CS3400 - Principles of Software Engineering

Software Engineering for Multicore Systems

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Thanks!
- Quiz I - 30 marks, End Sem - 50 marks, Take home assignments - 20
- There will be two assignments - total 10 marks.
- During the lecture time - you can get additional 5 marks.
- How? - Ask a *good* question, Answer a *chosen* question, Make a good point! Take 0.5 marks each. Max one mark per day per person.
- Plagiarism - A good word to know. A bad act to own.

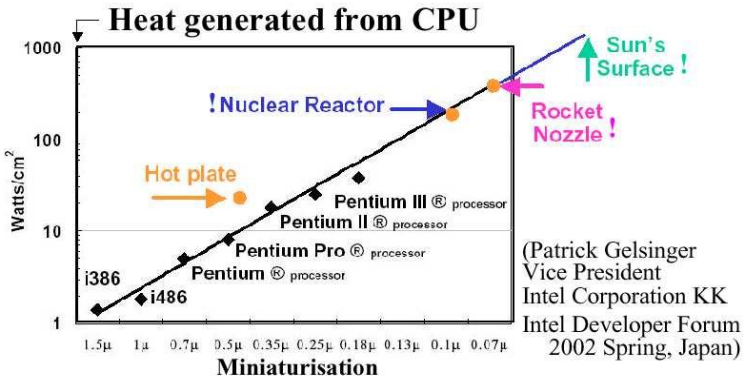
Contact (Anytime) :
Email: nvk@cse.iitm.ac.in, Skype (nvkrishna77), Office: BSB 352.

What, When and Why of Software Engineering

- **What**: Software engineering is a profession dedicated to designing, implementing, and modifying software so that it is of *higher* quality, *more* affordable, maintainable, and *faster* to build.
- **When** Phrase coined in 1968 by NATO.
- **Why Study**
  - Software Crisis.
    - difficulty of writing correct, understandable, and verifiable computer programs.
    - The roots of the software crisis are complexity, expectations, and change.
  - Money Magazine and Salary.com, rated “software engineering” as the best job in the United States in 2006.

Why Multicores?

Power density continues to get worse



(Patrick Gelsinger
Vice President
Intel Corporation KK
Intel Developer Forum
2002 Spring, Japan)

Surpassed hot-plate power density in 0.5µm

Not too long to reach nuclear reactor

Focus on increasing the number of computing cores.

What, When Multicores? **Why not** Multiprocessors

- **What** A multi-core processor is composed of two or more independent cores. Composition involves the interconnect, memory, caches.
- **When** IBM POWER4, the world's first dual-core processor, released in 2001.
- **Why not Multi-processors**
  - An application can be "threaded" across multiple cores, but not across multi-CPUs – communication across multiple CPUs is fairly expensive.
  - Some of the resources can be shared. For example, on Intel Core Duo: L2 cache is shared across cores, thereby reducing further power consumption.
  - Less expensive: A single CPU board with a dual-core CPU Vs a dual board with 2 CPUs.

Challenges Involved

- Harnessing parallelism
  - How to map parallel activities to different cores? How to distribute data?
- Locality: Data and threads
- Minimizing the communication overhead
- Exploring fine grain parallelism (SIMDization), coarse grain parallelism (SPMDization).
- Assist threads
- Dynamic code profiling and optimizations.
- Programmability issues.

Programmability issues

- With hardware becoming increasingly multi-core, software developed without attention to parallel processing capabilities of the hardware will typically under-utilize the hardware - Example?
- When software is designed to operate in a multi-threaded or multi-processed manner, how the threads are mapped to the cores becomes an important issue - Why?
- Software that is critically dependent on multi-threading is always based on assumptions regarding the *thread-safety* of the function calls - Why?
- Multi-threading of software is generally very important to applications that involve human interactivity.
- Understanding different levels of parallelism.

A simple example: thread safety (more details later)

```
function int Withdraw(int amount){
     if (balance > amount) {
           balance = balance - amount;
           return SUCCESS;
     }
     return FAIL;
}
```

- Say `balance` = 100.
- Two parallel threads executing `Withdraw`(80)
- At the end of the execution, it may so happen that both of the withdrawals are successful. Further `balance` can still be 20!

Parallelism types

Instruction level parallelism.

- Parallelism at the machine-instruction level.
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 20 years.

Thread level parallelism.

- This is parallelism on a more coarser scale.
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP. Multicores are the way out to exploit the TLP.

What type of applications benefit from Multi-cores?

- Nearly All !
- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with Thread-level parallelism (as opposed to instruction-level parallelism)
- To build applications that benefit from Multi-cores, we have to understand multi-cores, on how they differ from unicore machines.

Outline

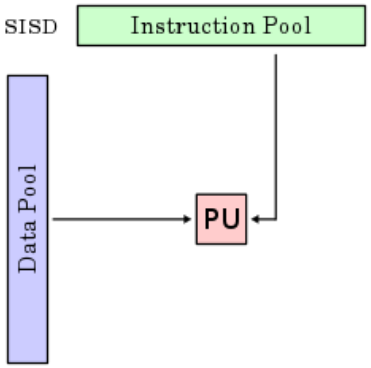
Flynn's Taxonomy.

Categorization of computers based on number of instruction and data streams[1].

- SISD: Single instruction Single Data - x86: sequential computer which exploits no parallelism in instruction or data streams.
- SIMD: Single instruction Multiple Data - Vector machines: A computer which exploits multiple data streams against a single instruction stream.
- MISD: Multiple instruction Single Data - Space Shuttle - Multiple instructions operate on a single data stream.
- MIMD: Multiple instruction Multiple Data - Bluegene, Cell - Multiple autonomous processors simultaneously executing different instructions on different data.

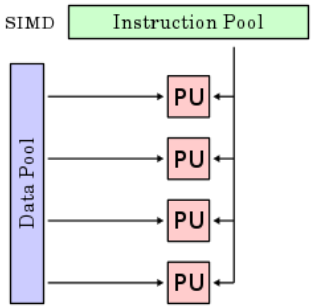
[1] Flynn, M. (1972). "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. C-21: 948.

SISD



- Traditional Von Neumann Architecture, all traditional computations.
- a single processor, a uniprocessor, executes a single instruction stream, to operate on data stored in a single memory.
- Pipelined execution allowed.

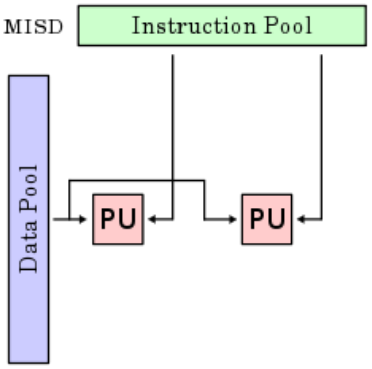
SIMD




```
for (int i=0;i<16;++i) A[i] = B[i] + C[i]
```

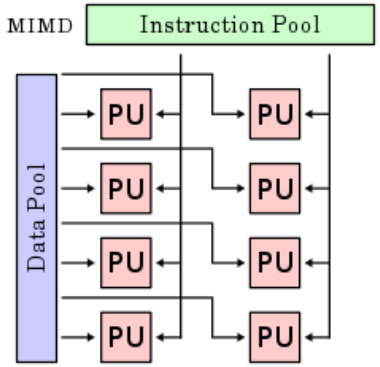
- Fetching / Write a bulk of data is efficient than single units of data.
- A compiler level optimization to generate SIMD instructions.
- Not all algorithm can be vectorized - for instance, parsing.
- increases power consumption and chip area.
- Detecting SIMD patterns is non-trivial.

MISD



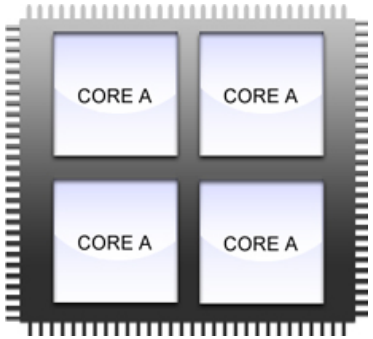
- Task replication for fault tolerance.
- Not used in practise. No known commercial system.

MIMD



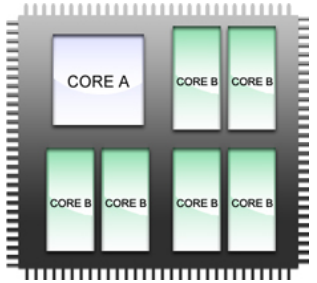
- Many processors that function asynchronously.
- Memory can be shared (less scalable) or distributed (memory consistency issues).
- Most of the modern parallel architectures fall into this category.

Different types of MIMD systems - homogeneous



- Homogeneous multi-core systems include only identical cores.
- Just as with single-processor systems, cores in multi-core systems may implement architectures like superscalar, VLIW, vector processing, SIMD, or multithreading.

Different types of MIMD systems - heterogeneous



- Mixture of different cores e.g.
  - a computational unit could be a general-purpose processor (GPP),
  - a special-purpose processor (i.e. digital signal processor (DSP)
  - a graphics processing unit (GPU)),
  - a co-processor, or custom acceleration logic
- Each core may be optimized for different roles.
- Clusters are often heterogeneous; future supercomputers mostly will be heterogeneous systems. Examples: Grids, lab clusters.
- What are hybrid multicore systems?

Pros and Cons

Homogeneous CPU multi-cores

Pros:

- Easier programming environment
- Easier migration of existing code

Cons:

- Lack of specialization of hardware to different tasks
- Fewer cores per server today (24 in Intels Dunnington and 8 cores / 64 threads in Suns Niagara 2)

Heterogeneous multi-cores

Pros:

- Massive parallelism today
- Specialization of hardware for different tasks.

Cons:

- Developer productivity - requires special training.
- Portability - e.g. software written for GPUs may not run on CPUs.
- Organization - multiple GPUs and CPUs in a grid need their work allocated and balanced and event-based systems need to be supported.

Challenges Involved (revisited)

- Harnessing parallelism
  - How to map parallel activities to different cores? How to distribute data?
- Locality: Data and threads. What is the challenge?
- Minimizing the communication overhead
- Exploring fine grain parallelism (SIMDization), coarse grain parallelism (SPMDization).
- Assist threads
- Dynamic code profiling and optimizations.
- Programmability issues.

Outline

Parallel Computing

Computation is done in parallel to take advantage of a) parallel computing elements, b) waiting time in different computations.

- A program is a collection of interacting processes (logged by the Operating System). Different address space,
- A process can be a collection of one or more threads. Share address space.
- A thread may contain multiple parallel tasks/activities. Even share the stack space.

Context Switching (processes)	$\geq$ CST (thread)	$\geq$ CST (tasks)
State information (processes)	$\geq$ SI (thread)	$\geq$ SI (tasks)

One of the main challenges: Mapping tasks/threads/processes onto hardware threads to improve load balancing.

Synchronous and Asynchronous events

- Synchronous events : One must happen after the other.
- Asynchronous events: Can happen in parallel.

```
int[] mergesort(int[]A,int L,int H){
   if (H - L <= 1) return;
   int m = (L+H)/2;
   A = mergesort(A, L, m);
   A = mergesort(A, m+1, H);
   return merge(A, L,m, m+1, H); }
int[] merge(int[]A,int L1,int H1,int L2,int H2){
  int[]result = ArrayCopy(A);
  int L1=0, L2=0, H1=A1.length-1, H2=A2.length-1;
  while (H1 - L1 > 0 OR H2 - L2 > 0){
    if( H1 - L1 > 0 AND H2 - L2 > 0 ) {
      if (A[L1] <= A[L2]) { result[r++] = A[L1++]; }
      else { result[r++] = A[L2++]; }
    }else if (H1 - L1 > 0) { result[r++] = A[L1++];}
     else if (H2 - L2 > 0) { result[r++] = A[L2++];}
  }
  return result; }
```

Synchronous and Asynchronous events

- Synchronous events : One must happen after the other.
- Asynchronous events: Can happen in parallel.

```
int[] mergesort(int[]A,int L,int H){
   if (H - L <= 1) return;
   int m = (L+H)/2;
   A1 = mergesort(A, L, m);
   A2 = mergesort(A, m+1, H);
   return merge(A1, A2); }
```

```
int[] merge(int[]A1, int []A2){
  int[]result = new int [A1.length + A2.length];
  int L1=0, L2=0, H1=A1.length-1, H2=A2.length-1;
  while (A1.length > 0 OR A2.length > 0){
    if ( A1.length > 0 AND A2.length > 0 ) {
       if (A[L1] <= A[L2]) { result[r++] = A[L1++]; }
       else { result[r++] = A[L2++]; }
    } else if (A1.length > 0) { result[r++] = A[L1++];}
      else if (A2.length > 0) { result[r++] = A[L2++];}
  }
  return result; }
```

Can you parallelize the function merge?

Activity/Thread creation - examples

- MPI: A program when invoked, is executed on multiple execution units.
- Number of threads is decided by the runtime.
-

```
MPI_Init(...);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

if(myid == 0) { // main thread.
   ...
} else { // other threads
  ...
}
MPI_Finalize();
```

Activity/Thread creation - examples

X10/HJ:

- Activity creation.

```
S0;
async  {
   S1
}
async  {
   S2
}
S4
S5
```

- Parallel loop

```
foreach (i:[1..n]){
  S
}
```

Activity/Thread creation - examples

X10/HJ: Abstraction of a *place*

- place: consists of data and some activities. An abstraction of a computing unit.
- Number of places fixed per execution.
- distribution: a map from indices to places.
- An array can be distributed, so can a parallel loop!

```
distribution D = block([1..100]);

int [D] A; // declares an array A distributed over D.

ateach (p: D){
   S;
}
// 100 iterations of S.
// Iteration p runs at the place D(p).
```

Communication across threads

Tasks/Threads/Processes need to communicate with each other for the program to make progress.

- Remote procedure calls.
- Shared memory.
- Message Passing.
- Synchronization.
- Examples: Files, Signals, Socket, Message queue, pipe, semaphore, shared memory, asynchronous message passing, memory mapped file.

Remote Procedure Calls

A subroutine or procedure to execute in another address space (core/processor), with no explicit coding.

- Typically, RPC is an synchronous event. While the server is processing the call the client is blocked.
- Easy to program, especially in reliable environments.
- Compared to local calls, a remote procedure may fail. Why?
- How to handle failure?
- By using RPC, programmers of distributed applications avoid the details of the interface with the network.
- The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.
- Examples: C, Java RMI, CORBA.
- Read yourself.

Shared memory

A large common RAM shared and simultaneously accessed by the multiple cores.
Note: Communication inside a task via memory is not generally referred to as 'shared memory'.

- Easy to visualize for the programmer.
- Communication can be fast.
- (Partitioned) Global Address Space.
- Scalable, especially for small number of cores.
- Not easily scalable for large number of cores.
- Cache coherence issues - Say a core updates its local cache - how to reflect the changes in the shared memory such that data access is not inconsistent.
- `#pragma omp flush [a, b, c]` : A synchronization point where memory consistency is enforced.
- `#pragma omp parallel private (a)`

Message passing

- Allows communication between processes (threads) using specific message-passing system calls.
- All shared data is communicated through messages
- Physical memory not necessarily shared
- Allows for asynchronous events
- Does not require programmer to write in terms of loop-level parallelism
- scalable to distributed systems
- A more general model of programming, extremely flexible
- Considered extremely difficult to write
- Difficult to incrementally increase parallelism
- Traditionally - no implicitly shared data (allowed in MPI 2.0)

MPI in implementation

- MPI (Message Passing Interface): A standard programming environment for distributed-meory parallel computers.
- All the processes involved in the computation are launched together, when the program starts. Say you need 1024 processes, all of them start at once. They may or not have anything meaningful to do immediately.
- Each process has an unique id, Each message has a label.
- Message label: ID of sender, ID of receiver, tag for the message.
- Only the intended receiver, waiting for the message receives it.

  `int MPI_Send(buff, count, type, dest, tag, Comm) int MPI_Recv(buff, count, type, source, tag, Comm, *stat)`

  `buff` : Pointer to buffer
  `count` : # of elem of `buff`.
  `type` : type of elem of `buff`.
  `dest` : destination id.
  `source` : source id
  `tag` : message tag.
  `stat` : status information.
- Deceptively simple, low level, yet extremely powerful abstraction.

Synchronizations

Task/Thread/Process Synchronization

- Tasks handshake or join at different program points to synchronize or commit.
- Achieved via locks, monitors, semaphores, barriers.
- Examples: C mutexes, Java *synchronized*, HJ/X10 *finish, atomic, clocks*.

Data Synchronization

- Keeping multiple copies of the data in coherence.

- Easy to program
- Most popular form of communication.
- Can lead to deadlocks.
- Data races still is an issue.

Synchronization examples

- Java `synchronized` methods - only one thread enters the code.

```
synchronized boolean Withdraw(int amount){
    ...
}
```

- Java `wait-notify`: `wait` waits for `notify` message from others.

```
synchronized(lockObject) {
   while (!condition) {lockObject.wait();}
   action;
}
```

- Java `Lock` : Mutex locks. (Make sure to unlock. Else?) What is the problem with the following code?

```
Lock lock = new ReentrantLock();
...
lock.lock();
while(list.notEmpty()){... Traverse the list}
lock.unlock();
```

Synchronization examples (contd)

X10 (from IBM)/HJ (from Rice universty)

- finish : Join operation
- clocks: Used for quiescence detection.
- An activity can register / deregister onto clocks dynamically.
- `next`: suspends an activity till all clocks that the current activity is registered can advance.

```
finish {
  async clocked (c1) {
      S1
      next;
      S2 }
  async clocked (c1) {
      S3
      next;
      S4 }
}
S5
```

Outline

Speedups in Parallel Programs

- Say a serial Program P takes T units of time.
- Q: How much time will the best parallel version P' take (when run on N number of cores)? $\frac{T}{N}$ units?
- Linear speedups is almost unrealizable, especially for increasing number of compute elements.
- $T_{total} = T_{setup} + T_{compute} + T_{finalization}$
- T_{setup} and $T_{finalization}$ may not run concurrently - represent the execution time for the non-parallelizable parts of code.
- Best hope : $T_{compute}$ can be fully parallelized.
- $T_{total}(N) = T_{setup} + \frac{T_{compute}}{N} + T_{finalization} \ldots\ldots\ldots (1)$
- Speedup $S(N) = \frac{T_{total}(1)}{T_{total}(N)}$
- Chief factor in performance improvement : Serial fraction of the code.

Amdahl's Law

- Serial fraction $\gamma = \frac{T_{setup}+T_{finalization}}{T_{total}(1)}$
- Fraction of time spent in parallelizable part = $(1-\gamma)$

$$T_{total}(N) = \underbrace{\gamma \times T_{total}(1)}_{\text{serial code}} + \underbrace{\frac{(1-\gamma) \times T_{total}(1)}{N}}_{\text{parallel code}}$$

$$= \left(\gamma + \frac{1-\gamma}{N}\right) \times T_{total}(1)$$

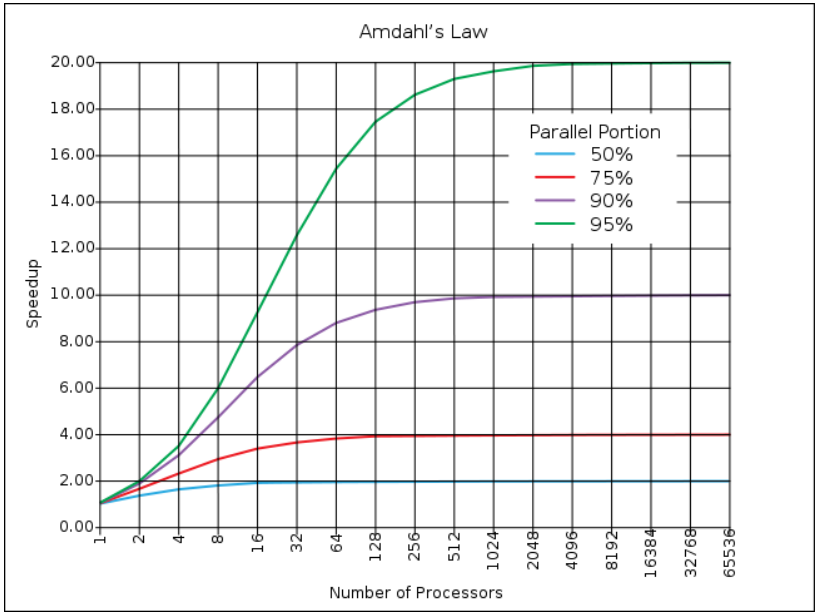
$$\text{Speedup } S(N) = \frac{T_{total}(1)}{\left(\gamma+\frac{1-\gamma}{N}\right)\times T_{total}(1)} = \frac{1}{\left(\gamma+\frac{1-\gamma}{N}\right)} \approx \frac{1}{\gamma} \quad \ldots \text{Amdahl's Law}$$

- Max speedup is inversely proportional to the serial fraction of the code.

Implications of Amdahl's law

- As we increase the number of parallel compute units, the speed up need not increase - an upper limit on the usefulness of adding more parallel execution units.
- For a given program maximum speedup nearly remains a constant.
- Say a parallel program spends only 10% of time in parallelizable code. If the code is fully parallelized, as we aggressively increase the number of cores, the speedup will be capped by ($\sim$) 1.11$\times$.
- Say a parallel program spends only 10% of time in parallelizable code. Q: How much time would you spend to parallelize it?
- Amdahl's law helps to set realistic expectations for performance gains from the parallelization exercise.
- Mythical Man-month - Essays on Software Engineering. Frederic Brooks.

Peaking via Amdahl's law



Limitations of Amdahl's law

- An over approximation : In reality many factors affect the parallelization and even fully parallelizable code does not result in linear speed ups.
- Overheads exist in parallel task creations/termination/synchronization.
- Does not say anything about the impact of cache - may result in much more or far less improvements.
- Dependence of the serial code on the parallelizable code - can the parallelization in result in faster execution of the serial code?
- Amdahl's law assumes that the problem size remains the same after parallelization: When we buy a more powerful machine, do we play only old games or new more powerful games?

Discussion: Amdahl's Law

- When we increase the number of cores - the problem size is also increased in practise.
- Also, naturally we use more and more complex algorithms, increased amount of details etc.
- Given a fixed problem, increasing the number of cores will hit the limits of Amdahl's law. However, if the problem grows along with the increase in the number of processors - Amdahl's law would be pessimistic
- Q: Say a program P has been improved to P' (increase the problem size) - how to keep the running time same? How many parallel compute elements do we need?

Gustafson's Law

- Invert the parameters in Eq(1):
  $T_{total}(1) = T_{setup} + N \times T_{compute}(N) + T_{finalization}$ ..... (2)
- Scaled serial fraction $\gamma_{scaled} = \frac{T_{setup} + T_{finalization}}{T_{total}(N)}$.
- $T_{total}(1) = \gamma_{scaled} \times T_{total}(N) + N \times (1 - \gamma_{scaled}) \times T_{total}(N)$
- $S(N) = N + (1 - N) \times \gamma_{scaled}$ ..... (Gustafson's Law)
- We are increasing the problem size. If we increase the number of parallel compute units - execution time may remain same (provided γ_{scaled} remains constant).
- It means that speedup is linear in N. Is it contradictory to Amdahl's law?

Comparison Amdhal's law and Gustafson's law

- Say we have program that takes 100s. The serial part takes 90s and the parallelizable part takes 10s.
- If we parallelize the parallel part (over 10 compute elements) the total time taken = 90 + $\frac{10}{10}$ = 91s.

Amdahl's law:	Gustafson's law:
$\gamma = 0.9$	$\gamma_{scaled} = \frac{90}{91} = 0.99$
Speedup $\approx \frac{1}{0.9} = 1.1$	Speedup$(10) = 10 + (1 - 10) \times 0.99 = 1.1$

- Speedups indicated by both Gustafson's Law and Amdahl's law are same.
- Gustafson's Law gives a better understanding for problems with varying sizes.

Bottlenecks in Parallel applications

- Traditional programs running on Von-Neumann Architectures - memory latency.
- The “memory wall” is the growing disparity of speed between CPU and memory outside the CPU chip.
- In the context of multi-core systems, the role of memory wall?
- Communication latency plays a far major role.
- Communication = task creation, sending data, synchronization etc.
- $T_{messaage-transfer} = \alpha + \frac{N}{\beta}$.
  - α communication latency - time it takes to send a single empty message.
  - β bandwidth of the communication medium. (bytes/sec)
  - N length of the message.

Reducing the communication latency cost

- A typical program involves, computation, communication and idling (why?).
- Overlap computation, communication and idle time.
  - Start the communication as early as possible. [always good?]
  - Instead of idling - do work of some other worker.
- Advantageous to aggregate communications into larger chunks.
- Avoid sending *self*-messages. (Why and How?)
- Ideal and useful parallelism.

Outline

Relevant X10 syntax

- `async S` : creates an asynchronous activity.
- `finish S` : ensures activity termination.

```
// Parent Activity
finish {
        S1; // Parent Activity
        async {
                S2; // Child Activity
        }
        S3; // Parent activity continues
}
S4;
```

Relevant X10 syntax (contd)

- **foreach** `(i: [1..n])` `S`

  ≡

  `for (i: [1..n])` **async** `S`
- **async**(p) clocked (c1, c2) S: creates an activity registered over clocks `c1, c2`.
- **next** : clock barrier

```
finish {
  async clocked (c1) {
     S1
     next;
     S2 }
  async clocked (c1) {
     S3
     next;
     S4 }
}
S5
```

Gap between Ideal and Useful parallelism

```
foreach(p: [1..1024]) {
    S1;
}
```

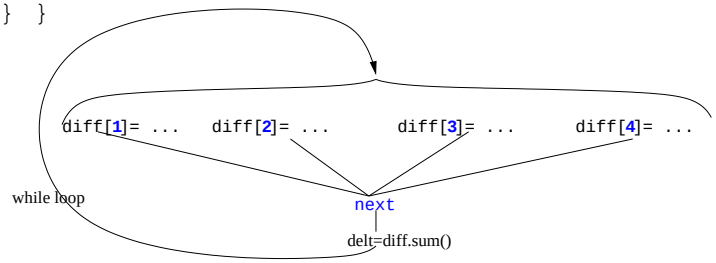
```
foreach (q: [1..16]){
   for (p: [1..64]){
     S1;
   }
}
```

- Programmers express *ideal* parallelism – over-specify.
- Only part of the parallelism is *useful*, for the target machine.
  - Synchronizations and Exceptions.

[Chunking parallel loops in the presence of synchronization, ICS 2009, Jun Shirako, Jisheng Zhao, V. Krishna Nandivada, Vivek Sarkar.]

Loop chunking Hardness

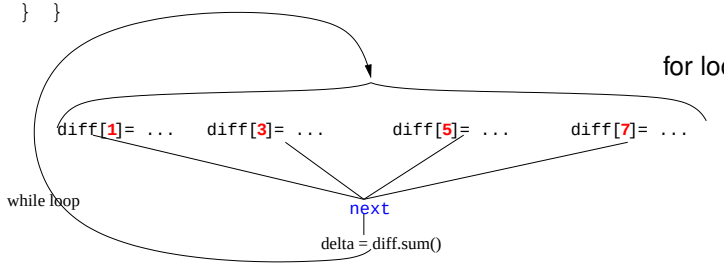
```
delta = epsilon+1;
clock ph = new clock();
foreach ( j : [1:n] ) clocked(ph) {
    while ( delta > epsilon ) {
      newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
      diff[j] = Math.abs(newA[j]-oldA[j]);
      next ;  // barrier
      ... delta = diff.sum(); ...
    } }
```



Loop chunking hardness - safety

```
delta = epsilon+1;
clock ph = new clock();
foreach ( jj : [1:n:2] ) clocked(ph) {
  for (int j = jj ; j <  min(jj+2,n) ; j++) {
    while ( delta > epsilon ) {
      newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
      diff[j] = Math.abs(newA[j]-oldA[j]);
      next ;  // barrier
      ... delta = diff.sum(); ...
    } } }
```

for loop j = 1



Chunking in the presence of exceptions

- Exception semantics of asyncs : caught only at the finish.
- Semantics of the chunked loop must match that of the original loop in the presence of exceptions.

```
foreach (i: Ie(R,g))                    ≠   for (p: Ie(R,g))
    S // contains no barriers                    S
```

- Each loop iteration must throw the exceptions in an asynchronous way.

```
for(p:  Ie(R, g))
 try {
  S
 } catch (Exception e){
   async throw e;
 }
```

What is the right chunking policy?

Assume N elements, P chunks, one dimension.

- Blocked distribution - $\{0, 1, \ldots \frac{N}{P}-1\}, \{\frac{N}{P}, \frac{N}{P}+1, \ldots 2\times\frac{N}{P}-1\}, \ldots \{(P-1)\times\frac{N}{P}, \ldots N-1\}$
- Cyclic distribution - $\{0, P, \ldots\}, \{1, P+1, \ldots\}, \ldots \{P-1, 2\times P-1, \ldots\}$
- Blocked cyclic distribution (blocking factor m, cycle stride c) - $\{0, 1, \ldots m-1, c\times m, c\times m+1, \cdots\}, \{m, m+1, \ldots 2\times m-1, 2\times c\times m, 2\times c\times m+1, \cdots\}, \cdots$
  - Interesting in multi dimensional data.
- Dynamic distribution
  - Create one activity per core.
  - Enable activities to dynamically share chunks of parallel iterations - based on different heuristics.
- Arbitrary distribution: Allow user to specify arbitrary distributions; a function from indices to places.

Step by step procedure for loop chunking

```
foreach (point p:  R) phased(⟨phaser-regs⟩)        ⟹   foreach (point g:  Ig(R)) phased(⟨phaser-regs⟩)
    S                                                       i-foreach (point p:  Ie(R, g)) phased
                                                              S
```

1. Loop Interchange:

```
i-foreach (point p :  R1) phased
  for (point q :  R2)
  // R2 is assumed to be independent of p
    S // S contains no break/continue statements
```
⟹
```
for (point q :  R2)
  i-foreach (point p :  R1) phased
    S
```

2. Loop Unswitching:

```
i-foreach (point p :  R1) phased
  if (e)
  // e is assumed to be independent of p
    S
```
⟹
```
if (e)
  i-foreach (point p :  R1) phased
    S
```

3. Loop Distribution:

```
i-foreach (point p :  R1) phased
  { S1; S2; }
```
⟹
```
i-foreach (point p :  R1) phased
  S1;
i-foreach (point p :  R1) phased
  S2;
```

4. Next Contraction:

```
i-foreach (point p :  R1) phased
  // Region R1 is assumed to be non-empty.
  next
```
⟹
```
next
```

Outline

Language syntax (contd)

Language syntax (contd)

- **forall** : parallel loop.

```
forall (point [i]: [1..n])
    S
```

≡

```
finish for (point [i]: [1..n])
    async S
```

- New Challenges: Scalable Synchronization and Communication.

```
for(i=0;i<n;++i){
 forall(j: [1..m]){
    S
 } }
```

# barriers created	n
# activities created	m × n
Max # parallel activities	m

```
forall(j: [1..m]){
 for(i=0;i<n;++i){
    S
 } }
```

# barriers created	1
# activities created	m
Max # parallel activities	m

forall distillation: What's the big deal?

```
delta=epsilon+1;iters=0;
while (delta > epsilon) {
 forall (j : [1:n]) {
  B[j]=(A[j-1]+A[j+1])/2.0;
  diff[j]=abs(B[j]-A[j]);
 } // forall
 // sum and exchange
 delta=diff.sum(); iters++;
 t=B;B=A;A=t;
} // while
```

```
delta=epsilon+1;iters=0;
forall (j : [1:n]) {
 while (delta > epsilon) {
  B[j]=(A[j-1]+A[j+1])/2.0;
  diff[j]=abs(B[j]-A[j]);
  // sum and exchange
  delta = diff.sum(); iters++;
  t=B;B=A;A=t;
 } // while
} // forall
```

Challenges: (1) Data dependence (2) Exceptions

[Reducing task creation and termination overhead in explicitly parallel programs, PACT 2010, Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, Vivek Sarkar.]

forall Distillation and Exceptions

- Exception semantics of **forall**: caught only at the implicit finish.
- Semantics of the translated code must match that of the original code in the presence of exceptions.

```
for (i:[1..n])
  forall (point p: R)
    S
```

≠

```
forall (point p: R)
  for (i:[1..n])
    S
```

- In iteration `i` of the `for` loop, if one ore more exceptions are thrown in the inner **forall** loop ⇒ iteration `i+1, i+2, ...` are not executed.

```
boolean excp = false;
forall (point p :  R)
  for (i:  [1..n]) {
    try {S;}
    catch (Exception e) {excp = true; throw e;}
    next;
    // synchronization ensures no data race for excp;
    if (excp == true)  break; }
```

Impact of data locality?

- The improvements from distillation:
  - From reducing the task creation and termination overheads.
  - From locality.
- How to measure the impact of locality?
  - Add code to compensate for the reduced task creation and termination.
- Our experience:
  - On a Niagara T2 system (dual-socket, socket = 8 cores x 8 hardware threads).

Benchmark	8 hardware threads			64 hardware threads		
	Unopt	Locality	Opt	Unopt	Locality	Opt
CG	16.40	10.87	9.37	11.67	12.07	1.40
MG	19.03	12.28	12.07	4.11	4.00	2.81
SOR	11.37	6.89	6.56	2.72	2.79	1.01
LUFact	32.34	19.53	18.39	13.28	14.28	3.19
MolDyn	65.51	33.19	32.69	10.45	7.97	5.58

Step by step procedure for distillation

Useless assignment - design a step-by-step procedure for `forall` distillation.

Sources

- Patterns for Parallel Programming: Sandors, Massingills.
- multicoreinfo.com
- Wikipedia
- fixstars.com
- Jernej Barbic slides.
- Loop Chunking in the presence of synchronization.