并行程序设计原理

北京大学信息学院

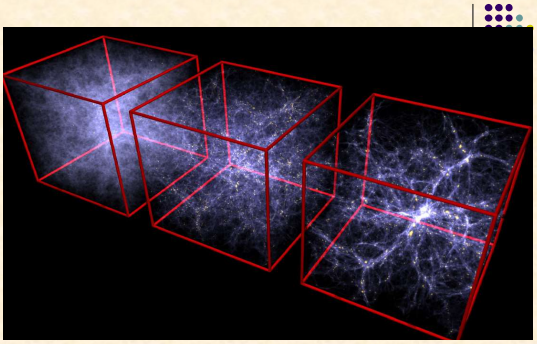
内容

- 并行体系结构
- 低层并行编程介绍（CUDA、MPI、Pthread、SCIF、GlobalArrays、OpenMP Offload）
- 典型并行算法
- 高级并行编程（OpenMP、Parray、MapReduce）
- 并行理论（Petri网、线性逻辑、进程代数）

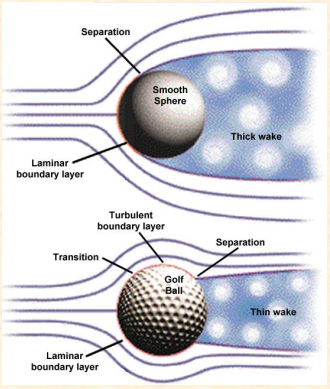
Getting Started

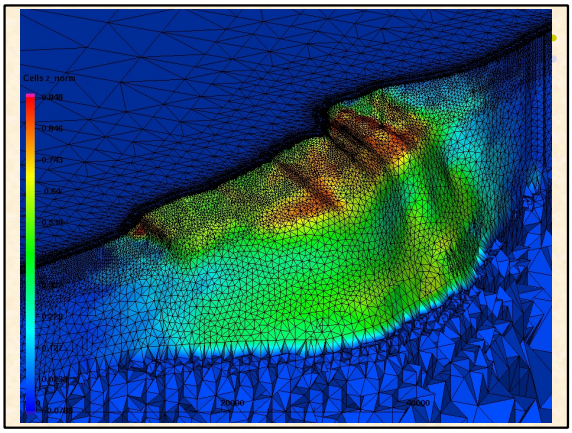
- http://www.nvidia.com/object/cuda_get.html

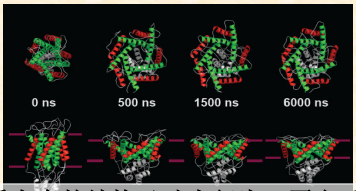
并行计算的应用



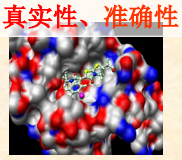
cosmological numerical simulations Springel et al. (2001)

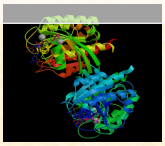






蛋白质自身的结构及动态行为、蛋白质与其他分子之间的相互识别及化学反应机理——真实性、准确性





How does Google work?

1. The web server sends the query to the index servers. The content inside the index servers is similar to the index in the back of a book--it tells which pages contain the words that match any particular query term.

2. The query travels to the doc servers, which actually retrieve the stored documents. Snippets are generated to describe each search result.

3. The search results are returned to the user in a fraction of a second.

- http://www.g works.html

Shading

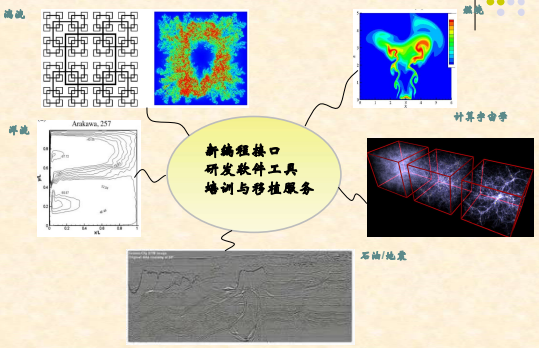
Modern Computer Games

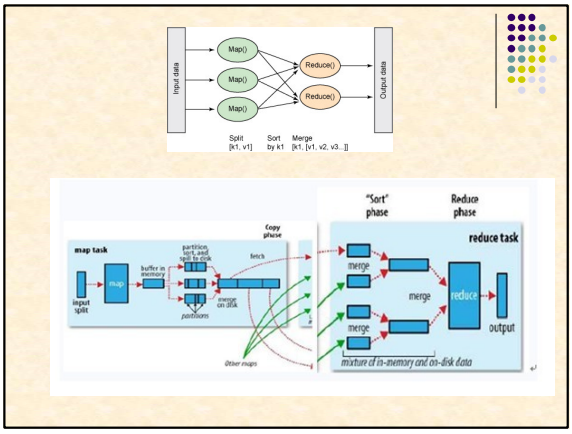
The obvious:

- Rendering
- Pixel shading
- Physics simulation, collision detection
- Game world simulation
- Artificial intelligence, path finding

http://cag.csail.mit.edu/crg/papers/sweeney06games.pdf

对北大各学科的辐射和服务作用

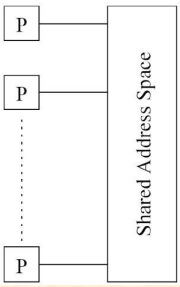




并行体系结构

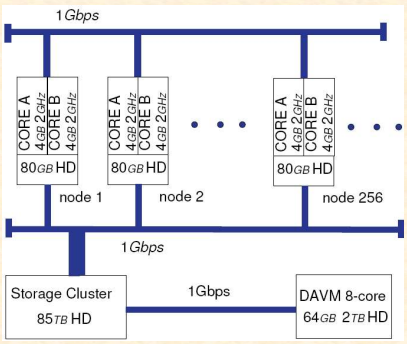
Shared Memory Architectures

Logical machine model

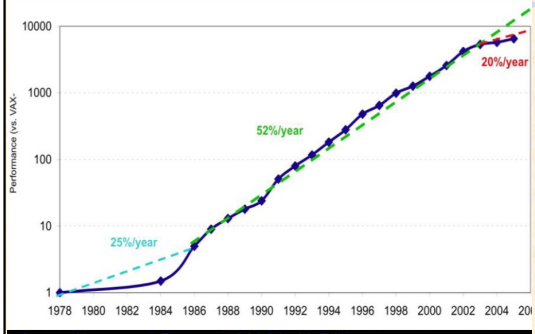


- **Hardware model**
  - —processors
  - —shared global memory
    - – equidistant from processors
- **Software model**
  - —threads
  - —shared variables
  - —communication
    - – read shared data
    - – write shared data
  - —synchronization

Clusters at ICC, Durham



Single-Thread Processor Performance vs Calendar Year



Source: Hennessy & Patterson, CAAQA, 4th Edition

Historical Concurrency in Top 500 Systems

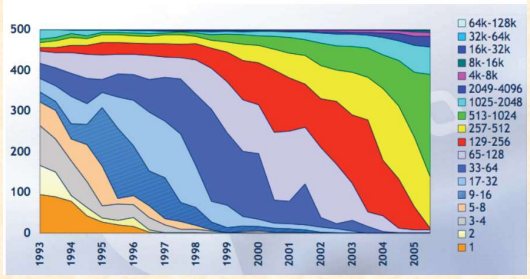
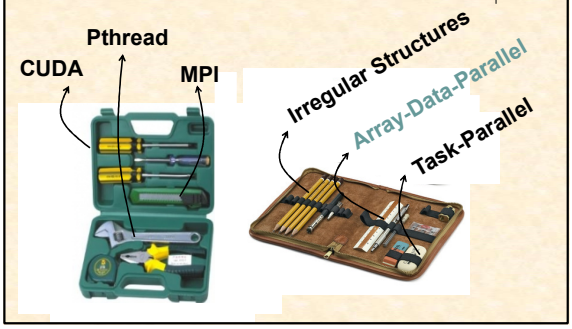


Figure Credit: www.top500.org, Nov 15, 2005

Modern-Day Supercomputers

- Tianhe 1A: 1 GPU/ 2CPUs
- Tianhe 2: 3 MICs/ 2CPUs
- Tsubame：3GPU/ 2CPUs
- Mole-8.5: 6GPUs/2CPUs
- PKU McClus: 2GPUs/1CPU

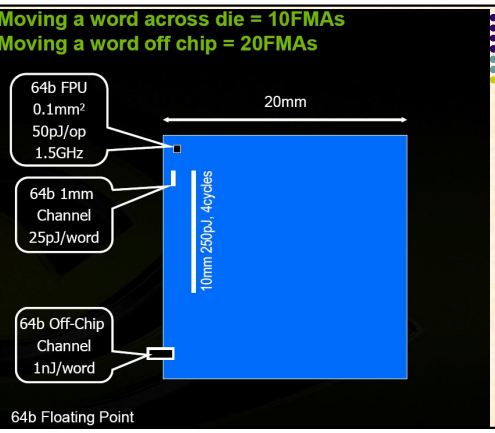
Parallel Programming: Toolbox vs Writing Case



Bill Dally (CS, Stanford & NVIDIA CTO)

- Single-threaded processor: performance is no longer scale.
  Performance = Parallelism
- Chips are power limited and most power is spent moving data
  Efficiency = Locality
- Amdahl's law not applicable to most future applications.

Moving a word across die = 10FMAs
Moving a word off chip = 20FMAs



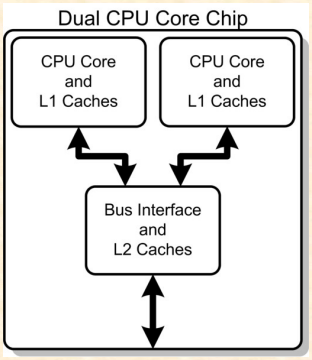
Exploiting parallelism and locality requires:

Many efficient processors
(To exploit parallelism)

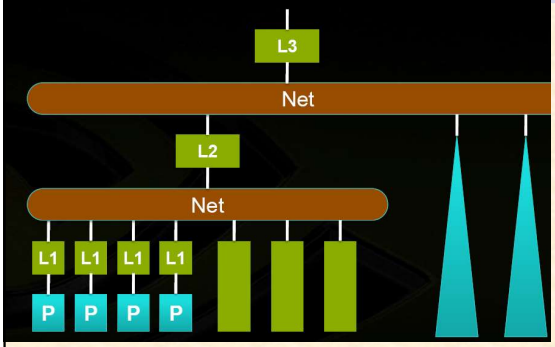
An exposed storage hierarchy
(To exploit locality)

A programming system that abstracts this

Multi-core

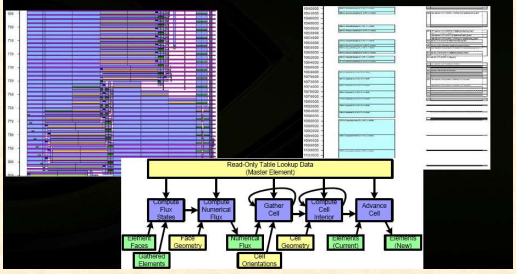


Tree-structured machines



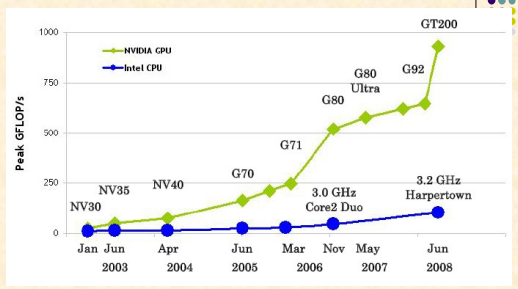
Optimize use of scarce bandwidth

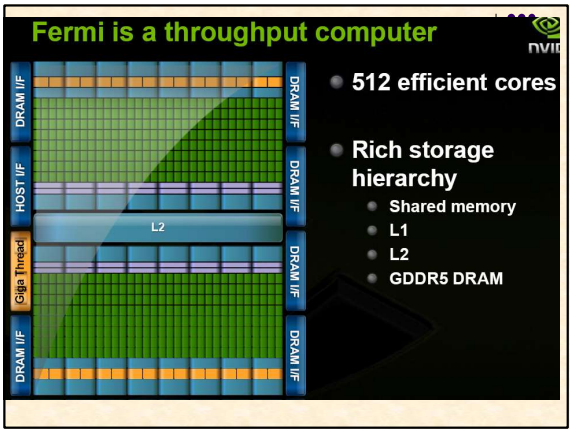
- Provide rich, exposed storage hierarchy
- Explicitly manage data movement on this hierarchy
  - Reduces demand, increases utilization

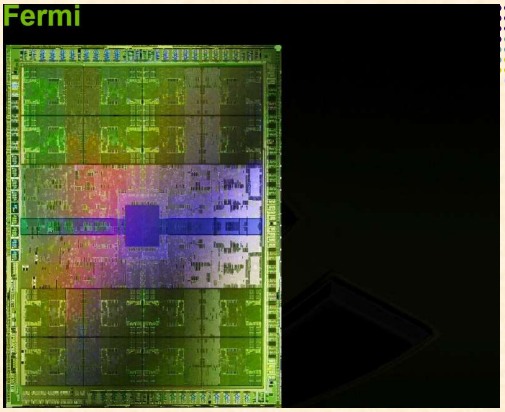


Avoid Denial Architecture

- Single thread processors are in denial about parallelism and locality
- They provide two illusions:
  - Serial execution - Denies parallelism
    - Tries to exploit parallelism with ILP - inefficient & limited scalability
  - Flat memory - Denies locality
    - Tries to provide illusion with caches – very inefficient when working set doesn't fit in the cache
- These illusions inhibit performance and efficiency





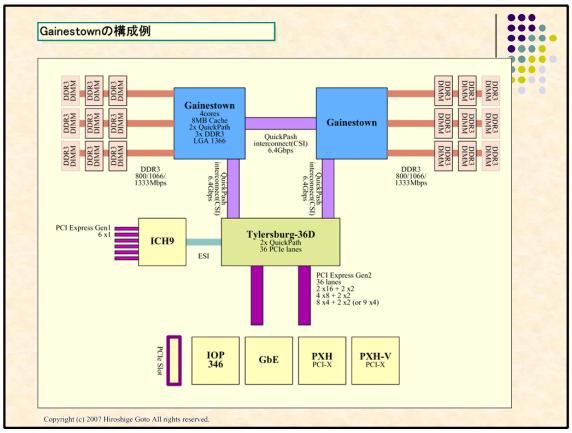


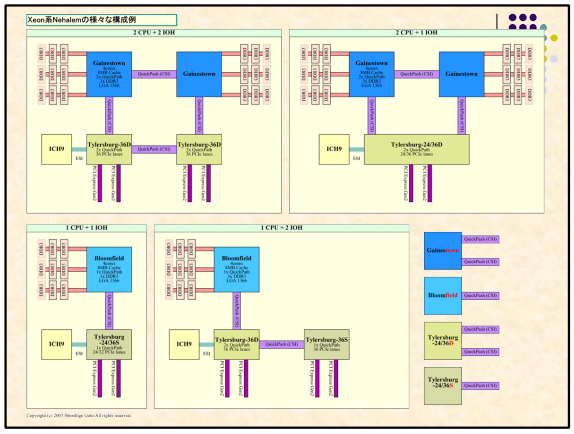
Memory Hierarchy

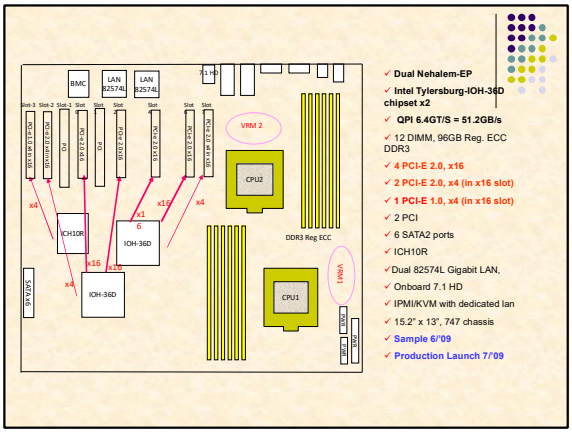
- Host memory (pageable or pinned)
- Memory slots adjacent to specific a PCI slot
- Memory segments up to max continuous length
- Device memory specific to a GPU
- L2 cache of Fermi
- Constant memory
- Shared memory (L1 cache of Fermi)
- Registers

Different NVidia GPU models

- GTX260
- GTX 280: 240 cores, 65nm, 602 MHz, 512b, 141.7GB/s, 1GB, 236w.
- GTX 285: 240 cores, 55nm, 649 MHz, 512b, 159GB/s, 1GB, 183w.
- GTX 295: 480 cores, 55nm, 576 MHz, 448b, 1.7GB, 224GB/s, 289w.
- C1060: 240 cores, 55nm, 602 MHz, 512b, 102GB/s, 4GB, 160w.
- S1070: 960 cores, 55nm, 512b, 408GB/s, 16GB, 800w.
- C2050: 448 cores, 40nm, 512b, ?GB/s, 3~6GB
- Kepler 1536cores



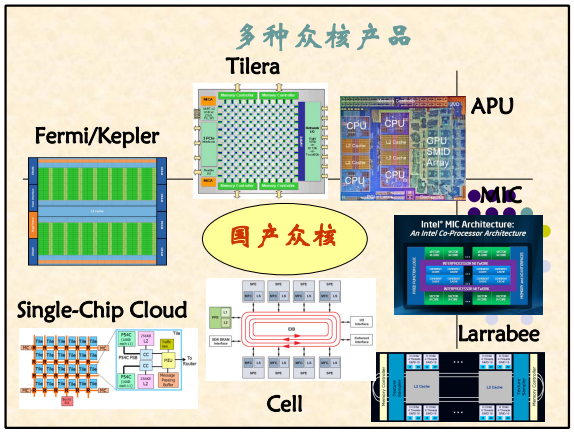


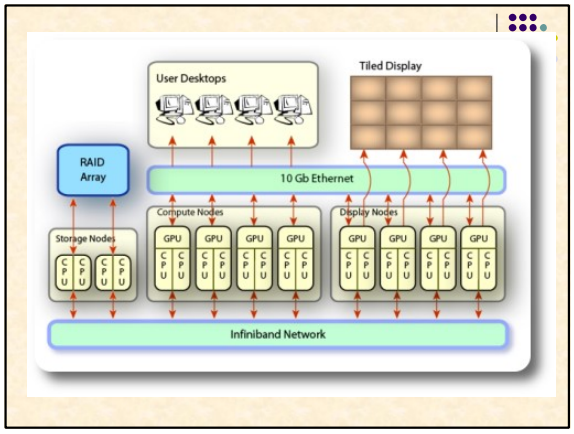


PCI Bandwidth for GPUs

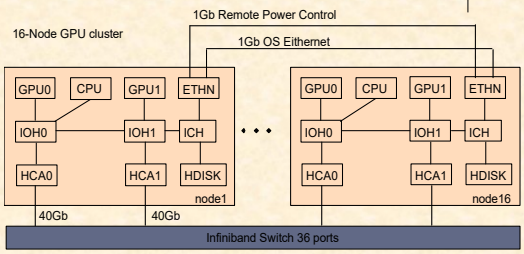
- CPU
  - Bandwidth 20~30GB/s
  - Cache hit rate not good for large data procssing (experiment: memcpy 3GB/s for one thread and 7GB/s for multi-threads)
  - Random access better than GPU
- GPU
  - Bandwidth 100~160GB/s
  - Coalesced access (73GB/s ~122GB/s)
- CPU-PCI-GPU: singlex 2.5~6GB/s (pinned, experiment), IOH dependency
- Infiniband: duplex upto 4GB/s (experiment using IB on PKU cluster), IOH dependency

众核集群技术

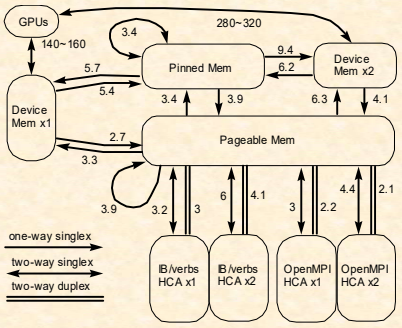


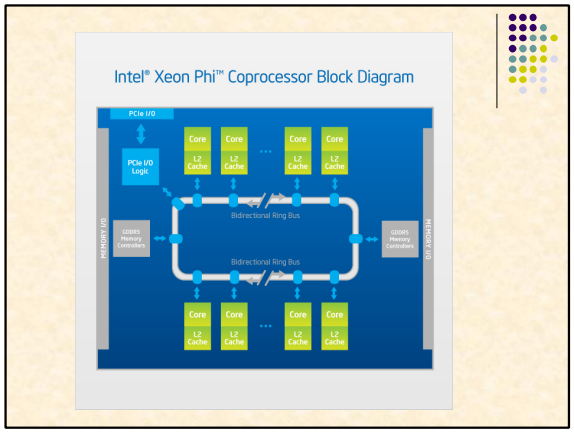


PKU McClus for Experiment

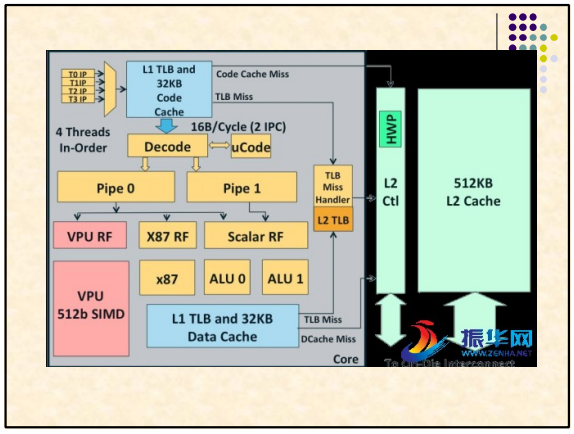


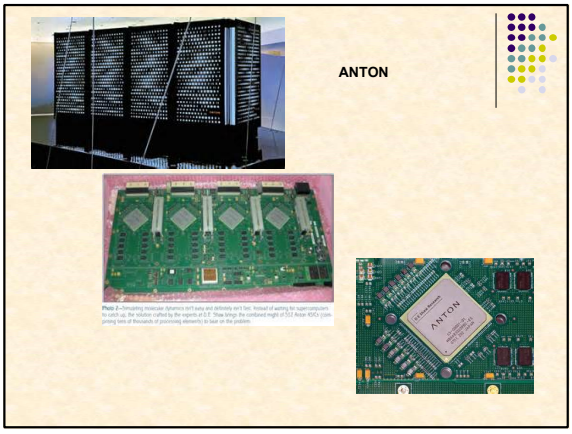
程序员管理不同速度的存储器

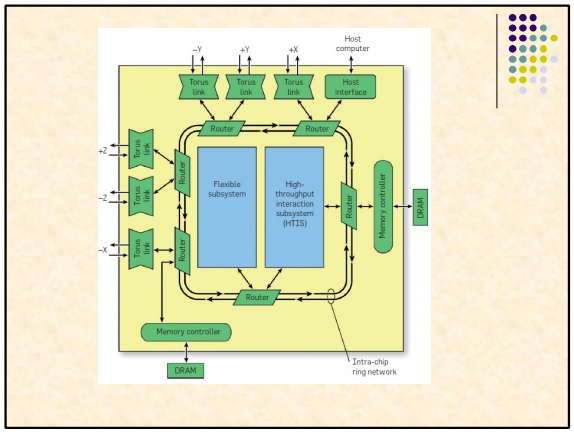



Intel® Xeon Phi™ Coprocessor Block Diagram
PCIe I/O
PCIe I/O Logic
Core
L2 Cache
Bidirectional Ring Bus
GDDR5 Memory Controllers
MEMORY I/O


天河二号
TH-2 High Performance Computer System


T0 IP
T1 IP
T2 IP
T3 IP
L1 TLB and 32KB Code Cache
Code Cache Miss
TLB Miss
4 Threads In-Order
16B/Cycle (2 IPC)
Decode
uCode
Pipe 0
Pipe 1
TLB Miss Handler
L2 TLB
VPU RF
X87 RF
Scalar RF
VPU 512b SIMD
x87
ALU 0
ALU 1
L1 TLB and 32KB Data Cache
TLB Miss
DCache Miss
Core
HWP
L2 Ctl
512KB L2 Cache

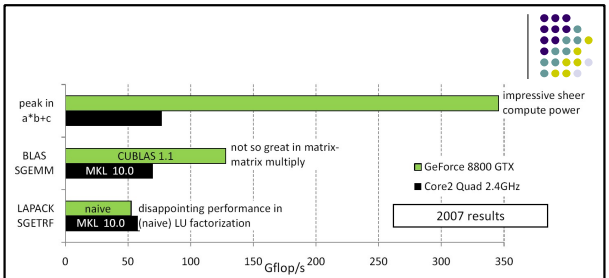

ANTON


Host computer
Torus link
Host interface
Router
Flexible subsystem
High-throughput interaction subsystem (HTIS)
Memory controller
DRAM
Intra-chip ring network

SGEMM

- Vasily Volkov, James Demmel: *Benchmarking GPUs to tune dense linear algebra*. SC 2008

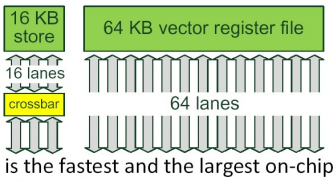
Some following slides are from the presentation of the above paper at SC'08.



- Memory bandwidth scales with number of memory controllers

GPU (NVIDIA GeForce)	8600 GTS	9800 GTX	GTX 280
Processor cores	4	16	30
Compute capability (a+b*c)	93 Gflop/s	429 Gflop/s	624 Gflop/s
Memory controllers	2	4	8
Memory bandwidth	32 GB/s	70 GB/s	141 GB/s

GPU Memory Hierarchy



- Register file is the fastest and the largest on-chip memory
  - **Keep as much data as possible in registers**
  - However, register file is constrained to vector operations
  - Can live with it — vectorized codes are common in HPC
- Shared memory permits indexed and shared access
  - However, it is 2–4x smaller and has 4x lower bandwidth than registers
    - Only 1 operand in shared memory is allowed versus 4 register operands
  - Moreover, some instructions run slower if using shared memory
  - **Use shared memory as a communication device only**
  - Avoid communication to improve performance

Peak Throughput in Multiply-and-Add

- How much parallelism is *enough* to get the peak?
- Run **1 thread per processor core**
  - Purpose: smallest amount that can control all computing resources
- Assume **sufficient instruction-level parallelism** in the program
  - Purpose: hide pipeline latency
- Choose the shortest vector length that yields the peak
  - Purpose: satisfy inherent data-parallelism constraints
- Result: **98% of arithmetic peak at VL = 64**
  - Therefore, VL=64 is recommended for all compute-bound codes
- However, we never could surpass 66% of peak is using an operand in shared memory
  - We believe this is an inherent bottleneck in the architecture
  - We use this number in the throughput bounds below

Matrix-Matrix Multiply: $C = C + A*B$

- GPU requires using block algorithms in matrix-matrix multiply:
  - Peak rates on one of the latest GPUs are 624 Gflop/s and 141 GB/s
  - This corresponds to 0.23 bytes per flop
  - But naïve matrix-matrix multiply requires 4 bytes per flop
  - Thus, it is bandwidth-bound unless data is reused 18 times
  - Using MxN blocks in C yields $2/(1/M+1/N)$ average reuse
- Use vector algorithms to efficiently use vector registers
  - Such as used on IBM 3090 Vector Facility and Cray X1:
  - Keep A's and C's blocks in registers
  - Keep B's block in a shared storage
  - No other sharing is needed if C's height = VL. We know VL=64 is best
- Choose large enough width of C's block
  - 16 is enough as 2/(1/64+1/16) = 26-way reuse
- Choose a convenient thickness for A's and B's blocks

```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;
    __shared__ float bs[16][17];
    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    const float *Blast = B + k;
    do
    {
#pragma unroll
        for( int i = 0; i < 16; i += 4 )
            bs[threadIdx.x][threadIdx.y+i]  = B[i*ldb];
        B += 16;
        __syncthreads();
#pragma unroll
        for( int i = 0; i < 16; i++, A += lda )
        {
            c[0] += A[0]*bs[i][0];   c[1] += A[0]*bs[i][1];   c[2] += A[0]*bs[i][2];   c[3] += A[0]*bs[i][3];
            c[4] += A[0]*bs[i][4];   c[5] += A[0]*bs[i][5];   c[6] += A[0]*bs[i][6];   c[7] += A[0]*bs[i][7];
            c[8] += A[0]*bs[i][8];   c[9] += A[0]*bs[i][9];   c[10] += A[0]*bs[i][10];c[11] += A[0]*bs[i][11];
            c[12] += A[0]*bs[i][12];c[13] += A[0]*bs[i][13];c[14] += A[0]*bs[i][14];c[15] += A[0]*bs[i][15];
        }
        __syncthreads();
    } while( B < Blast );
    for( int i = 0; i < 16; i++, C += ldc )
        C[0] = alpha*c[i] + beta*C[0];
}
```

Compute pointers to the data

Declare the on-chip storage

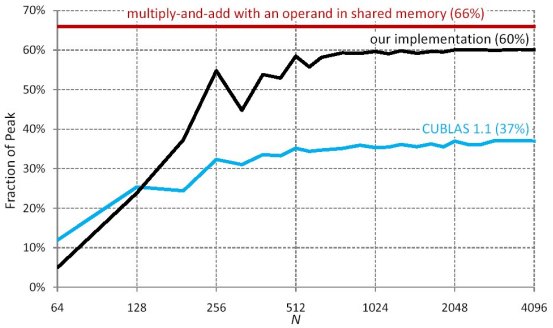
Read next B's block

The bottleneck:
Read A's columns
Do Rank-1 updates

Store C's block to memory

Our code vs. CUBLAS 1.1

Performance in multiplying two *NxN* matrices on GeForce 8800 GTX:

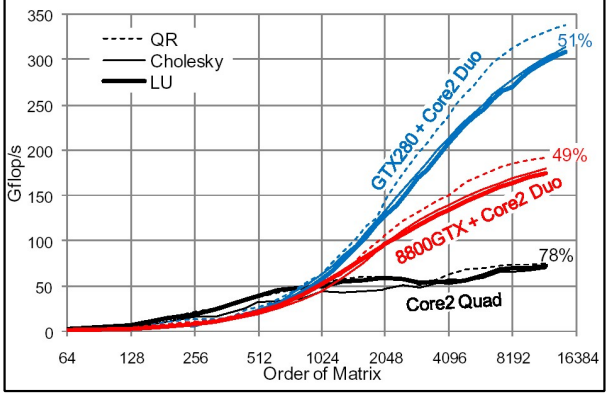


What causes CUBLAS 1.1 to run slower than our code?

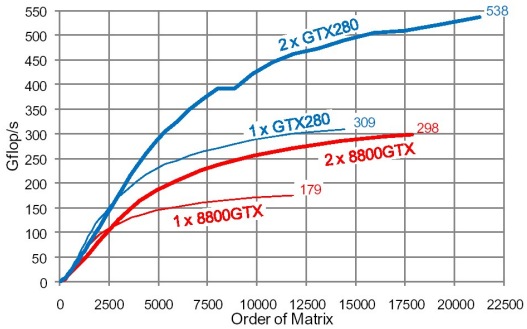
Fast Matrix Factorizations using GPUs

- Use GPU to compute matrix-matrix multiplies only
- Factorize panels on the CPU
- Use look-ahead to overlap computations on CPU and GPU
- Use right-looking algorithms to have more threads in SGEMM
  - Better load balance in the GPU workload, better latency hiding
- Use row-major layout on GPU in LU factorization
  - Requires extra (but fast) matrix transpose for each CPU-GPU transfer
- Substitute triangular solves $LX=B$ with multiply by L^{-1}
  - Provably stable if we do this only when $//L^{-1}// < fixed\_threshold$
  - Small pivot growth nearly always assumes small $//L^{-1}//$
  - Accuracy of LU assumes small pivot growth anyway
- Use two-level and variable size blocking as finer tuning
  - Thicker blocks impose lower bandwidth requirements in SGEMM
  - Variable size blocking improves CPU/GPU load balance
- Use column-cyclic layout when computing using two GPUs
  - Requires no data exchange between GPUs in pivoting
  - Cyclic layout is used on GPUs only so does not affect panel factorizations

Performance Results



LU Factorization using Two GPUs



- Second GPU allows 1.7x higher rates
- More than half-teraflop using two GPUs

Advanced Tips

- Every round reading 2KB with 256 threads grouped into 2 blocks, each of 128 threads
- At least 2 blocks if using blockwise synchronizations
- Total number of active threads >192 for pipeline efficiency

X. Cui, Y. Chen and H. Mei. Improving performance of matrix multiplication and FFT on GPU.

15th ICPADS, pp 42-48, IEEE Computer Society, 2009.

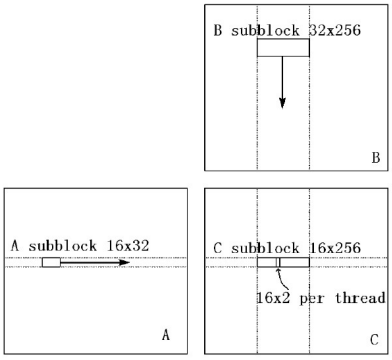


Figure 3: Matrix multiplication: A×B=C with all arrays in row major

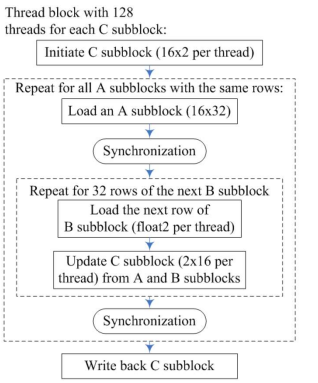
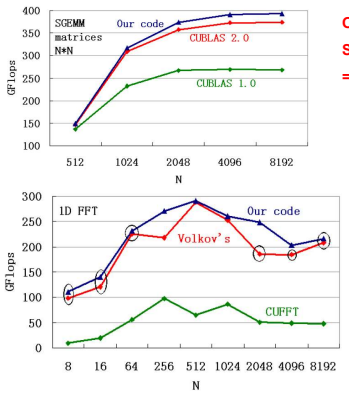


Figure 4: Thread block function for matrix-matrix multiplication.



On GTX 285
SGEMM
= 448 Gflops!

Hand-Tuned SGEMM on GT200 GPU

Lung-Sheng Chien

Department of Mathematics, Tsing Hua university, R.O.C. (Taiwan)

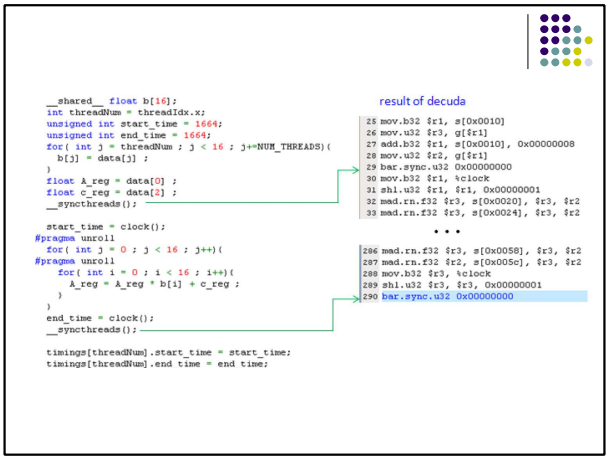
	GTX295[1]	GTX285	TeslaC1060
# of Streaming Processor	240	240	240
Core Frequency	1242MHz	1476 MHz	1.3 GHz
Memory Speed	999MHz	1242 MHz	800 MHz
Memory Interface	448-bit (7 channel)	512-bit (8 channel)	512-bit (8 channel)
Memory Bandwidth (GB/s)	112	159	102
SP, peak (Gflop/s)	894	1063	933
SP without dual issue	596.2	708.5	624
DP, peak (Gflop/s)	74.5	88.6	78
DRAM (MByte)	896	1024	4096

There are two kinds of MAD when operands are "float",

(1) "MAD dest, src1, src2, src3" corresponds to $dest = src1 \times src2 + src3$ where dest, src1, src2 and src3 are all registers.

(2) "MAD dest, [smem], src2, src3" corresponds to $dest = [smem] \times src2 + src3$ where [smem] denotes shared memory.

These two MAD operations have different pipeline latency and throughput. In our method, we change the later in Volkov's code to the former to improve performance.



NUM_THREADS	1	64	128	192	224	256	288	320	384	512
Minimum time	34.6	34.6	34.7	38.6	42.4	46.6	54.0	60.2	72.1	96.1
Maximum time	34.6	34.6	34.8	39.3	43.2	49.5	54.7	60.6	72.7	96.9
Total time for one a = a*b_smem +c	34.6	34.6	34.6	36	42	48	54	60	72	96

Table 2: average number of cycles per "MAD dest, [smem], src2, src3" on TeslaC1060. Pipeline latency is 34.6 cycle and throughput is 6 cycle /warp.

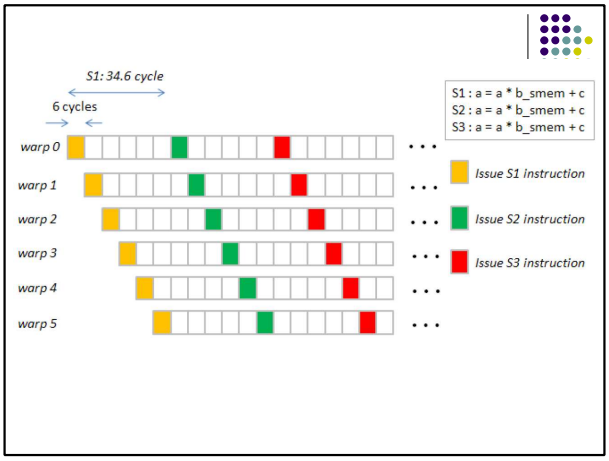
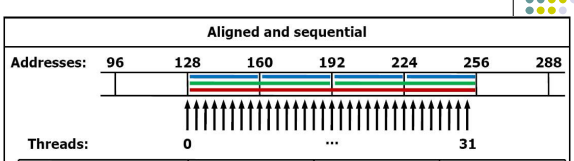


Table A-1. CUDA-Enabled Devices with Compute Capability, Number of Multiprocessors, and Number of CUDA Cores

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	1.1	14	112
Tesla S1070	1.3	4x30	4x240
Tesla C1060	1.3	30	240

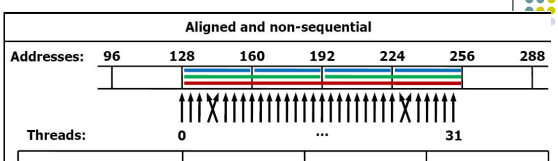
	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of instructions per kernel	2 million				

Aligned and sequential



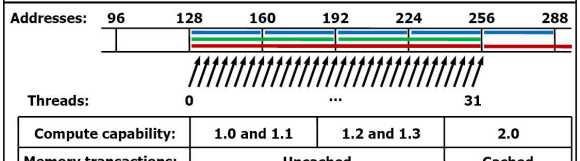
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential

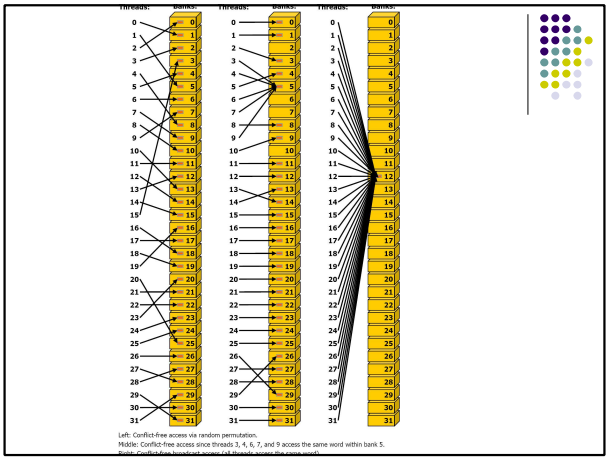


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Misaligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256



Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

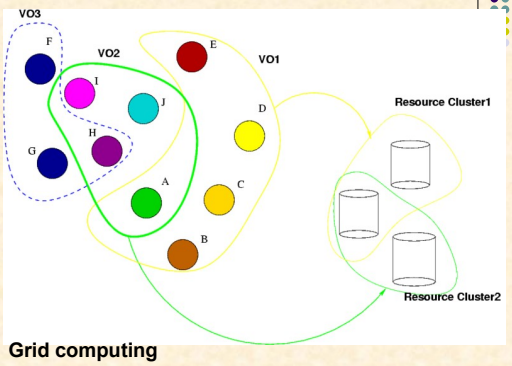
了解各种（非众核）并行程序设计思想

Programmable Memory Hierarchy

- Registers
- Cache
- Physical memory
- Virtual memory
- Local disk storage
- RAID (via LAN) / Distributed shared-memory
- Data Grid

Ideas of parallel computation

- Multi-threading (e.g. Java)
- Concurrent processes (e.g. Unix)
- SIMD (e.g. Connection Machine-2)
- SPMD (e.g. MPI)
- MIMD (e.g. CORBA)
- Message-passing
- Memory-sharing
- Bulk-Synchronous Parallelism
- Work-flow task parallelism, data parallelism



Grid computing

MPI集群编程示例

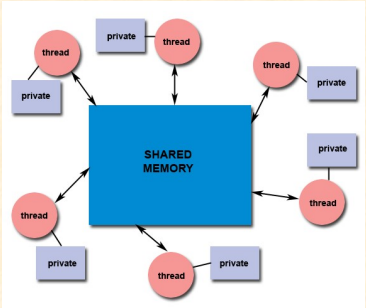
IB/verbs 编程示例

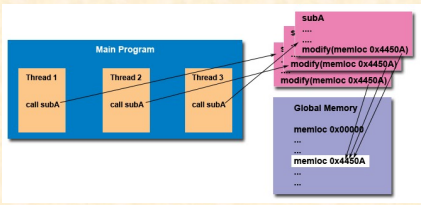
```
while (totscnt < (user_param->iters * user_param->numofqps)  || totccnt < (user_param->iters * user_param->numofqps) ) {
  /* main loop to run over all the qps and post each time n messages */
  for (index =0 ; index < user_param->numofqps ; index++) {
      ctx->wr.wr.rdma.remote_addr = rem_dest[index]->vaddr;
      ctx->wr.wr.rdma.rkey = rem_dest[index]->rkey;
      qp = ctx->qp[index];
      ctx->wr.wr_id      = index ;
      while (ctx->scnt[index] < user_param->iters && (ctx->scnt[index] - ctx->ccnt[index]) < user_param->maxpost[index]) {
      tposted[totscnt] = get_cycles();
      if (ibv_post_send(qp, &ctx->wr, &bad_wr)) {
            fprintf(stderr, "Couldn't post send: qp index = %d qp scnt=%d total scnt %d\n",
                    index,ctx->scnt[index],totscnt);
            return 1;
      }
      ctx->scnt[index]= ctx->scnt[index]+1;
      ++totscnt;
    }
  }
  /* finished posting now polling */
  if (totccnt < (user_param->iters * user_param->numofqps) ) {
    int ne;
    do {
      ne = ibv_poll_cq(ctx->cq, 1, &wc);
    } while (ne == 0);
    tcompleted[totccnt] = get_cycles();  // XXXX ask for the timing
    if (ne < 0) {
      fprintf(stderr, "poll CQ failed %d\n", ne);
      return 1;
    }
```

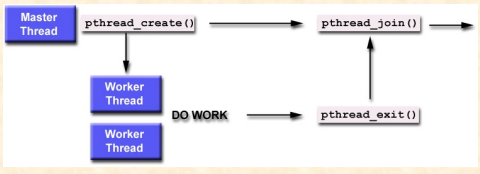
Pthread多核编程示例

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

void *print_message_function( void *ptr );

main()
{
pthread_t thread1, thread2;
char *message1 = "Thread 1";
char *message2 = "Thread 2";
int iret1, iret2;

/* Create independent threads each of which will execute function */

iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

/* Wait till threads are complete before main continues. Unless we */
/* wait we run the risk of executing an exit which will terminate */
/* the process and all threads before the threads have completed. */

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);

printf("Thread 1 returns: %d\n",iret1);
printf("Thread 2 returns: %d\n",iret2);
exit(0);
}

void *print_message_function( void *ptr )
{
char *message;
message = (char *) ptr;
printf("%s \n", message);
}
```

PTHREAD







Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

```
#include <unistd.h>    /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <pthread.h>   /* POSIX Threads */
#include <semaphore.h>  /* Semaphore */
void handler ( void *ptr );
sem_t mutex;int counter; /* shared variable */
int main(){   int i[2];
pthread_t thread_a; pthread_t thread_b;
i[0] = 0; /* argument to threads */   i[1] = 1;
sem_init(&mutex, 0, 1);     /* initialize mutex to 1 - semaphore */
pthread_create (&thread_a, NULL, (void *) &handler, (void *) &i[0]);
pthread_create (&thread_b, NULL, (void *) &handler, (void *) &i[1]);
pthread_join(thread_a, NULL);   pthread_join(thread_b, NULL);
sem_destroy(&mutex); /* destroy semaphore */
exit(0);} /* main() */
```

```
void handler ( void *ptr ) {
    int x;    x = *((int *) ptr);
    printf("Thread %d: Waiting to enter critical region...\n", x);
    sem_wait(&mutex);     /* down semaphore */
/* START CRITICAL REGION */
    printf("Thread %d: Now in critical region...\n", x);
    printf("Thread %d: Counter Value: %d\n", x, counter);
    printf("Thread %d: Incrementing Counter...\n", x);   counter++;
    printf("Thread %d: New Counter Value: %d\n", x, counter);
    printf("Thread %d: Exiting critical region...\n", x);
 /* END CRITICAL REGION */
    sem_post(&mutex);     /* up semaphore */
    pthread_exit(0); /* exit thread */
}
```

MPI

A Typical MPI Program

```
#include "mpi.h"
          :
main(int argc, char** argv) {
          :
    /* No MPI functions called before this */
    MPI_Init(&argc, &argv);
          :
    MPI_Finalize();
    /* No MPI functions called after this */
          :
} /* main */
```

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv) {
    int my_rank;        /* Rank of process */
    int p;              /* Number of processes */
    int source;         /* Rank of sender */
    int dest;           /* Rank of receiver */
    int tag = 50;       /* Tag for messages */
    char message[100];  /* Storage for the message */
    MPI_Status status;  /* Return status for receive */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if (my_rank != 0) {
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen(message)+1 to include '\0' */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
            tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
```

```
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize();
} /* main */
```

When the program is compiled and run with two processes, the output should be

```
Greetings from process 1!
```

If it's run with four processes, the output should be

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```

MPI_Send and MPI_Receive

```
int MPI_Send(void* message, int count,
    MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)

int MPI_Recv(void* message, int count,
    MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Status* status)
```

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

(Explicit) Synchronisation

```
int MPI_Barrier(MPI_Comm comm)
```

MPI_Barrier provides a mechanism for synchronizing all the processes in the communicator comm. Each process blocks (i.e., pauses) until every process in comm has called MPI_Barrier.

Broadcasting

A communication pattern that involves all the processes in a communicator is a *collective communication*. As a consequence, a collective communication usually involves more than two processes. A *broadcast* is a collective communication in which a single process sends the same data to every process. In MPI the function for broadcasting data is MPI_Bcast:

```
int MPI_Bcast(void* message, int count,
    MPI_Datatype datatype, int root, MPI_Comm comm)
```

```
void Get_data2(int my_rank, float* a_ptr, float* b_ptr,
    int* n_ptr) {
    int root = 0;  /* Arguments to MPI_Bcast */
    int count = 1;

    if (my_rank == 0)
      {
          printf("Enter a, b, and n\n");
          scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
      }
      MPI_Bcast(a_ptr, 1, MPI_FLOAT, root,
          MPI_COMM_WORLD);
      MPI_Bcast(b_ptr, 1, MPI_FLOAT, root,
          MPI_COMM_WORLD);
      MPI_Bcast(n_ptr, 1, MPI_INT, root,
          MPI_COMM_WORLD);
} /* Get_data2 */
```

Reduction

```
int MPI_Reduce(void* operand, void* result,
    int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm)
```

Operation Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical And
MPI_BAND	Bitwise And
MPI_LOR	Logical Or
MPI_BOR	Bitwise Or
MPI_LXOR	Logical Exclusive Or
MPI_BXOR	Bitwise Exclusive Or
MPI_MAXLOC	Maximum and Location of Maximum
MPI_MINLOC	Minimum and Location of Minimum

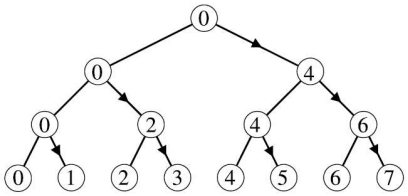
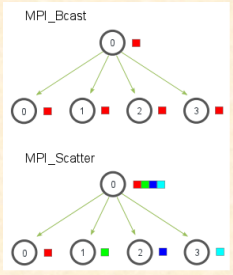


Figure 1: Processors configured as a tree

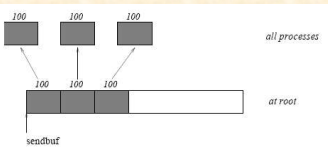
Yet Another Piece of Advice

Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors. (*End of advice to implementors.*)



Scattering

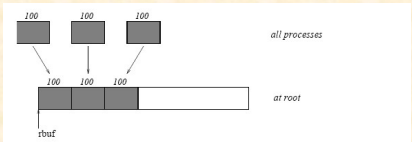
```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)
```



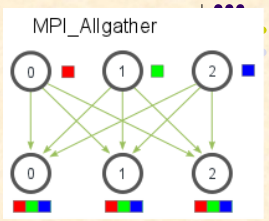
The root process scatters sets of 100 ints to each process in the group.

Gathering

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int *recvcounts, int *displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

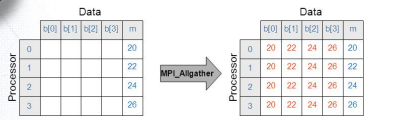


The root process gathers 100 ints from each process in the group.

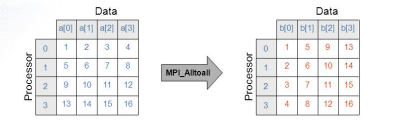


MPI_Allgather(void* send_data,
int send_count,
MPI_Datatype send_datatype,
void* recv_data,
int recv_count,
MPI_Datatype recv_datatype,
MPI_Comm communicator
)

MPI_Allgather / MPI_Alltoall




```
MPI_Allgather(&m,1,MPI_INT,b,1,MPI_INT,MPI_COMM_WORLD);
```



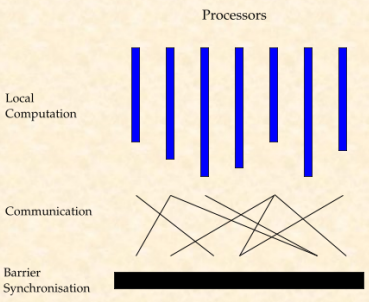

```
MPI_Alltoall(a,1,MPI_INT,b,1,MPI_INT,MPI_COMM_WORLD);
```

MPI 实现 alltoall.c

```
520         /* Do the pairwise exchanges */
521         for (i=1; i<comm_size; i++) {
522             if (pof2 == 1) {
523                 /* use exclusive-or algorithm */
524                 src = dst = rank ^ i;
525             }
526             else {
527                 src = (rank - i + comm_size) % comm_size;
528                 dst = (rank + i) % comm_size;
529             }
530
531             mpi_errno = MPIC_Sendrecv_ft(((char *)sendbuf +
532                                           dst*sendcount*sendtype_extent),
533                                          sendcount, sendtype, dst,
534                                          MPIR_ALLTOALL_TAG,
535                                          ((char *)recvbuf +
536                                           src*recvcount*recvtype_extent),
537                                          recvcount, recvtype, src,
538                                          MPIR_ALLTOALL_TAG, comm, &status, errflag);
539             if (mpi_errno) {
540                 /* for communication errors, just record the error but continue */
541                 *errflag = TRUE;
542                 MPIU_ERR_SET(mpi_errno, MPI_ERR_OTHER, "**fail");
543                 MPIU_ERR_ADD(mpi_errno_ret, mpi_errno);
544             }
545         }
546     }
```

Communication Groups

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group first_row_group;
MPI_Comm first_row_comm;
int row_size;
int* process_ranks;

/* Make a list of the processes in the new
 * communicator */
process_ranks = (int*) malloc(q*sizeof(int));
for (proc = 0; proc < q; proc++)
    process_ranks[proc] = proc;

/* Get the group underlying MPI_COMM_WORLD */
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

/* Create the new group */
MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks, &first_row_group);
MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
    &first_row_comm);
```



GPU集群LINPACK

LINPACK (wiki)

- The **LINPACK Benchmarks** are a measure of a system's floating point computing power. Introduced by Jack Dongarra, they measure how fast a computer solves a dense *N* by *N* system of linear equations *Ax* = *b*, which is a common task in engineering. The solution is obtained by Gaussian elimination with partial pivoting, with 2/3·*N*^3 + 2·*N*^2 floating point operations. The result is reported in millions of floating point operations per second.
- Massimiliano Fatica, *Accelerating linpack with CUDA on heterogenous clusters*, GPGPU'09.

Slides extracted from the above paper

Hardware Configuration

1. SUN Ultra 24 workstation with an Intel Core2 Extreme Q6850 (3.0GHz) CPU and 8GB of memory plus a Tesla C1060 card.
2. Cluster with 8 nodes, each node connected to half of a Tesla S1070 system, containing 4 GPUs, so that each node is connected to 2 GPUs. Each node has 2 Intel Xeon E5462 (2.8GHz with 1600Mhz FSB) and 16GB of memory. The nodes are connected with SDR (Single Data Rate) Infiniband.

Pinned Memory

```
// Regular malloc/free
 double *A;
 A = malloc(N*N*sizeof(double));
 free(A);


// Page-locked version
 double *A;
 cudaMallocHost(A,N*N*sizeof(double));
 cudaFreeHost(A);
```

PCI Bandwidth

	Sun Ultra 24	
	Pageable memory	Pinned memory
H2D	2132 MB/s	5212 MB/s
D2H	1882 MB/s	5471 MB/s

	Supermicro 6015TW	
	Pageable memory	Pinned memory
H2D	2524 MB/s	5651 MB/s
D2H	2084 MB/s	5301 MB/s

Solving Linear Equations

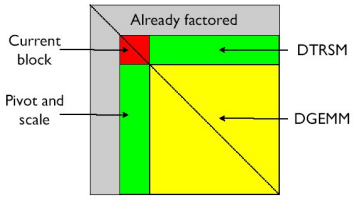


Figure 1: LU factorization: the grey area represents the portion of the matrix already factored. The red area is the current block being factorized. Once this factorization is ready, it is applied to the sub-matrix on the right. The final step is to update the trailing sub-matrix in yellow.

LU Decomposition

$L\,U = A.$

$A\,x = (L\,U)\,x = L\,(U\,x) = b.$

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\begin{bmatrix} l_{11}\,u_{11} & l_{11}\,u_{12} & l_{11}\,u_{13} \\ l_{21}\,u_{11} & l_{21}\,u_{12} + l_{22}\,u_{22} & l_{21}\,u_{13} + l_{22}\,u_{23} \\ l_{31}\,u_{11} & l_{31}\,u_{12} + l_{32}\,u_{22} & l_{31}\,u_{13} + l_{32}\,u_{23} + l_{33}\,u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

first solve $L\,y = b$ for y. This can be done by forward substitution

$$y_1 = \frac{b_1}{l_{11}}$$

$$y_i = \frac{1}{l_{ii}}\left(b_i - \sum_{j=1}^{i-1} l_{ij}\,y_j\right)$$

for $i = 2 \ldots N$. Then solve $U\,x = y$ for x. This can be done by back substitution

$$x_N = \frac{y_N}{u_{N,N}}$$

$$x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{N} u_{ij}\,x_j\right)$$

for $i = N-1 \ldots 1$.

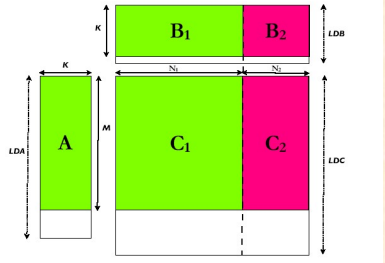


Figure 3: The green portion is performed on the GPU, while the red one is performed on the CPU)

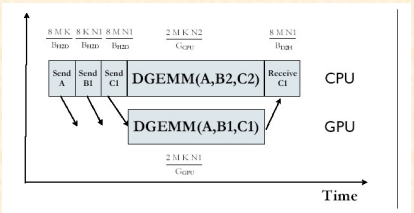


Figure 4: Data flow to split the DGEMM call between CPU cores and GPU.

B_{H2D}: Bandwidth from host to device expressed in GB/s

G_{GPU}: Sustained performance of DGEMM on the GPU expressed in $GFlops$

G_{CPU}: Sustained performance of DGEMM on the CPU expressed in $GFlops$

B_{D2H}: Bandwidth from device to host expressed in GB/s

A DGEMM call on the host CPU performs $2KMN$ operations, so if the CPU cores can perform this operation at G_{CPU} the total time is:

$$T_{CPU}(M,K,N) = 2\frac{MKN}{G_{CPU}}$$

The total time to offload a DGEMM call to the GPU has an I/O component that accounts for both the data transfer from the CPU memory space to the GPU memory space and vice versa plus a computational part once the data is on the GPU. We can express this time as:

$$T_{GPU}(M,K,N) = 8\frac{(MK+KN+MN)}{B_{H2D}} + 2\frac{MKN}{G_{GPU}} + 8\frac{(MN)}{B_{D2H}}$$

the factor 8 is the size of a double in bytes.
The optimal split will be

$T_{CPU}(M,K,N2) = T_{GPU}(M,K,N1)$ with $N = N1+N2$

For an initial approximation of the optimal split fraction $\eta = N1/N$, we can omit the transfer time ($O(N^2)$) compared to the computation ($O(N^3)$). From a simple manipulation, the optimal split is

$$\eta = \frac{G_{GPU}}{G_{GPU} + G_{CPU}}$$

On the cluster, where the quad core Xeon has a DGEMM performance of 40 GFlops and the GPU a DGEMM performance of 82 GFlops, this formula predicts $\eta = 0.67$, very close to the optimal value of 0.68 found by experiments.

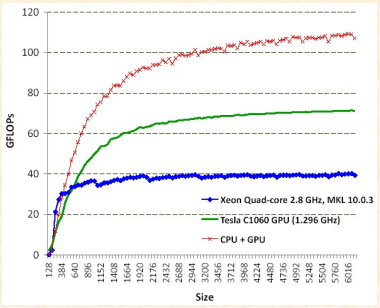
```
// Copy A from CPU memory to GPU memory devA
   status = cublasSetMatrix (m, k     , sizeof(A[0]), A, lda, devA, m_gpu);
// Copy B1  from CPU memory to GPU memory devB
   status = cublasSetMatrix (k ,n_gpu, sizeof(B[0]), B, ldb, devB, k_gpu);
// Copy C1  from CPU memory to GPU memory devC
   status = cublasSetMatrix (m, n_gpu, sizeof(C[0]), C, ldc, devC, m_gpu);

// Perform  DGEMM(devA,devB,devC) on GPU
// Control immediately return to CPU
   cublasDgemm('n', 'n', m, n_gpu, k, alpha, devA, m,devB, k, beta, devC, m);

// Perform  DGEMM(A,B2,C2) on CPU
   dgemm_cpu('n','n',m,n_cpu,k, alpha, A, lda,B+ldb*n_gpu, ldb, beta,C+ldc*n_gpu, ldc);

// Copy devC  from GPU memory to CPU memory C1
   status = cublasGetMatrix (m, n, sizeof(C[0]), devC, m, C, *ldc);
```

It turns out that on Intel systems using Front Side Bus (FSB), it is better not to overlap the transfer to the GPU with computations on the CPU (the memory system cannot supply data to both the PCIe and the CPU at maximum speed).

The DGEMM function call in CUBLAS maps to several different kernels depending on the size of the matrices. The best performance is achieved when M is multiple of 64 and K and N multiple of 16. Performance numbers for different choices of M, K and N are shown in table 5. When all the above conditions are satisfied, the GPU can achieve 82.4 Gflops, 95% of the peak double precision performance.

M	K	N	M%64	K%16	N%16	$Gflops$
448	400	12320	Y	Y	Y	82.4
12320	400	1600	N	Y	Y	72.2
12320	300	448	N	N	Y	55.9
12320	300	300	N	N	N	55.9

Table 3: DGEMM performance on the Tesla S1070 GPU (1.44 GHz) with data resident in GPU memory



The peak performance of the CPU is 48 GFlops, a CPU-only Linpack sustains performance in the $70\% - 80\%$ range for very large problem sizes. The CUDA accelerated version with pinned memory (limited today to 4GB per MPI process) delivers 83.2 Gflops, 66% of the combined 125 GFlops system peak performance (the Tesla C1060 has a peak performance of 77.7 Gflops). The accelerated solution also delivers good performance for small problem sizes and is not very sensitive to the size of NB.

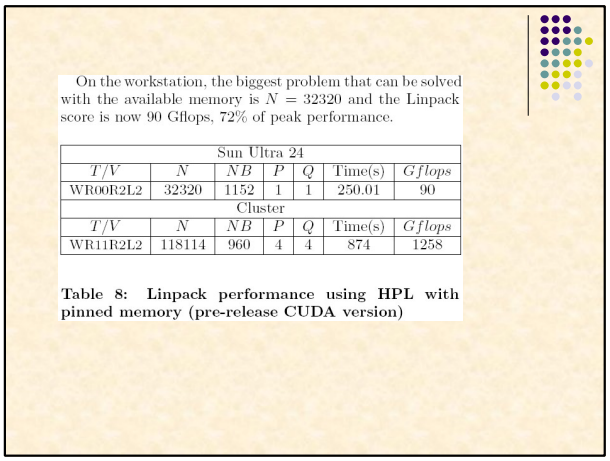
T/V	N	NB	P	Q	Time(s)	$Gflops$
WR00L2L2	23040	960	1	1	97.91	83.28
WR00L2L2	7432	960	1	1	5.47	50.01
WR00L2L2	7432	1152	1	1	5.47	53.56

Table 4: Linpack performance on a Sun Ultra 24 workstation with 1 Intel(R) Core2 Extreme Q6850 (3.0Ghz) and a Tesla C1060 using HPL with pinned memory for different N and NBs.

Table 5 shows the same results using pageable memory. The use of pinned vs. regular pageable memory increases the performance from 70 to 84 Gflops.

T/V	N	NB	P	Q	Time(s)	$Gflops$
WR00L2L2	23040	960	1	1	117.06	69.66
WR00L2L2	23040	1152	1	1	117.06	70.64
WR00L2L2	7432	960	1	1	6.09	44.94
WR00L2L2	7432	1152	1	1	5.97	45.86

Table 5: Linpack performance on a Sun Ultra 24 workstation with 1 Intel(R) Core2 Extreme Q6850 (3.0Ghz) and a Tesla C1060 using HPL with pageable memory



On the workstation, the biggest problem that can be solved with the available memory is $N = 32320$ and the Linpack score is now 90 Gflops, 72% of peak performance.

Sun Ultra 24						
T/V	N	NB	P	Q	Time(s)	$Gflops$
WR00R2L2	32320	1152	1	1	250.01	90
Cluster						
T/V	N	NB	P	Q	Time(s)	$Gflops$
WR11R2L2	118114	960	4	4	874	1258

Table 8: Linpack performance using HPL with pinned memory (pre-release CUDA version)

一致性
时态逻辑
安全性与进展性

用一条直线来表示系统中每一个处理器的运行，时间从左边开始到右边。每一个共享内存的操作我们把它写在处理器的直线上。两个主要的操作为“读”和“写”，用下面的表达式表示：
W(var) value
该表达式的含义为：将值value写到共享变量var中，而
R(var) value
该表达式的含义为：读取共享变量var，获取它的值value.
比如：W(x)1 表示：将1写到x中，而R(y)3表示：读取变量y的值3.
更多的操作（特别是同步操作）在需要的时候我们再来定义他的记号。为简单起见，假设所有的变量都初始化为0.
我们要特别注意一点，在高级语言中的一条语句（比如x = x + 1;）通常将会涉及到几次内存操作。如果x之前为0，则那条高级语言中的语句将变成（不考虑其它处理器）：
P1：R(x)0 W(x)1

严格一致性(Strict Consistency):

- 任意read(x)操作都要读到**最新**的write(x)的结果。依赖于绝对的全局时钟。

P1：W(x)1

P2: R(x)1 R(x)1
这表示，“处理器P1将1写到变量x中；一段时间之后处理器P2读取x的值1。然后再读取一次，获得相同的值。
我们再给出一个符合”内存严格一致性模型”的场景：
P1： W(x)1

P2：R(x)0 R(x)1
这一次，处理器P2先执行，它先读取x的值0，当它第二次读取x时却获得了处理器P1写入的x的值1。注意这两个场景能够通过将同一个程序在同一个处理器上执行两次而获得。
我们给出一个不符合“内存严格一致性模型”的场景：
P1： W(x)1

P2: R(x)0 R(x)1

顺序一致性 (Sequential Consistency):

- “（并发程序在多处理器上的）任何一次执行结果都相同，就像所有处理器的操作按照某个顺序执行，各个微处理器的操作按照其程序指定的顺序进行。”。

在一个写操作发生之前看到该写操作的结果是可能的，
P1: W(x)1

P2: R(x)1

P1: W(x)1

P2: R(x)1 R(x)2

P3: R(x)1 R(x)2

P4: W(x)2
这个场景是合法的顺序一致性内存模型的原因是，下面的交替操作在严格一致性内存模型中将会是合法的：
P1: W(x)1

P2: R(x)1 R(x)2

P3: R(x)1 R(x)2

P4: W(x)2
下面是一个不符合顺序一致性内存模型的场景：
P1: W(x)1

P2: R(x)1 R(x)2

P3: R(x)2 R(x)1

P4: W(x)2

顺序一致性在分布式算法中的应用例

- 最短路径迭代计算收敛终止判定
  - 每个线程独立反复运行Dijkstra算法，并记录第k轮有多少结点被更新；
  - 无更新且其他线程见过该线程最新一轮则进入终止准备状态；
  - 一个线程见到其他线程相关结点更新后先退出终止准备状态，再告知对方已经见过。
  - 原子性判定所有线程都终止准备状态则终止。

弱一致性 (Weak Consistency):

- 如果我们仅仅将竞争访问分为同步与非同步访问，并且同时要求符合下列条件，那么我们就得到了“弱一致性”：
- 1.对同步变量的访问是“顺序一致性”的
- 2.直到之前对所有同步变量的写操作完成之后，才允许访问这些同步变量。
- 3.直到之前对同步变量的访问完成之后，才允许我们访问（读或写）这些同步变量。

下面给出一个符合“弱一致性”的场景，显示了“弱一致性”的真正用途：

P1: W(x)1 W(x)2

P2: R(x)0 R(x)2 S R(x)2

P3: R(x)1 S R(x)2

换一句话说，根本没有要求一个处理器广播它对变量的修改，直到一个同步访问的发生。在一个基于网络而不是总线的分布式系统中，这能够极大的减少信息的互通（注意到，在现实中没有人会故意写一个具有这样行为的程序；你绝对不想去读一个别人正在更新的变量。读操作必须发生在S之后。我提到过一些同步算法，比如松弛算法，它不要求内存一致性的概念。这些算法在“弱一致性”系统中不能工作，因为在“弱一致性”系统中推迟了数据的交流直到同步点。）

因果一致性 （Casual Consistency）

- **有因果关系的写操作必须按照它们的因果关系的顺序被看到，没有因果关系的写操作可以以任意顺序被别的进程看到。**

思考题：

1. **画出一个满足顺序一致但不满足因果一致的情形**
2. **最短路径迭代计算收敛终止判定算法可否使用因果一致性**

Specifying a Simple Clock

$HCini \triangleq hr \in \{1, \ldots, 12\}$

$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$

MODULE *HourClock*

EXTENDS *Naturals*

VARIABLE hr

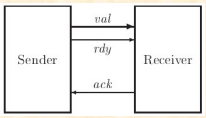
$HCini \triangleq hr \in (1 \,..\, 12)$

$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$

$HC \triangleq HCini \wedge \Box[HCnxt]_{hr}$

THEOREM $HC \Rightarrow \Box HCini$

An Asynchronous Interface



$$\begin{bmatrix} val = 26 \\ rdy = 0 \\ ack = 0 \end{bmatrix} \xrightarrow{Send\ 37} \begin{bmatrix} val = 37 \\ rdy = 1 \\ ack = 0 \end{bmatrix} \xrightarrow{Ack} \begin{bmatrix} val = 37 \\ rdy = 1 \\ ack = 1 \end{bmatrix} \xrightarrow{Send\ 4}$$

$$\begin{bmatrix} val = 4 \\ rdy = 0 \\ ack = 1 \end{bmatrix} \xrightarrow{Ack} \begin{bmatrix} val = 4 \\ rdy = 0 \\ ack = 0 \end{bmatrix} \xrightarrow{Send\ 19} \begin{bmatrix} val = 19 \\ rdy = 1 \\ ack = 0 \end{bmatrix} \xrightarrow{Ack} \cdots$$

MODULE *AsynchInterface*

EXTENDS *Naturals*
CONSTANT *Data*
VARIABLES *val*, *rdy*, *ack*

$$
\begin{array}{ll}
\mathit{TypeInvariant} \triangleq & \land\ \mathit{val} \in \mathit{Data} \\
& \land\ \mathit{rdy} \in \{0,1\} \\
& \land\ \mathit{ack} \in \{0,1\}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Init} \triangleq & \land\ \mathit{val} \in \mathit{Data} \\
& \land\ \mathit{rdy} \in \{0,1\} \\
& \land\ \mathit{ack} = \mathit{rdy}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Send} \triangleq & \land\ \mathit{rdy} = \mathit{ack} \\
& \land\ \mathit{val}' \in \mathit{Data} \\
& \land\ \mathit{rdy}' = 1 - \mathit{rdy} \\
& \land\ \text{UNCHANGED } \mathit{ack}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Rcv} \triangleq & \land\ \mathit{rdy} \neq \mathit{ack} \\
& \land\ \mathit{ack}' = 1 - \mathit{ack} \\
& \land\ \text{UNCHANGED } \langle \mathit{val}, \mathit{rdy} \rangle
\end{array}
$$

$$\mathit{Next} \triangleq \mathit{Send} \lor \mathit{Rcv}$$

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{val}, \mathit{rdy}, \mathit{ack} \rangle}$$

THEOREM $\mathit{Spec} \Rightarrow \Box\, \mathit{TypeInvariant}$

A: "The moon is a satellite of the earth"
B: "The moon is rising"
C: "The moon is setting"

■ (denoting "always") and ◆ (denoting "eventually")

D: ◆ *B*	"The moon will be rising eventually"
E: ■ ◆ *B*	"The moon will be rising again and again"
F: ■ (*B* ⇒ ◆ *C*)	"Moon rise leads to moon setting"

◆ terminated

(■ ◆ sent) ⇒ (◆ delivered)

■ (*P* ⇒ ◆ *Q*)

◊□ P
□ ◊P
◊□ ◊P
□ ◊□ P

□ □ P = ?
◊◊P = ?
¬□ ¬P = ?
¬◊ ¬P = ?

An execution of a concurrent program can be viewed as an infinite sequence of states:

$\sigma = s_0 s_1 \ldots$.

Each state after s_0 results from executing a single atomic action in the preceding state. (For a terminating execution, an infinite sequence is obtained by repeating the final state.) A *property* is a set of such sequences. Since a program also defines a set of sequences of states, we say that a property *holds* for a program if the set of sequences defined by the program is contained in the property.

Informally, a safety property stipulates that some 'bad thing' does not happen during execution [2]. Examples of safety properties include mutual exclusion, deadlock freedom, partial correctness, and first-come-first-serve. In *mutual exclusion*, the proscribed 'bad thing' is two processes executing in critical sections at the same time. In *deadlock freedom* it is deadlock. In *partial cor-*

We now formalize safety.[1] Let S be the set of program states, S^ω the set of infinite sequences of program states, and S* the set of finite sequences of program states. An execution of any program can be modeled as a member of S^ω. We call elements of S^ω *executions* and elements of S* *partial executions* and write $\sigma \models P$ when execution σ is in property P. Finally, let σ_i denote the partial execution consisting of the first i states in σ.

Safety properties

For P to be a safety property, if P does not hold for an execution, then at some point some 'bad thing' must happen. Such a 'bad thing' must be irremediable because a safety property states that the 'bad thing' never happens during execution. Thus, P is a *safety property* if and only if the following holds.

Safety

$$(\forall \sigma : \sigma \in S^{\omega} : \sigma \not\models P \Rightarrow (\exists i : 0 \leqslant i : (\forall \beta : \beta \in S^{\omega} : \sigma_i \beta \not\models P))).$$

任何无限运行序列，如若违反P性质，则存在其有限前缀，使得该前缀之所有无限扩展均违反P。

Liveness properties

Informally, a liveness property stipulates that a 'good thing' happens during execution [2]. Examples of liveness properties include starvation freedom, termination, and guaranteed service. In *starvation freedom*, which states that a process makes progress infinitely often, the 'good thing' is making progress. In *termination*, which asserts that a program does not run forever, the 'good thing' is completion of the final instruction. Finally, in *guaranteed service*,[2] which states that every request for service is satisfied eventually, the 'good thing' is receiving service.

We now formalize liveness. A partial execution α is *live* for a property P if and only if there is a sequence of states β such that $\alpha\beta \models P$. A *liveness property* is one for which every partial execution is live. Thus, P is a liveness property if and only if the following holds.

Liveness

$$(\forall \alpha : \alpha \in S^{*} : (\exists \beta : \beta \in S^{\omega} : \alpha\beta \models P)).$$

任何有限运行序列均可以被无限扩展为一个满足P的无限序列。

Theorem 1. *Every property* P *is the intersection of a safety property and a liveness property.*

Proof. Let $\overline{P}$ be the smallest safety property containing P and let L be $\neg(\overline{P} - P)$. Then,

$$\begin{aligned} L \cap \overline{P} &= \neg(\overline{P} - P) \cap \overline{P} \\ &= (\neg\overline{P} \cup P) \cap \overline{P} \\ &= (\neg\overline{P} \cap \overline{P}) \cup (P \cap \overline{P}) \\ &= P \cap \overline{P} = P. \end{aligned}$$

It remains to show that L is dense, and hence a liveness property. By way of contradiction, suppose there is a nonempty open set O contained in $\neg$L and thus L is not dense. Then, $O \subseteq (\overline{P} - P)$. Consequently, $P \subseteq (\overline{P} - O)$. The intersection of two closed sets is closed, so $\overline{P} - O$ is closed and thus a safety property. This contradicts the hypothesis that $\overline{P}$ is the smallest safety property containing P. □

进程代数

Communicating Sequential Processes

C. A. R. Hoare

VMS—the simple vending machine
VMC—the complex vending machine
and the letters P, Q, R (occurring in laws) stand for arbitrary processes.

3. The letters x, y, z are variables denoting events.
4. The letters A, B, C stand for sets of events.
5. The letters X, Y are variables denoting processes.
6. The alphabet of process P is denoted αP, e.g.,
   $\alpha VMS = \{coin, choc\}$
   $\alpha VMC = \{in1p, in2p, small, large, out1p\}$

The process with alphabet A which never actually engages in any of the events of A is called $STOP_A$. This describes the behaviour of a broken object:

Let x be an event and let P be a process. Then

$(x \rightarrow P)$ (pronounced "x then P")

$\alpha(x \rightarrow P) = \alpha P$ provided $x \in \alpha P$

X1 A simple vending machine which consumes one coin before breaking

$(coin \rightarrow STOP_{\alpha VMS})$

X2 A simple vending machine that successfully serves two customers before breaking

$(coin \rightarrow (choc \rightarrow (coin \rightarrow (choc \rightarrow STOP_{\alpha VMS}))))$

X3 A counter starts on the bottom left square of a board, and can move only up or right to an adjacent white square

$\alpha CTR = \{up, right\}$
$CTR = (right \rightarrow up \rightarrow right \rightarrow right \rightarrow STOP_{\alpha CTR})$

$\alpha CLOCK = \{tick\}$

$CLOCK = (tick \rightarrow CLOCK)$

$$
\begin{aligned}
&CLOCK \\
&\quad = (tick \rightarrow CLOCK) && \text{[original equation]} \\
&\quad = (tick \rightarrow (tick \rightarrow CLOCK)) && \text{[by substitution]} \\
&\quad = (tick \rightarrow (tick \rightarrow (tick \rightarrow CLOCK))) && \text{[similarly]}
\end{aligned}
$$

$tick \rightarrow tick \rightarrow tick \rightarrow \cdots$

X1 A perpetual clock

$CLOCK = \mu X : \{tick\} \bullet (tick \rightarrow X)$

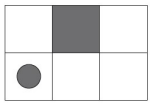
Choice

$(x \rightarrow P \mid y \rightarrow Q)$

$\alpha(x \rightarrow P \mid y \rightarrow Q) = \alpha P$ provided $\{x, y\} \subseteq \alpha P$ and $\alpha P = \alpha Q$

The bar $\mid$ should be pronounced "choice": "x then P choice y then Q"

X1 The possible movements of a counter on the board



are defined by the process

$(up \rightarrow STOP \mid right \rightarrow right \rightarrow up \rightarrow STOP)$

X2 A machine which offers a choice of two combinations of change for 5p (compare 1.1.2 X3 and X4, which offer no choice).

$$CH5C = in5p \rightarrow (out1p \rightarrow out1p \rightarrow out1p \rightarrow out2p \rightarrow CH5C \mid out2p \rightarrow out1p \rightarrow out2p \rightarrow CH5C)$$

The choice is exercised by the customer of the machine. □

X3 A machine that serves either chocolate or toffee on each transaction

$$VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X)$$

□

X4 A more complicated vending machine, which offers a choice of coins and a choice of goods and change

$$VMC = (in2p \rightarrow (large \rightarrow VMC \mid small \rightarrow out1p \rightarrow VMC) \mid in1p \rightarrow (small \rightarrow VMC \mid in1p \rightarrow (large \rightarrow VMC \mid in1p \rightarrow STOP)))$$

X7 A copying process engages in the following events

in.0—input of zero on its input channel
in.1—input of one on its input channel
out.0—output of zero on its output channel
out.1—output of one on its output channel

Its behaviour consists of a repetition of pairs of events. On each cycle, it inputs a bit and outputs the same bit

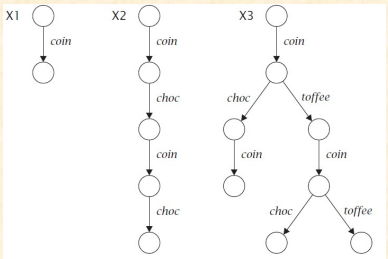
$$COPYBIT = \mu X \bullet (in.0 \rightarrow out.0 \rightarrow X \mid in.1 \rightarrow out.1 \rightarrow X)$$

Pictures

$(coin \rightarrow STOP_{\alpha VMS})$

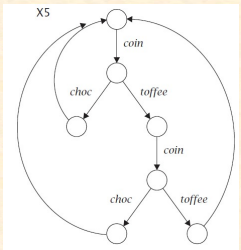
$(coin \rightarrow (choc \rightarrow (coin \rightarrow (choc \rightarrow STOP_{\alpha VMS}))))$

$VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X)$



X5 A machine that allows its customer to sample a chocolate, and trusts him to pay after. The normal sequence of events is also allowed

$$VMCRED = \mu X \bullet (coin \rightarrow choc \rightarrow X \mid choc \rightarrow coin \rightarrow X)$$



Laws

L1 $(x : A \rightarrow P(x)) = (y : B \rightarrow Q(y)) \equiv (A = B \wedge \forall x : A \bullet P(x) = Q(x))$

$(x \rightarrow P \mid y \rightarrow Q) = (y \rightarrow Q \mid x \rightarrow P)$

$(x \rightarrow P) \neq STOP$

$(c \rightarrow P) \neq (d \rightarrow Q)$ if $c \neq d$

$(c \rightarrow P) = (c \rightarrow Q) \equiv P = Q$

$(coin \rightarrow choc \rightarrow coin \rightarrow choc \rightarrow STOP) \neq (coin \rightarrow STOP)$

$\mu X \bullet F(X) = F(\mu X \bullet F(X))$

Traces

$\langle x, y \rangle$ consists of two events, x followed by y.
$\langle x \rangle$ is a sequence containing only the event x.
$\langle\rangle$ is the empty sequence containing no events.

$\langle coin, choc, coin, choc \rangle$

$s \frown t$

$\langle coin, choc \rangle \frown \langle coin, toffee \rangle = \langle coin, choc, coin, toffee \rangle$

$\langle in1p \rangle \frown \langle in1p \rangle = \langle in1p, in1p \rangle$

$\langle in1p, in1p \rangle \frown \langle\rangle = \langle in1p, in1p \rangle$

L1 $s \frown \langle\rangle = \langle\rangle \frown s = s$

L2 $s \frown (t \frown y) = (s \frown t) \frown u$

X1 The only trace of the behaviour of the process *STOP* is $\langle\rangle$. The notebook of the observer of this process remains forever blank

$$traces(STOP) = \{\langle\rangle\}$$

□

X2 There are only two traces of the machine that ingests a coin before breaking

$$traces(coin \rightarrow STOP) = \{\langle\rangle, \langle coin\rangle\}$$

□

X3 A clock that does nothing but *tick*

$$\begin{aligned} traces(\mu X \bullet tick \rightarrow X) &= \{\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \ldots\} \\ &= \{tick\}^* \end{aligned}$$

As with most interesting processes, the set of traces is infinite, although of course each individual trace is finite. □

X4 A simple vending machine

$$traces(\mu X \bullet coin \rightarrow choc \rightarrow X) = \{\, s \mid \exists n \bullet s \leq \langle coin, choc\rangle^n \,\}$$

□

L1 $traces(STOP) = \{\, t \mid t = \langle\rangle \,\} = \{\langle\rangle\}$

L2 $$\begin{aligned} traces(c \rightarrow P) &= \{\, t \mid t = \langle\rangle \vee (t_0 = c \wedge t' \in traces(P)) \,\} \\ &= \{\langle\rangle\} \cup \{\, \langle c\rangle \frown t \mid t \in traces(P) \,\} \end{aligned}$$

L3 $$\begin{aligned} &traces(c \rightarrow P \mid d \rightarrow Q) = \\ &\quad \{\, t \mid t = \langle\rangle \vee (t_0 = c \wedge t' \in traces(P)) \vee (t_0 = d \wedge t' \in traces(Q)) \,\} \end{aligned}$$

If P and Q are processes with the same alphabet, we introduce the notation

$$P \parallel Q$$

to denote the process which behaves like the system composed of processes P and Q interacting in lock-step synchronisation as described above.

Examples

X1 A greedy customer of a vending machine is perfectly happy to obtain a toffee or even a chocolate without paying. However, if thwarted in these desires, he is reluctantly prepared to pay a coin, but then he insists on taking a chocolate

$$\begin{aligned} GRCUST = (&toffee \rightarrow GRCUST \\ \mid\ &choc \rightarrow GRCUST \\ \mid\ &coin \rightarrow choc \rightarrow GRCUST) \end{aligned}$$

When this customer is brought together with the machine *VMCT* (1.1.3 X3) his greed is frustrated, since the vending machine does not allow goods to be extracted before payment. Similarly, *VMCT* never gives a toffee, because the customer never wants one after he has paid

$$(GRCUST \parallel VMCT) = \mu X \bullet (coin \rightarrow choc \rightarrow X)$$

X2 A foolish customer wants a large biscuit, so he puts his coin in the vending machine *VMC*. He does not notice whether he has inserted a large coin or a small one; nevertheless, he is determined on a large biscuit

$$\begin{aligned} FOOLCUST = (&in2p \rightarrow large \rightarrow FOOLCUST \\ \mid\ &in1p \rightarrow large \rightarrow FOOLCUST) \end{aligned}$$

Unfortunately, the vending machine is not prepared to yield a large biscuit for only a small coin

$$(FOOLCUST \parallel VMC) = \mu X \bullet (in2p \rightarrow large \rightarrow X \mid in1p \rightarrow STOP)$$

L1 $P \parallel Q = Q \parallel P$

The next law shows that when three processes are assembled, it does not matter in which order they are put together

L2 $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$

Thirdly, a deadlocked process infects the whole system with deadlock; but composition with $RUN_{\alpha P}$ (1.1.3 X8) makes no difference

L3A $P \parallel STOP_{\alpha P} = STOP_{\alpha P}$

L3B $P \parallel RUN_{\alpha P} = P$

The next laws show how a pair of processes either engage simultaneously in the same action, or deadlock if they disagree on what the first action should be

L4A $(c \rightarrow P) \parallel (c \rightarrow Q) = (c \rightarrow (P \parallel Q))$

L4B $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP$ if $c \neq d$

L4 $(x : A \rightarrow P(x)) \parallel (y : B \rightarrow Q(y)) = (z : (A \cap B) \rightarrow (P(z) \parallel Q(z)))$

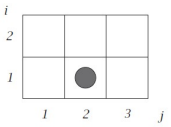
Example

X1 Let $P = (a \rightarrow b \rightarrow P \mid b \rightarrow P)$
and $Q = (a \rightarrow (b \rightarrow Q \mid c \rightarrow Q))$
Then

$$\begin{aligned} &(P \parallel Q) = \\ &= a \rightarrow ((b \rightarrow P) \parallel (b \rightarrow Q \mid c \rightarrow Q)) && \text{[by L4A]} \\ &= a \rightarrow (b \rightarrow (P \parallel Q)) && \text{[by L4A]} \\ &= \mu X \bullet (a \rightarrow b \rightarrow X) && \text{[since the recursion is guarded.]} \end{aligned}$$

□

X2



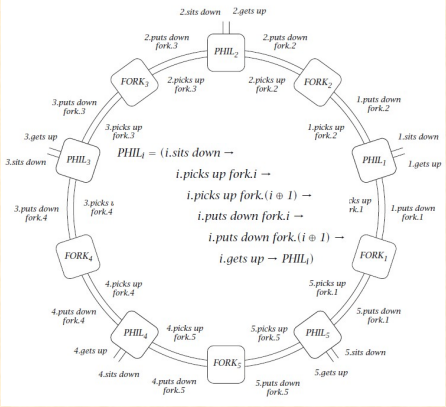
A counter starts at the middle bottom square of the board, and may move within the board either *up*, *down*, *left* or *right*. Let

$$\alpha P = \{up, down\}$$
$$P = (up \rightarrow down \rightarrow P)$$
$$\alpha Q = \{left, right\}$$
$$Q = (right \rightarrow left \rightarrow Q \mid left \rightarrow right \rightarrow Q)$$

哲学家进餐问题

$$\alpha PHIL_i = \{i.sits\ down, i.gets\ up, i.picks\ up\ fork.i, i.picks\ up\ fork.(i \oplus 1), i.puts\ down\ fork.i, i.puts\ down\ fork.(i \oplus 1)\}$$

$$\alpha FORK_i = \{i.picks\ up\ fork.i, (i \ominus 1).picks\ up\ fork.i, i.puts\ down\ fork.i, (i \ominus 1).puts\ down\ fork.i\}$$



$$FORK_i = (i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow FORK_i \mid (i \ominus 1).picks\ up\ fork.i \rightarrow (i \ominus 1).puts\ down\ fork.i \rightarrow FORK_i)$$

$$PHILOS = (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4)$$
$$FORKS = (FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4)$$
$$COLLEGE = PHILOS \parallel FORKS$$

expensive. The solution finally adopted was the appointment of a footman, whose duty it was to assist each philosopher into and out of his chair. His alphabet was defined as

$$\bigcup_{i=0}^{4} \{i.sits\ down, i.gets\ up\}$$

This footman was given secret instructions never to allow more than four philosophers to be seated simultaneously. His behaviour is most simply defined by mutual recursion. Let

$$U = \bigcup_{i=0}^{4} \{i.gets\ up\} \qquad D = \bigcup_{i=0}^{4} \{i.sits\ down\}$$

$FOOT_j$ defines the behaviour of the footman with j philosophers seated

$$FOOT_0 = (x : D \rightarrow FOOT_1)$$
$$FOOT_j = (x : D \rightarrow FOOT_{j+1} \mid y : U \rightarrow FOOT_{j-1}) \quad \text{for } j \in \{1, 2, 3\}$$
$$FOOT_4 = (y : U \rightarrow FOOT_3)$$

A college free of deadlock is defined

$$NEWCOLLEGE = (COLLEGE \parallel FOOT_0)$$

3.2 Nondeterministic or

If P and Q are processes, then we introduce the notation

$$P \sqcap Q \qquad (P \text{ or } Q)$$

to denote a process which behaves either like P or like Q, where the selection between them is made arbitrarily, without the knowledge of control of the external environment. The alphabets of the operands are assumed to be the same

$$\alpha(P \sqcap Q) = \alpha P = \alpha Q$$

Examples

X1 A change-giving machine which always gives the right change in one of two combinations

$$CH5D = (in5p \rightarrow ((out1p \rightarrow out1p \rightarrow out1p \rightarrow out2p \rightarrow CH5D) \sqcap (out2p \rightarrow out1p \rightarrow out2p \rightarrow CH5D)))$$

□

X2 *CH5D* may give a different combination of change on each occasion of use. Here is a machine that always gives the same combination, but we do not know initially which it will be (see 1.1.2 X3, X4)

$$CH5E = CH5A \sqcap CH5B$$

3.2.1 Laws

The algebraic laws governing nondeterministic choice are exceptionally simple and obvious. A choice between P and P is vacuous

L1 $P \sqcap P = P$ (idempotence)

It does not matter in which order the choice is presented

L2 $P \sqcap Q = Q \sqcap P$ (symmetry)

A choice between three alternatives can be split into two successive binary choices. It does not matter in which way this is done

L3 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ (associativity)

The occasion on which a nondeterministic choice is made is not significant. A process which first does x and then makes a choice is indistinguishable from one which first makes the choice and then does x

L4 $x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q)$ (distribution)

L5 $(x : B \rightarrow (P(x) \sqcap Q(x))) = (x : B \rightarrow P(x)) \sqcap (x : B \rightarrow Q(x))$

L6 $P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)$

L7 $(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$

L8 $f(P \sqcap Q) = f(P) \sqcap f(Q)$

However, the recursion operator is *not* distributive, except in the trivial case where the operands of $\sqcap$ are identical. This point is simply illustrated by the difference between the two processes

$P = \mu X \bullet ((a \rightarrow X) \sqcap (b \rightarrow X))$

$Q = (\mu X \bullet (a \rightarrow X)) \sqcap (\mu X \bullet (b \rightarrow X))$

4.2 Input and output

Let v be any member of $\alpha c(P)$. A process which first outputs v on the channel c and then behaves like P is defined

$(c!v \rightarrow P) = (c.v \rightarrow P)$

The only event in which this process is initially prepared to engage is the communication event $c.v$.

A process which is initially prepared to input any value x communicable on the channel c, and then behave like $P(x)$, is defined

$(c?x \rightarrow P(x)) = (y : \{ y \mid channel(y) = c \} \rightarrow P(message(y)))$

Example

X1 Using the new definitions of input and output we can rewrite 1.1.3 X7

$COPYBIT = \mu X \bullet (in?x \rightarrow (out!x \rightarrow X))$

where $\alpha in(COPYBIT) = \alpha out(COPYBIT) = \{0, 1\}$ □

We shall observe the convention that channels are used for communication in only one direction and between only two processes. A channel which is used only for output by a process will be called an output channel of that process; and one used only for input will be called an input channel. In both cases, we shall say loosely that the channel name is a member of the alphabet of the process.

When drawing a connection diagram (Section 2.4) of a process, the channels are drawn as arrows in the appropriate direction, and labelled with the name of the channel (Figure 4.1).

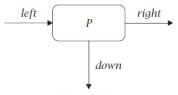
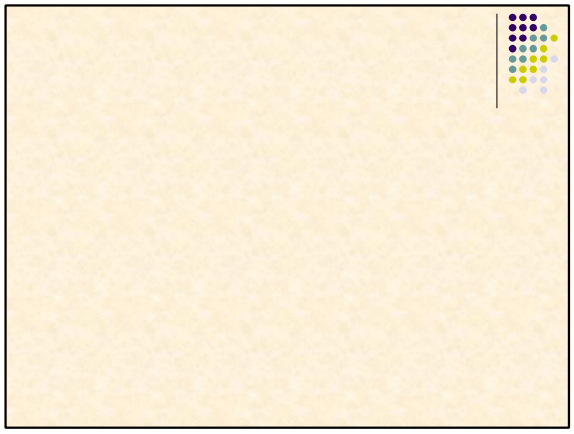
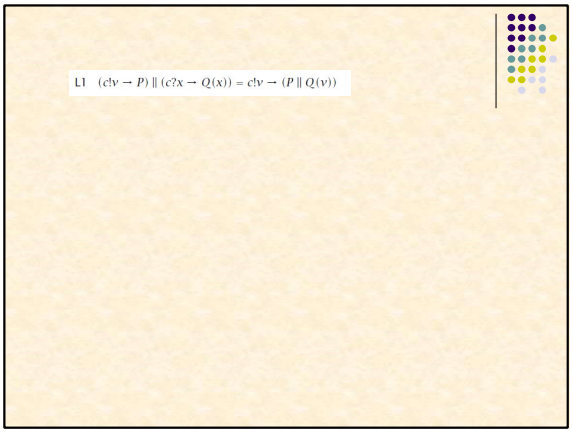


Figure 4.1

L2 $((c!v \rightarrow P) \parallel (c?x \rightarrow Q(x))) \setminus C = (P \parallel Q(v)) \setminus C$

where $C = \{ c.v \mid v \in \alpha c \}$




L1 $(c!v \to P) \parallel (c?x \to Q(x)) = c!v \to (P \parallel Q(v))$

TEMPORAL LOGIC

Heiko Krumm

University of Dortmund, Department of Computer Science

Symbolic logic generally supports the reasoning with propositions, i.e., with statements to be evaluated to true or false. Temporal logic is a special branch of symbolic logic focussing on propositions whose truth values depend on time. That contrasts with the classical logic point of view where the truth value of a repeatedly uttered proposition must always be the same and must neither depend on the modalities of the repetition nor on additional information. Temporal propositions typically contain some (explicit or implicit) reference to time conditions, while classical logic deals with timeless propositions. For instance consider the following examples:

A: "The moon is a satellite of the earth"
B: "The moon is rising"
C: "The moon is setting"

Proposition *A* can be viewed as timeless, since it is true in past, present, and future. In contrast, the propositions *B* and *C* have a temporalized aspect and refer to the implicit time condition "now". Consequently temporal logic applies to time-related universes of discourse where behaviors and courses of events are of interest. As classical logic formulas can characterize static states and properties, temporal logic formulas can describe sequences of state changes and properties of behaviors.

Classical logic comprises different logics; several variants of propositional logic, first order predicate logic, etc., exist with different sets of logical operators and inference rules. Likewise some temporal logics were proposed which differ with respect to their formula syntax, semantics, and expressiveness. A temporal logic, however, basically results from an extension of a classical propositional or predicate logic by temporal quantifiers introducing temporalized modalities. Usually, there are at least the two quantifiers ■ (denoting "always") and ◆ (denoting "eventually") and typical formulas are similar to following examples:

D: ◆ *B* "The moon will be rising eventually"
E: ■ ◆ *B* "The moon will be rising again and again"
F: ■ (*B* ⇒ ◆ *C*) "Moon rise leads to moon setting"

The example formula *D* is true, if the moon is rising now or will be rising in some future point of time. Formula *E* exemplifies that combinations of temporal quantifiers can denote more complex time conditions, e.g., "always eventually" can correspond to the natural language term "again and again". Finally, formula *F* is an example of a "leads-to" pattern describing that always a precondition *B* will eventually result in a postcondition *C*.

Due to its temporal quantifiers temporal logic is a convenient and appropriate means to reason with time-related propositions. Indeed, classical logic can also handle temporal properties, but the formulas tend to be complicated since points of time have to be explicitly represented in the underlying universe. The formula *E* may serve as example and underpin the usefulness of temporal logics. The easy-to-read temporal logic formula *E* corresponds to following predicate logic formula: "For all subjects *x* a subject *y* exists such, that – if *x* is a point of time – *y* is a point of time equal or later to *x* and the moon is rising at *y*".

History

Temporal logic is rooted in the field of exact philosophy and is a variant of modal logic. Modal logic deals with propositions which are interpreted with respect to a set of possible worlds. The truth value of propositions depends on the respective world and basically two operators "necessarily" and "possibly" exist which denote that a proposition is true in all possible worlds res. in some possible worlds. Even the ancient Greek philosophy schools of the Megarians, Stoics, and Peripatetics as well as Aristotle used some temporalized form of these modal operators. During the Middle Ages Arabian and European logicians resumed and refined the ancient approaches in order to discern different types of necessity and possibility. In modern times, the interest in symbolic logic grew during the first half of the 20th century, and – with some delay – new modal and temporal logic approaches occurred. First publications date back to the 1940's.

In particular, the logicians Prior, Rescher, Kripke, and Scott contributed to the development of modern temporal logic. Kripke presented a formal possible world semantics for modal logics. Prior proposed a temporal interpretation. An ordered set of possible worlds can correspond to a temporal sequence of states. In result, the two basic modal operators "necessarily" and "possibly" become the temporal quantifiers "always" and "eventually". Based on the linearity of time additional operators like "next" and "until" as well as past operators were introduced. Rescher and Urquhart outlined the history and introduced several major systems of temporal logic in [5]. In 1974, Burstall proposed the application of

temporal logic in computer science for the first time. Pnueli improved this approach in [4], which is regarded as the classic source of temporal logic based program specification and verification.

Computer Science Application

In several fields of computer science there is a needs for the formal description of event-discrete processes and the corresponding reasoning. In the main, we have to mention the formal specification and verification of so-called reactive systems, the formalization of real-life processes as well as the semantics of natural language commands to be modeled in artificial intelligence, and finally the handling of dynamic consistency conditions in data base systems.

We focus on reactive systems. In particular, Manna and Pnueli recognized in [3] that reactive systems are of growing interest and that temporal logic is well-suited for their formal specification and verification. In contrast to those programs which transform starting states into final results and which may be specified by pre- and postconditions, reactive systems interact with their environment during runtime and the course of interactions and system events is essential. The range of reactive systems is wide and growing. It comprises embedded systems, process control systems, and all types of interactive, concurrent or distributed hard- and software systems. Due to their inherent concurrency, their elaborated fault-tolerance, coordination, and interaction mechanisms distributed systems are rather complex reactive systems and usually need particular design and development tools which support the formal handling of dynamic aspects. Here, temporal logic is profitably applied with respect to following topics:

1. Formal specification: Temporal logic formulas serve as precise, concise and binding descriptions of systems and components (e.g., as proposed by Lamport, Manna, and Pnueli in [2] res. [3]).
2. Formal verification: The rules of a temporal logic proof calculus are applied to show the correctness of a temporal logic specification with respect to more abstract system specifications (e.g., in [2] and [3]).
3. Requirements description: During the early system design the results of the requirements constraining the functional system behavior are represented by a set of temporal logic formulas.
4. Specification checks: Even if the design specifications use other means than temporal logic (e.g., other formal description techniques, *see* SDL, Estelle, and LOTOS, *see also* UNITY), temporal logic may be applied additionally in order to describe requirements and plausibility conditions. Meanwhile several approaches exist which support the tool-based checking of formal system specifications with respect to temporal logic conditions (*see* Model Checking).

Linear and Branching Time

Usually, a temporal logic can be classified as so-called linear-time logic which considers behaviors modeled as linear sequences of states. Within one behavior, each state has exactly one future. Additionally, so-called branching-time logics are known. Here, the formulas refer to tree-structured behaviors where a state can have several futures. The behavior-trees can directly correspond to tree models of non-deterministic systems (e.g., synchronization and communication trees, *see* Calculus of Communicating Systems). A corresponding prominent branching-time logic is CTL (computation tree logic, proposed by Clarke, Emerson, and Sistla in [1]). Its temporal quantifiers directly support the navigation in behavior trees.

Non-deterministic systems, however, have not necessarily to be modeled by behavior trees. Likewise, a set of linear state sequences can form a model of a non-deterministic system where each state sequence corresponds to one possible evolution of the system. In comparison with this linear-time approach, branching-time logics additionally provide for notions of potential behaviors since branching-time formulas can describe properties of branches which correspond to subsets of the possible execution sequences while linear-time formulas generally state properties of all possible sequences.

Variants

Besides of the mentioned distinctions between temporal propositional and predicate logics and between linear-time and branching-time logics, there exist further variants. Some introduce additional temporal quantifiers like “always in the past”, “sometimes in the past”, “next”, “precedes”, “until”, and “leads-to”. Others extend the time model, e.g., in order to describe time-intervals or real-time quantifications. Furthermore, partial-order temporal logics were proposed which directly refer to partial-order representations of concurrency (*see* Concurrency Model).

Example

To exemplify the application of temporal logic for the specification and verification of systems we outline some formula and proof patterns proposed by Lamport in [2] with respect to the Temporal Logic of Actions TLA which is a compact linear-time logic for the reasoning on state-transition systems. He considers the two commonly known classes of properties, invariance and eventuality. Moreover, the correctness of design refinements can be proven with respect to the preservation of properties.

An invariance property P is expressed by a formula "$\blacksquare P$" where P is a predicate logic formula describing a set of execution states. Inter alia P may specify following typical correctness conditions of a system:

1. Partial correctness: P is an implication of the form "system terminated $\Rightarrow$ correct results computed".
2. Deadlock freedom: P applies to a set of states, the system is not deadlocked.
3. Mutual exclusion: P asserts that at most one process is in a critical section.

By means of auxiliary history variables all interesting safety properties of a system can be expressed as invariance properties (*see* Safety Property).

The formal proof of invariance properties is supported by a proof rule applying induction on the course of system execution steps. At first, one proves that each initial state implies P. Furthermore, each transition class of the system has to be considered. Each transition has to transform states where P is true into successor states where P is true again.

Eventuality properties assert that some events will eventually happen during each execution of a system. The following typical properties can be easily expressed in temporal logic:

1. Termination: A formula of the form "$\blacklozenge$ terminated" can assert that each execution leads to a state where the system is terminated.
2. Live service: Each state representing that a service request is pending will be followed by a state the request is served: "$\blacksquare$ (requested $\Rightarrow \blacklozenge$ served)".
3. Fair message transfer: If a message is sent often enough over a loose channel, then it is eventually delivered: "$(\blacksquare \blacklozenge$ sent$) \Rightarrow (\blacklozenge$ delivered)".

Eventuality properties can express the typical liveness requirements of systems (*see* Liveness Property).

The proof of eventuality properties can be reduced to the proof of a series of transitive leads-to properties of the form "$\blacksquare (P \Rightarrow \blacklozenge Q)$". The proof of a single leads-to property is supported by the so-called lattice rule which is based on the existence of a well-founded order. The order asserts that a finite number of execution steps is sufficient to reach a state where Q is true.

Systems can be described by formulas on abstract levels as well as on more implementation-near ones. Thus, specifications, refinement steps of a design, and implementations can be represented. That is of great interest, since valid implications correspond to system refinements which are correct in the usual understanding of system developers. Let the formula *Spec* describe a system *S* on a more abstract level. A formula *Impl* describes a correct refinement of S, if the implication formula "*Impl* $\Rightarrow$ *Spec*" is provable.

References

[1] E.M. Clarke, E.A. Emerson, and A.P. Sistla, *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, 8(2): 244-263, 1986

[2] L. Lamport, *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, 16(3):872-923, 1994

[3] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992

[4] A. Pnueli, *The Temporal Logic of Programs*, Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pp. 46-57, 1977

[5] N. Rescher and A. Urquhart, *Temporal Logic*, Springer-Verlag, 1971

Cross Reference:

CTL *see* Temporal Logic

Formal Specification *see* Temporal Logic

Formal Verification *see* Temporal Logic

TLA *see* Temporal Logic

Dictionary Terms:

Concurrency Model

Model representing the global dynamics of a system which consists of concurrently acting components. Mainly, there are two types of concurrency models. Interleaving models induce a total temporal ordering of all component actions. Thus, the system is assumed to perform a global sequence of actions and the model reduces concurrency to non-determinism. In contrast, partial-order models represent the temporal independence of concurrent events directly. Concurrent events are not comparable with respect to the order of events.

Liveness Property

Property of a system concerning its dynamics and expressing that the system will eventually show a particular behavior within a finite period of time. Together with safety properties (*see* Safety Property) liveness properties can be used to characterize the principal functionality of distributed systems.

Safety Property

Property of a system concerning its dynamics and expressing that the system behavior never injures particular conditions, e.g., never enters forbidden states. Together with liveness properties (*see* Liveness Property) safety properties can be used to characterize the principal functionality of distributed systems.