Parallel Processing

Chris Davis, Sonja Keserovic, Bryce Morsello, Sachin Patel, Colin Reid, Erick Smith

Introduction

Moore's Law[1] states that the number of transistors that can be placed on a microchip at a reasonable price will double approximately every two years. For the last few decades, computational throughput has tracked this growth. There are a few reasons why this will not continue to be the case. As transistor density grows, heat becomes an increasing issue. Also, as the complexity of interactions between transistors grows, latency between computational units becomes a factor. Importantly, as the number of transistors grows, the latency of transistor switching has not improved at nearly the same rate.

In order to continue to improve the total throughput of computational machines, one solution class is to increase the parallelism of that computation. Hardware engineers are still able to grow the size of multi-core machines, and the calculating components of individual cores can be distributed across the chip for the same exponential theoretical processing growth. Taking advantage of the ability to perform portions of a calculation at the same time requires different hardware approaches, and may require increasing changes to software. Intel's founder Andrew Grove thinks this is the inflection point[2] – "the time in the life of a business when its fundamentals are about to change".

We don't really know how to program parallel computers efficiently - not even after decades of experience. They are much more difficult to design and implement for than for sequential ones. The kind of bugs that are common in parallel programs are nondeterministic, difficult to find and fix. With multiple processing units operating simultaneously, there isn't even a clear definition of "stopping" a computation before it's complete. There is also lack of good, scalable parallel algorithms. Many parallel algorithms scale up to 8 cores, then there are no more improvements – or the algorithm performs worse when the number of cores increases.

Current Hardware Solutions

Hardware manufacturers have kept pace with Moore's Law in transistor density through 2004[3], but the limits of existing technology for transistor density have caused the clock speed to fall significantly off the curve[4]. The result is that processor architects have more aggressively started exploiting thread level parallelism (TLP) by replicating cores, rather than trying to find additional instruction level parallelism, or continue to increase clock speed.

Application-Specific Integrated Circuits(ASICs)

An end-user's experience of the speed of their computation is based on the overall performance of the machine. An ASIC is a custom-designed circuit that performs a specific function, such as implementing a complete cellular telephone on a single chip. With custom hardware, lower power consumption and explicit hardware parallelism can be gained, and software developers typically write only a hardware driver, which itself may be purely sequential, or event-driven.

Instruction-Level Parallelism (ILP)

Instruction-level parallelism is the process of executing existing instruction code faster by executing it partially in parallel. This is done using a handful of techniques:

Pipelining

Executing a single instruction requires fetching the instruction from cache or main memory, evaluting the instruction, possibly determining a branch point, and possibly accessing data from cache or main memory. The individual tasks of this execution can be done concurrently by different transistors on a single core. Like assembling a sandwich at Subway[6], there are multiple "stations" that each perform a portion of the operation before passing it on to another worker. Although an individual instruction won't be executed any faster with this method, the total throughput increase at the limit is porportional to the number of stations.

Superscalar

Instead of starting one new instruction on each clock cycle, a superscalar processor allows up to *n* instructions to be issued at a time. Logically you can think of multiple pipelines all pulling instuctions from the same isntruction stream. It is actually quite rare that *n* instructions can all be initiated at a time, but it isn't uncommon to get more than one instruction issued per clock.

Out-of-order Execution

Individual instructions can be reordered to take advantage of their independence. If a particular instruction doesn't have a dependency on logically "previous" instructions, it can be executed in tandem. This result can often be accomplished by the compiler during its optimization phase.

Speculative Execution

While instruction reordering provides some benefits, there are

limits. Additional performance can be gleaned by executing code speculatively, before the branch decision has been made.

Hardware Architecture

As the logical limits of optimizing single cores are approached, hardware manufacturers are already creating multiprocessor machines.

Shared vs. Distributed Memory

The hardware architecture of parallel computers can fall under either of two categories: *shared memory* or *distributed memory*. *Shared memory* refers to computers in which all processors have access to all of the memory in a global address space. Changes made to memory by one processor are visible to all of the other processors. This requires cache coherency protocols to ensure that all of the caches have a consistent view of memory. Shared memory computers can be further divided into two categories: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). UMA machines are also know as Symmetric Multiprocessor (SMP) machines, and are the typical architecture in today's commercial multi-core machines, such as Intel's Core2 Duo. In these computers, processors have equal access times to memory. In NUMA machines, there is still a global address space, but access to a processor's local memory is faster than access to remote memory of another processor.

The other category of parallel computers is distributed memory systems. In distributed memory, processors have their own local memory and there is no global address space. Instead, in order to communicate with other processors, the programmer must explicitly define send and receive messages. The network that allows communication between processors can vary widely. For example, a cluster can be thought of as a distributed memory system, where the method of communication is over Ethernet and commodity processors are used. Many supercomputers use distributed memory, such as the Cray T3E and IBM SP2.

Each type of system carries it advantages and disadvantages. Shared memory provides the simplest programming model, since there is a global address space. This is consistent with uniprocessor machines, and the programmer does not have to worry about data locality and sharing between processors. However, the disadvantage is that it doesn't scale well for a large number of processors because access to memory becomes a bottleneck and cache coherency protocols do not scale well. In order to avoid memory bus contention, the processor to memory communication can be implemented as a crossbar switch, where every processor is connected to every memory unit. However, there are still scaling issues because the size of the crossbar switch increases proportionally to the number of processors multiplied by the number of memory units. This can lead to increased costs, the need to lower clock frequency, and/or power issues.

The distributed memory systems solve these scalability issues, as they can scale to thousands of processors. Each processor can access its own local memory without having to worry about contention with other processors. However, the disadvantage is that it requires a new programming model, such as the Message Passing Interface (MPI). So far, this has been the main hindrance from wide-spread adoption.

In order to try to get the benefit of both approaches, a hybrid approach called distributed/shared memory systems (DSM) has been implemented. In this case, a block of CPUs are implemented with shared memory and then multiple blocks are connected through a network. The number of CPUs per block and the network topology can vary between systems. Computing clusters are a popular form of this, where each SMP is loosely coupled with its own OS image, and the SMPs are connected through a standard network. Supercomputers, such as the IBM SP3, are now often implemented as a cluster that is highly tuned and contains custom interconnects.

Multithreaded Processors

Historically, processor architects have investigated various approaches for increasing the throughput of their machines. One scheme for increasing utilization is to have more than one instruction stream ready for execution at a given time; thereby taking advantage of thread level parallelism (TLP). From the operating systems perspective, more than one thread can be scheduled for execution at a time. Early processors which implemented this idea would switch threads if there was a high latency operation running on one thread. This is known a course grained multithreading.

Fine grained multithreading takes this idea a bit further. It essentially switches between threads in a round robin fashion. This hides instruction latencies of all kinds. The throughput of the machine as a whole is increased at the expense of any particular thread. The Cray/Tera MTA machine made extensive use of this machine. In fact, they believed that this approach could hide so much latency that they could even forgo the use of processor side data caches.

Simultaneous Multithreading (SMT), an idea originating at the University of Washington, extends this idea a bit more. Just as out of order execution attempts to increase the pool of instructions available for execution by looking ahead in the instruction stream, SMT processors attempt to increase the size of this pool of instructions by producing a set of available instructions from multiple threads at the same time. The processor holds the state for two or more threads at the same time. The operating system will have scheduled both of the threads for execution, and the processor itself will now find instructions ready for execution from either instruction stream. With a relatively small increase in transistors and complexity, you can theoretically improve the throughput of the processor significantly. This technique has been used in practice in a number of processors. Digital Equipment Corp. built a version of the Alpha processor with SMT support and the chip area only increased 15%, yet the performance throughput for some applications was 3-4x. Intel has shipped versions of the Pentium IV and Xeon processors with SMT support. The Intel brand name for this technol-

ogy is Hyperthreading. Sun and IBM also include SMT support in their Niagara and Power processor brands.

Although SMT provides a number of performance benefits with a minimal complexity overhead, it does have some shortcomings. For example, the performance of any one thread will typically be lower, although throughput overall is increased. Additionally, there are some scenarios were performance as a whole can suffer. For example, because SMT processors don't duplicate most of the chip, but instead share the functional units – similar to an ordinary out of order processor – the threads may compete with each other. Consider a spin lock for example? One thread spins doing useless work taking up processor resources, while the other thread tries to get useful work done. In this case throughput suffers. Another pitfall is cache conflict. Intel had a problem in early HT processors where the caches backing the stack for each thread would conflict with each other, leading to high cache miss rates. These types of problems can and have been mitigated, but are never fully solved.

Single Instruction Multiple Data (SIMD)/Vector Processors

In addition to increasing parallelism by finding instructions which can execute at the same time, processor architects have introduced special instructions known as SIMD instructions which allow some level of explicit parallelism to be expressed within a sequential instruction stream. For example, a single SIMD instruction would operate on more than one piece of data at a time. Imagine adding two long vectors together. Instead of iterating through a loop once for each data item, you could iterate fewer times, because at each iteration a SIMD instruction would let you perform the operation on multiple data elements. This explicit parallelism (encoded in a sequential instruction stream) allows for significant speed boosts in some applications. Most modern processors support some form of SIMD under the title of "Multimedia" instructions.

Supercomputers of the past also relied heavily on the vector processing idea to increase parallelism – and thus performance.

Intel vs. AMD

Although IBM was first to manufacture a multicore design with its PowerPC 970 in 2002, it is Intel and AMD's products we hear the most about. Both are similar in speed but with some significant design differences.

AMD introduced a dual core Opteron chip in May 2005. This was significant as it was the first mainstream offering of a dual core chip in a consumer grade machine. Intel did not have a dual core offering for over six months.

The Opteron chip contains two 32/64 bit cores on a single chip. Each processor has a 64k L1 cache (both data and instruction). There is also a separate 1 MB L2 cache per processor. Fast on-chip communication between the two processors is achieved through the system request interface (SRI). The SRI handles the memory coherency responsibilities, ensuring both processors see a single memory image. Requests to RAM (or other processors) are implemented using the industry standard HyperTransport technology (HT).

Intel introduced its Core Duo Pentium in early 2006. It contains two 32-bit cores on a single chip. Each core has its own 32k L1 data and instruction cache. There is a single 2 MB (or 4MB depending on chip interation) L2 cache that is shared between processors. Fast on-chip communication between processors occurs through shared memory (not using SRI). The Front Side Bus (FSB) controller mediates transfers between the L2 cache and RAM.

Why Can't Sequential Processor Performance Continue to Scale?

There are three main obstacles currently impeding this continued growth:

1. The ILP Wall

2. The Power Wall

3. The Memory Wall

It is becoming increasingly hard to find additional Instruction Level Parallelism (ILP) in a sequential instruction stream. Techniques such as Out of order execution, register renaming, branch prediction, speculation, pipelining, superscalar, and vector operations have been extremely beneficial in general. All of these schemes, however, break down in some situations. For example, control-dependent computation (with lots of branches) or data-dependent memory addressing (ie, pointer chasing) do not perform well with these schemes. In practice we are limited to just a few instructions per clock cycle.

In the past, it was possible to keep power usage roughly constant. However, this was typically accomplished by reducing voltage as transistor sizes got smaller. The voltage can't be lowered much further, however. The threshold voltage of the transistor is impeding further reduction in voltage, and the result is increase power usage per unit area. Static (leakage) power is also getting worse as the voltages go down.

Processor performance has been growing much faster than memory subsystem performance. If the processor can't access memory fast enough, then the processor spends all its time waiting on memory. Techniques have been employed to hide memory latencies, but as the memory latencies increase these techniques are not enough. Increasing cache sizes can help, but the increases have to be significant to maintain throughput.

The designs of the AMD and Intel chips seem similar. The main difference between both designs is the position of the L2 cache. In the AMD chip, the L2 is private to the owning processor, while the Intel chip shares the L2 cache. For the AMD chip, managing SRI on the "back" of L2 gives processors more private memory but more importantly, the coherence information can be easily combined with that of other processors, leading to a global architecture known as symmetric multiprocessor (SMP). This also helps the AMD design scale better to larger numbers of cores over the Intel design. Intel's FSB causes a bottleneck that decreases performance as the numbers of cores grow.

While many of us are familiar with the hardware currently offered by Intel and AMD, it is important to note the work done by other companies. Sun, IBM and Cray have created various configurations of multi-processor systems. These systems are similar in that they both have large numbers of cores/processors, yet their shared memory and layout configurations are significantly different.

Sun Microsystems

The most notable multi-processor system produced by Sun is the Sun Fire E25K. The system has a total of 72 processors spread across 18 boards (4 per board). Each board is connected by 3 18x18 crossbars, each dedicated to addresses, data and responses. It is these crossbars that provide the communication capabilities of the system. Crossbar implementations provide great communication performance for multi-processor systems up to a point. This is because each node in the system has to be connected to every other node. In SUN's system, each board needs to be connected to the 17 other boards. Thus, there are 18x18 connections. Beyond this point, the performance degrades due to communication on the crossbar.

IBM

Another significant chip made by IBM is the Cell processor. It first appeared on the market in 2005 and had its first official use in the Sony Playstation 3. The Cell is composed of a 64-bit PowerPC core as well as 8 specialized cores. These 8 specialized cores are known as Synergistic Processing Elements (SPEs). These are typically used to perform vector operations with high floating point performance. This is one of the reasons the Cell is attractive as a gaming and scientific computing platform. What truly makes the Cell unique is its Element Interconnect Bus (EIB). The EIB is used to perform communication between the main core, SPEs and other components (total of 12 components in all). This differs greatly from Intel's Front Side Bus and AMDs crossbar. The EIB is implemented as a circular ring. Each component on the ring is at most 6 hops away from its furthest neighboring component. The architects originally had planned on implementing a crossbar, but chose the EIB due to space constraints on the chip. Also of note is the memory model used by the Cell. The chip uses a co-processor model where the primary PowerPC core has access to all global memory and is responsible for managing the read/write streams to the 8 SPEs. This master core bottleneck as well as the circular implementation of the EIB suggests that the Cell will not scale further beyond its current implementation.

IBM BlueGene

A review of parallel computing would not be complete without mentioning the BlueGene. This system dwarfs those previously mentioned as it has 65,536 dual core nodes. Each of these nodes has two 440 PowerPC processors which contain 32k in private L1 cache. The processors also have an L3 cache of size 3MB which they share. It is also important to note that each processor has a fairly humble speed: 770 MHz. Yet, the BlueGene makes up for this by sheer number of processors. The most unique feature of the BlueGene other than its size is the arrangement of the processors. These are arranged in a 3-dimensional torus network. Each node is connected to its 6 closest neighbors. If a processor needs to communicate with a node other than one of those 6, the data needs to be sent through the network of the torus mesh. Also, each node in the system can only access a portion of the overall memory (512MB per node). Thus, if a node requires more memory it will have to be shared among the other nodes and the data communicated. This is known as a distributed address space memory model.

As is illustrated by differences in the the described hardware, there are great differences from one parallel architecture to another. Developers of parallel applications need to take this into account when architecting their software as different memory models and inter-core communication can have a great impact on their runtime performance.

Implications on Software

While the hardware continues to advance and the availability of implicit parallelism gets consumed with optimizations, what is the impact on software. Runtime libraries may provide some of the answer, where existing sequential programs that call into libraries can be partially parallelized by improving the runtimes themselves. The remainder will have to be taken up by new programming language paradigms.

Operating Systems

As multiple CPU architectures become more commonplace, the operating system scheduler must become more intelligent in how it schedules threads and processes to the available CPUs. In addition, each of the CPUs in the system may have an unequal relationship with each other. For example, there could be a CPU topology where the bottom layer consists of a single Simultaneous Multithreading (SMT) physical processor which exposes two logical processors. At the next level, there could be two SMT cores grouped into a Symmetric Multiprocessing (SMP) domain, where each SMT core has equal access time to local memory. At the highest level, there could be two SMPs grouped together which make up a Non-Uniform Memory Access (NUMA) domain (see Figure 1). Moving threads or process load within a SMT physical processor is cheap, because both logical processors within the physical processor share the same memory, cache, and execution units. However, moving threads or processes from one NUMA node to another is

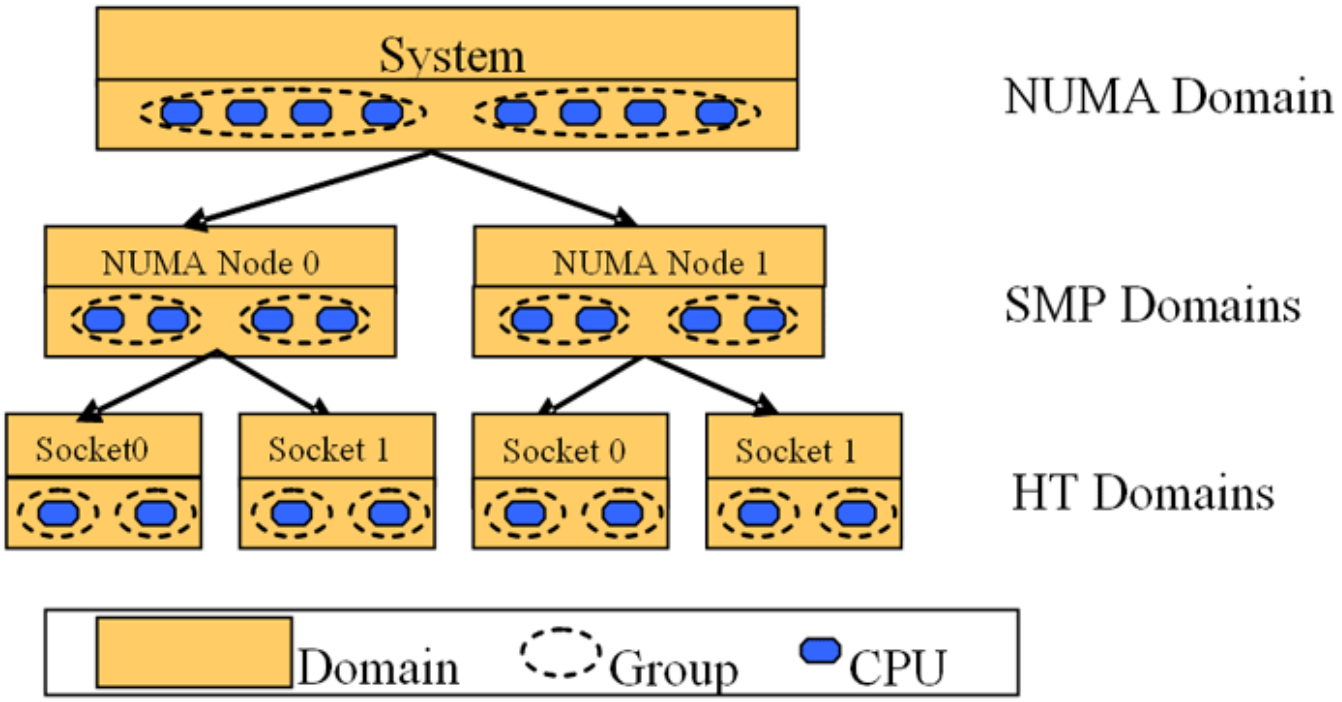

Figure 1

expensive, since the memory access time is longer for remote memory.

Linux handles balancing the load across all of the CPUs in an efficient manner by defining scheduling domains. In the example above, three domains would be defined: SMT, SMP, and NUMA. These domains contain the policy for how scheduling decisions are made. Within the SMT domain, the balancing attempts occur often, even when the imbalance in load is small. For example, if a sleeping thread is awakened, normally the thread would stay on the same processor since its data is likely to be cached on that processor. However, if another processor shares the same cache, then it is fine to move it to another processor if it is idle. Within the NUMA domain, balancing attempts are made very rarely, since the cost of moving a process between nodes is very high. Most of the time, a process will only be scheduled to another NUMA node when creating a new process. In addition, there is the option to further tune the system through the use of processor affinity. This can be used to specify an ideal processor to run a particular process on.

While there is significant support in today's commercial operating systems for multiprocessor execution, there is more work that can be done. For example, task scheduling can be improved to adapt to the workload. Typically, tasks that share the same data, such as threads that belong to the same process, will be scheduled across cores that share the same last-level cache in order to minimized resource contention. However, if the tasks share primarily read-only data, it may be better to replicate the data across the caches and distribute the tasks to all idle processors. There is currently research being done, called Micro Architectural Scheduling Assist, to use performance counters to track the shared resource usage and better predict the optimal scheduling. Another area of future research is in cache-fair thread scheduling. If a thread happens to get scheduled with another thread that uses up a majority of the cache, then it will end up running much slower than normal, even if it is at a higher priority than the other thread. Cache-fair thread scheduling attempts to address this by estimating the cache miss rate that the thread would incur under normal or fair conditions and compensating the thread by giving it more thread quantum to run if the actual miss rate falls lower than the fair miss rate.

In order to scale into the thousands of processors, many researchers have advocated a radically different operating system architecture. One example is the Wisdom parallel operating system, where virtual processors are used to abstract away the knowledge of the real physical processor topology. In this system, there can be an infinite number of virtual processors that can be multiplexed onto real processors. Each virtual processor handles the executing of one task and message passing is used to communicate between virtual processors. In addition, the code that makes up the traditional kernel, such as the scheduler, is distributed across many processors so that there isn't a bottleneck. Overall, there is a lot of room for operating system innovation to address massively parallel computers.

Emerging Programming Models and Runtimes

Highly parallel machines are programmed differently than classical von Neumann computers. The notion of a simple, linear program flow mutating the system from one state to another is no longer sufficient. Rather, parallel applications exhibit markedly different characterictics than conventional software.

They are typically event- or I/O-driven, highly asynchronous, and expressed in terms of small units of work that can be intelligently scheduled according to the resources available on a given system at a given time.

Event- and I/O-driven programming models are nothing new. Graphical user interfaces and commodity web servers provide classic examples of each, and neither need be developed using emerging parallel methodologies. But these models can be used in much more powerful ways on emerging, massively parallel hardware. Program components can be engineered as aggregating functions over multiple concurrent input sources, and the software can intelligently arbitrate between these inputs to efficiently process data that arrives at roughly the same time. Work can be cancelled if a concurrent computation determines that it is no longer necessary, and conversely, it can be performed eagerly in antipation of speeding up other concurrent tasks. These principles are demonstrated well in the Microsoft Robotics Concurrency and Coordination Runtime, which is designed to arbitrate concurrent streaming inputs from multiple ports.

As an example of a more conventional application of this kind of a runtime, consider a client application that pulls stock information from the web in near real time. Clearly the user's mouse clicks are an input to the system, and in a conventional architecture it might be acceptable to drive all computations directly from a UI message pump. However, this architecture begins to crumble if the computations need information from the web that must be loaded on demand with latencies in the tens or hundreds of milliseconds. If the user asked for a custom analysis of a particular mutual fund, for instance, the application might dispatch dozens of requests to the web in parallel to investigate the component equities, and then generate even more requests as a result of the equity information. On a conventional runtime it would be challenging for the developer to do anything more sophisticated than wait for all the requests to return, processing them nearly sequentially. A typical multithreaded application today would parallelize the work only to the extent that each response could be preprocessed on its own thread before aggregating the results.

It is easy to see how complex the application would have to be if the user expected it to fully utilize their 32-core laptop CPU. They might expect it to be responsive, so that they could click around, triggering concurrent computations and web requests and rendering graphs as data arrives and as calculations complete. The application data cache, for example, would have to cope with high contention on individual cache entries. If one computation has just requested a particular resource, a second computation that also needs the same resource should wait for the first request to complete instead of dispatching another request. A developer could choose to design the system so that the second thread would block, but then a large number of threads could rapidly emerge and coordinating them to cancel or share redundant work would become difficult. Serious deadlocks and race conditions could also arise, and blocking is therefore often avoided in this type of parallel application.

So to avoid blocking, an asynchronous, callback-oriented cache would likely lie at the heart of this stock application. And to keep the callbacks from blocking, the asynchrony ultimately driven by the long Internet I/O latencies would rapidly propagate throughout the code. This would be a difficult application to write today, and the relative cost of speed-of-light latencies on the web is only getting worse. Throw in a natural interface that accepts voice, gaze, multitouch and device mesh inputs, and the problem becomes orders of magnitude more complex. Add a rich, animated, framerate-oriented graphical UI as an output, and the problem becomes horrific since large amounts of work could be wasted preparing frames that are never even rendered. The many-core CPUs of the future will have lots to do; our challenge is to keep them working on the right things.

Parallel runtimes and models help developers simplify this kind of heavily parallel application by inverting the architecture and concentrating on functions over asynchronous inputs rather than on steps necessary to produce outputs. This is a much more resilient approach in light of the massive asynchrony we can expect of our applications in the future. The functional approach has been long proven in products like SQL Server, Excel, Matlab and Photoshop: these successful applications were designed with asynchronous functions as their internal building blocks in order to keep them responsive. But the complexity of these applications reflect how hard it can be to design and use custom, parallelizable patterns and practices for every project.

Thankfully, emerging parallel runtimes are designed to factor out the common features of these popular asynchronous and functional approaches, so that they can be used by a broader population of software developers. Yet source code based on these runtimes can be barely recognizable. For example, the following snippet of code from an excellent MSDN article[7] demonstrates the asynchronous, I/O driven structure of a highly parallel sample application:

```
Port<WebResponse> responsePort = null;
Port<Exception> failurePort = null;
Port<DateTime> timeoutPort = new Port<DateTime>();

for (Int32 n = 0; n < c_ImageUrls.Length; n++)
{
    WebRequest webReq = WebRequest.Create(c_ImageUrls[n]);
    ApmToCcrAdapters.GetResponse(webReq, ref responsePort, ref
failurePort);
}

dq.EnqueueTimer(TimeSpan.FromMilliseconds(2000), timeoutPort);

Arbiter.Activate(dq,
    Arbiter.Choice(
        Arbiter.Receive(false, failurePort, delegate(Exception
e)
            {
                Msg("At least 1 GetResponse failed");
            }
        ),

    Arbiter.Receive(false, timeoutPort, delegate(DateTime dt)
        {
            Msg("Some requests did not complete within 2 sec-
onds.");
        }
    ),

    Arbiter.MultipleItemReceive(false, responsePort, c_ImageUrls.
```

```
Length,
          delegate(WebResponse[] responses) {
               foreach (WebResponse response in responses)
               {
                    Byte[] data = new Byte[response.ContentLength];
                    response.GetResponseStream().Read( data, 0,
data.Length);
                    Msg("ResponseUrl={0}", response.ResponseUri);
               }
          }
     ))
);
```

This type of structure has almost no relation to traditional imperative control flow, yet it is built using familiar C#/.NET primitives, such as delegates, anonymous methods and iterators. And even iterators themselves can assume a new form in a highly concurrent asynchronous program:

```
private static IEnumerator<ITask> SaveWebSiteToFile() {
    WebResponse webResponse = null;

    yield return Arbiter.Choice(ApmToCcrAdapters.GetResponse(
        WebRequest.Create("http://Wintellect.com")),
        delegate(WebResponse wr) {
            Msg("Got web data"); webResponse = wr; },
        delegate(Exception e) { Msg("Failed to get web data"); });

    if (webResponse == null) yield break;

    FileStream fs = new FileStream(@"WebData.html", FileMode.Cre-
ate,
        FileAccess.Write, FileShare.Write, 8 * 1024,
            FileOptions.Asynchronous);
    using (fs) {
        Byte[] webData = new Byte[10000];
        Int32 numbytes = webResponse.GetResponseStream().Read(
            webData, 0, webData.Length);
        Array.Resize(ref webData, numbytes);

        yield return Arbiter.Choice(ApmToCcrAdapters.Write(fs,
             webData, 0, webData.Length),
            delegate(EmptyValue ev) {
                Msg("Wrote web data to file"); },
            delegate(Exception e) {
                Msg("Failed to write web data to file"); });
    }
}
```

A complete dissection of these code samples cannot be accommodated within this paper's space constraints, yet the essential elements of a parallel runtime are readily apparent from these examples. Tasks are expressed as small units of work without an explicit parallel execution plan, yet the runtime is able to generate chores and tasks that can be optimally executed at runtime according to the system's available resources. Execution is deferred until necessary, and an intelligent implementation could actively identity opportunities to eagerly execute or terminate tasks based on runtime profiling, high level branch prediction, and computation based on concurrent inputs.

The runtime clearly requires software developers to undertake extreme contortions in their coding styles, and these examples show the strain that parallel programs exert on conventional programming languages. This is why programming languages are themselves are changing so profoundly in response to these hardware trends.

Computer Language Implications

The growth of parallel processing hardware has led to the necessity for programmers to write code that utilizes the available parallelization. While instruction level parallelization has provided some implicit benefit, and compilers and runtimes can solve problems without explicit participation of the end-level programmer, this has only limited benefit. Compilers are not able to deduce the intention of a program and rewrite it in a new way.

Major programming languages today have mechanisms for using the threads paradigm for explicitly parallelizing applications. Starting and stopping threads, and sychronizing them with mutexes, locks, and critical sections is the job of the coder, with no help from the language itself or runtime libraries. There are dozens of new programming languages today for writing parallel programs that are based on languages with significant code bases. These are "sequential-like" languages, with added keywords and mechanisms built in, to allow compiler and runtime-level support for parallelization.

Another approach to creating parallel programs is to use functional programming languages. Functional languages were primarily developed for modeling mathematics. Functional languages define computation in terms of mathematical functions. A happy side effect of this design is that, since functions are stateless and have no side effects, any evaluations can be done where function inputs are known. Analogous to these languages is Google's MapReduce, which is an implicitly parallelizable paradigm for creating programs that is dissimilar to sequential programming.

Farther out is the possibility of the dominance of intentional programming languages[6]. The purported advantage of this design is to reduce code size, minimize the consequences of system-wide code changes, and reduce programmer-induced bugs. A consequence is that, since the intention of the algorithms is captured at the highest level, the compiler is exposed to the parallelism of the algorithms.

Transactional Memory

Transactional memory (TM) is an alternative way to coordinate access to shared data. It's not a "silver bullet" for addressing parallel programming complexity – it just shifts much of the burden of synchronizing and coordinating parallel computations from a programmer to a compiler, runtime, and/or hardware. The main challenge is to build efficient TM infrastructure either in software or hardware (or both). This area is still under active research and there are no mainstream implementations yet.

The idea of TM comes from database systems, which were successful at exploiting concurrency for years (e.g. many queries can execute at the same time and programmer doesn't need to know about this). The secret lies in database transactions.

In the database world, a transaction is indivisible sequence of actions. It has four properties (ACID):

Atomicity – either all actions are performed or none of them

are executed.
Consistency – after transaction succeeds it leaves data in consistent state. Consistency property is application specific.
Isolation – each transaction produces a correct result, regardless of any other transactions executing at the same time.
Durability – once a transaction commits, its results are permanent and available to other transactions. Note that this property is not useful for TM since memory states don't need to be permanent.

ACI properties of transactions provide a useful abstraction for synchronization of access to shared data. If a set of instructions can be executed in context of a transaction, then *atomicity* ensures all instructions will execute (or fail) and *isolation* ensures there will be no interference between multiple transactions even if they operate on the same shared data. These two properties have potential to greatly simplify concurrent programming.

Here is an example of a simple transaction:

```
atomic
{
  account1.Credit(amount);
  account2.Debit(amount);
}
```

An *atomic* block ensures atomicity and isolation, while the programmer needs to ensure consistency. There is no explicit locking and synchronization required and the programmer doesn't need to know if there are any other parts of code that access any of the objects modified inside an atomic block. There is also no need to know anything about implementation of methods called from an atomic block (e.g. what other objects might be modified as side effects). All those details can be safely abstracted from a programmer while underlying runtime and/or hardware track various interactions and ensure correctness.

The main challenges in implementing a TM system are coming up with good semantics that fit naturally with existing languages, libraries, OSs, etc and making its performance acceptable. There are many aspects of TM semantics that need further research. For example, what's the best way to deal with non-reversible operations like IO? Should they be allowed inside atomic blocks? Allowing them would make TM blend naturally with existing code but consequences are severe – TM systems would need to guarantee that any atomic block with IO operations will always succeed (i.e. there will be no rollback). This would affect performance of course – another area that requires lots of research to find the right balance of guarantees given by TM and amount of overhead required to keep those promises.

TM is a promising technology that could simplify development of parallel software but there are still lots of questions that need to be answered.

Conclusion

Is parallelism the solution to the growing sequential hardware constraints? There have been significant advances in new approaches to creating algorithms, such as Google's MapReduce[10]; significant improvements in instruction-level parallelism; improvements to compiler optimizations to find available parallelism in existing programs. Symmetric parallelism can increase the speed of a computation which has available parallelism, at rates potentially even faster than the density growth of transistors on a chip, and operating systems can implement efficiency mechanisms to assure that ready threads can be run.

But Craig Zilles of the University of Illinois says "It's very hard to write a correct sequential program. It will be only harder to write a correct parallel program." And it's true that many of the current developments are bounded by the amount of implicit parallelism in sequentially-written programs.

It's clear that massively parallel computational hardware is in our future, and it seems evident that software designers will have to offer parallelism to the hardware in a new way. The open questions are: How big a change will developers have to make, will those changes be extensions to predominant current languages, and how much help from runtimes and "smart" hardware will we get?

References

1. Moore's Law: Electronics Magazine 19 April 1965

2. Andrew Grove, Only the Paranoid Survive 1999

3. Intel Product introductions

4. The Free Lunch is Over, Herb Sutter

5. James Larus, Ravi Rajwar: Transactional Memory (Synthesis Lectures on Computer Architecture), Morgan & Claypool Publishers

6. Charles Simonyi, Intentsoft.com

7. Craig Zilles, University of Illinois

8. John L. Hennessy, David A. Patterson: Computer Architecture - A quantitatie Approach, Fourth Edition, Morgan Kaufmann

9. Burton Smith, "Mainstream Parallel Computing", Microsoft Research Talk

10. MapReduce, Jeff Dean, Sanjay Ghemawat, Google Research

11. (http://msdn.microsoft.com/en-us/magazine/cc163556.aspx) Jeffrey Richter on the Concurrency and Coordination Runtime

12. Suresh Siddha, Venkatesh Pallipadi, Asit Mallick. Process Scheduling Challenges in the Era of Multi-Core Processors. http://download.intel.com/technology/itj/2007/v11i4/9-process/9-Process_Scheduling_Challenges.pdf

13. Paul Austin, Kevin Murray, Andy Wellings. The Design of an Operating System for a Scalable Parallel Computing Engine. http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol21/issue10/spe052pa.pdf

14. Parallel Computing on Wikipedia. http://en.wikipedia.org/wiki/Parallel_computing