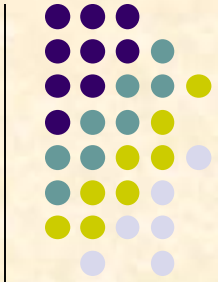
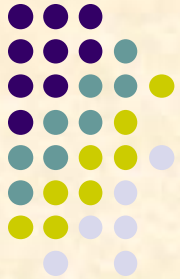


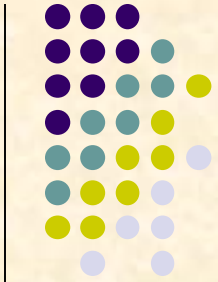
了解各种（非众核）并行程序设计思想

Programmable Memory Hierarchy



- Registers
- Cache
- Physical memory
- Virtual memory
- Local disk storage
- RAID (via LAN) / Distributed shared-memory
- Data Grid
- Web 3.0

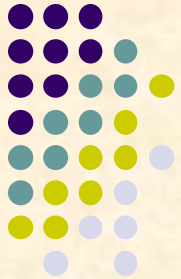
Ideas of parallel computation



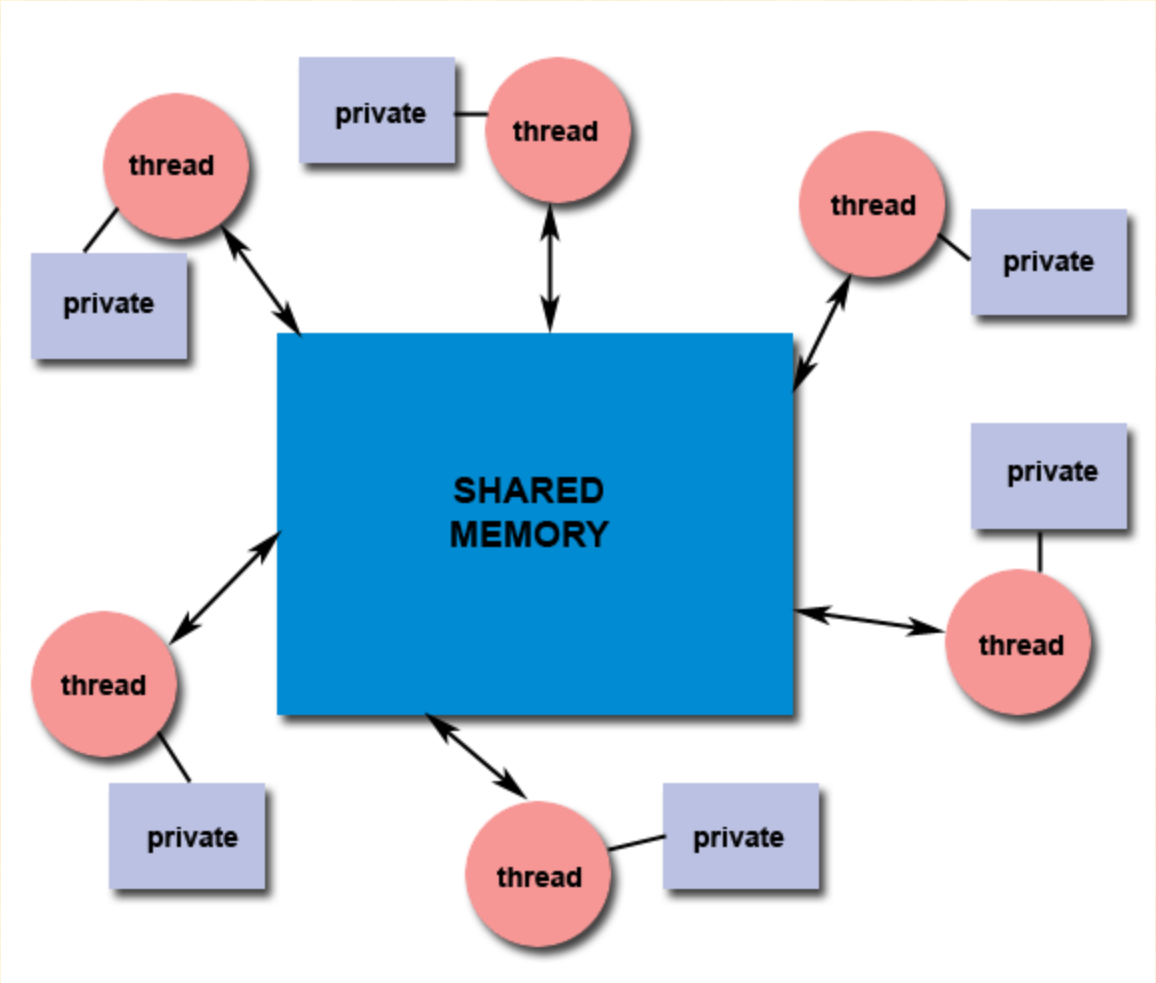
- Multi-threading (e.g. Java)
- Concurrent processes (e.g. Unix)
- SIMD (e.g. Connection Machine-2)
- SPMD (e.g. MPI)
- MIMD (e.g. CORBA)
- Message-passing
- Memory-sharing
- PGAS
- Bulk-Synchronous Parallelism
- Work-flow task parallelism, data parallelism

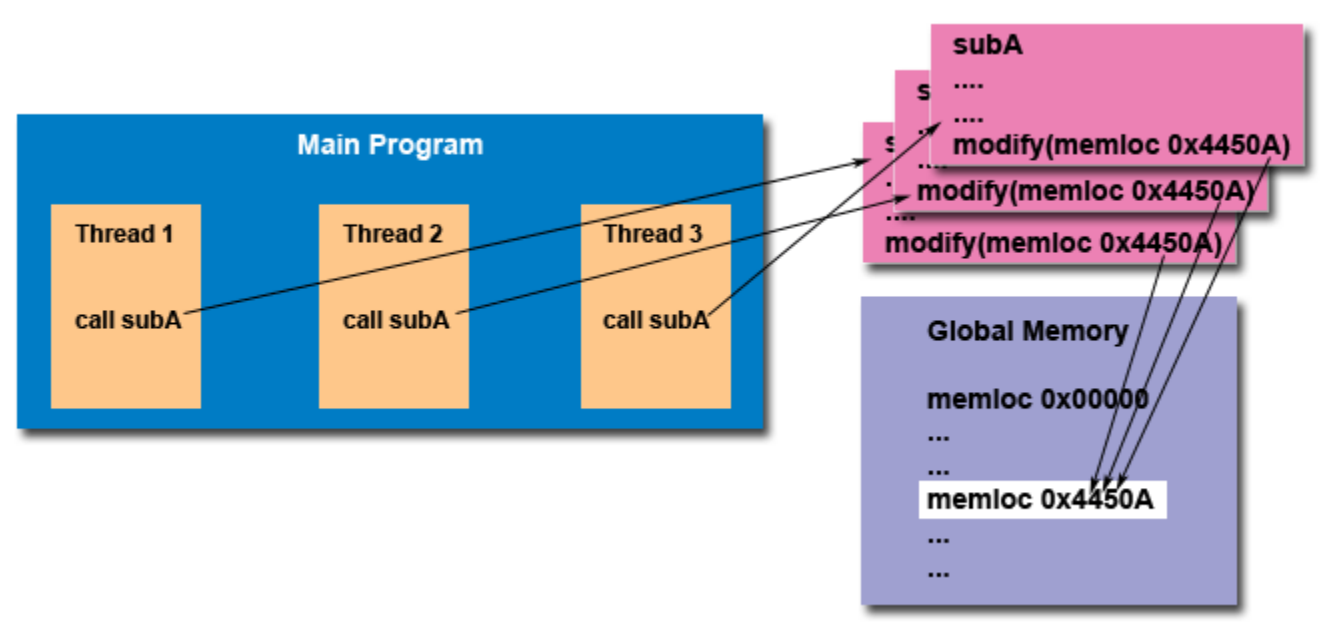
IB/verbs 编程示例

```
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->maxpostsofqpinit
      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;
    }
```



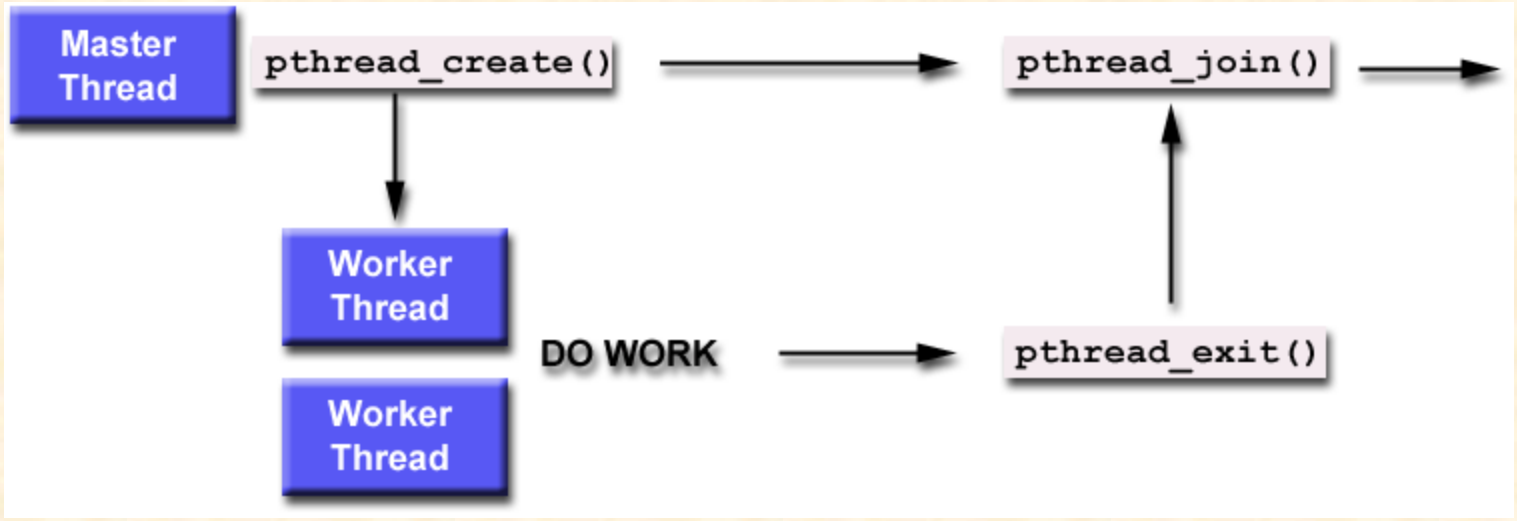
PTHREAD


private
thread
thread
private
thread
private
SHARED
MEMORY
private
thread
thread
private
thread
private

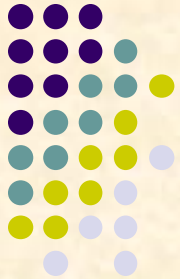

Main Program
Thread 1
call subA
Thread 2
call subA
Thread 3
call subA
subA
....
....
modify(memloc 0x4450A)
modify(memloc 0x4450A)
modify(memloc 0x4450A)
Global Memory
memloc 0x00000
...
...
memloc 0x4450A
...
...

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**

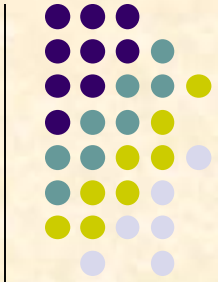
x += 200


Master Thread
pthread_create()
pthread_join()
Worker Thread
DO WORK
pthread_exit()
Worker Thread

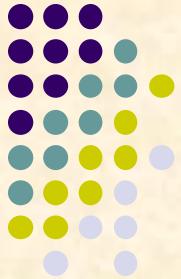
- **pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (start_routine)(void*), void *arg)**
- **pthread_exit (void * retval)**
- **pthread_join (pthread_t thread, void** threadreturn)**
- **sem_init(sem_t *sem, int pshared, unsigned int val);**
- **sem_wait(sem_t *sem);**
- **sem_post(sem_t *sem);**



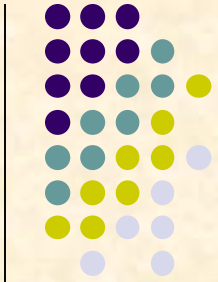
```
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
#include <semaphore.h>
void handler ( void *ptr );
sem_t mutex;
int counter;
int main(){
    int i[2];  i[0] = 0; i[1] = 1;
    pthread_t  thread_a, thread_b;
    sem_init(&mutex, 0, 1);
    pthread_create (&thread_a, NULL, (void *) &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);
    exit(0);
}
```



```
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: Counter Value: %d\n", x, ++counter);
/* END CRITICAL REGION */

    sem_post(&mutex);      /* up semaphore */
    pthread_exit(0); /* exit thread */
}
```



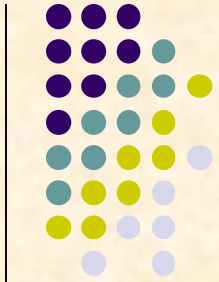
OpenMP



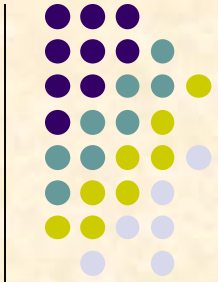
```
#include<omp.h>
int main() {
    #pragma omp parallel
        for(int i=0; i<10; i++)
            printf("i = %d\n", i);
    #pragma omp parallel for
        for(int i=0; i<10; i++)
            printf("i = %d\n", i);
}
```

MPI

A Typical MPI Program

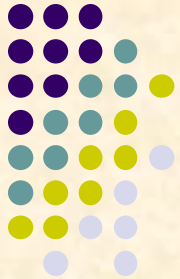


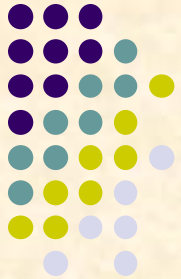
```
#include "mpi.h"
          :
main(int argc, char** argv) {
          :
    /* No MPI functions called before this */
    MPI_Init(&argc, &argv);
          :
    MPI_Finalize();
    /* No MPI functions called after this */
          :
} /* main */
```



```
#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

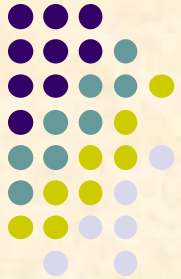
```
Greetings from process 1!
```

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

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

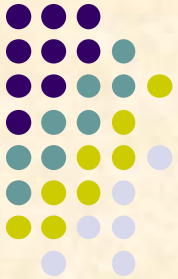
MPI_Send and MPI_Receive

```
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	

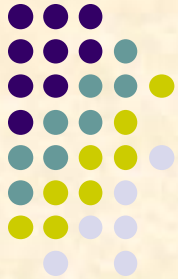
(Explicit) Synchronisation



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

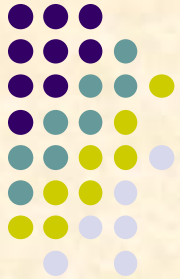
MPI_Barrier provides a mechanism for synchronizing all the processes in the communicator comm. Each process blocks (i.e., pauses) until every process in comm has called MPI_Barrier.

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

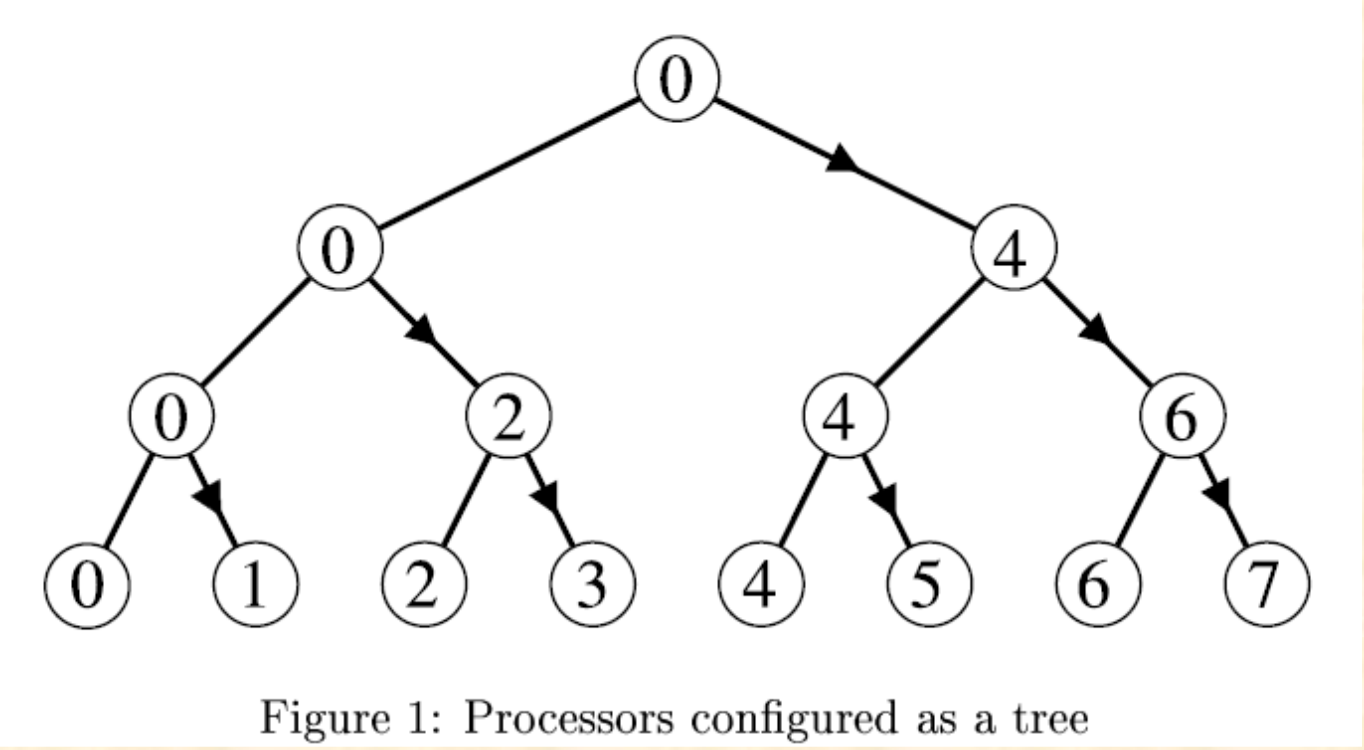
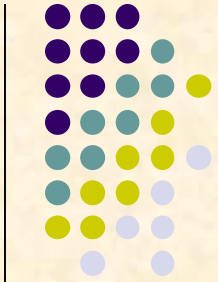
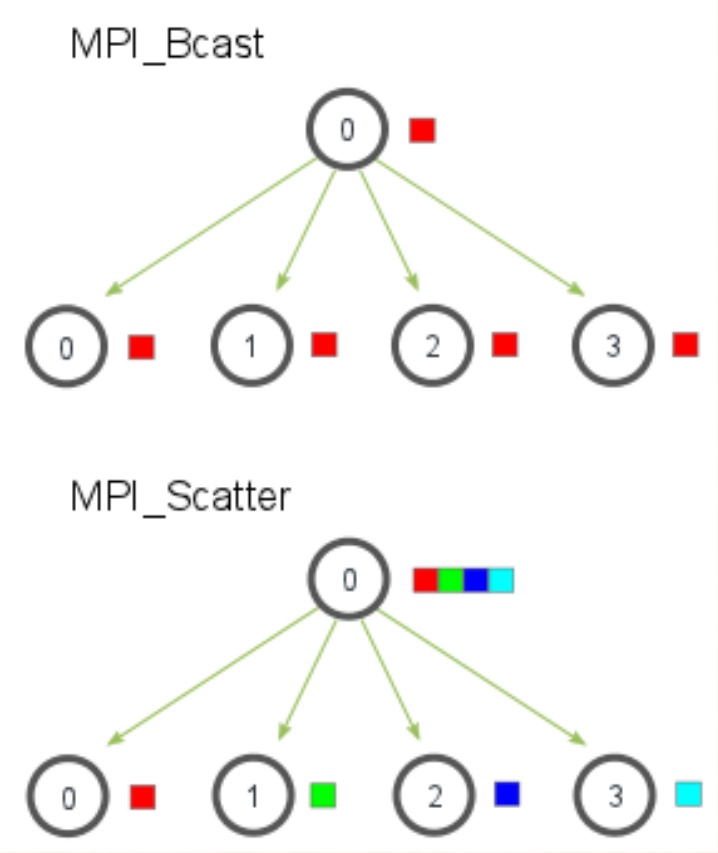


Figure 1: Processors configured as a tree

Yet Another Piece of Advice

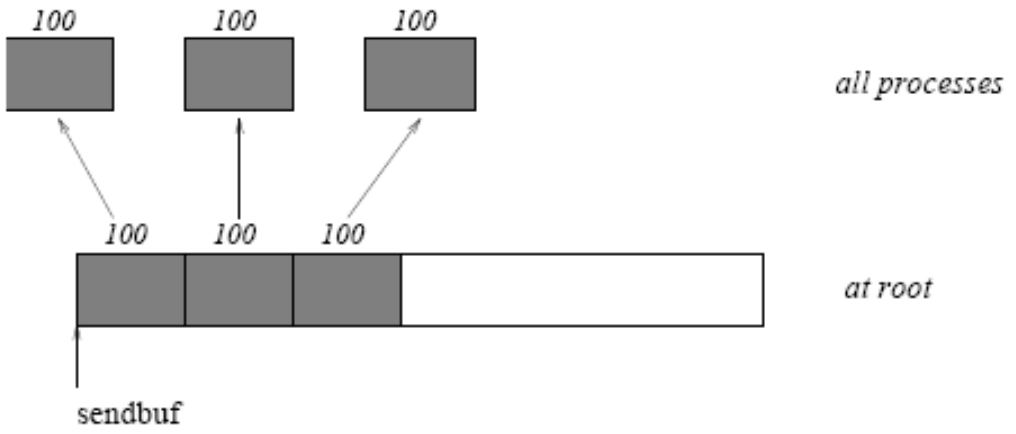


Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors.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


MPI_Bcast
0
0
1
2
3
MPI_Scatter
0
0
1
2
3

Scattering

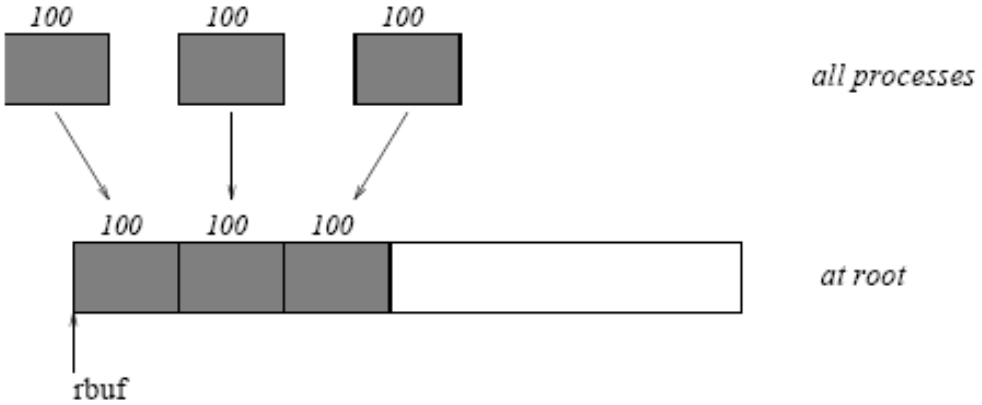
```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)
```



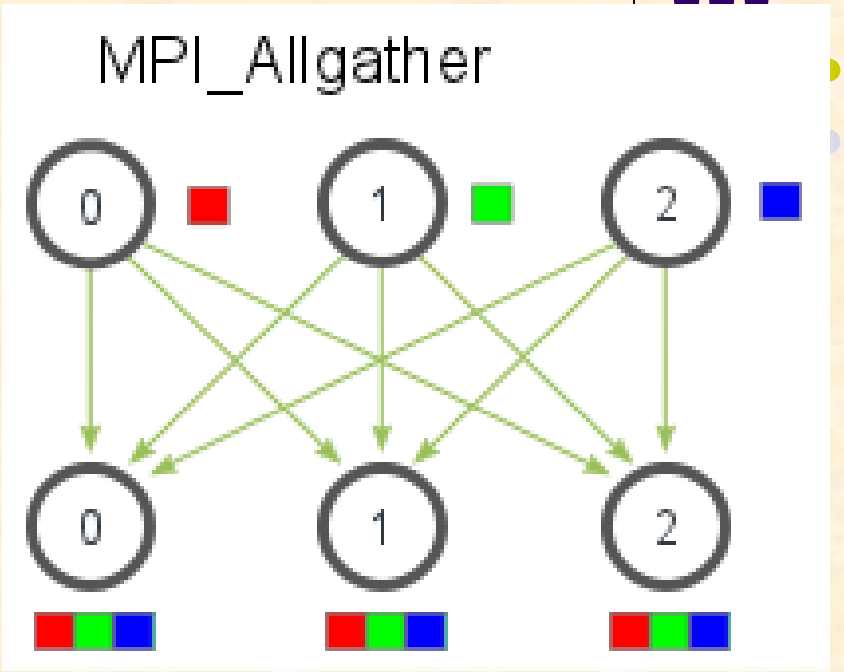
The root process scatters sets of 100 ints to each process in the group.

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_Allltoall

Data

Processor	b[0]	b[1]	b[2]	b[3]	m
0					20
1					22
2					24
3					26

MPI_Allgather

Data

Processor	b[0]	b[1]	b[2]	b[3]	m
0	20	22	24	26	20
1	20	22	24	26	22
2	20	22	24	26	24
3	20	22	24	26	26

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

Data

Processor	a[0]	a[1]	a[2]	a[3]
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

MPI_Alltoall

Data

Processor	b[0]	b[1]	b[2]	b[3]
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

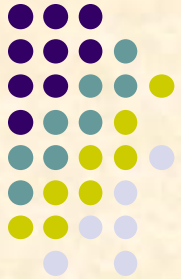
```
MPI_Alltoall(a,1,MPI_INT,b,1,MPI_INT,MPI_COMM_WORLD);
```

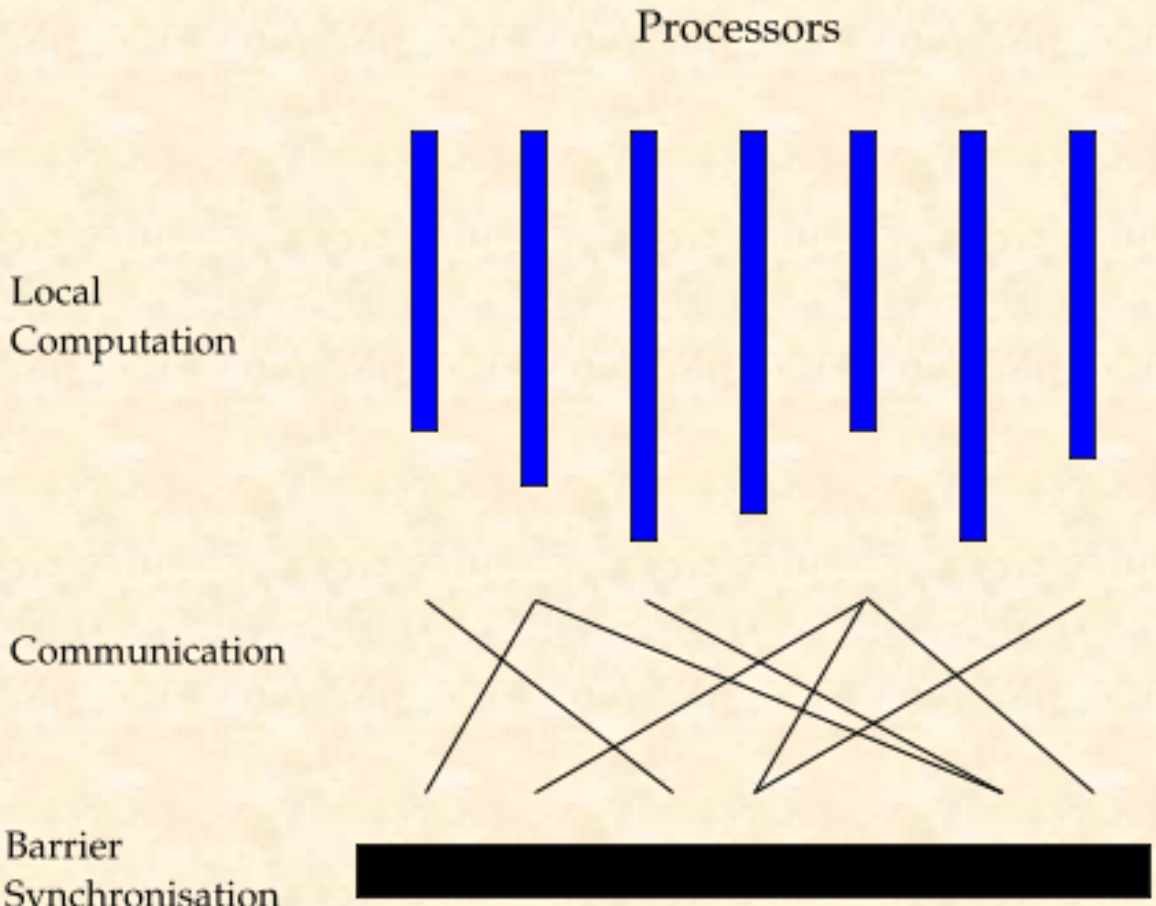
MPI 实现 alltoall.c

```
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

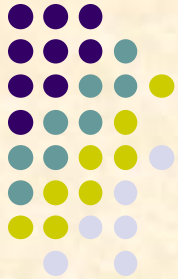
```
MPI_Group MPI_GROUP_WORLD;
MPI_Group first_row_group;
MPI_Comm first_row_comm;
int row_size;
int* process_ranks;

/* 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);
```




Processors
Local
Computation
Communication
Barrier
Synchronisation

GPU集群LINPACK

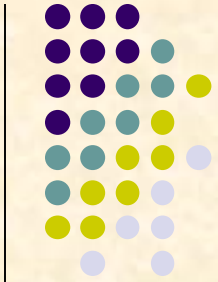
LINPACK (wiki)



- 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

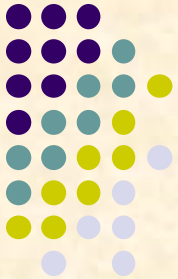
Hardware Configuration



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

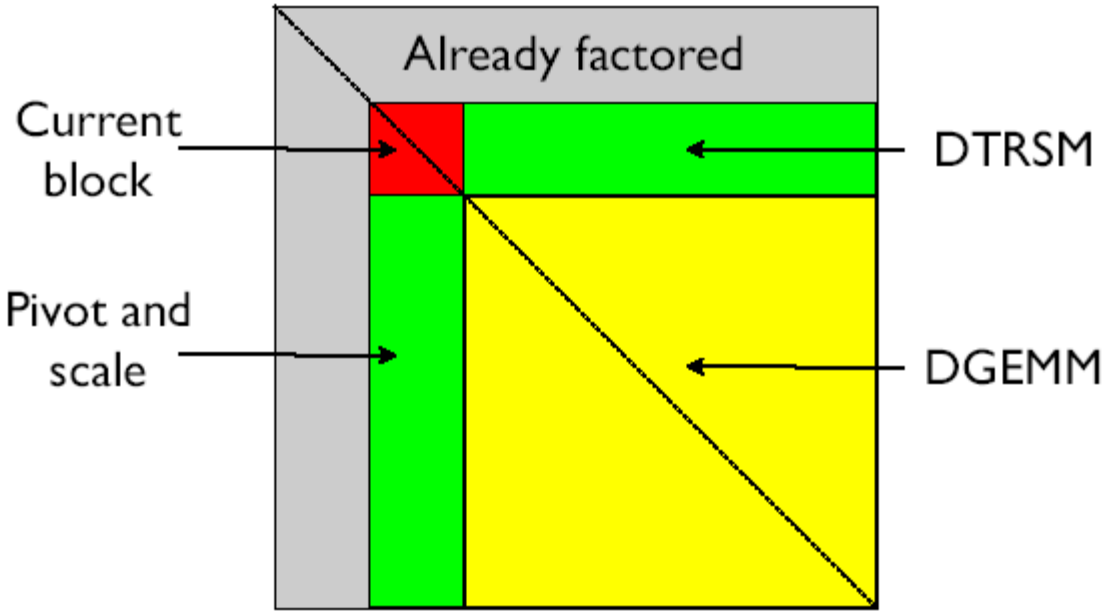


Figure 1: LU factorization: the grey area represents the portion of the matrix already factored. The red area is the current block being factorized. Once this factorization is ready, it is applied to the sub-matrix on the right. The final step is to update the trailing sub-matrix in yellow.

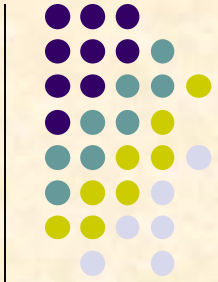
LU Decomposition

$$L\,U = A.$$

$$A\,\mathbf{x} = (L\,U)\,\mathbf{x} = L\,(U\,\mathbf{x}) = \mathbf{b},$$

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

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



first solve $\mathbf{L}\,\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$. 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 $\mathbf{U}\,\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. This can be done by back substitution

$$x_N = \frac{y_N}{u_{NN}}$$

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

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

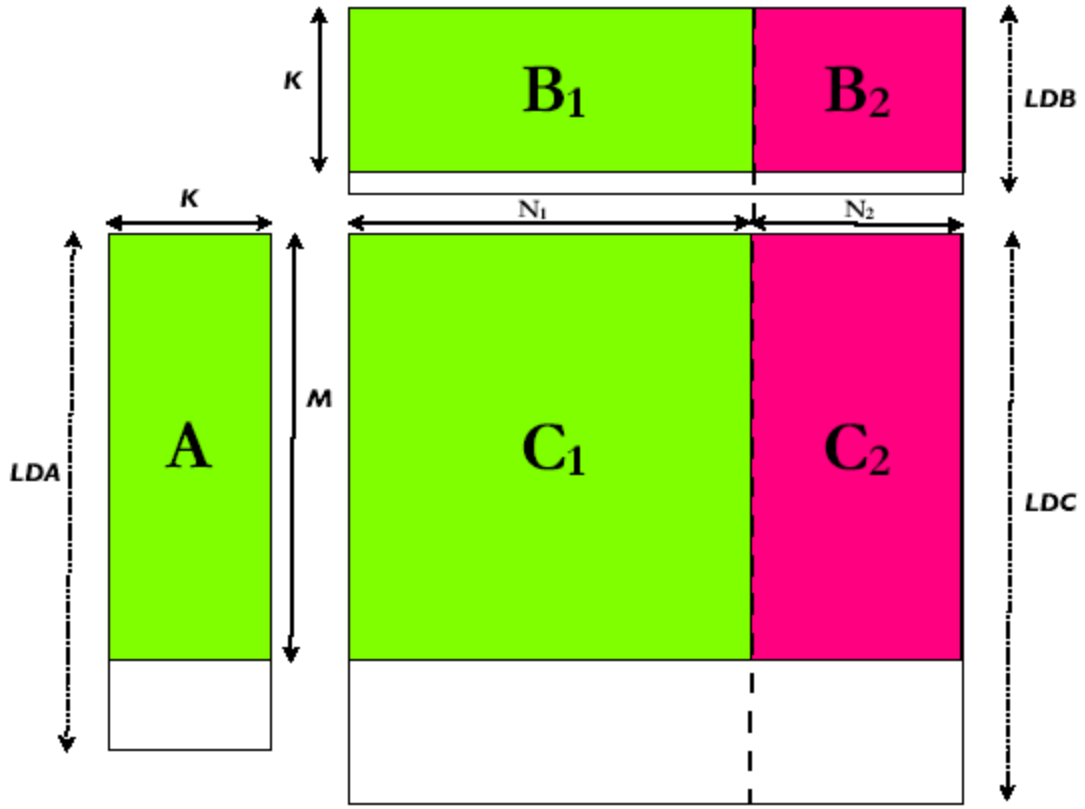


Figure 3: The green portion is performed on the GPU, while the red one is performed on the CPU)

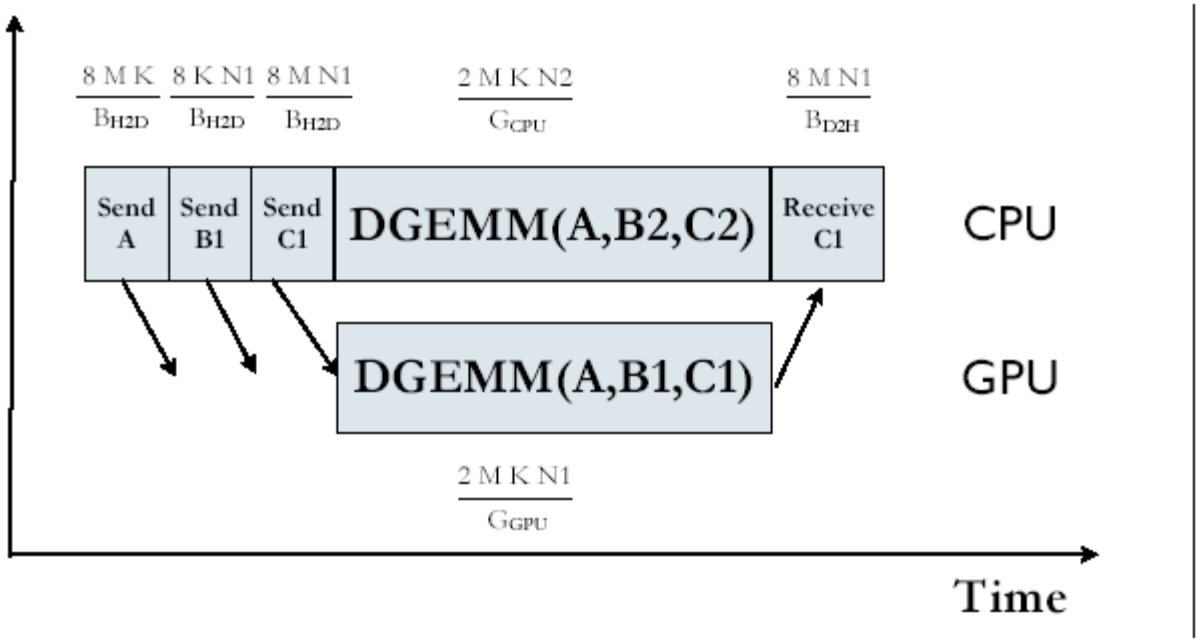


Figure 4: Data flow to split the DGEMM call between CPU cores and GPU.

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:

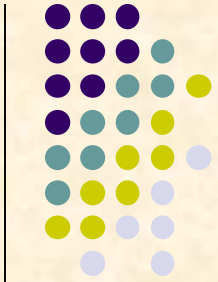
$$T_{CPU}(M,K,N) = 2\frac{MKN}{G_{CPU}}$$

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

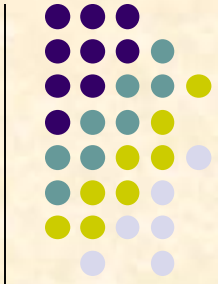
$$T_{CPU}(M,K,N2) = T_{GPU}(M,K,N1) \quad \text{with} \quad N = N1+N2$$



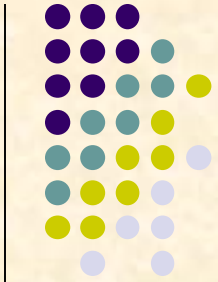
For an initial approximation of the optimal split fraction $\eta = N1/N$, we can omit the transfer time ($O(N^2)$) compared to the computation ($O(N^3)$). From a simple manipulation, the optimal split is

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

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);
```

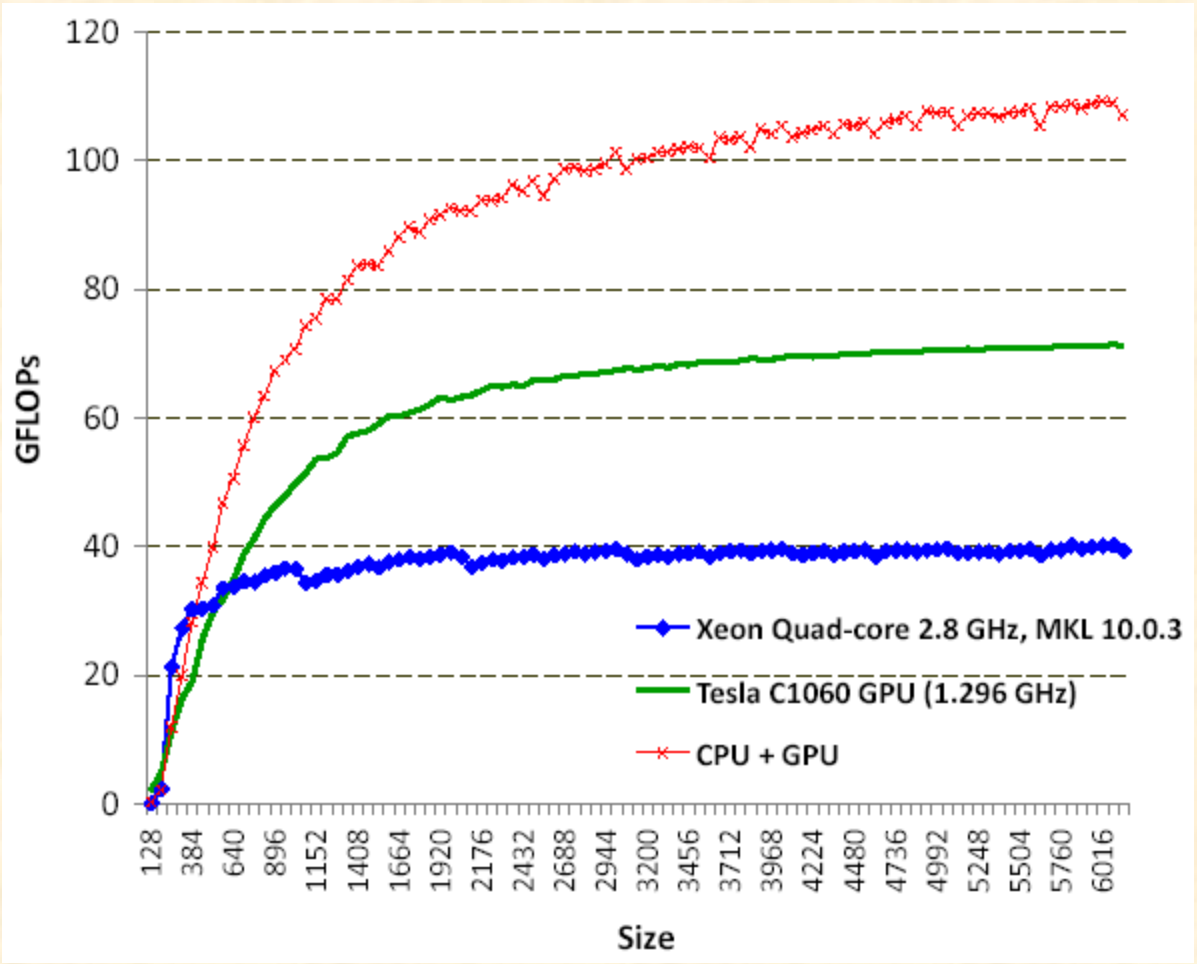


It turns out that on Intel systems using Front Side Bus (FSB), it is better not to overlap the transfer to the GPU with computations on the CPU (the memory system cannot supply data to both the PCIe and the CPU at maximum speed).

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


120
100
80
60
40
20
0
GFLOPs
128
384
640
896
1152
1408
1664
1920
2176
2432
2688
2944
3200
3456
3712
3968
4224
4480
4736
4992
5248
5504
5760
6016
Size
Xeon Quad-core 2.8 GHz, MKL 10.0.3
Tesla C1060 GPU (1.296 GHz)
CPU + GPU