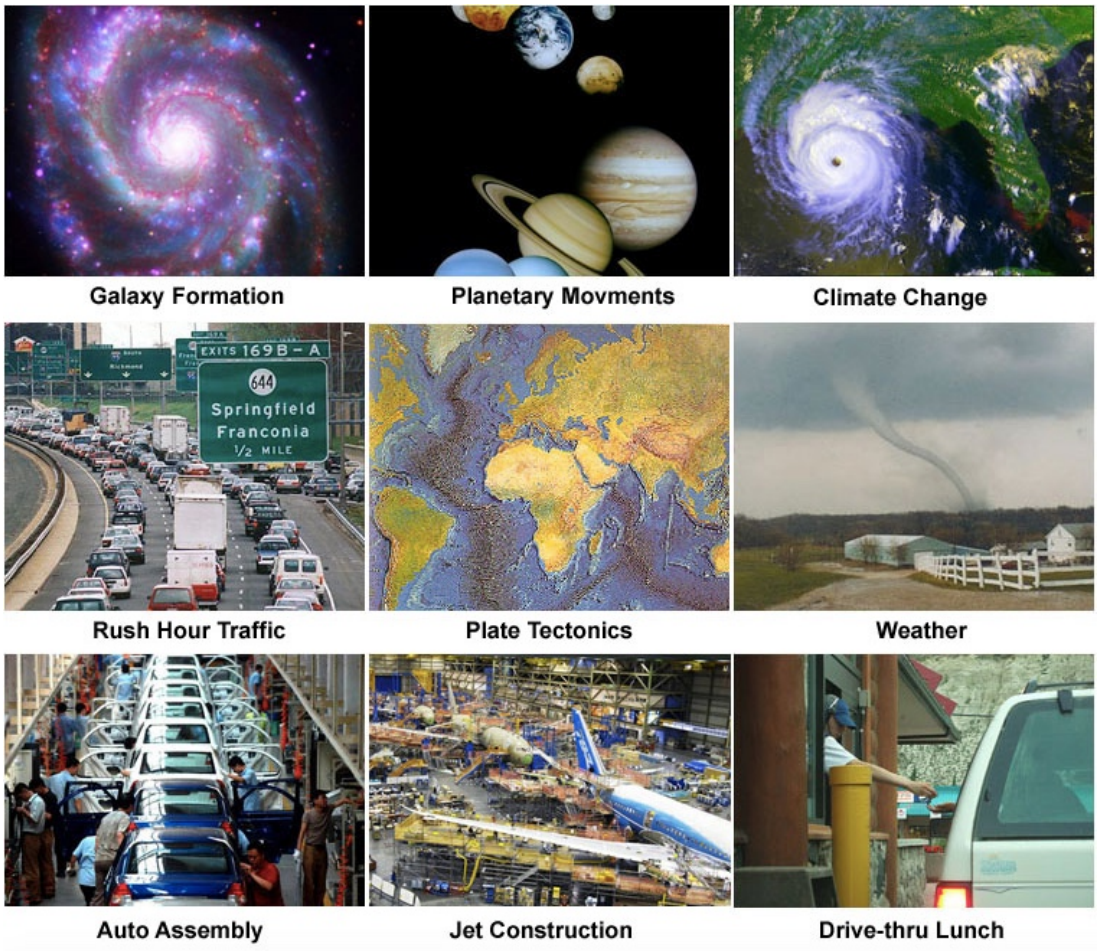
1990's and early 2000s



Today
(no more free lunch)



Application Push



How Parallel Programming?

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

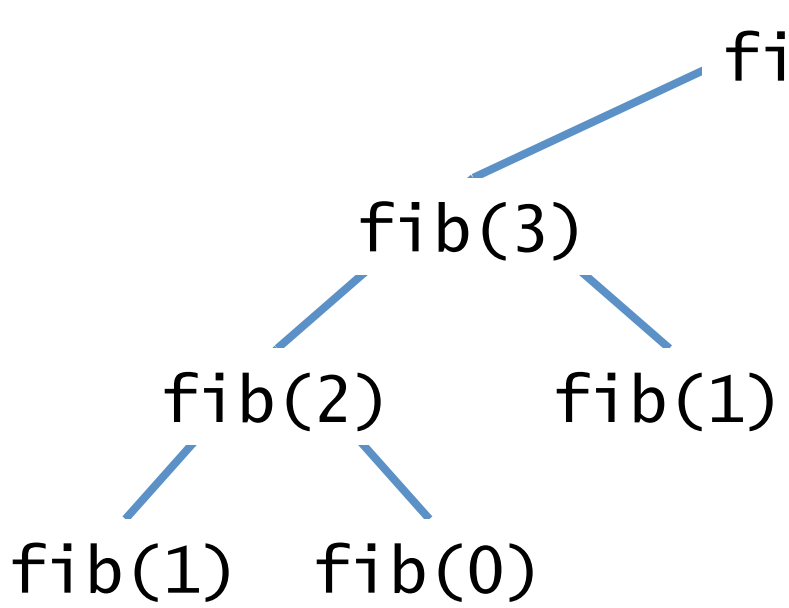
```c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

int main(int argc, char *argv[]) {
  uint64_t n = atoi(argv[1]);
  uint64_t result = fib(n);
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Disclaimer to Algorithms Police

This recursive program is a poor way to compute the nth Fibonacci number, but it provides a good didactic example.

Can we write a parallel version of this **Fibonacci** code using Pthreads?

How to Parallelize Fibonacci using async and finish Constructs ?




```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}
```

Key idea for parallelization

The calculations of `fib(n-1)` and `fib(n-2)` can be executed simultaneously without mutual interference.

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

How Parallel Programming?

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Pthread based implementation of Fibonacci program

How Parallel Programming?

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Original code.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Pthread based implementation of Fibonacci program

How Parallel Programming?

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Structure for thread arguments.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread based implementation of Fibonacci program

How Parallel Programming?

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Function called when thread is created.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread based implementation of Fibonacci program

How Parallel Programming?

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

No point in creating thread if there isn't enough to do.

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread based implementation of Fibonacci program

How Parallel Programming?

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1]
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Marshal input argument to thread.

Pthread based implementation of Fibonacci program

How Parallel Programming?

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Create thread to execute `fib(n-1)`.

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread based implementation of Fibonacci program

How Parallel Programming?

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(v
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Main program executes fib(n-2) in parallel.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Pthread based implementation of Fibonacci program

How Parallel Programming?

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Block until the auxiliary thread finishes.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread based implementation of Fibonacci program

How Parallel Programming?

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Add the results together to produce the final output.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Pthread based implementation of Fibonacci program

How Parallel Programming?

Issues with Pthreads

Overhead	The cost of creating a thread >10^4 cycles ⇒ coarse-grained concurrency. (Thread pools can help.)
Scalability	Fibonacci code gets at most about 1.5 speedup for 2 cores. Need a rewrite for more cores.
Modularity	The Fibonacci logic is no longer neatly encapsulated in the fib() function.
Code Simplicity	Programmers must marshal arguments (shades of 1958!) and engage in error-prone protocols in order to load-balance.

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Tutorial Outline

- Parallel Programming
  - What, Why, How?
- **Parallel Programming using Habanero C/C++ library (HClib)**
  - **Task parallelism (async, finish)**
  - Loop parallelism (forasync)
  - Functional parallelism (futures, promises)
  - Dataflow parallelism (async-await)
- Advanced topics: programming for processor affinities, heterogeneous parallelism, cluster parallelism
- Appendix: HClib installation and usage information

HClib: Parallel Programming with Ease

- What is Habanero C library?
  - Tasks based parallel programming library
  - A C/C++ implementation of Habanero parallel programming constructs
  - Shares ideas with Habanero Java library
  - C++ APIs are based on C++11 lambda functions
  - Supports heterogeneous (GPU) computing
  - Integrated with HPC libraries such as OpenSHMEM and UPC++ for hybrid shared and distributed memory parallel programming

Async and Finish Statements for Task Creation and Termination

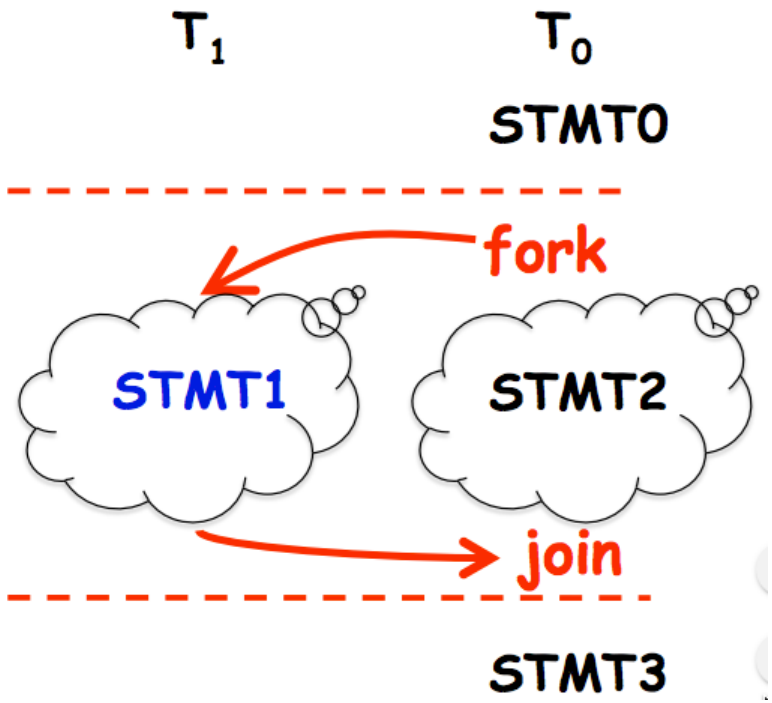
async S

- Creates a new child task that executes statement S

finish S

- Execute S but wait until all async in S's scope have terminated

```
// T0 (Parent task)
STMT0;
finish {    //Begin finish
  async {
    STMT1; //T1 (Child task)
  }
  STMT2;    //Continue in T0
            //Wait for T1
}           //End finish
STMT3;      //Continue in T0
```



HClib: Parallel Programming with Ease

```
#include <inttypes.h>
#include "hclib_cpp.h"

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    hclib::finish([&x, &y, n]() {
        hclib::async([&x, n]() {
            x = fib(n-1);
        }); //end of async
        y = fib(n-2);
    }); //end of finish
    return (x + y);
  }
}
```

```
int main(int argc, char *argv[]) {
  char const *deps[] = {"system"};
  hclib::launch(deps, 1, [&]() {
    uint64_t n = atoi(argv[1]);
    uint64_t result = fib(n);
    printf("Fibonacci of %" PRIu64 "
      is %" PRIu64 ".\n", n, result);
  });
  return 0;
}
```

High productivity due to serial elision

- In most cases, async-finish can be simply added in any sequential algorithm (without any significant changes) to get the corresponding parallel version of the algorithm

Launching HClib

```
int main(int argc, char ** argv) {
    char const *deps[] = { "system" };
    hclib_launch(deps, 1, program_lambda); //HCLib program running
    // HCLib program is finished, back to sequential code
    ….
}
```

Async and Finish using the C++ Interface

```
hclib::async ( [capture_list] ( )
{
  <Stmt>
});

hclib::finish ([capture_list] ( ) {
  <Stmt>
});
```

```
using namespace hclib;

async ( [capture_list] ( ) {
  <Stmt>
});

finish ([capture_list] ( ) {
  <Stmt>
});
```

Async and Finish using the C Interface

```
void async_fct(void * arg) {<Stmt>}

/* void hclib_async(generic_frame_ptr fp, void *arg,
      hclib_future_t **futures, const int nfutures,
      hclib_locale_t *locale); */

hclib_async(async_fct, NULL, NO_FUTURE, 0, ANY_PLACE);

hclib_start_finish();
hclib_end_finish();
```

Fibonacci using the C Interface

```
#include <inttypes.h>
#include "hclib.h”

typedef struct {
  uint64_t n;
  uint64_t res;
} FibArg;

void fib(void* raw_args) {
  FibArgs* args = raw_args;
  if (args->n < 2) {
    args->res = n;
  } else {
    FibArgs left = {args->n-1, 0};
    FibArgs right = {args->n-2, 0};
    hclib_start_finish();
    hclib_async(fib, &left, NO_FUTURE,
                        0, ANY_PLACE);
    fib(&right);
    hclib_end_finish();
    args->res = left.res + right.res;
  }
}
```

```
void taskMain(void* raw_args) {
  char** argv = raw_args;
  int n = atoi(argv[1]);
  FibArgs args = {n, 0};
  fib(&args);
  printf("Fibonacci of %" PRIu64 "
    is %" PRIu64 ".\n”,n ,args->res);
}
```

```
int main(int argc, char *argv[]) {
  char const *deps[] = {“system”};
  hclib_launch(taskMain, argv,
                      deps, 1);
  return 0;
}
```


Tutorial Outline

- Parallel Programming
  - What, Why, How?
- **Parallel Programming using Habanero C/C++ library (HClib)**
  - Task parallelism (async, finish)
  - **Loop parallelism (forasync)**
  - Functional parallelism (futures, promises)
  - Dataflow parallelism (async-await)
- Advanced topics: programming for processor affinities, heterogeneous parallelism, cluster parallelism
- Appendix: HClib installation and usage information

Loop Level Parallelism in HClib

```
void foo() {
  loop_domain_1d *loop = new loop_domain_1d (SIZE)};
  finish([=]() {
    forasync1D (loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, false, FORASYNC_MODE_RECURSIVE);
  });
}
```

- loop_domain_1d *loop = new loop_domain_1d (lowBound, highBound, tileSize, loopStride};
- loop_domain_1d *loop = new loop_domain_1d (lowBound, highBound, tileSize};
- loop_domain_1d *loop = new loop_domain_1d (lowBound, highBound};

- loop_domain_2d *loop = new loop_domain_2d (highBound1, highBound2}; // simplest
- loop_domain_3d *loop = new loop_domain_3d (highBound1, highBound2, highBound3};

- forasync1D(loop_domain_1d* loop, lambda_function, seq, mode); //similarly forasync2D & 3D
- "seq": boolean value to specify sequential or parallel execution. By default its false.
- "mode": for runtime optimizations. This is also optional argument with default as recursive
  - FORASYNC_MODE_RECURSIVE: recursively partition total iteration space until "tileSize" is reached
  - FORASYNC_MODE_FLAT: chunk iterations into blocks of length "tileSize"

Parallelizing Vector Addition

```
for (uint64_t i=0; i<N; i++) {
  C[i] = A[i] + B[i];
}
```

- Sequential vector addition
  - How to parallelize using forasync1d?

Parallelizing Vector Addition

```
loop_domain_1d *loop_info = new loop_domain_1d(N);

finish {[=]() {
  forasync1D (loop_info, [=] (uint64_t i) {
      C[i] = A[i] + B[i];
  }, false, FORASYNC_MODE_RECURSIVE);
});
```

- Parallel vector addition
  - High productivity using forasync1D

HClib Documentation

- http://habanero-rice.github.io/hclib/
- https://budimlic.github.io/hclib/

Tutorial Outline

- Parallel Programming
  - What, Why, How?
- **Parallel Programming using Habanero C/C++ library (HClib)**
  - Task parallelism (async, finish)
  - Loop parallelism (forasync)
  - **Functional parallelism (futures, promises)**
  - Dataflow parallelism (async-await)
- Advanced topics: programming for processor affinities, heterogeneous parallelism, cluster parallelism

Functional Parallelism: Adding Return Values to Async Tasks

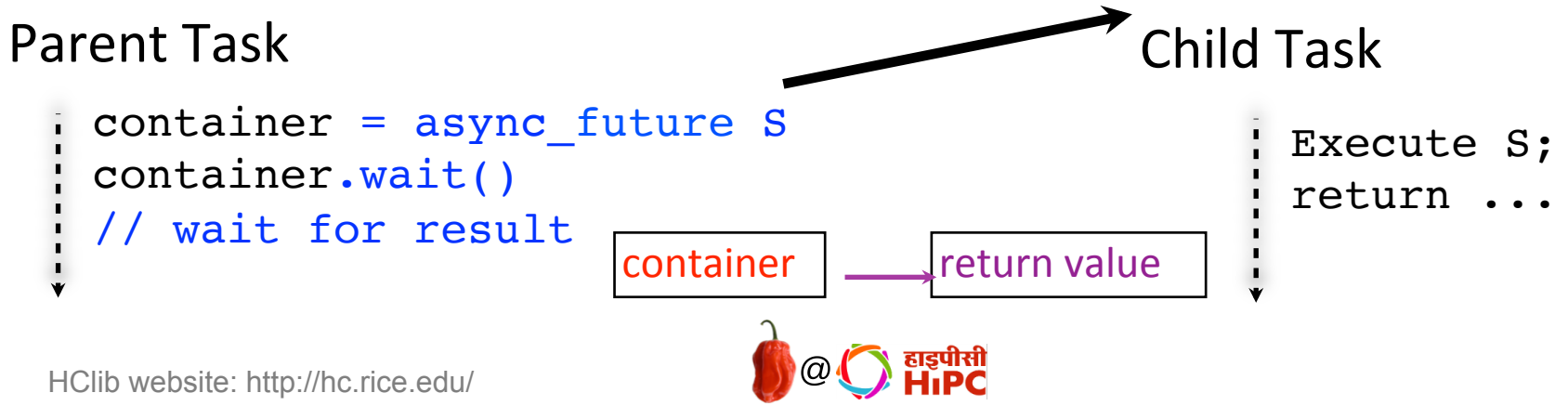
Example Scenario (C++ pseudocode, similar syntax for Java)

```
// Parent task creates child async task
future_t<Integer> container = async_future S;
. . .
// Later, parent examines the return value
Integer sum = container.wait();
```

Two issues to be addressed:

1) Distinction between container and value in container (box)

2) Synchronization to avoid race condition in container accesses



HClib Futures: Tasks with Return Values

future_t<T> *f = async_future { S }

- Creates a new child task that executes S, which must terminate with a return statement and return value
- Async expression returns a pointer to a container of type future_t

T result = f.wait();

- wait() evaluates f and blocks if f's value is unavailable
- Unlike finish which waits for all tasks in the finish scope, a wait operation only waits for the specified async_future

Example: Two-way Parallel Array Sum using Future Tasks

```
1. // Parent Task T1 (main program)
2. // Compute sum1 (lower half) & sum2 (upper half) in parallel
3. future_t<Integer> sum1 = async_future { // Future Task T2
4.   int sum = 0;
5.   for(int i = 0; i < X.length / 2; i++) sum += X[i];
6.   return sum;
7. };
8. future_t<Integer> sum2 = async_future { // Future Task T3
9.    int sum = 0;
10.  for(int i = X.length / 2; i < X.length; i++) sum += X[i];
11.  return sum;
12. };
13. // Task T1 waits for Tasks T2 and T3 to complete
14. int total = sum1.wait() + sum2.wait();
```


Background: Functional Programming

- Eliminate side-effects
  - emphasizes functions whose results that depend only on their inputs and not on any other program state
  - calling a function, f(x), twice with the same value for the argument x will produce the same result both times

Helpful Link: http://en.wikipedia.org/wiki/Functional_programming

Another Example: Binomial Coefficient

- The coefficient of the x^k term in the polynomial expansion of the binomial power $(1 + x)^n$
- Number of sets of k items that can be chosen from n items
- Indexed by *n* and *k*
  - written as C(n, k)
  - read as “n choose k”
- Factorial Formula: $C(n, k) = \left(\frac{n!}{k!(n-k)!} \right)$
- Recursive Formula

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$$

Base cases: $C(n, n) = C(n, 0) = C(0, k) = 1$

Helpful Link: http://en.wikipedia.org/wiki/Binomial_coefficient

Example: Binomial Coefficient (Recursive Sequential version)

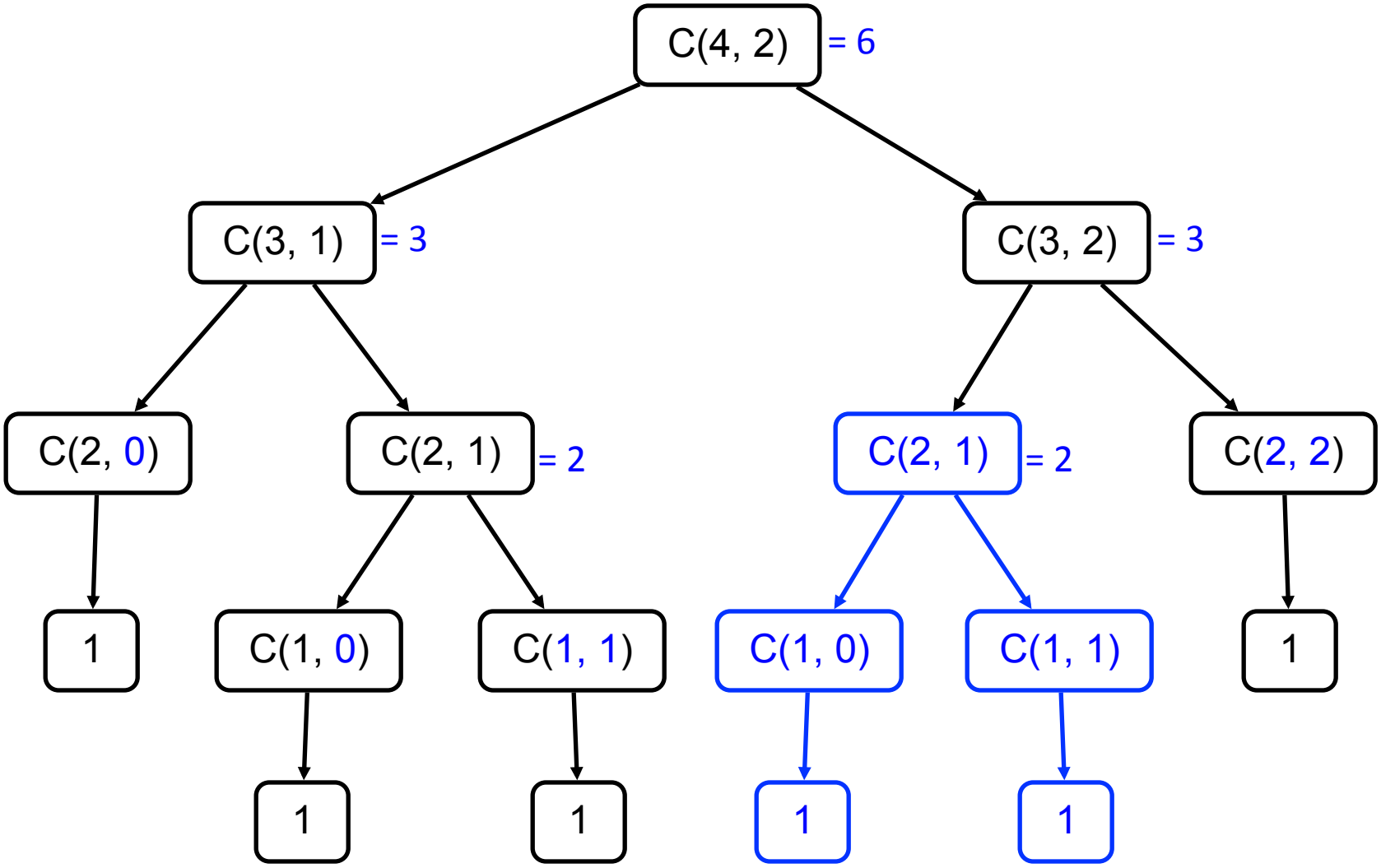
```
1. int choose(int N, int K) {
2.     if (N == 0 || K == 0 || N == K) {
3.         return 1;
4.     }
5.     int left  = choose (N-1, K - 1);
6.     int right = choose (N-1, K);
7.     return left + right;
8. }
```


Example: Binomial Coefficient (Parallel Recursive Pseudocode)

```
1.  Integer choose(int N, int K) {
2.      if (N == 0 || K == 0 || N == K) {
3.          return 1;
4.      }
5.      future_t<Integer> left  =
6.                  async_future { return choose (N-1, K-1); }
7.      future_t<Integer> right =
8.                   async_future { return choose (N-1, K); }
9.      return left.wait() + right.wait();
10. }
```

- Use of futures supports incremental parallelization with low developer effort

What inefficiencies do you see in the recursive Binomial Coefficient algorithm?



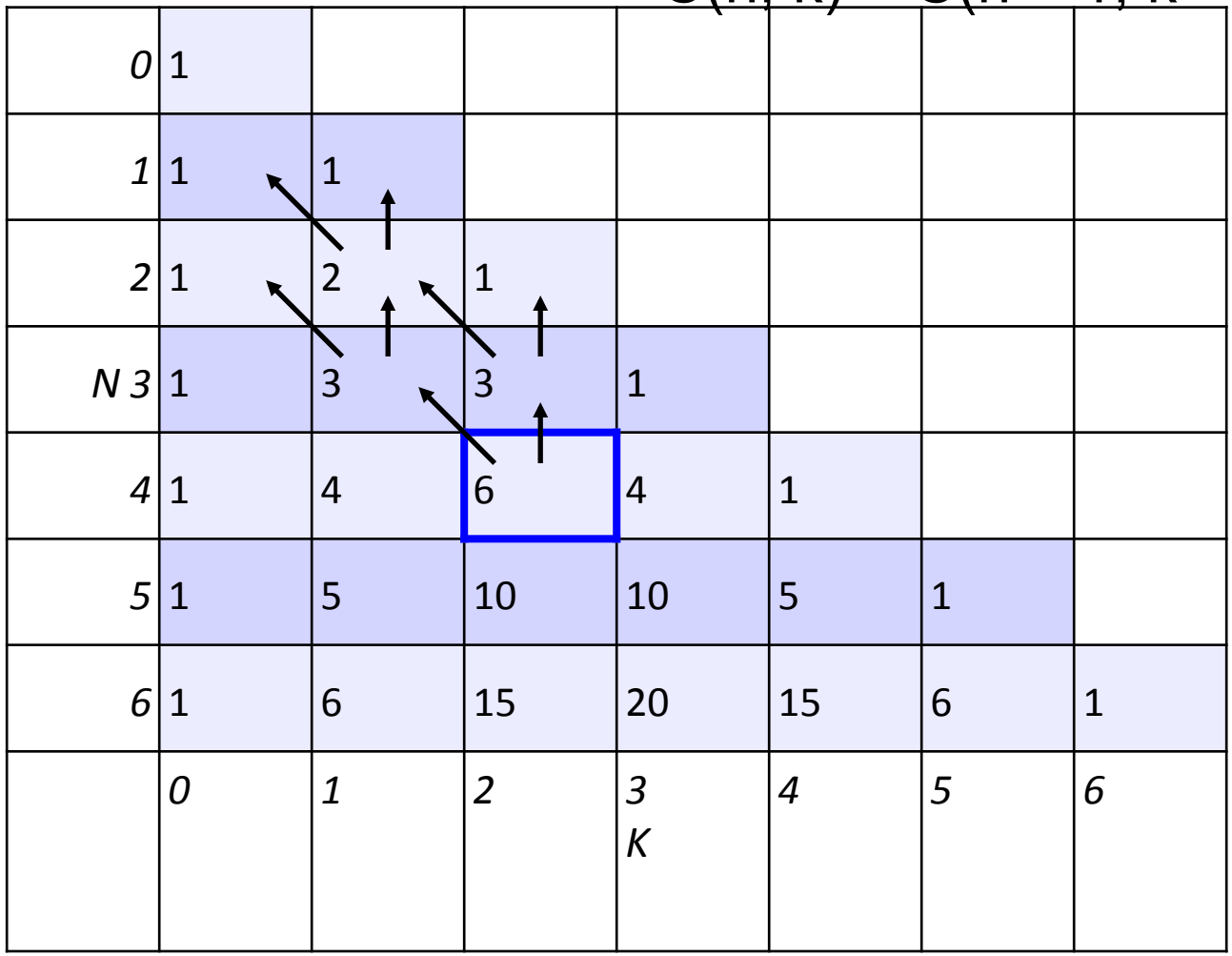
Memoization

- Memoization - saving and reusing previously computed values of a function rather than recomputing them
  - A optimization technique with space-time tradeoff
- A function can only be memoized if it is *referentially transparent*, i.e. functional
- Related to caching
  - memoized function "remembers" the results corresponding to some set of specific inputs
  - memoized function populates its cache of results transparently on the fly, as needed, rather than in advance

Helpful Link: http://en.wikipedia.org/wiki/Memoization

Pascal's Triangle is an example of Memoization

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$$



Example: Binomial Coefficient (sequential memoized version)

```
1.  Map<Pair<Int, Int>, Int> cache = new ...;
2.  int choose(int N, int K) {
3.      Pair<Int, Int> key = Pair.factory(N, K);
4.      if (cache.contains(key)) {
5.          return cache.get(key);
6.      }
7.      if (N == 0 || K == 0 || N == K) {
8.          return 1;
9.      }
10.     int left  = choose (N - 1, K - 1);
11.     int right = choose (N - 1, K);
12.     int result = left + right;
13.     cache.put(key, result);
14.     return result;
15. }
```

Example: Binomial Coefficient (parallel memoized version w/ futures)

```
1.  Map<Pair<Int, Int>, future_t<Integer>> cache = new ...;
2.  Integer choose(final int N, final int K) {
3.      final Pair<Int, Int> key = Pair.factory(N, K);
4.      if (cache.contains(key)) {
5.          return cache.get(key).wait();
6.      }
7.      future_t<Integer> f = async_future {
8.        if (N == 0 || K == 0 || N == K) return 1;
9.        future_t<Integer> left = async_future { return choose (N-1, K-1); }
10.       future_t<Integer> right = async_future { return choose (N-1, K); }
12.       return left.wait() + right.wait();
13.     }
14.     f = cache.putIfAbsentElseGet(key, f);
15.     return f.wait();
16. }
```

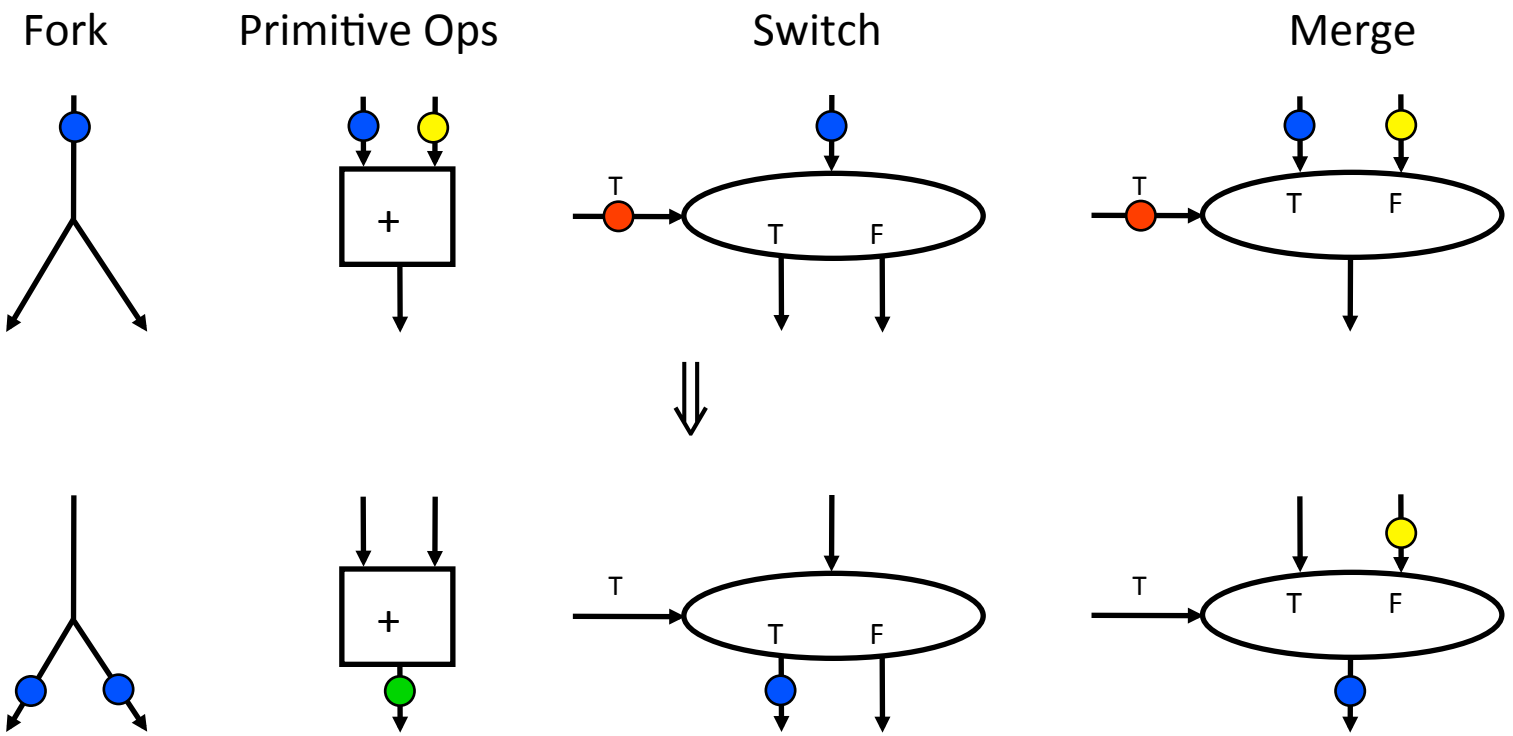
- Assumes availability of a “thread-safe” cache library, e.g., ConcurrentHashMap

Tutorial Outline

- Parallel Programming
  - What, Why, How?
- **Parallel Programming using Habanero C/C++ library (HClib)**
  - Task parallelism (async, finish)
  - Loop parallelism (forasync)
  - Functional parallelism (futures, promises)
  - **Dataflow parallelism (async-await)**
- Advanced topics: programming for processor affinities, heterogeneous parallelism, cluster parallelism

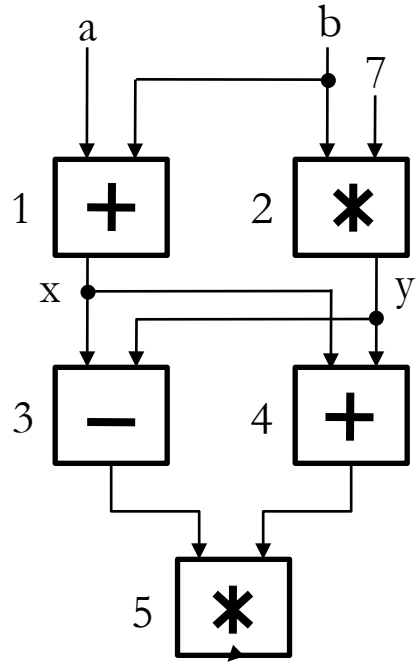
Dataflow Computing

- Original idea: replace machine instructions by a small set of dataflow operators



Example instruction sequence and its dataflow graph

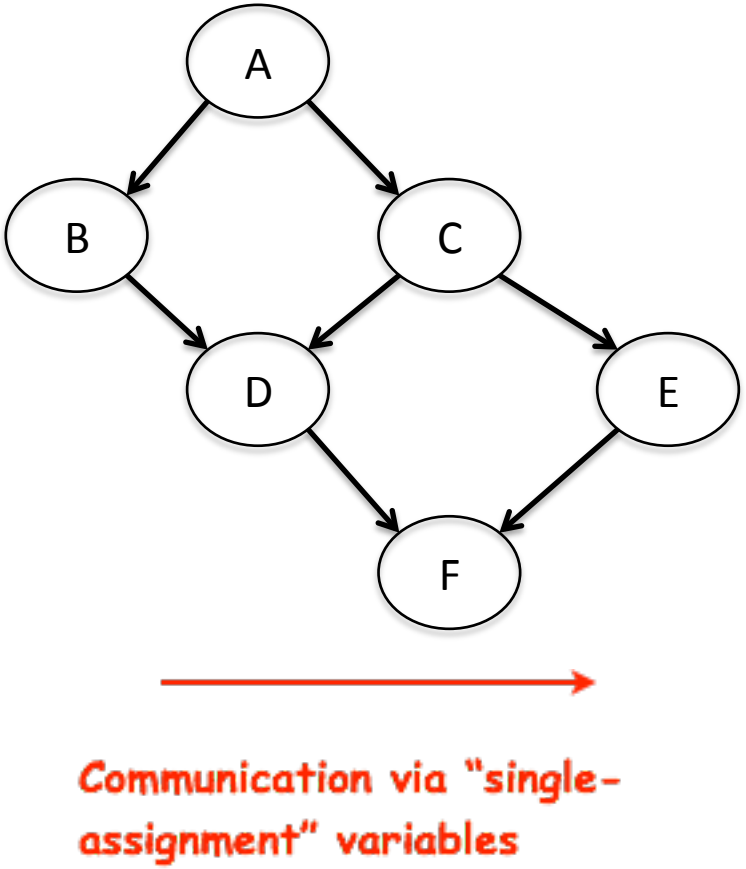
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



An operator executes <u>when all its input values are present</u>; copies of the result value are distributed to the destination operators.

No separate branch instructions

Dataflow Programming



- General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
  - Static dataflow ==> graph fixed when program execution starts
  - Dynamic dataflow ==> graph can grow dynamically
- Semantic guarantees: race-freedom, determinism
- Deadlocks are possible due to unavailable inputs (but they are deterministic in the absence of data races)

hclib::promise v/s hclib::future

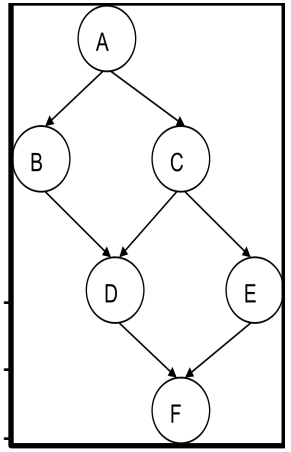
- *"A promise is an object that can store a value of type T to be retrieved by a future object (possibly in another thread), offering a synchronization point"*
  - Writable end of an object
- *"A future is an object that can retrieve a value from some provider object or function, properly synchronizing this access if in different threads"*
  - Readable end of an object

Data-Driven Task (DDT) in HClib

async_await(lambda, fObj_1, fObj_2,……,fObj_n)

- Task is kept on hold until all the future objects in the parameter list are ready with values inside them
  - i.e. the put has been performed on the promise end of each of these future objects

Using Data-Driven Tasks to implement an arbitrary Computation Graph




```
1.  hclib::promise_t<uint64_t> *pA = new hclib::promise_t<uint64
2.  hclib::promise_t<uint64_t> *pB = new hclib::promise_t<uint64
3.  hclib::promise_t<uint64_t> *pC = new hclib::promise_t<uint64
4.  hclib::promise_t<uint64_t> *pD = new hclib::promise_t<uint64_t>();
5.  hclib::promise_t<uint64_t> *pE = new hclib::promise_t<uint64_t>();
6.  hclib::async([=](){…; pA->put(…)}); // Task A
7.  hclib::async_await([=](){…; pB->put(…)}, pA->get_future());// Task B
8.  hclib::async_await([=](){…; pC->put(…)}, pA->get_future());// Task C
9.  hclib::async_await([=](){…; pD->put(…)}, pB->get_future(),
10.                    pC->get_future());// Task D
11. hclib::async_await([=](){…; pE->put(…)}, pC->get_future());// Task E
12. hclib::async_await([=](){…}, pD->get_future(),
13.                    pE->get_future());// Task F
14. // Note that creating a "producer" task after its "consumer"
15. // task is permitted with DDTs, but not with futures
16. // Statements 6, 7, 8, 9, 11, 12 can be arbitrarily permuted!
```


Differences between Futures and DDFs/DDTs

- Consumer task blocks on get() for each future that it reads, whereas async-await does not start execution till all DDFs are available
- Future tasks cannot deadlock (assuming no data races), but it is possible for a DDT to block indefinitely ("deadlock") if one of its input DDFs never becomes available
- DDTs and DDFs are more general than futures
  - Producer task can only write to a single future object, where as a DDT can write to multiple DDF objects
  - The choice of which future object to write to is tied to a future task at creation time, where as the choice of output DDF can be deferred to any point with a DDT
  - Consumer tasks can be created before the producer tasks in DDTs
- DDTs and DDFs can be more implemented more efficiently than futures
  - An "asyncAwait" statement does not block the worker, unlike a future.get()

What is Deadlock?

- A parallel program execution contains a deadlock if some task's execution remains incomplete due to it being blocked indefinitely awaiting some condition
  - Often arises from concurrent accesses to shared resources, but can also occur due to incorrect synchronization in parallel programming
- Example of a program with a deadlocking execution

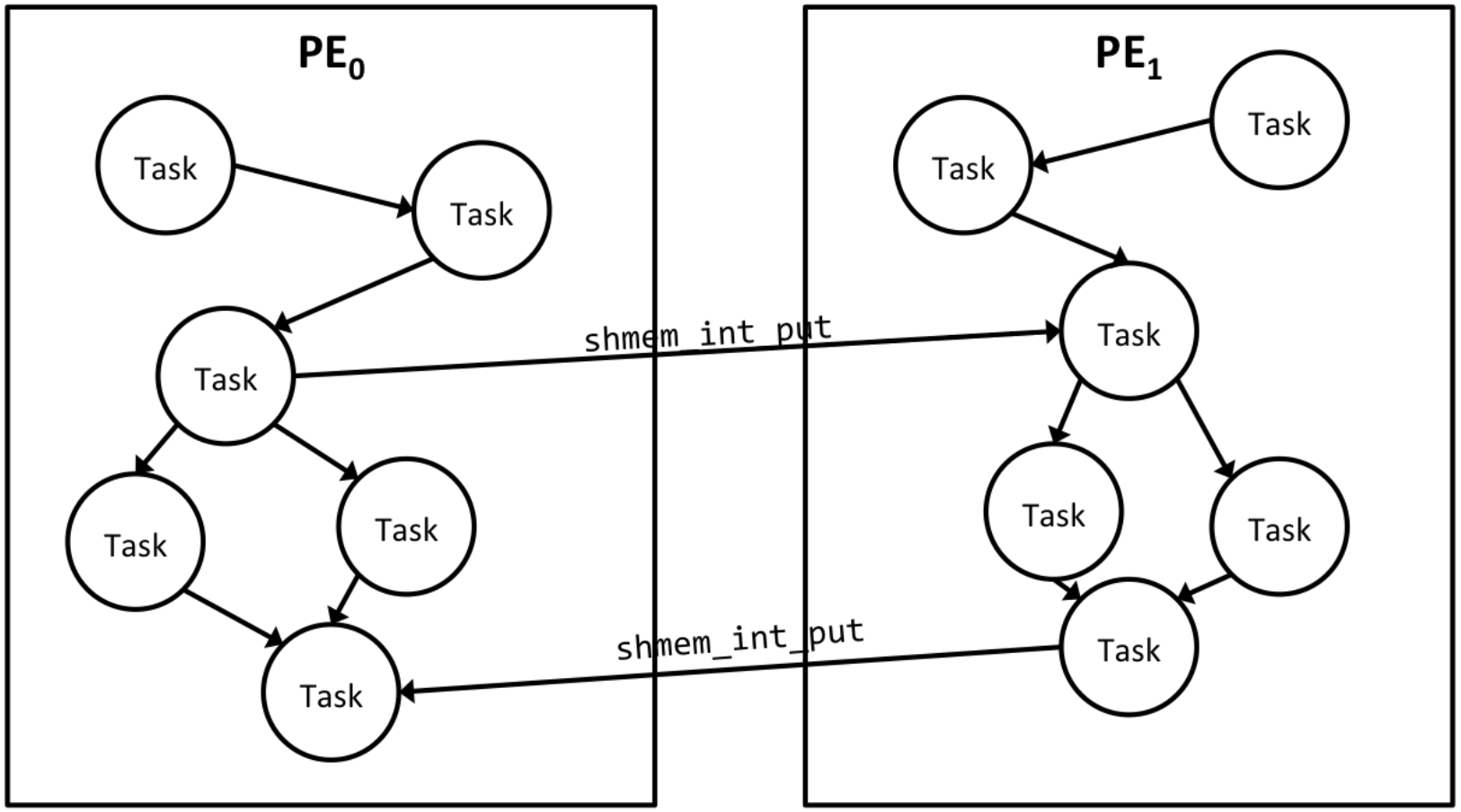
```
hclib::promise_t<uint64_t> *left = new hclib::promise_t<uint64_t>();
hclib::promise_t<uint64_t> *right = new hclib::promise_t<uint64_t>();
async_await ( left ) right.put(rightBuilder()); // Task1
async_await ( right ) left.put(leftBuilder()); // Task2
```

- In this case, Task1 and Task2 are in a deadlock cycle

Tutorial Outline

- Parallel Programming
  - What, Why, How?
- Parallel Programming using Habanero C/C++ library (HClib)
  - Task parallelism (async, finish)
  - Loop parallelism (forasync)
  - Functional parallelism (futures, promises)
  - Dataflow parallelism (async-await)
- **Advanced topics: programming for processor affinities, heterogeneous parallelism, cluster parallelism**

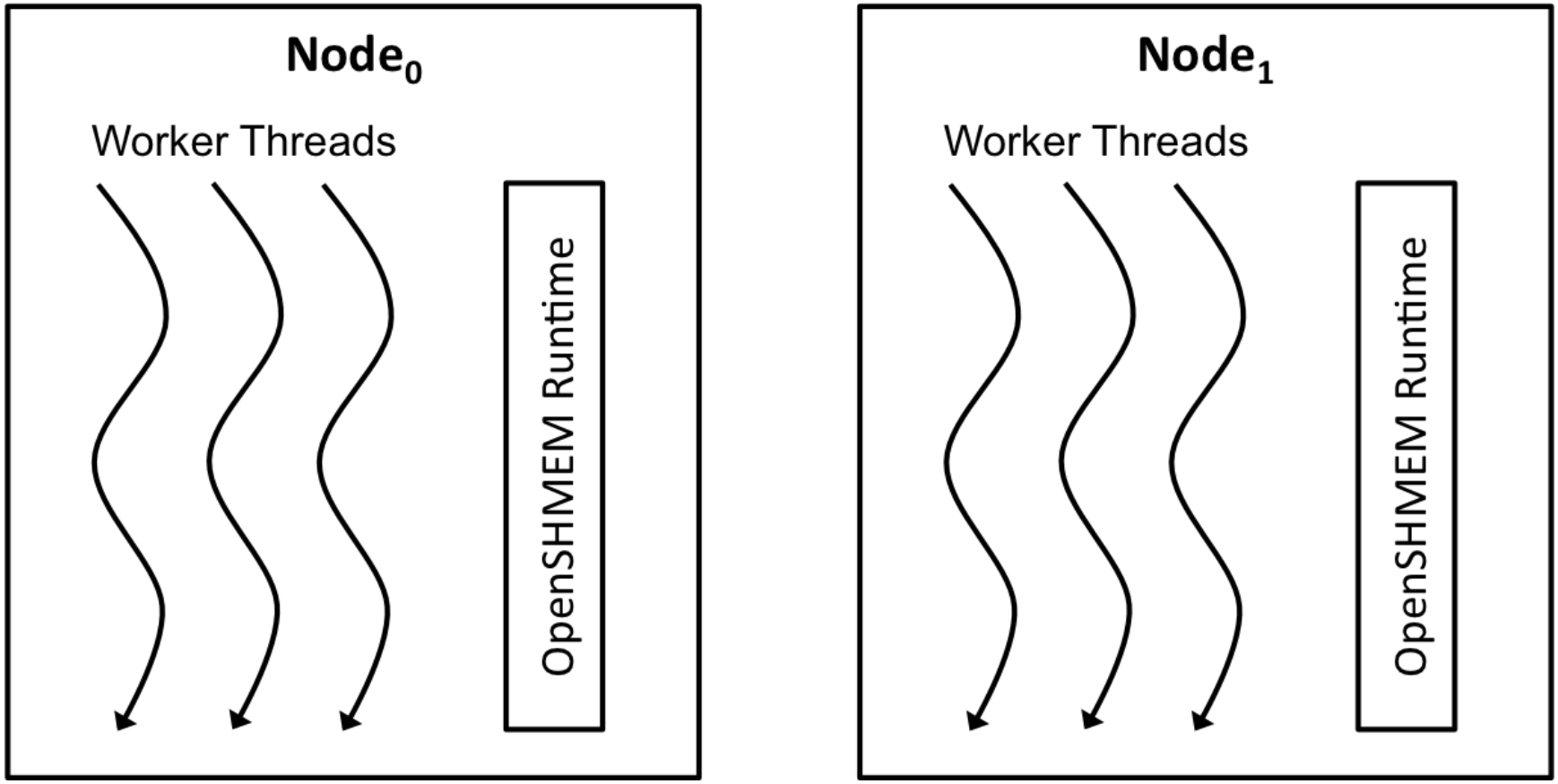
AsyncSHMEM: HClib + OpenSHMEM



2

"Integrating Asynchronous Task Parallelism with OpenSHMEM." Max Grossman, Vivek Kumar, Zoran Budimlic, Vivek Sarkar. OpenSHMEM Workshop, August 2016.

AsyncSHMEM Under the Covers



AsyncSHMEM: Fork-Join APIs

- `shmem_task_scope_begin`: Start a new synchronization scope for asynchronously launched tasks (`#pragma omp taskgroup`)
- `shmem_task_scope_end`: End the current synchronization scope for asynchronously launched tasks (end scope of `#pragma omp taskgroup`)
- `shmem_async_task`: Launch a single asynchronous task (`#pragma omp task`), tasks are single-threaded, may block without blocking the underlying worker, may spawn new tasks.
- `shmem_parfor_task`: Launch a parallel loop (`#pragma omp parallel for`, `taskloop`)

AsyncSHMEM: Fork-Join APIs Example

```
shmem_task_scope_begin();
{
    shmem_async_task(foo, NULL);

    shmem_async_task(bar, NULL);
}
// Wait for foo and bar tasks to complete
shmem_task_scope_end();
```

AsyncSHMEM: Futures APIs

- `shmem_malloc_promise`: Create promise object(s)
- `shmem_async_future`: Launch a single asynchronous task whose execution is predicated on a specified future (`#pragma omp task depend`, with caveats)
- `shmem_parfor_future`: Launch a parallel for loop predicated on a specified future
- `shmem_satisfy_promise`: Satisfy a specific promise object
- `shmem_future_wait`: Block current thread on execution of provided future

AsyncSHMEM: Futures APIs Example

```
static void foo(void *arg) {
    shmem_satisfy_promise((shmem_promise *)arg);
}
shmem_promise *prom = shmem_malloc_promise();

shmem_async_future(bar, NULL, prom);

shmem_async_task(foo, prom);
```

AsyncSHMEM: Communication Driven Tasks

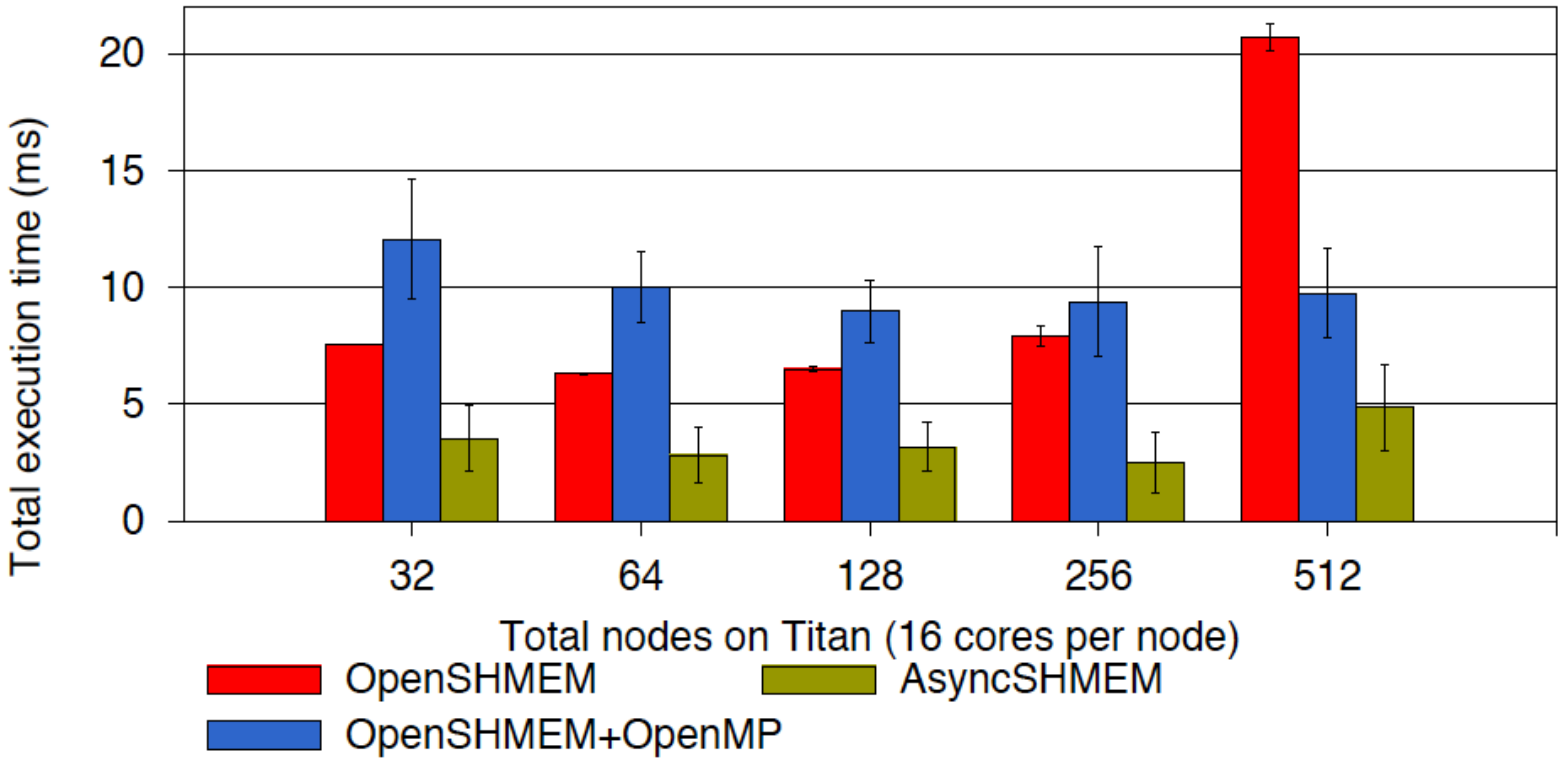
- `shmem_async_when`: Make the execution of a task predicated on the satisfaction of a condition.
- `shmem_wait_until_any` [1]: Wait until the satisfaction of any of the specified conditions, similar to `shmem_wait_until`.
- `shmem_async_when_any`: Same as above, but with multiple conditions.
- [1] http://www.openshmem.org/redmine/issues/215

AsyncSHMEM: Communication Driven Tasks Example

```
int pe = shmem_my_pe();
if (pe < shmem_n_pes() - 1) {
    shmem_int_async_when(&shared_var, SHMEM_CMP_EQ,
                                pe + 1, async_body);
}

if (pe > 0) {
    shmem_int_put(&shared_var, &pe, 1, pe - 1);
}

shmem_barrier_all();
```

AsyncSHMEM: Experimental Results

Unbalanced Tree Search (UTS)



We are heading towards Extreme Heterogeneity in HPC at all levels …

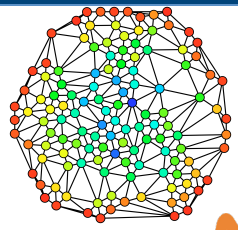
Applications

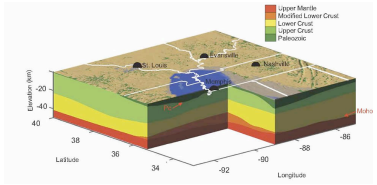
Runtimes

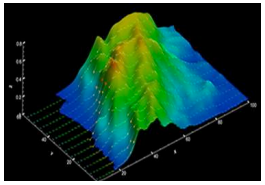
Processor

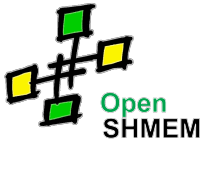
Memory

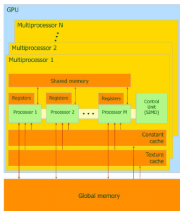
Network

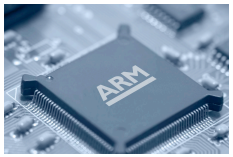


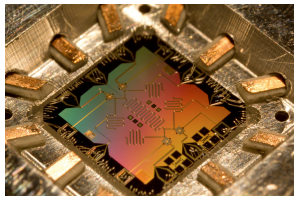


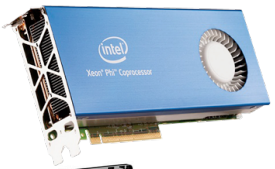


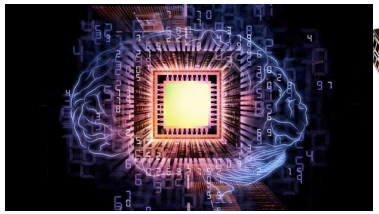


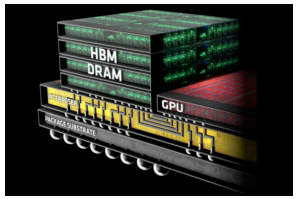






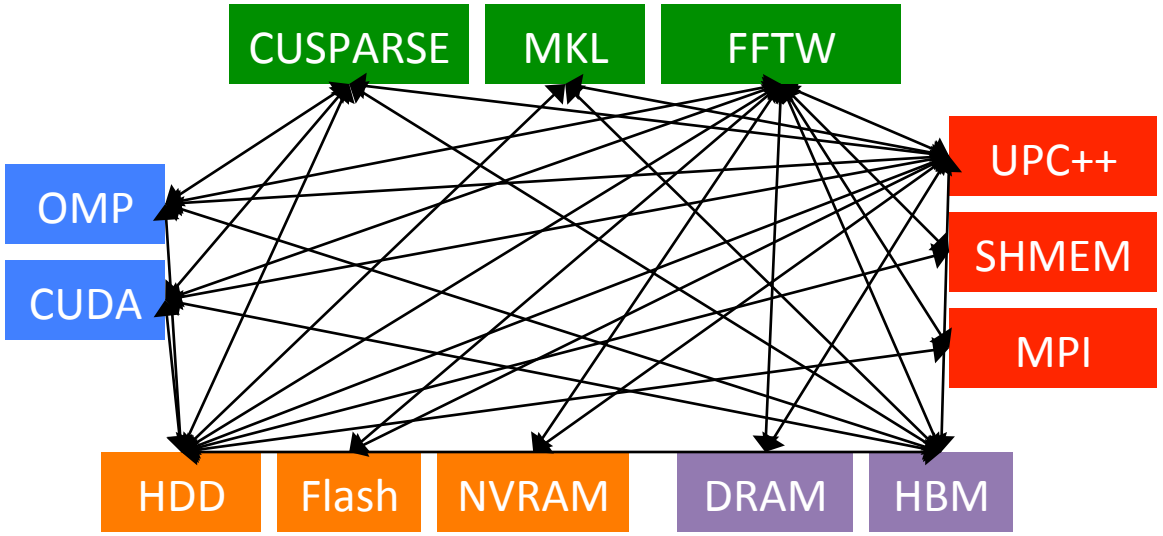






Heterogeneity Requires Composability

HiPER (Highly Pluggable, Extensible, and Reconfigurable Framework for HPC) is a runtime and programming model based on HClib for composing heterogeneous scheduling libraries.



1. "A Pluggable Framework for Composable HPC Scheduling Libraries. Max Grossman, Vivek Kumar, Nick Vrvilo, Zoran Budimlic, Vivek Sarkar. The Seventh International Workshop on Accelerators and Hybrid Exascale Systems (AsHES). May 2017.
2. "Integrating Asynchronous Task Parallelism with OpenSHMEM." Max Grossman, Vivek Kumar, Zoran Budimlic, Vivek Sarkar. OpenSHMEM Workshop, August 2016.

Related Work

HPC Runtimes supporting heterogeneous machines:

- *High-level, auto-scheduling*: Legion, Charm++
- *Mid-level, more tunable by the programmer*: Chapel, X10
- *Low-level*: HPX, OmpSs, StarPU

Library composition:

- NVSHMEM: OpenSHMEM for GPUs
- GPU-Aware MPI: Direct GPU-to-GPU communication between nodes
- AsyncSHMEM, HCMPI, Habanero-UPC++: Asynchronous tasking + communication
- Lithe: Define an API for libraries to share cores

HiPER

An extension of the HClib parallel programming system and its hierarchical place trees for heterogeneity.

Explored graph platform representations.

Developed the concept of generalized work-stealing.

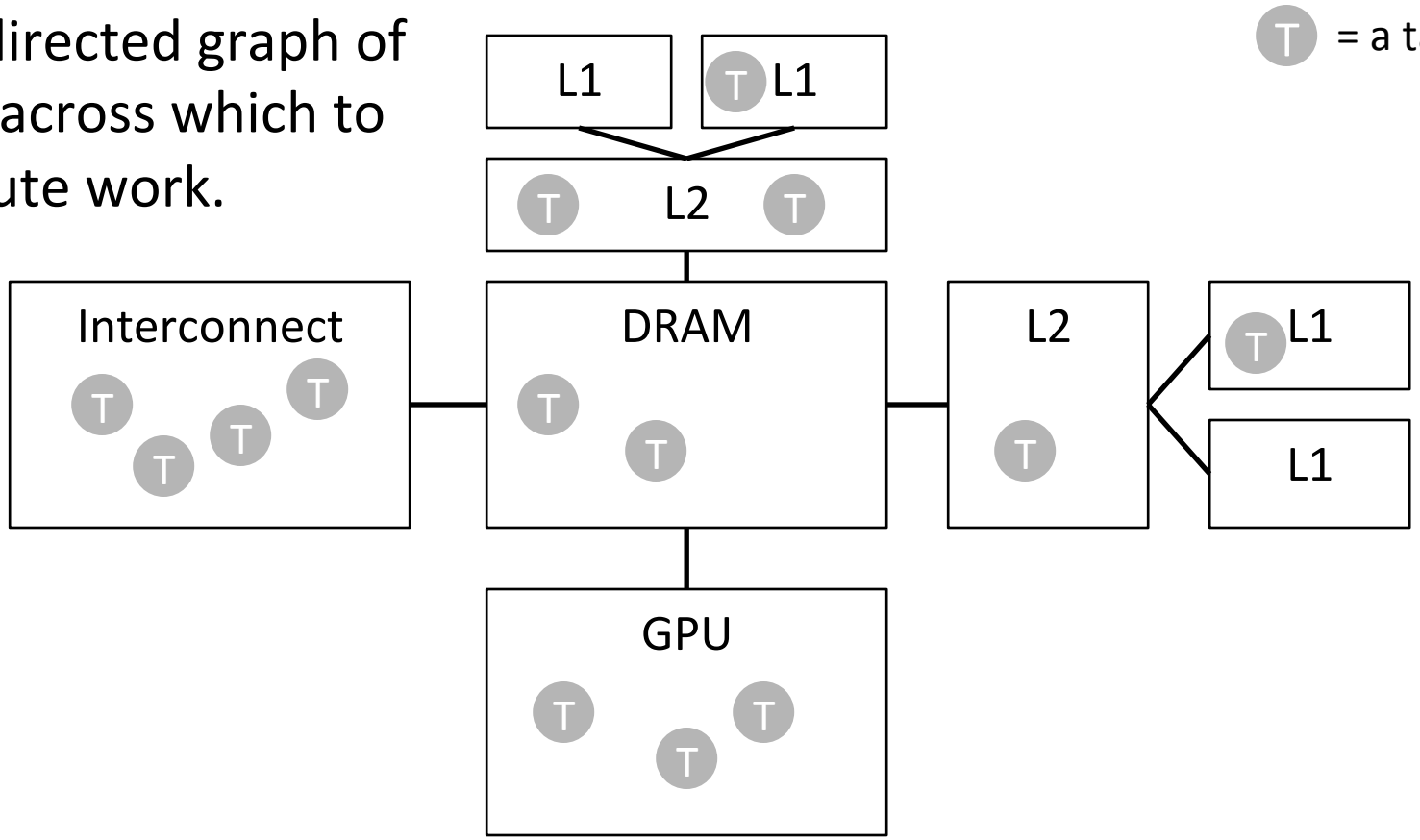
Developed the concept of pluggable library modules for work-stealing runtimes.

Explored API extensions, tooling, scheduling problems enabled by this work in the context of MPI, OpenSHMEM, CUDA, UPC++.

Open source and robust implementation that enables rapid integration and composition of HPC libraries.

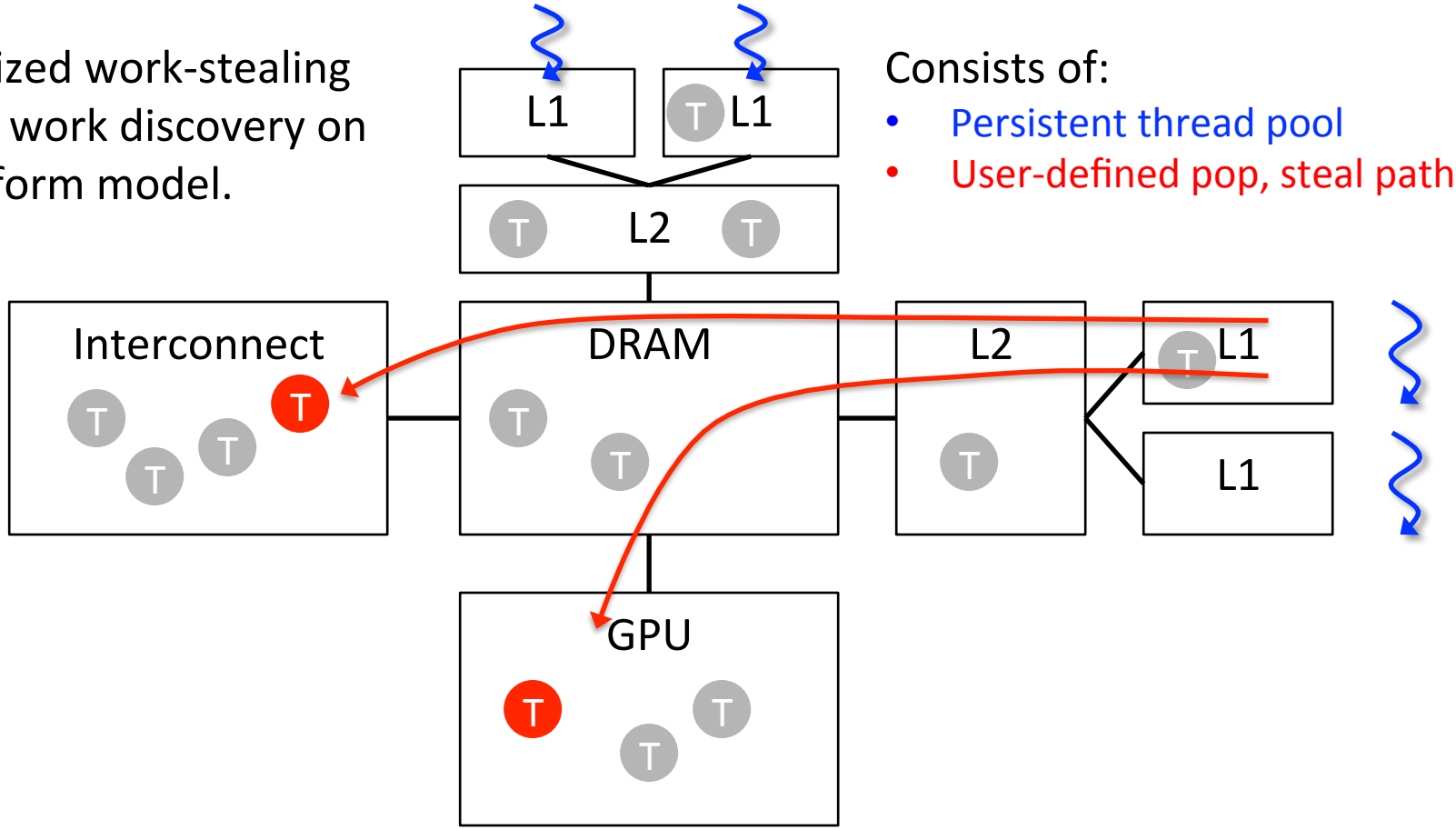
HiPER Platform Model

An undirected graph of *places* across which to distribute work.



HiPER Generalized Work-Stealing

Generalized work-stealing controls work discovery on the platform model.



Consists of:

- Persistent thread pool
- User-defined pop, steal paths

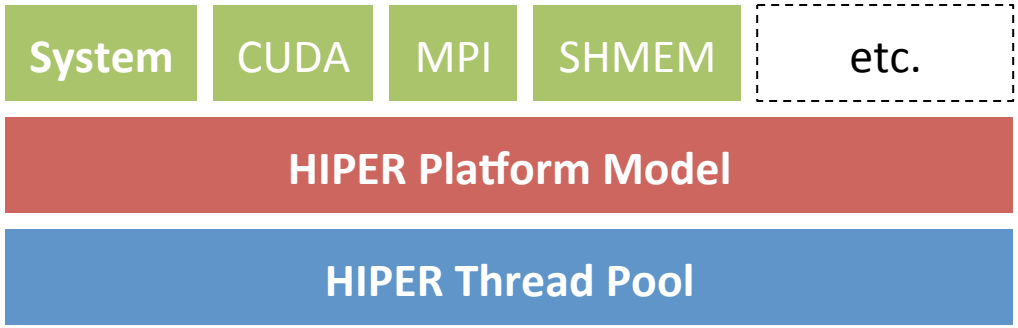
HiPER Pluggable Modules

Platform model & generalized work-stealing handle **work discovery and execution**.

Modules add user APIs that spawn tasks on the platform model, (**work creation)**.

Built-in **system module** adds basic task-parallel, future-based, etc. APIs.

Developers add custom modules for third-party software packages (CUDA, MPI, UPC++, and OpenSHMEM).



A Simple Example Module

```
void shmem_int_put(int *dst, const int *src, size_t nelems, int pe);
```

```
void hiper::shmem_int_put(int *dst, const int *src, size_t nelems,
        int pe) {
    finish([dst, src, nelems, pe] {
        async_at(nic, [dst, src, nelems, pe] {
            ::shmem_int_put(dst, src, nelems, pe);
        });
    });
}
```

Only executed by worker threads with the NIC place on their place path.

Suspension point for the calling task

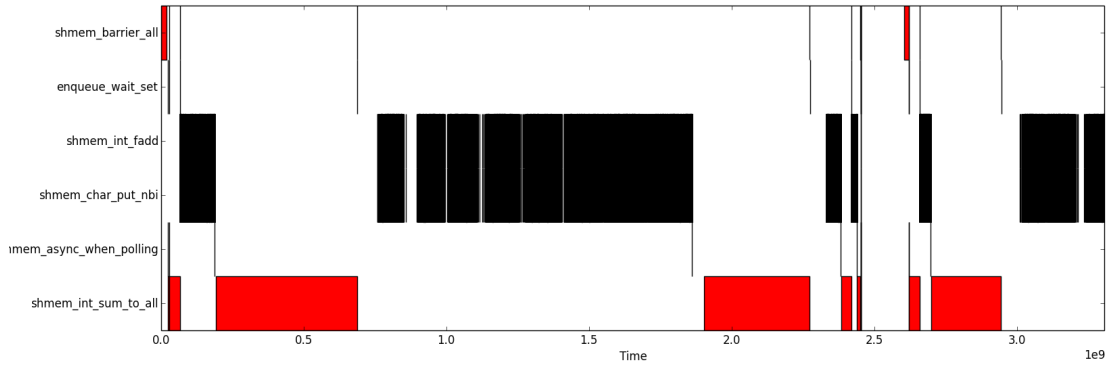
Enables New Functionality

APIs:

```
void shmem_int_async_when(volatile int *ivar, int cmp_value, auto lambda);
```

Novel API, executes asynchronous task `lambda` once `*ivar == cmp_value`.

Tools:



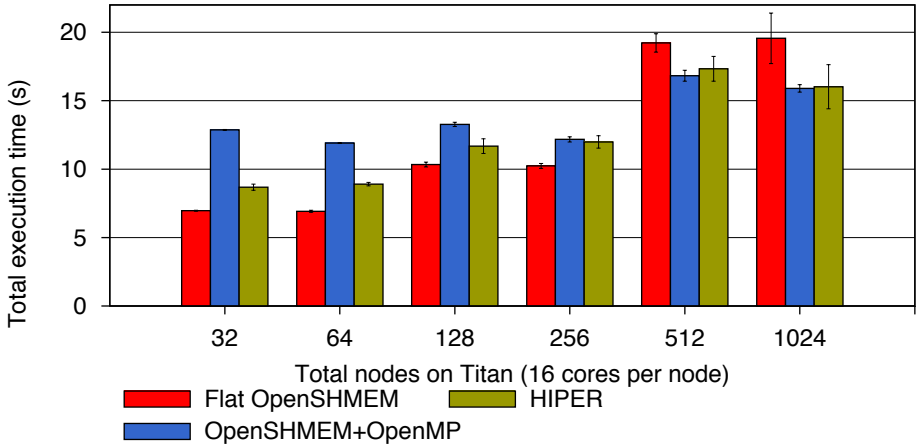
Unified runtime scheduling

HiPER Evaluation

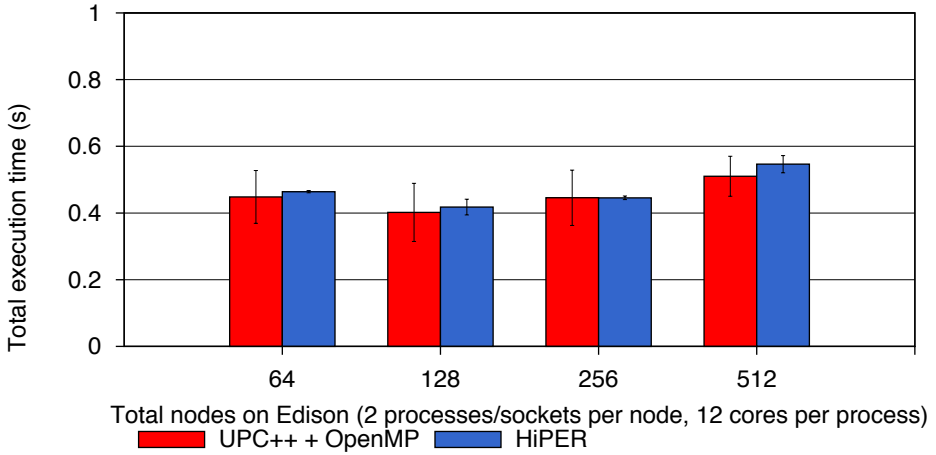
Application	Platform	Dataset	Scaling
ISx	Titan	2^{29} keys per node	Weak
HPGMG	Edison	log2_box_dim=7 boxes_per_rank=8	Weak
UTS	Titan	T1XXL	Strong
Graph500	Titan	2^{29} nodes	Strong

HiPER Evaluation – Regular Applications

HIPER is low-overhead, no impact on performance for regular applications



ISx



HPGMG Solve Step

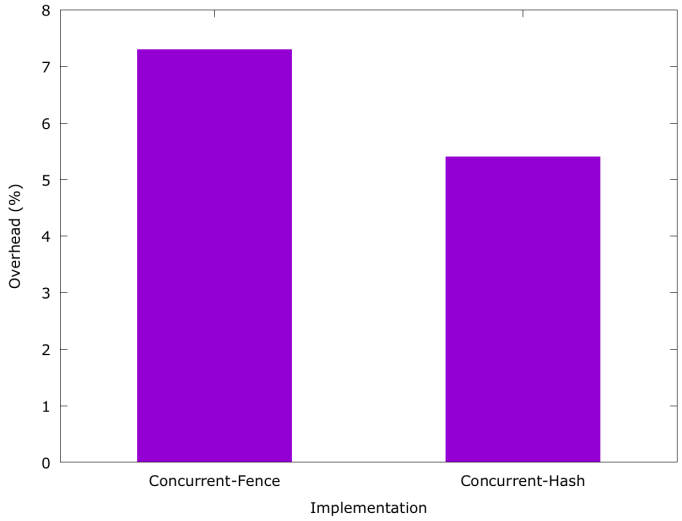
1. "Integrating Asynchronous Task Parallelism with OpenSHMEM." Max Grossman, Vivek Kumar, Zoran Budimlic, Vivek Sarkar. OpenSHMEM Workshop, August 2016.
2. "A Pluggable Framework for Composable HPC Scheduling Libraries". Max Grossman, Vivek Kumar, Nick Vrvilo, Zoran Budimlic, Vivek Sarkar. Submitted for publication to IPDPS 2017.

HiPER Evaluation – Graph500

HiPER used for concurrent (not parallel) programming in Graph500.

Rather than periodic polling, use novel `shmem_async_when` APIs to trigger local computation on incoming RDMA.

Reduces code complexity, hands scheduling problem to the runtime.

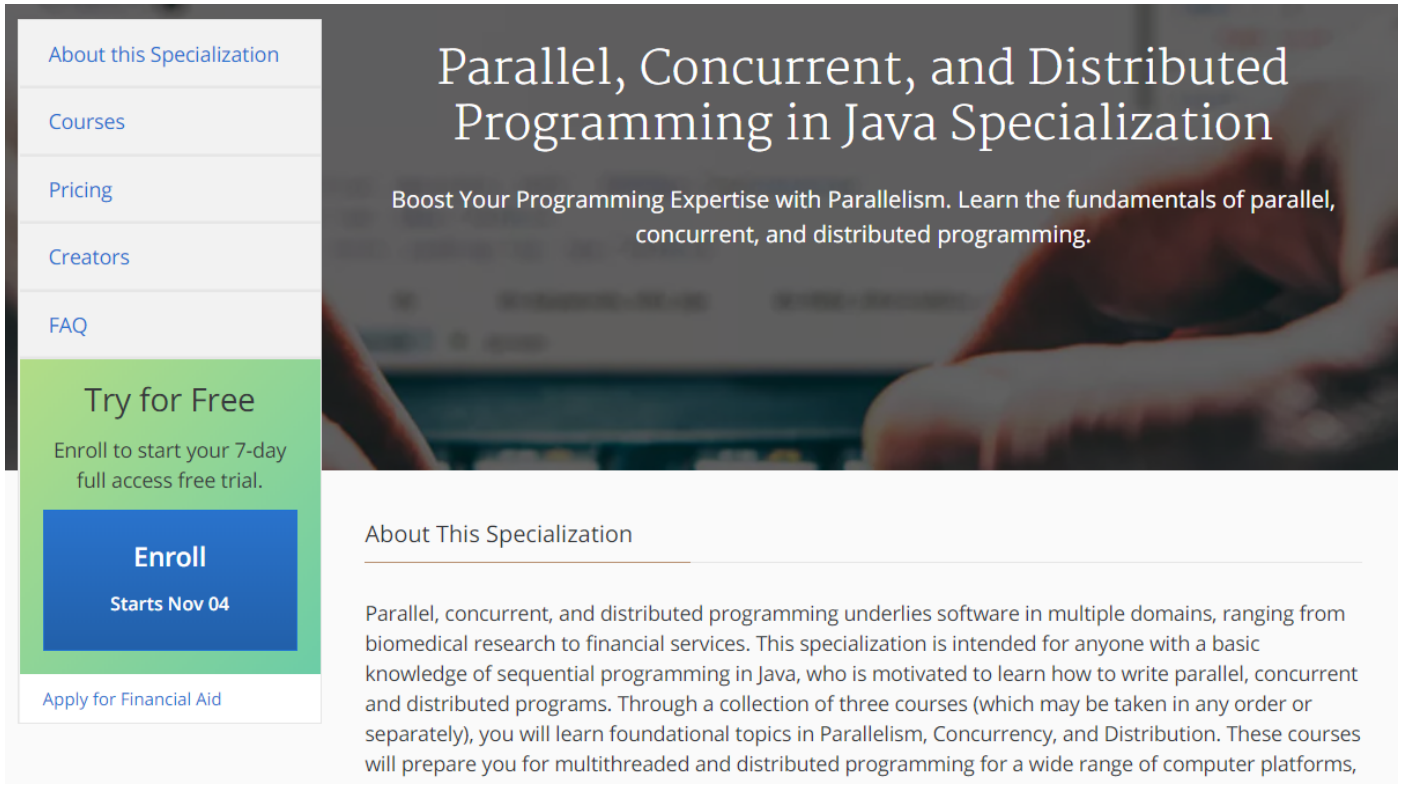


"Graph500 on OpenSHMEM: Using a Practical Survey of Past Work to Motivate Novel Algorithmic Developments". Max Grossman. Los Alamos National Laboratory Technical Report. December 2016.

Tutorial Summary

- Parallel Programming
  - What, Why, How?
- Parallel Programming using Habanero C/C++ library (HClib)
  - Task parallelism (async, finish)
  - Loop parallelism (forasync)
  - Functional parallelism (futures, promises)
  - Dataflow parallelism (async-await)
- Advanced topics: programming for processor affinities, heterogeneous parallelism, cluster parallelism
- **Appendix: HClib installation and usage information**

Additional Teaching Resources in Coursera Specialization on "Parallel, Concurrent, and Distributed Programming in Java" (C++ version under development)



https://www.coursera.org/specializations/pcdp

Parallel, Concurrent, and Distributed Programming in Java

Split into three courses on parallelism, concurrency, and distribution.

Each course divided into four weeks. Each week includes:

- 5 lecture videos
- A lecture summary following each video
- A graded multiple choice quiz
- A demonstration video, showing some application of concepts from the week
- A graded mini-project exercising some concepts from the week, usually graded by correctness and performance on the Coursera auto-grading cloud.

Instructor-learner interaction occurs primarily through per-course forums.

coursera

Week-by-week Syllabus for Coursera Specialization

- Parallel Programming course
  1. Task Parallelism
  2. Functional Parallelism
  3. Loop Parallelism
  4. Data Flow Synchronization and Pipelining
- Concurrent Programming course
  1. Threads and Locks
  2. Critical Sections and Isolation
  3. Actors
  4. Concurrent Data Structures
- Distributed Programming course
  1. Distributed Map Reduce
  2. Client-Server Programming
  3. Message Passing
  4. Combining Distribution and Multithreading

Downloading HClib

- git clone https://github.com/habanero-rice/hclib.git
- Dependencies
  - automake
  - gcc >= 4.8.4, or clang >= 3.5 (must support -std=c11 and -std=c++11)
  - Modules
    - OpenSHMEM, MPI, UPC++, CUDA

Using HClib

- Installation
  - Please read the following file: https://github.com/habanero-rice/hclib/blob/master/README.md
- Sample examples along with instructions on building and executing
  - Available in repository https://github.com/habanero-rice/hclib/blob/master/test/tutorial/hipc18/