HPC – Algorithms and Applications

Fundamentals – Parallel Architectures, Models, and Languages

Michael Bader
Winter 2014/2015
Part I

Parallel Architectures

(sorry, not everywhere the latest ones ... )
Multicore CPUs – Intel’s Nehalem Architecture

- example: quad-core CPU with shared and private caches
- simultaneous multithreading: 8 threads on 4 cores
- memory architecture: Quick Path Interconnect (replaced Front Side Bus)
Multicore CPUs – Intel’s Nehalem Architecture (2)

- NUMA (non-uniform memory access) architecture: CPUs have “private” memory, but uniform access to remote memory
- max. 25 GB/s bandwidth

Intel® QuickPath Technology

(source: Intel – Nehalem Whitepaper)
Manycore CPU – Intel “Many Integrated Core”

Intel® MIC Architecture: 
An Intel Co-Processor Architecture

Many cores and many, many more threads
Standard IA programming and memory model

(source: Intel/K. Skaugen – SC’10 keynote presentation)
Manycore CPU – Intel Xeon Phi Coprocessor

- coprocessor = works as an extension card on the PCI bus
- ≈ 60 cores, 4 hardware threads per core
- simpler architecture for each core, but
- wider vector computing unit (8 double-precision floats)
- next generation (Knights Landing) announced to be available as standalone CPU
GPGPU – NVIDIA Fermi

Hardware Execution
CUDA’s hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

An Overview of the Fermi Architecture
The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 SMs of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

Fermi’s 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

(source: NVIDIA – Fermi Whitepaper)
GPGPU – NVIDIA Fermi (2)

(source: NVIDIA – Fermi Whitepaper)
GPGPU – NVIDIA Fermi (3)

General Purpose Graphics Processing Unit:

- 512 CUDA cores
- improved double precision performance
- shared vs. global memory
- new: L1 und L2 cache (768 KB)
- trend from GPU towards CPU?

Memory Subsystem Innovations
NVIDIA Parallel DataCache™ with Configurable L1 and Unified L2 Cache

Working with hundreds of GPU computing applications from various industries, we learned that while Shared memory benefits many problems, it is not appropriate for all problems. Some algorithms map naturally to Shared memory, others require a cache, while others require a combination of both. The optimal memory hierarchy should offer the benefits of both Shared memory and cache, and allow the programmer a choice over its partitioning. The Fermi memory hierarchy adapts to both types of program behavior.

Adding a true cache hierarchy for load / store operations presented significant challenges. Traditional GPU architectures support a read-only “load” path for texture operations and a write-only “export” path for pixel data output. However, this approach is poorly suited to executing general purpose C or C++ thread programs that expect reads and writes to be ordered. As one example: spilling a register operand to memory and then reading it back creates a read after write hazard; if the read and write paths are separate, it may be necessary to explicitly flush the entire write / “export” path before it is safe to issue the read, and any caches on the read path would not be coherent with respect to the write data.

The Fermi architecture addresses this challenge by implementing a single unified memory request path for loads and stores, with an L1 cache per SM multiprocessor and unified L2 cache that services all operations (load, store and texture). The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations. The 64 KB memory can be configured as either 48 KB of Shared memory with 16 KB of L1 cache, or 16 KB of Shared memory with 48 KB of L1 cache. When configured with 48 KB of shared memory, programs that make extensive use of shared memory (such as electrodynamic simulations) can perform up to three times faster. For programs whose memory accesses are not known beforehand, the 48 KB L1 cache configuration offers greatly improved performance over direct access to DRAM.
Future Parallel Computing Architectures?

Not exactly sure how the hardware will look like . . .
(CPU-style, GPU-style, something new?)

However: **massively parallel programming** required

- revival of vector computing
  → several/many FPUs performing the same operation
- hybrid/heterogenous architectures
  → different kind of cores; dedicated accelerator hardware
- different access to memory
  → cache and cache coherency
  → small amount of memory per core
- new restrictions → power efficiency, heat management, . . .
Part II

Parallel Models
The PRAM Model(s)

Concurrent or Exclusive Read/Write Access:

- **EREW** exclusive read, exclusive write
- **CREW** concurrent read, exclusive write
- **ERCW** exclusive read, concurrent write
- **CRCW** concurrent read, concurrent write
Exclusive/Concurrent Read and Write Access

exclusive read

X1  X2  X3  X4  X5  X6

concurrent read

X  Y

exclusive write

X1  X2  X3  X4  X5  X6

concurrent write

X  Y
Example: Minimum Search on the PRAM

“Binary Fan-In”:

```
4  7  3  9  5  6  10  8
4  3  5  8
3  5
3
```
Minimum on the PRAM – Implementation

MinimumPRAM( L:Array[1..n]) : Integer {

! n assumed to be 2^k
! Model: EREW PRAM

for i from 0 to k−1 do {
    for j from 1 to n by 2^(i+1) do in parallel
        if L[j+2^i] < L[j]
            then L[j] := L[j+2^i];
    end if;
}

return L[1];

Complexity: $T(n) = \Theta(\log n)$ on $\frac{n}{2}$ processors
Lockstep Execution of parallel for:

- Parallel for-loops (i.e., with extension in parallel) are executed “in lockstep”.
- Any instruction in a parallel for-loop is executed at the same time (and “in sync”) by all involved processors.
- If an instruction consists of several substeps, all substeps are executed in sync.
- If an if-then-else statement appears in a parallel for-loop, all processors first evaluate the comparison at the same time. Then, all processors on which the condition evaluates as true execute the then branch. Finally, all processors on which the condition evaluates to false execute the else branch.

Lockstep Example:

```plaintext
for i from 1 to n do in parallel {
    if U[i] > 0
    then F[i] := (U[i]−U[i−1]) / dx
    else F[i] := (U[i+1]−U[i]) / dx
    end if
}
```

- First, all processors perform the comparison U[i]>0
- All processors where U[i]>0 then compute F[i]; note that first all processors read U[i] and then all processors read U[i−1] (substeps!); hence, there is no concurrent read access!
- Afterwards, the else-part is executed in the same manner by all processors with U[i]≤0
Parallel External Memory – Memory Scheme

[Arge, Goodrich, Nelson, Sitchinava, 2008]
Parallel External Memory – History

Extension of the classical I/O model:
- large, global memory (main memory, hard disk, etc.)
- CPU can only access smaller working memory (cache, main memory, etc.) of $M$ words each
- both organised as cache lines of size $L$ words
- algorithmic complexity determined by memory transfers

Extension of the PRAM:
- multiple CPUs access global shared memory (but locally distributed)
- EREW, CREW, CRCW classification (for local and external memory)
- similar programming model (sychronised execution, e.g.)
Parallel External Memory – Comments

Load and Stores to/from the Cache:

- “out of core” style: assume that it is possible to explicitly control to load variables into cache and (not) to evict them from cache → algorithm can specify cache behavior
- “cache oblivious” style: assume that the cache is intelligent and can even look into the future – variables will only be evicted, if they are no longer used; if the cache needs to evict variables for capacity reasons, it will evict those variables that lead to the fewest overall loads & stores

Memory Complexity in Parallel External Memory Model:

- count the number of cache line transfers between memory and cache
- underlying assumption: memory access determines execution time
- thus: prerequisite for applying the roofline model
Compute-Bound vs. Memory-Bound Performance

Consider a memory-bandwidth intensive algorithm:

- you can do a lot more flops than can be read from memory
- **computational intensity** of a code:
  number of performed flops per accessed byte

**Memory-Bound Performance:**

- computational intensity smaller than critical ratio
- you could execute additional flops "for free"
- speedup only possible by reducing memory accesses

**Compute-Bound Performance:**

- enough computational work to "hide" memory latency
- speedup only possible by reducing operations
The Roofline Model

[Williams, Waterman, Patterson, 2008]
Roofline Model – Comments

Drawing the Roofline Model:

- bandwidth phase: if \( a \) is the arithmetic intensity (Flops per byte), then a memory throughput of \( b \) GB/s leads to \( ba \) GFlop/s
  we use a log-log plot: \( \log(ba) = \log b + \log a = \log b + x \) (if \( x := \log x \)); hence, in the log-log-plot the bandwidth line has unit gradient
- if the arithmetic intensity is 1 and the memory throughput is \( b \) GB/s, the CPU executes \( b \) GFlop/s \( \Rightarrow \) at \( a = 1 \) the bandwidth-part of the roofline is equal to the bandwidth

Calculating the Arithmetic Intensity:

- the roofline model typically only counts accesses to main memory; accesses to cache are ignored
- thus: improving cache use can increase the arithmetic intensity of a kernel
- roofline model can be adapted to consider specific cache level instead of main memory

“Ceilings for Performance”:

- reduced memory bandwidth due to non-optimal access pattern (or similar) \( \Rightarrow \) lower available memory bandwidth for your kernel
- code cannot exploit instruction-level parallelism, vectorization, fused-multiply-add instructions, etc. \( \Rightarrow \) lower available peak performance
Interconnection Networks

- multiple CPUs with private memory
- CPUs connected via interconnection network
- new: topology of the network explicitly considered

Example: 1D mesh (linear array) of processors:
2D Processor Mesh (Array)

Problem: Broadcast
- information transported by at most 1 processor per step
- phase 1: propagate along first line
- phase 2: propagate along columns
Broadcast on the 2D Mesh – Implementation

Broadcast_2Dmesh (P[1,1]:X, n) {

! Model: 2D mesh p[i,j] with n×n processors

input: P[1,1]:X ! element to be broadcasted

for j from 1 to n−1 do
    P[1,j +1]:X <<< P[1,j]:X

for i from 1 to n−1 do
    for P[i , j ]: 1≤j≤n do in parallel
        P[i+1,j ]: X <<< P[i,j]:X
    end in parallel

! value of X is now available on each processor:

output: P[i , j ]: X, range 1≤ i,j ≤ n
}

Time complexity: 2n – 2 steps on an n×n mesh of processors
Interconnection-Network Programs
(Conventions for Execution)

Notation for variables:
- $P[i,j]$: $X$ stresses that $X$ is a **local** variable on processor $P[i,j]$
  - each processor has a distinct $X$ (exactly one element of the array)
  - array is stored in a distributed fashion on all processors
- in comparison to PRAM: here $X$ would need to be an array $X[1..n]$

Input/Output Configurations:
- **output**: $P[i,j]$: $X$, **range** $1 \leq i,j \leq n$ means that for the given range of processor indices the processor-local variables $X$ contain the (distributed) output
- **input** $P[i,k]$: $A,B,C$, **range** $P[i,k]$: $1 \leq i,k \leq p$ (compare matrix multiplication example) means that the matrices $A,B,C$ are stored distributed on the given range of processors; each processor holds only one element (or block) of $A,B,C$ in its local memory

Explicit Data Transfer (Send/Receive):
- $P[i+1,j]$: $X$ $\ll\ll$ $P[i,j]$: $X$ means that the content of the local variable $X$ on processor $P[i,j]$ is copied (transferred) into the local variable $X$ on processor $P[i+1,j]$
  - in an MPI program, this might correspond to a pair of send and receive calls
Bulk Synchronous Parallelism

- suggested as a “bridging model” between software and hardware aspects
- hardware model:
  - multiple CPUs with private memory
  - CPUs connected via point-to-point network
Bulk Synchronous Parallelism – Execution Model

Computation organised into sequence of “Super Steps”:

1. each CPU executes a sequence of operations (on local data, synchronised from the last super step)
2. CPUs send and receive point-to-point messages (no broadcasts or send/receive to/by multiple CPUs allowed)
3. synchronisation at a barrier

Goal:

- estimate time for steps 1, 2, 3 based on CPU speed $\gamma$, bandwidth $\beta$, and latency $\lambda$
- estimated execution time thus: $T = s(n \cdot \gamma + m \cdot \beta + \lambda)$ for $s$ steps with $n$ operations and $m$ sent/received bytes
Summary: Parallel Algorithmic Models

Each model highlights specific properties of the hardware:

- limits to parallelism (number of processors) → PRAM
- influence of (parallel) caches → Parallel External Memory
- compute- or memory-bound performance → Roofline
- transfer of messages → Interconnection Networks
- computation vs. communication time → BSP

“All models are wrong – some of them are useful”

- more than one model required to analyze the performance of parallel algorithm on real machines
- pick the model that fits best to an anticipated bottleneck
Performance Evaluation – Speed-Up

Definition:
- $T(p)$: runtime on $p$ processors
- *speed-up* $S(p)$ quantifies the improvement factor in processing speed:

$$S(p) := \frac{T(1)}{T(p)}, \text{ typically: } 1 \leq S(p) \leq p$$

Absolute vs. Relative Speed-Up:
- *absolute speed-up*: best sequential algorithm for the mono-processor system is compared to the best parallel algorithm for the multi-processor system
- *relative speed-up*: compare the same (parallel) algorithm on mono- and multi-processor system
Parallel Efficiency

**Definition:**
- efficiency $E(p)$ relates speed-up $S(p)$ to the number of processors $p$:
  \[ E(p) := \frac{S(p)}{p} \]
- indicates the relative improvement in processing speed
- typically: $0 \leq E(p) \leq 1$
- again: *absolute* vs. *relative* efficiency
Scalability

- reduction of execution time, if the number of processors is increased
- quantitative: speed-up or parallel efficiency
- strong scalability: increase number of processors for fixed problem size
- weak scalability: increase number of processors and increase problem size
- qualitative: is there an improvement at all?

“If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?”

(Seymour Cray)
Amdahl’s Law

Assumptions:

- program consists of a sequential part $s$, $0 \leq s \leq 1$, which can not be parallelised (synchronisation, data I/O, etc.)
- parallelisable part, $1 - s$, can be perfectly parallelised (perfect speed-up on arbitrary number of processors)
- execution time for the parallel program on $p$ processors:

$$T(p) = s \cdot T(1) + \frac{1 - s}{p} \cdot T(1)$$
Amdahl’s Law (2)

- resulting speed-up:

\[
S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{s \cdot T(1) + \frac{1-s}{p} \cdot T(1)} = \frac{1}{s + \frac{1-s}{p}}
\]

- consider increasing number of processors:

\[
\lim_{p \to \infty} S(p) = \lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s}
\]

- Amdahl’s law: speed-up is bounded by \( S(p) \leq \frac{1}{s} \)

- message: any inherently sequential part will destroy scalability once the number of processors becomes big enough
Gustafson’s Law

Assumptions:

- Amdahl: sequential part stays for increased problem size
- Gustavson: assume that any sufficient large problem can be efficiently parallelised
- fixed-time concept:
  - parallel execution time is normalised to $T(p) = 1$
  - this contains a non-parallelisable part $\sigma$, $0 \leq \sigma \leq 1$
- execution time on the mono-processor:

  $$T(1) = \sigma + p \cdot (1 - \sigma)$$

- thus: sequential part of total work gets smaller with increasing $p$
Gustafson’s Law (2)

- resulting speed-up (as $T(p) = 1$):

$$S(p) = \sigma + p \cdot (1 - \sigma) = p - \sigma(p - 1)$$

- resulting parallel efficiency:

$$E(p) = \frac{S(p)}{p} = \frac{\sigma}{p} + (1 - \sigma) \rightarrow 1 - \sigma$$

- more realistic: larger problem sizes, if more processors are available; parallelisable parts typically increase
Part III

Parallel Languages
OpenMP

- shared-memory application programming interface (API)
- extends *sequential* programs by directives to help compiler generate parallel code
- available for Fortran or C/C++
- *fork-join-model*: programs will be executed by a team of cooperating threads
- memory is shared between threads, except few private variables
Matrix-Vector Product in OpenMP

```c
void mvp(int m, int n, double* restrict y,
         double** restrict A, double* restrict x)
{
    int i, j;
    #pragma omp parallel for default(none) \ 
        shared(m,n,y,A,x) private(i, j)
    for (i=0; i<n; i++) {
        y[i] = 0.0;
        for (j=0; j<n; j++) {
            y[i] += A[i][j]*x[j];
        }
    } /*-- end of omp parallel for --*/
}
```
OpenMP Directives

OpenMP directives are inserted as \#pragma:

\#pragma omp parallel for default(none) \ shared(m,n,y,A,x) private(i, j )

Advantages:

• directives will be ignored by compilers that do not support OpenMP
• sequential and parallel program in the same code!
• incremental programming approach possible (add parallel code sections as required)
OpenMP’s Memory Model

```c
#pragma omp parallel ... default(none) \shared(m,n,y,A,x) private(i, j)
```

- memory usually shared between threads – here: matrix and vectors
- however, certain variables are private: loop variables (here: indices), temporary variables, etc.
- if not specified, default settings apply – here: `default(none)` to switch off all default settings
- programmer is responsible to sort out concurrent accesses! (even if default settings are used)
Pthreads

- standardised programming interface according to POSIX (Portable Operating System Interface)
- thread model is a generalised version of the UNIX process model (forking of threads on shared address space)
- scheduling of threads to CPU cores done by operating system
Pthreads – Typical Methods

- `pthread_create(...)`: (main) thread creates a further thread that will start by executing a specified function
- `pthread_join(...)`: thread will wait until a specified thread has terminated (useful for synchronisation)
- `pthread_cancel(...)`: cancel another thread
- Functions to synchronise data structures:
  - `mutex`: mutual exclusive access to data structures;
  - `cond`: wait for or signal certain conditions
- Functions to take influence on scheduling
- Etc.
Programming Patterns

Pthreads allow arbitrary coordination of threads; however, certain programming patterns are common, e.g.:

- **Master-Slave** model:
  master thread controls program execution and parallelisation by delegating work to slave threads

- **Worker** model:
  threads are not hierarchically organised, but distribute/organise the operations between themselves (example: jobs retrieved from a work pool)

- **Pipelining** model:
  threads are organised via input/output relations: certain threads provide data for others, etc.
Example: Matrix Multiplication

```c
#include <pthread.h>
typedef struct {
    int size, row, column;
    double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void thread_mult(matrix_type_t *work) {
    int i, row = work->row, col = work->column;
    work->MC[row][col] = 0.0;
    for (i = 0; i < work->size; i++)
        work->MC[row][col] +=
            work->MA[row][i] * work->MB[i][col];
}
```
Example: Matrix Multiplication (cont.)

```c
void main() {
    double MA[8][8], MB[8][8], MC[8][8];
    pthread_t thread[8*8];
    for (int row=0; row<8; row++)
        for (int col=0; col<8; col++) {
            matrix_type_t *work = (matrix_type_t *) malloc( /* ... */ );
            work->size = 8; work->row = row; work->col = col;
            work->MA = MA; work->MB = MB; work->MC = MC;
            pthread_create(&thread[col+8*row], NULL,
                            (void*) thread_mult, (void*) work);
        }
    for (int i=0; i<8*8; i++) pthread_join(thread[i], NULL);
}
```

(example from: Rauber&Rünger: Parallele Programmierung)
Java Threads

- object-oriented language design explicitly includes threads
- class Thread to represent threads – can be extended to implement customised threads (inherits start(), run() methods, etc.)
- interface Runnable:
  classes that implement Runnable, i.e., provide a method run() can be used to create a thread:

  ```java
  Thread th = new Thread(runnableObject);
  ```

- keyword synchronized for methods that should be treated as a critical region
- methods wait() and notify() in class Object
Example: Matrix Multiplication

class MatMult extends Thread {
    static int a [], b [], c [], n=3;
    int row;
    MatMult(int _row) {
        row = _row; this.start();
    }
    public void run() {
        for (int i=0; i<n; i++) {
            c[row][i] = 0.0;
            for (int j=0; j<n; j++)
                c[row][i] = c[row][i] + a[row][j]*b[j][i];
        }
    } /* class MatMult t.b.c. */
Example: Matrix Multiplication (cont.)

```java
public static void main() {
    a = new int[n][n]; b = new int[n][n]; c = new int[n][n];
    /* ... initialise a,b,c ... */
    MatMult mat = new MatMult[n];
    for (int i=0; i < n; i++) mat[i] = new MatMult(i);
    try {
        for (int i=0; i < n; i++) mat[i].join();
    } catch(Exception E) { /* ... */ }
}
```

(cmp. example in Rauber&Rünger: Parallele Programmierung)
Unified Parallel C (UPC)

- extension of C, specified in the ISO C99 standard
- based on a *distributed shared memory* model: physically distributed memory with shared address space
- **PGAS** language: “partitioned global address space”
- single program, multiple data: every program is executed in parallel on specified number of threads
- variables are private by default, but can be declared as shared
- consistency model can be varied: strict vs. relaxed
Example: Matrix Multiplication

Declaration of variables:

```c
shared [N*N/THREADS] int a[N][N];
shared [N/THREADS] int b[N][N];
shared [N*N/THREADS] int c[N][N];
```

Variables have an affinity towards threads:

- *block-cyclic* distribution of variables top threads
- `a` and `c` declared with a block size of `N*N/THREADS` → block-oriented distribution of rows to threads
- `b` declared with a block-size of `N/THREADS` → block-oriented distribution of columns to threads
- Affinity can reflect physical distribution of data
Example: Matrix Multiplication (cont.)

Code excerpt for matrix multiplication:

```upc
forall (i=0; i<N; i++;&a[i][0])
    /* &a[i][0] specifies that iteration will be executed by thread that has affinity to a[i][0] */
    for (j=0; j<N; j++) {
        c[i][j] = 0;
        for (l=0; l<N; l++) c[i][j] += a[i][l]*b[l][j];
    }
    upc_barrier;
```

(source: Rauber & Rünger: Parallele Programmierung)
Further PGAS Languages

- Co-Array Fortran (CAF)
  → will become part of the next Fortran standard
- Titanium (similar to UPC, but for Java)
- X10: extension of Java;
  *globally asynchronous, locally synchronous*: add “places” that execute threads
- Chapel
- Fortress
Further Example: Intel PBB

“Intel Parallel Building Blocks”:
- language extensions & libraries for C/C++
- **Intel Cilk Plus**: language extension for simple loop & task oriented parallelism for C/C++
- **Intel Threading Building Blocks**: C++ template library to support task parallelism
- **Intel Array Building Blocks**: C++ template library to support vector parallelism
Examples – Intel Cilk Plus

Intel Cilk:

```c
void mergesort(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right)/2;
        cilk_spawn mergesort(a, left, mid);
        mergesort(a, mid, right);
        cilk_sync;
        merge(a, left, mid, right);
    }
}
```

(source: Intel)
Examples – Intel ArBB

Intel ArBB:

```c
void matvec_product(const dense<f32, 2>& matrix,
                    const dense<f32>& vector,
                    dense<f32>& result)
{
    result = add_reduce(matrix
                         * repeat_row(vector, matrix.num_rows()));
}
```

(source: Intel)
Message Passing – MPI

Abstraction of a distributed memory computer

- MPI run consists of a set of processes with separate memory space
- processes can exchange data by sending messages
- no explicit view on network topology required

SIMD/MIMD?? … → MPMD/SPMD

- processes can run different programs (“codes”)
  → *multiple program multiple data* (MPMD)
- more common (and simpler):
  processes run instances of the same program (“code”)
  → *single program multiple data* (SPMD)
MPI Example: “Hi there” . . .

```c
#include "mpi.h"
int main( int argc, char **argv )
{
    int myrank;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)
        send_a_message();
    else if (myrank == 1)
        receive_a_message();
    MPI_Finalize();
}
```
void send_a_message() {
    char message[40];
    strcpy(message, "Mr. Watson, come here, I want you.");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 110, MPI_COMM_WORLD);
}

void receive_a_message() {
    char message[40];
    MPI_Status status;
    MPI_Recv(message, 40, MPI_CHAR, 0, 110, MPI_COMM_WORLD, &status);
    printf ("received: %s:\n", message);
}
References (Languages)