Parallelism and operating systems

M. Frans Kaashoek

MIT CSAIL

With input from Eddie Kohler, Butler Lampson, Robert Morris, Jerry Saltzer, and Joel Emer

Parallelism is a major theme at SOSP/OSDI

Real problem in practice, from day 1

Parallel programming is either:

- **a cakewalk**: No sharing between computations
- **a struggle**: Sharing between computations
  - race conditions
  - deadly embrace
  - priority inversion
  - lock contention
  - ...

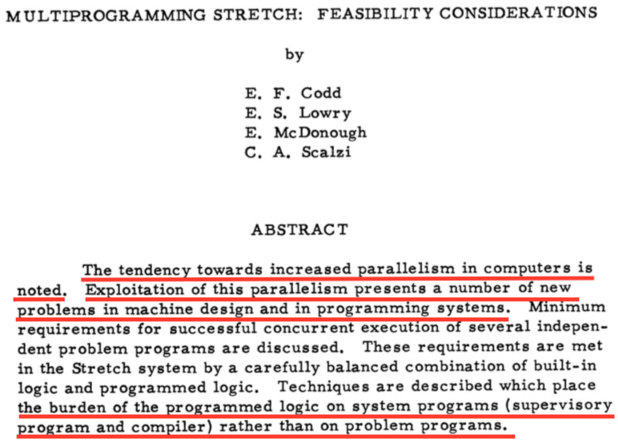
SOSP/OSDI is mostly about avoiding struggling for programmers

Parallelism is a major theme before SOSP

Several forms of parallelism

- User-generated parallelism
- I/O parallelism
- Instruction-level parallelism

An example: Stretch [IBM TR 1960]:



MULTIPROGRAMMING STRETCH: FEASIBILITY CONSIDERATIONS

by

E. F. Codd
E. S. Lowry
E. McDonough
C. A. Scalzi

ABSTRACT

The tendency towards increased parallelism in computers is noted. Exploitation of this parallelism presents a number of new problems in machine design and in programming systems. Minimum requirements for successful concurrent execution of several independent problem programs are discussed. These requirements are met in the Stretch system by a carefully balanced combination of built-in logic and programmed logic. Techniques are described which place the burden of the programmed logic on system programs (supervisory program and compiler) rather than on problem programs.

Three types of parallelism in operating systems

1. **User parallelism**
   - Users working concurrently with computer
2. **I/O concurrency**
   - Overlap computation with I/O to keep a processor busy
3. **Multiprocessors parallelism**
   - Exploit several processors to speedup tasks

The first two may involve only **1** processor

This talk: 4 phases in OS parallelism

Phases	Period	Focus
Time sharing	60s/70s	Introduction of many ideas for parallelism
Client/server	80s/90s	I/O concurrency inside servers
SMPs	90s/2000s	Multiprocessor kernels and servers
Multicore	2005s-now	All software parallel

Phases represent major changes in commodity hardware

In reality phases overlap and changes happened gradually

Trend: More programmers must deal with parallelism

Talk is **not** comprehensive

Phase 1: Time sharing

Many users, one computer

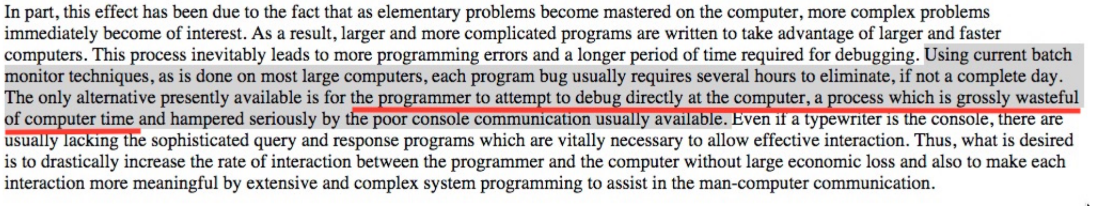
- Often 1 processor

[IBM 7094, 1962]

Standard approach: batch processing

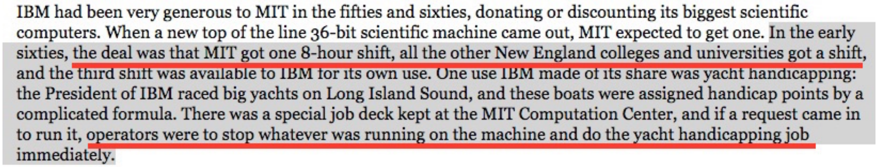
Run one program to completion, then run next

A pain for interactive debugging [SJCC 1962]:



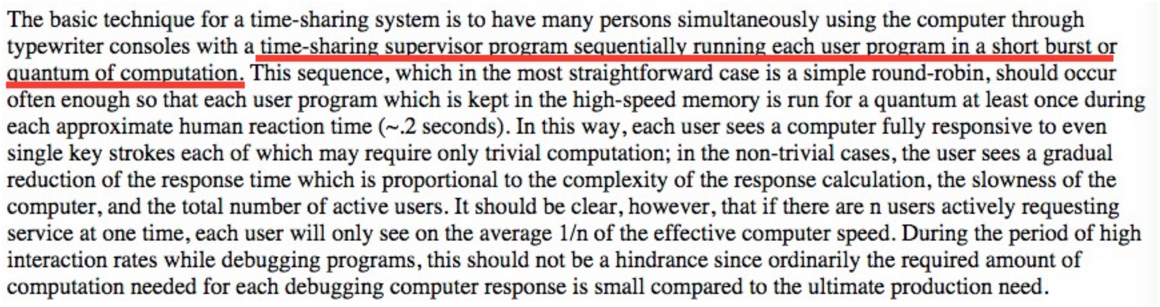
In part, this effect has been due to the fact that as elementary problems become mastered on the computer, more complex problems immediately become of interest. As a result, larger and more complicated programs are written to take advantage of larger and faster computers. This process inevitably leads to more programming errors and a longer period of time required for debugging. Using current batch monitor techniques, as is done on most large computers, each program bug usually requires several hours to eliminate, if not a complete day. The only alternative presently available is for the programmer to attempt to debug directly at the computer, a process which is grossly wasteful of computer time and hampered seriously by the poor console communication usually available. Even if a typewriter is the console, there are usually lacking the sophisticated query and response programs which are vitally necessary to allow effective interaction. Thus, what is desired is to drastically increase the rate of interaction between the programmer and the computer without large economic loss and also to make each interaction more meaningful by extensive and complex system programming to assist in the man-computer communication.

Time-sliced at 8-hour shifts [http://www.multicians.org/thvv/7094.html]:



IBM had been very generous to MIT in the fifties and sixties, donating or discounting its biggest scientific computers. When a new top of the line 36-bit scientific machine came out, MIT expected to get one. In the early sixties, the deal was that MIT got one 8-hour shift, all the other New England colleges and universities got a shift, and the third shift was available to IBM for its own use. One use IBM made of its share was yacht handicapping: the President of IBM raced big yachts on Long Island Sound, and these boats were assigned handicap points by a complicated formula. There was a special job deck kept at the MIT Computation Center, and if a request came in to run it, operators were to stop whatever was running on the machine and do the yacht handicapping job immediately.

Time-sharing: exploit user parallelism



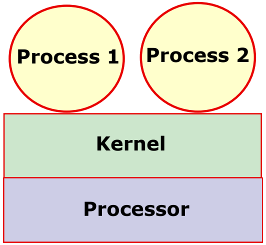
The basic technique for a time-sharing system is to have many persons simultaneously using the computer through typewriter consoles with a time-sharing supervisor program sequentially running each user program in a short burst or quantum of computation. This sequence, which in the most straightforward case is a simple round-robin, should occur often enough so that each user program which is kept in the high-speed memory is run for a quantum at least once during each approximate human reaction time (~.2 seconds). In this way, each user sees a computer fully responsive to even single key strokes each of which may require only trivial computation; in the non-trivial cases, the user sees a gradual reduction of the response time which is proportional to the complexity of the response calculation, the slowness of the computer, and the total number of active users. It should be clear, however, that if there are n users actively requesting service at one time, each user will only see on the average 1/n of the effective computer speed. During the period of high interaction rates while debugging programs, this should not be a hindrance since ordinarily the required amount of computation needed for each debugging computer response is small compared to the ultimate production need.

CTSS [SJCC 1962]
Youtube: “ctss wgbh” [https://www.youtube.com/watch?v=Q07PhW5sCEk, 1963]

Many programs: an opportunity for I/O parallelism

Multiprogramming [Stretch 1960, CTSS 1962]:

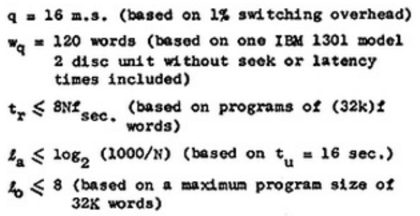
- On I/O, kernel switches to another program
- Later kernel resumes original program
- Benefit: higher processor utilization



Kernel developers deal with I/O concurrency

supervisor $<$ 5K 36-bit-words

Programmers write sequential code



q = 16 m.s. (based on 1% switching overhead)

w_q = 120 words (based on one IBM 1301 model 2 disc unit without seek or latency times included)

$t_r \leq 8Nf_{sec.}$ (based on programs of (32k)f words)

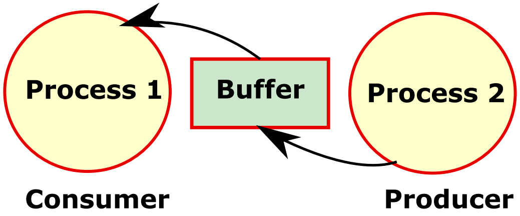
$\ell_a \leq \log_2 (1000/N)$ (based on t_u = 16 sec.)

$\ell_o \leq 8$ (based on a maximum program size of 32K words)

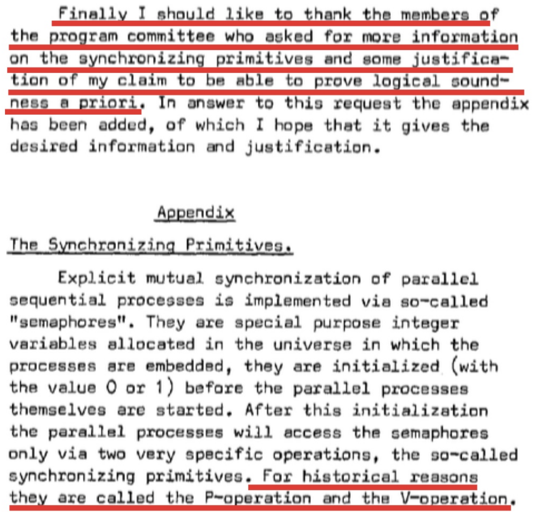
Challenge: atomicity and coordination

Example: the THE operating system [EWD123 1965, SOSP 1967]

- Technische Hogeschool Eindhoven (THE)
- OS organized as many “sequential” processes
  - A driver is a sequential process



The THE solution: semaphores



Finally I should like to thank the members of the program committee who asked for more information on the synchronizing primitives and some justification of my claim to be able to prove logical soundness a priori. In answer to this request the appendix has been added, of which I hope that it gives the desired information and justification.

Appendix

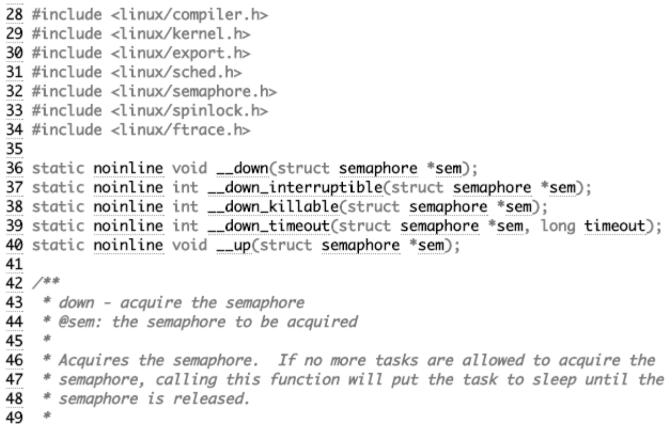
The Synchronizing Primitives.

Explicit mutual synchronization of parallel sequential processes is implemented via so-called "semaphores". They are special purpose integer variables allocated in the universe in which the processes are embedded, they are initialized (with the value 0 or 1) before the parallel processes themselves are started. After this initialization the parallel processes will access the semaphores only via two very specific operations, the so-called synchronizing primitives. For historical reasons they are called the P-operation and the V-operation.

[The “THE” multiprogramming system, First SOSP]

The THE solution: semaphores

Still in practice today



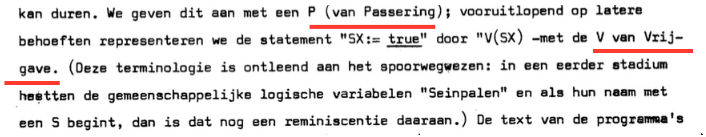
```
#include <linux/compiler.h>
#include <linux/kernel.h>
#include <linux/export.h>
#include <linux/sched.h>
#include <linux/semaphore.h>
#include <linux/spinlock.h>
#include <linux/ftrace.h>

static noinline void __down(struct semaphore *sem);
static noinline int __down_interruptible(struct semaphore *sem);
static noinline int __down_killable(struct semaphore *sem);
static noinline int __down_timeout(struct semaphore *sem, long timeout);
static noinline void __up(struct semaphore *sem);

/**
 * down - acquire the semaphore
 * @sem: the semaphore to be acquired
 *
 * Acquires the semaphore.  If no more tasks are allowed to acquire the
 * semaphore, calling this function will put the task to sleep until the
 * semaphore is released.
 *
```

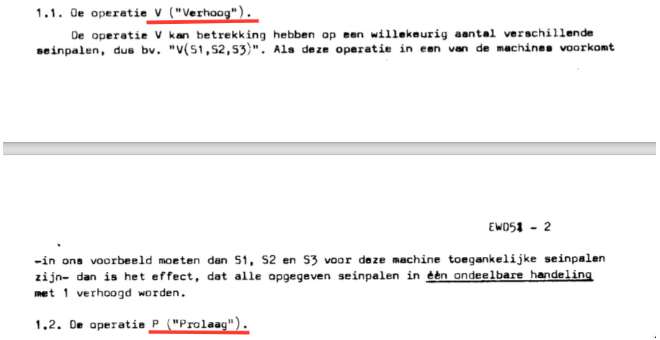
P & V?

passing (P) and **release** (V) [EWD35]



kan duren. We geven dit aan met een P (van Passering); vooruitlopend op latere behoeften representeren we de statement "SX:= true" door "V(SX) –met de V van Vrij-gave. (Deze terminologie is ontleend aan het spoorwegwezen: in een eerder stadium heetten de gemeenschappelijke logische variabelen "Seinpalen" en als hun naam met een S begint, dan is dat nog een reminiscentie daaraan.) De text van de programma's

portmanteau **try to reduce** (P) and **increase** (V) [EWD51]



1.1. De operatie V ("Verhoog").

De operatie V kan betrekking hebben op een willekeurig aantal verschillende seinpalen, dus bv. "V(S1,S2,S3)". Als deze operatie in een van de machines voorkomt

EWD51 – 2

–in ons voorbeeld moeten dan S1, S2 en S3 voor deze machine toegankelijke seinpalen zijn– dan is het effect, dat alle opgegeven seinpalen in één ondeelbare handeling met 1 verhoogd worden.

1.2. De operatie P ("Prolaag").

Time-sharing and multiprocessor parallelism

Early computers with several processors

- For example, Burroughs B5000 [1961]

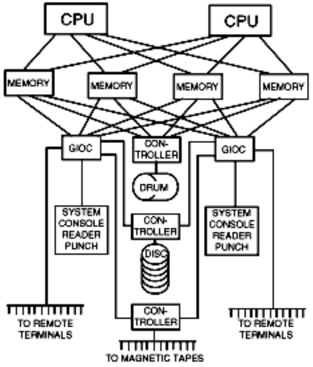
Much attention paid to parallelism:

- Amdahl's law for speedup [AFIPS 1967]
- Traffic control in Multics [Saltzer PhD thesis, 1966]
- Deadlock detection
- Locking ordering
- ...

I.e., Most ideas that you will find in an intro OS text

Serious parallel applications

- E.g., Multics Relational Database Store
  - Ran on 6-processor computer at Ford



[GE 645, Multics Overview 1965]

Time-sharing on minicomputers: just I/O parallelism

Minicomputers had only one processor

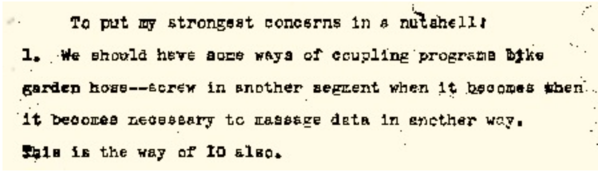
Multiprocessor parallelism de-emphasized

- Other communities develop processor parallelism further (e.g., DBs).

For example: Unix [SOSP 1973]

- Unix kernel implementation specialized for uniprocessors
- User programs are sequential
  - Pipelines enable easy-to-use user-level producer/consumer

```
$ cat todo.txt | sort | uniq | wc
      273     1361     8983
$
```



To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

[McIlroy 1964]

Phase 2: Client/server computing

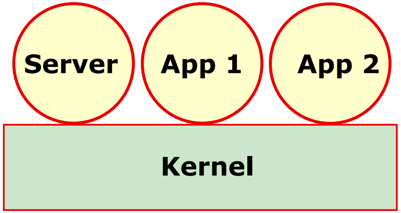
Computers inexpensive enough to give each user her own

Local-area networks and servers allow users to collaborate

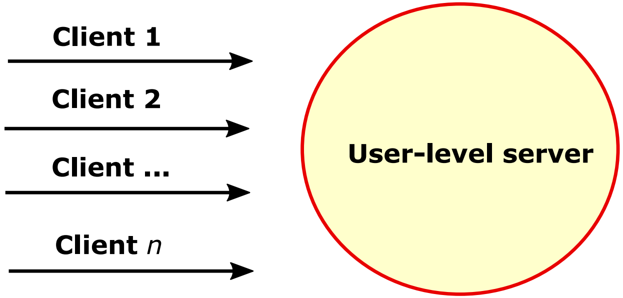
[Alto, Xerox PARC, 1975]

Goal: wide range of services

Idea: allow non-kernel programmers to implement services by supporting servers at user level

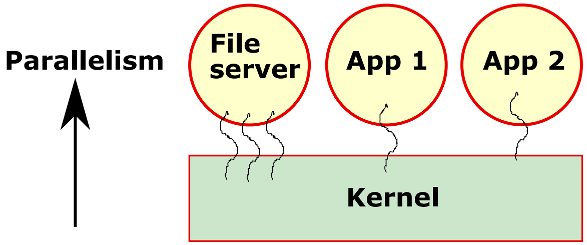


Challenge: user-level servers must exploit I/O concurrency



Some of the requests involve expensive I/O

Solution: Make concurrency available to servers



Kernel exposes interface for server developers

- Threads
- Locks
- Condition variables
- ...

Result: many high-impact ideas

New operating systems (Accent [SOSP 1981]/Mach [SOSP 1987], Topaz/Taos, V [SOSP 1983], etc.)

- Support for multithreaded servers encourages microkernel design

Much impact: e.g., Pthreads [POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]

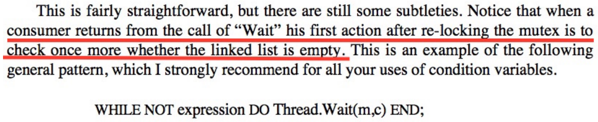
- Supported now by many widely-used operating systems

New programming languages (Mesa [SOSP 1979] , Modula2+, etc.)

- If you have multithreaded programs, you want automatic garbage collection
- Other nice features too (e.g., monitors, continuations)
- Influenced Java, Go, ...

Programming with threads

An introduction to programming with threads [Birrell tutorial 1989]



This is fairly straightforward, but there are still some subtleties. Notice that when a consumer returns from the call of "Wait" his first action after re-locking the mutex is to check once more whether the linked list is empty. This is an example of the following general pattern, which I strongly recommend for all your uses of condition variables.

```
WHILE NOT expression DO Thread.Wait(m,c) END;
```

Case study: Cedar and GVX window system [SOSP 1993]:

- Many threads
- Written over a 10 year period, 2.5M LoC

Design patterns:

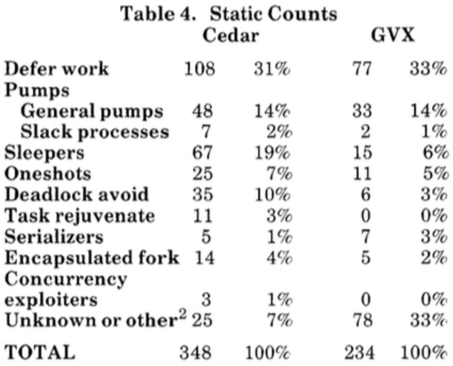
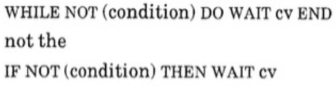


Table 4. Static Counts

	Cedar		GVX	
Defer work	108	31%	77	33%
Pumps				
General pumps	48	14%	33	14%
Slack processes	7	2%	2	1%
Sleepers	67	19%	15	6%
Oneshots	25	7%	11	5%
Deadlock avoid	35	10%	6	3%
Task rejuvenate	11	3%	0	0%
Serializers	5	1%	7	3%
Encapsulated fork	14	4%	5	2%
Concurrency exploiters	3	1%	0	0%
Unknown or other[2]	25	7%	78	33%
TOTAL	348	100%	234	100%

Bugs:



```
WHILE NOT (condition) DO WAIT cv END
not the
IF NOT (condition) THEN WAIT cv
```

The debate: events versus threads

Handle I/O concurrency with event handlers

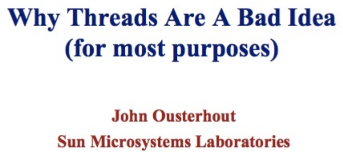
- Simple: no races, etc.
- Fast: No extra stacks, no locks

High-performance Web servers use events

Javascript uses events

The response: Why Events Are A Bad Idea [HotOS IX]

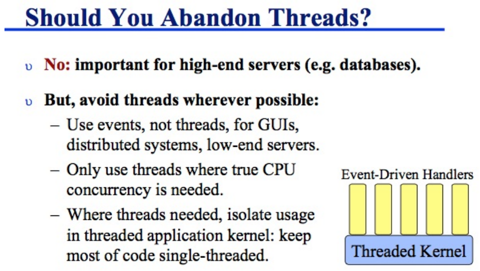
- Must break up long-running code paths
- “Stack ripping”
- No support for multiprocessor parallelism



Why Threads Are A Bad Idea (for most purposes)

John Ousterhout
Sun Microsystems Laboratories

[Keynote at USENIX 1995]

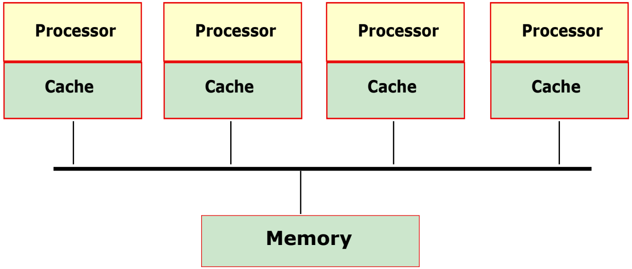


Should You Abandon Threads?

- **No: important for high-end servers (e.g. databases).**
- **But, avoid threads wherever possible:**
  - Use events, not threads, for GUIs, distributed systems, low-end servers.
  - Only use threads where true CPU concurrency is needed.
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded.

Event-Driven Handlers
Threaded Kernel

Phase 3: Shared-memory multiprocessors (SMPs)



Mid 90s: inexpensive x86s multiprocessors showed up with 2-4 processors

Kernel and server developers had take multiprocessor parallelism seriously

- E.g., Big Kernel Lock (BKL)
- E.g., Events **and** threads

Much research on large-scale multiprocessors in phase 3

Scalable NUMA multiprocessors: BBN Butterfly, Sequent, SGI, Sun, Thinking Machines, ...

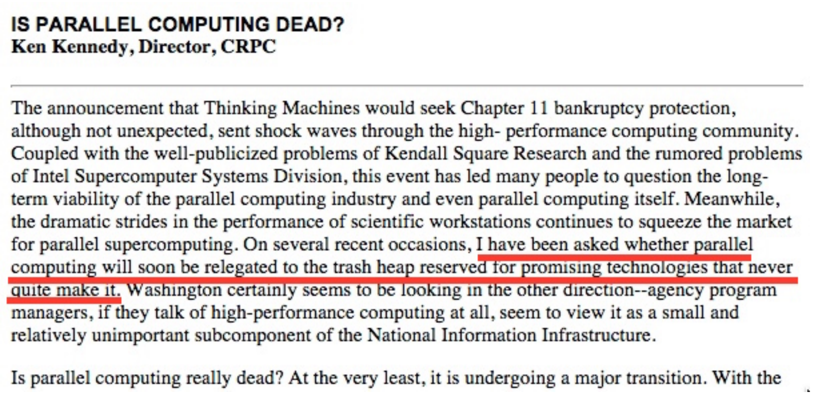
Many papers on scalable operating systems:

- Scalable locks [TOCS 1991]
- Efficient user-level threading [SOSP 1991]
- NUMA memory management [ASPLOS 1996]
- Read-copy update (RCU) [PDCS 1998, OSDI 1999]
- Scalable virtual machines monitor [SOSP 1997]
- ...

[VU, Tanenbaum, 1987]

Uniprocessor performance keeps doubling in phase 3

No real need for expensive parallel machine



IS PARALLEL COMPUTING DEAD?
Ken Kennedy, Director, CRPC

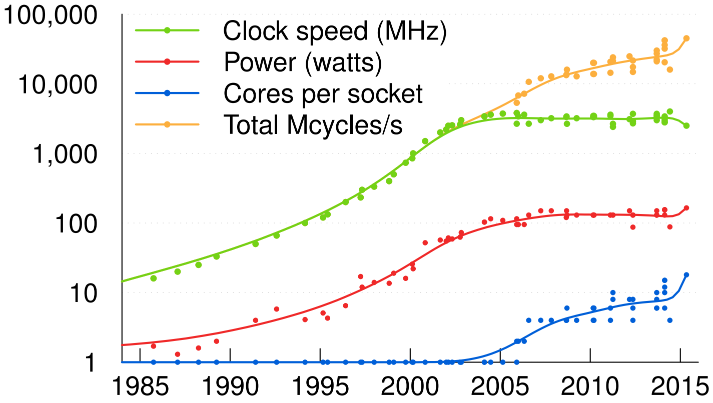
The announcement that Thinking Machines would seek Chapter 11 bankruptcy protection, although not unexpected, sent shock waves through the high- performance computing community. Coupled with the well-publicized problems of Kendall Square Research and the rumored problems of Intel Supercomputer Systems Division, this event has led many people to question the long-term viability of the parallel computing industry and even parallel computing itself. Meanwhile, the dramatic strides in the performance of scientific workstations continues to squeeze the market for parallel supercomputing. On several recent occasions, I have been asked whether parallel computing will soon be relegated to the trash heap reserved for promising technologies that never quite make it. Washington certainly seems to be looking in the other direction--agency program managers, if they talk of high-performance computing at all, seem to view it as a small and relatively unimportant subcomponent of the National Information Infrastructure.

Is parallel computing really dead? At the very least, it is undergoing a major transition. With the

[http://www.crpc.rice.edu/newsletters/oct94/director.html]

Panels at HotOS/OSDI/SOSP

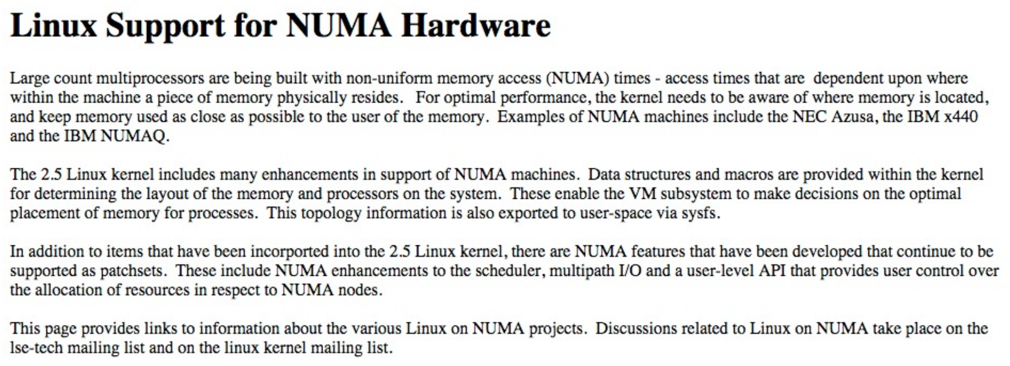
Phase 4: multicore processors



Achieving performance on commodity hardware requires exploiting parallelism

Scalable operating systems return from the dead

Several parallel computing companies switch to Linux



Linux Support for NUMA Hardware

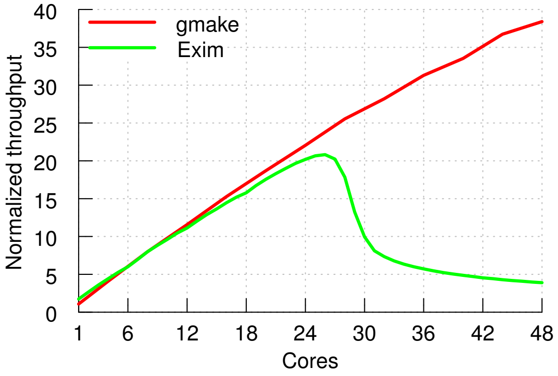
Large count multiprocessors are being built with non-uniform memory access (NUMA) times - access times that are dependent upon where within the machine a piece of memory physically resides. For optimal performance, the kernel needs to be aware of where memory is located, and keep memory used as close as possible to the user of the memory. Examples of NUMA machines include the NEC Azusa, the IBM x440 and the IBM NUMAQ.

The 2.5 Linux kernel includes many enhancements in support of NUMA machines. Data structures and macros are provided within the kernel for determining the layout of the memory and processors on the system. These enable the VM subsystem to make decisions on the optimal placement of memory for processes. This topology information is also exported to user-space via sysfs.

In addition to items that have been incorporated into the 2.5 Linux kernel, there are NUMA features that have been developed that continue to be supported as patchsets. These include NUMA enhancements to the scheduler, multipath I/O and a user-level API that provides user control over the allocation of resources in respect to NUMA nodes.

This page provides links to information about the various Linux on NUMA projects. Discussions related to Linux on NUMA take place on the lse-tech mailing list and on the linux kernel mailing list.

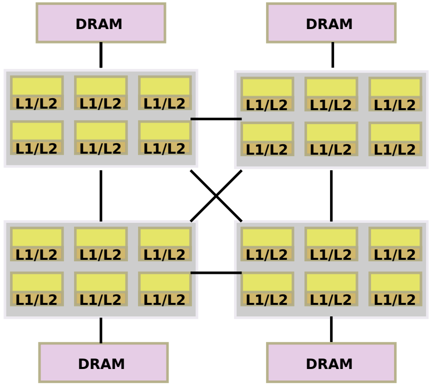
Many applications scale well on multicore processors



But, more applications stress parallelism in operating systems

- Some tickle new scalability bottlenecks
- Exim contends on a *single* reference counter in Linux [OSDI 2010, SOSP 2013]

Cache-line fetches are expensive



Read cache line written by another core: expensive! 100–10000 cycles (contention)

For reference, a creat system call costs 2.5K cycles

Avoiding cache-line sharing is challenging

Consider read-write lock

```
struct read_write_lock {
  int count;          // -1, write mode; > 0, read mode
  list_head waiters;
  spinlock wait_lock;
}
```

Problem: to acquire lock in **read** mode requires **modifying** `count`

- Fetching a remote cache line is expensive
- Many readers can cause performance collapse

Read-copy update (RCU) becomes popular

Readers read shared data without holding **any** lock

- Mark enter/exit read section in per-core data structure

Writer makes changes available to readers using an atomic instruction

- Free node when all readers have left read section

Lots of struggling to scale software [Recent OSDI/SOSP papers]

What will phase 4 mean for OS community?

What will commodity hardware look like?

- 1000s of unreliable cores?
- Many heterogeneous cores?
- No cache-coherent shared memory?

Barrelfish [SOSP 2009]

How to avoid struggling for programmers?

- Exploit transactional memory [ISCA 1993]?
- Develop frameworks for specific domains?
  - MapReduce [OSDI 2004], .., GraphX [OSDI 2014], ...
- Develop principles that make systems scalable by design? [SOSP 2013]

Stepping back: some observations

SOSP/OSDI papers had tremendous impact

- Many ideas can be found in today's operating systems and programming languages

Processes/threads have been good for managing computations

- OS/X 10.10.5 launches 1158 threads, 308 processes on 4-core iMac at boot

Shared memory and locks have worked well for concurrency and parallelism

Events vs. threads – have both?

Rewriting OSes to make them more scalable has worked surprisingly well (so far)

- From big kernel lock to fine-grained parallelism

Summary

Parallelism has moved up the software stack driven by changes in commodity hardware

- More and more programmers are writing parallel code

Today: to achieve performance on commodity hardware programmers **must** use parallelism

- Phase 1: time sharing (foundational ideas)
- Phase 2: client/server (concurrent servers)
- Phase 3: SMPs (parallel kernels and servers)
- Phase 4: multicore (all applications parallel)

Summary

Parallelism has moved up the software stack driven by changes in commodity hardware

- More and more programmers are writing parallel code

Today: to achieve performance on commodity hardware programmers **must** use parallelism

Prediction: Many more SOSP/OSDI papers on parallelism

- Phase 1: time sharing (foundational ideas)
- Phase 2: client/server (concurrent servers)
- Phase 3: SMPs (parallel kernels and servers)
- Phase 4: multicore (all applications parallel)