What parallel programming model do you recommend today to take advantage of the manycore processors of tomorrow? [closed]

Asked 16 years, 3 months ago Modified 5 years, 11 months ago

Viewed 6k times

46

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 2 years ago.

Improve this question

If you were writing a new application from scratch today, and wanted it to scale to all the cores you could throw at it tomorrow, what parallel programming model/system/language/library would you choose? Why?

I am particularly interested in answers along these axes:

1. Programmer productivity / ease of use (can mortals successfully use it?)

2. Target application domain (what problems is it (not) good at?)
3. Concurrency style (does it support tasks, pipelines, data parallelism, messages...?)
4. Maintainability / future-proofing (will anybody still be using it in 20 years?)
5. Performance (how does it scale on what kinds of hardware?)

I am being deliberately vague on the nature of the application in anticipation of getting good general answers useful for a variety of applications.

multicore parallel-processing

Share

Improve this question

Follow

edited Sep 27, 2010 at 2:12

Jeremy Friesner
73k ● 17 ● 141 ● 248

asked Sep 17, 2008 at 4:29

Doctor J
6,312 ● 6 ● 47 ● 41

22 Answers

Sorted by: Highest score (default)

Multi-core programming may actually require more than one paradigm. Some current contenders are:

27

1. MapReduce. This works well where a problem can be easily decomposed into parallel chunks.
2. Nested Data Parallelism. This is similar to MapReduce, but actually supports recursive decomposition of a problem, even when the recursive chunks are of irregular size. Look for NDP to be a big win in purely functional languages running on massively parallel but limited hardware (like GPUs).
3. Software Transactional Memory. If you need traditional threads, STM makes them bearable. You pay a 50% performance hit in critical sections, but you can scale complex locking schemes to 100s of processors without pain. This will not, however, work for distributed systems.
4. Parallel object threads with messaging. This really clever model is used by Erlang. Each "object" becomes a lightweight thread, and objects communicate by asynchronous messages and pattern matching. It's basically true parallel OO. This has succeeded nicely in several real-world applications, and it works great for unreliable distributed systems.

Some of these paradigms give you maximum performance, but only work if the problem decomposes cleanly. Others sacrifice some performance, but allow a wider variety of algorithms. I suspect that some combination of the above will ultimately become a standard toolkit.

Share Improve this answer

Follow

answered Sep 17, 2008 at 15:10

emk
60.9k ● 6 ● 47 ● 51

14

Two solutions I really like are join calculus (JoCaml, Polyphonic C#, Cω) and the actor model (Erlang, Scala, E, Io).

I'm not particularly impressed with Software Transactional Memory. It just feels like it's only there to allow threads to cling on to life a little while longer, even though they should have died decades ago. However, it does have three major advantages:

1. People understand transactions in databases
2. There is already talk of transactional RAM hardware
3. As much as we all wish them gone, threads are probably going to be the dominant concurrency model for the next couple of decades, sad as that may be. STM could significantly reduce the pain.

Share Improve this answer

Follow

answered Sep 17, 2008 at 5:04

Jörg W Mittag
369k ● 79 ● 453 ● 661

"threads are probably going to be the dominant concurrency model for the next couple of decades". From my point of view, threads have basically already disappeared. – J D Jun 4, 2012 at 23:34

11

The mapreduce/hadoop paradigm is useful, and relevant. Especially for people who are used to languages like perl, the idea of mapping over an array and doing some action on each element should come pretty fluidly and naturally, and mapreduce/hadoop just takes it to the next stage and says that there's no reason that each element of the array need be processed on the same machine.

In a sense it's more battle tested, because Google is using mapreduce and plenty of people have been using hadoop, and has shown that it works well for scaling applications across multiple machines over the network. And if you can scale over multiple machines across the network, you can scale over multiple cores in a single machine.

Share Improve this answer Follow

answered Sep 17, 2008 at 4:35

Daniel Papasian
16.4k ● 6 ● 31 ● 32

Annoyingly, I know of no MapReduce implementations which work in parallel (like Hadoop) but which are not distributed. Mongo and CouchDB have map/reduce implementations, but they don't parallelize the computations, and hadoop is a monster to set up if you just want to gain benefit from 4-8 cores in a single box. – Nick Bastin Jun 30, 2009 at 13:17

Map and Reduce are simply functional programming constructs. They can be easily implemented in any major language. Functional languages like F# already have these built in. If you need intra box parallelism use a library for you language. On .Net I have been using Task Parallel Library for quite sometime. – Steve Severance Aug 29, 2009 at 19:16

@Steve: Note that `MapReduce` is not `reduce g (map f xs)` but, rather, something much more complicated. – J D May 23, 2010 at 18:34

10

For .NET application I choose ".NET Parallel Extensions (PLINQ)" it's extremely easy to use and allows me to parallelize existing code in minutes.

1. It's simple to learn
2. Used to perform complex operations over large arrays of objects, so I can't comment on other applications
3. Supports tasks and piplines
4. Should be supported for a next couple of years, but who knows for sure?
5. CTP version has some performance issues, but already looks very promising.

Mono will likely get support for PLINQ, so it could be a cross-platform solution as well.

Share Improve this answer Follow

edited Sep 17, 2008 at 4:50

answered Sep 17, 2008 at 4:33

aku
124k ● 33 ● 176 ● 203

6

For heavy computations and the like, purely functional languages like Haskell are easily parallelizable without any effort on the part of the programmer. Apart from learning Haskell, that is.

However, I do not think that this is the way of the (near) future, simply because too many programmers are too used to the imperative programming paradigm.

Share Improve this answer Follow

answered Sep 17, 2008 at 4:46

Thomas
181k ● 55 ● 376 ● 501

2 -1: "purely functional languages like Haskell are easily parallelizable without any effort on the part of the programmer". Only if "parallelizable" means burning all of your cores without getting decent performance. In reality, purely functional languages like Haskell seriously impede parallelism. – J D May 23, 2010 at 18:41

1 I elaborated recently in a criticism of some published research on this subject: flyingfrogblog.blogspot.com/2010/06/… – J D Jun 5, 2010 at 11:08

2 In essence, purely functional languages make heavy use of immutable data structures. That places far more stress on the garbage collector (which has serial components) and memory bandwidth due to the unnecessary copying. The result is cache unfriendliness that forces every core to contend for the machine's shared memory, destroying scalability. That is why the Haskell guys only got 2.6× speedup on 8 cores when other published results show up to 5-7× speedups. – J D Jun 5, 2010 at 11:12

1 FWIW, the guys at MIT who produced Cilk (which was recently bought by Intel) explain in their papers about cache oblivious algorithms that it is essential to mutate data structures in place if you want to avoid that effect. – J D Jun 5, 2010 at 11:14

4 Note that the research Jon refers to isn't yet published as he claims, and one of the authors went to some effort to rebut his criticism in the comments on previous entry on Jon's blog. – Ganesh Sittampalam Jun 5, 2010 at 20:20

5

kamaelia is a **python framework** for building applications with lots of communicating processes.

>
>
> ## Kamaelia - Concurrency made useful, fun
>
> In Kamaelia you build systems from **simple components that talk to each other**. This speeds development, massively aids maintenance and also means you **build naturally concurrent software**. It's intended to be accessible by **any** developer, including novices. It also makes it fun :)

> What sort of systems? Network servers, clients, desktop applications, pygame based games, transcode systems and pipelines, digital TV systems, spam eradicators, teaching tools, and a fair amount more :)

See also the Question Multi-Core and Concurrency - Languages, Libraries and Development Techniques

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 ● 1

answered Sep 18, 2008 at 1:20
Sam Hasler
12.6k ● 10 ● 73 ● 106

4

I'm betting on communicating event loops with promises, as realized in systems like Twisted, E, AmbientTalk, and others. They retain the ability to write code with the same execution-model assumptions as non-concurrent/parallel applications, but scaling to distributed and parallel systems. (That's why I'm working on Ecru.)

Share Improve this answer Follow

answered Sep 17, 2008 at 4:36
Allen
5,090 ● 24 ● 30

Check out Erlang. Google for it, and watch the various presentations and videos. Many of the programmers and

2

architects I respect are quite taken with its scalability. We're using it where I work pretty heavily...

Share Improve this answer Follow

answered Sep 17, 2008 at 4:40

Kevin Little
12.9k ● 6 ● 42 ● 48

There's also Reia for Erlang wiki.reia-lang.org/wiki/Reia_Programming_Language – Sam Hasler Oct 2, 2008 at 15:30

2

As mentioned, purely functional languages are inherently parallelizable. However, imperative languages are much more intuitive for many people, and we are deeply entrenched in imperative legacy code. The fundamental issue is that pure functional languages express side-effects explicitly, while side effects are expressed implicitly in imperative languages by the order of statements.

I believe that techniques to declaratively express side effects (e.g., in an object oriented framework) will allow compilers to decompose imperative statements into their functional relationships. This should then allow the code to be automatically parallelized in much the same way pure functional code would be.

Of course, just as today it is still desirable to write certain performance-critical code in assembly language, it will still be necessary to write performance-critical explicitly parallel code tomorrow. However, techniques such as I outlined should help automatically take advantage of manycore architectures with minimal effort expended by the developer.

Share Improve this answer Follow

answered Sep 17, 2008 at 15:32
smo
913 ● 7 ● 11

2

I'm surprised nobody has suggested MPI (Message Passing Interface). While designed for distributed memory, MPI programs with essential and frequent global coupling (solving linear and nonlinear equations with billions of unknowns) have been shown to scale to 200k cores.

Share Improve this answer Follow

answered Mar 18, 2009 at 14:53
Jed
1,701 ● 18 ● 26

2

This question seems to keep appearing with different wording - maybe there are different constituencies within StackOverflow. [Flow-Based Programming](FBP) is a concept/methodology that has been around for over 30 years, and is being used to handle most of the batch processing at a major Canadian bank. It has thread-based implementations in Java and C#, although earlier implementations were fiber-based (C++, and mainframe Assembler - the one used at the bank). Most approaches to the problem of taking advantage of multicore involve trying to take a conventional single-threaded program and figure out which parts can run in parallel. FBP takes a different approach: the application is designed from the start in terms of multiple "black-box" components running asynchronously (think of a manufacturing assembly line). Since the interface between components is data streams, FBP is essentially language-independent, and therefore supports mixed-language applications, and domain-specific languages. For the same reason, side-effects are minimized. It could also be described as a "share nothing" model, and a MOM (message-oriented middleware). MapReduce seems to be a special case of FBP. FBP differs from Erlang mostly in that Erlang operates in terms of many short-lived threads, so green threads are more appropriate there, whereas FBP uses fewer (typically a few 10s to a few hundred) longer-lived threads. For a *piece* of a batch network that has been in daily use for over 30 years, see part of batch network. For a high-level design of an interactive app, see Brokerage app high-level design. FBP has been found to result in much more maintainable applications, and improved elapsed times - even on single core machines!

Share Improve this answer Follow

answered Jun 2, 2009 at 15:55

Paul Morrison
1,724 ● 3 ● 21 ● 35

1

Qt concurrent offers an implementation of MapReduce for multicore which is really easy to use. It is multiOS.

Share Improve this answer Follow

answered Sep 17, 2008 at 4:34

fulmicoton
15.9k ● 10 ● 55 ● 74

1

If your problem domain permits try to think about a share nothing model. The less you share between processes and threads the less you have to design complex concurrency models.

Share Improve this answer Follow

answered Sep 17, 2008 at 4:59

Daniel
563 ● 5 ● 11

1

See also the question [Multi-Core and Concurrency - Languages, Libraries and Development Techniques]()

Share Improve this answer Follow

edited May 23, 2017 at 11:51

Community Bot
1 ● 1

answered Oct 23, 2008 at 11:08

Sam Hasler
12.6k ● 10 ● 73 ● 106

1

A job-queue with multiple workers system (not sure of the correct terminology - message queue?)

Why?

Mainly, because it's an absurdly simple concept. You have a list of stuff that needs processing, then a bunch of processes that get jobs and process them.

Also, unlike the reasons, say, Haskell or Erlang are so concurrent/parallelisable(?), it's entirely language-agnostic - you can trivially implement such a system in C, or Python, or any other language (even using shell scripting), whereas I doubt bash will get software transactional memory or join-calculus anytime soon.

Share Improve this answer Follow

answered Jun 2, 2009 at 16:48

dbr
169k ● 69 ● 283 ● 347

As long as you have giant chunks of parallel work to schedule, this works great. Can you find those giant chunks? – Ira Baxter Aug 29, 2009 at 20:31

1

I really like the model Clojure has chosen. Clojure uses a combination of immutable data structures and software transactional memory.

Immutable data structures are ones that never change. New versions of the structures can be created with modified data, but if you have a "pointer" to a data structure, it will never change out from under you. This is good because you can access that data without worrying about concurrency problems.

Software transactional memory is discussed elsewhere in these answers but suffice it to say that it is a mechanism whereby multiple threads can all act upon some data and if they collide, one of the threads is rolled back to try again. This allows for much faster speed when the risk of collision is present but unlikely.

There is a video from author Rich Hickey that goes into a lot more detail.

Share Improve this answer Follow

answered Aug 28, 2009 at 4:18

Steve Rowe
19.4k ● 9 ● 53 ● 82

1

A worthwhile path might be OpenCL, which provides a means of distributing certain kinds of compute loads across heterogeneous compute resources, IE the same code will run on a multicore CPU and also commodity GPU's. ATI has recently released exactly such a toolchain. NVidia's CUDA toolchain is similar, although somewhat more restricted. It also appears that Nvidia has an OpenCL sdk in the works

It should be noted that this probably won't help much where the workloads are not of a data-parallel nature, for instance it won't help much with typical transaction processing. OpenCL is mainly oriented toward the kinds of computing that are math intensive, such as scientific/engineering simulation or financial modeling.

Share Improve this answer Follow

answered Aug 28, 2009 at 4:52

SingleNegationElimination

156k ● 35 ● 268 ● 306

1

> If you were writing a new application from scratch today, and wanted it to scale to all the cores you could throw at it tomorrow, what parallel programming model/system/language/library would you choose?

Perhaps the most widely applicable today is Cilk-style task queues (now available in .NET 4). They are great for problems that can be solved using divide and conquer with predictable complexity for subtasks (such as parallel `map` and `reduce` over arrays where the complexity of the function arguments is known as well as algorithms like quicksort) and that covers many real problems.

Moreover, this only applies to shared-memory architectures like today's multicores. While I do not believe this basic architecture will disappear anytime soon I do believe it must be combined with distributed parallelism at some point. This will either be in the form of a cluster of multicores on a manycore CPU with message passing between multicores, or in the form of a hierarchy of cores with predictable communication times between them. These will require substantially different programming models to obtain maximum efficiency and I do not believe much is known about them.

Share Improve this answer Follow

edited Jun 5, 2010 at 11:16

answered May 23, 2010 at 19:23

J D

48.6k ● 14 ● 174 ● 277

1

We've been using PARLANSE, a parallel programming langauge with explicit partial-order specification of concurrency for the last decade, to implement a scalable program analysis and transformation system (DMS Software Reengineering Toolkit) that mostly does symbolic rather than numeric computation. PARLANSE is a compiled, C-like language with traditional scalar data types character, integer, float, dynamic data types string and array, compound data types structure and union, and lexically-scoped functions. While most of the language is vanilla (arithmetic expressions over operands, if-then-else statements, do loops, function calls), the parallelism is not. Parallelism is expressed by defining a "precedes" relation over blocks of code (e.g, a before b, a before c, d before c) written as

```
(|;  a  (... a's computation)
     (<< a) b ( ... b's computation ... )
     (<< a) c ( ....c's computation ...)
     (>> c) d ( ... d's computation...)
)|;
```

where the << and >> operators refer to "order in time". The PARLANSE compiler can see these parallel computations and preallocate all the structures necessary for computation grains a,b,c,d, and generate custom code to start/stop each one, thus minimizing the overhead to start and stop these parallel grains.

See this link for *parallel* iterative deepening search for optimals solutions to the 15-puzzle, which is the 4x4 big-brother of the 8-puzzle. It only uses *potential* parallel as a parallelism construct **(|| a b c d)** which says there are no partial order constraints on the computations *a b c d*, but it also uses speculation and asynchronously aborts tasks that won't find solutions. Its a lot of ideas in a pretty small bit of code.

PARLANSE runs on multicore PCs. A big PARLANSE program (we've built many with 1 million+ lines or more) will have thousands of these partial orders, some of which call functions that contain others. So far we've had good results with up to 8 CPUs, and modest payoff with up to 16, and we're still tuning the system. (We think a real problem with larger numbers of cores on current PCs is memory bandwidth: 16 cores pounding a memory subsystem creates a huge bandwidth demand).

Most other languages don't expose the parallelism so it is hard to find, and the runtime systems pay a high price for scheduling computation grains by using general-purpose scheduling primitives. We think that's a recipe for disaster or at least poor performance because of Amhdahl's law: if the number of machine instructions to schedule a grain is large compared to the work, you can't be efficient. OTOH, if you insist on computation grains with many machine instructions to keep the scheduling costs relatively low, you can't find computation grains that are independent and so you don't have any useful parallelism to schedule. So the key idea behind PARLANSE is to minimize the cost of scheduling grains, so that grains can be small, so that there can be many of them found in real code. The insight into this tradeoff came from the abject failure of the pure dataflow paradigm, that did everything in parallel with tiny parallel chunks (e.g., the add operator).

We've been working on this on and off for a decade. Its hard to get this right. I don't see how folks that haven't been building parallel langauges and using/tuning them for this time frame have any serious chance of building effective parallel systems.

Share Improve this answer Follow

edited May 23, 2017 at 11:48

Community Bot

1 ● 1

answered Aug 28, 2009 at 3:53

Ira Baxter

95.2k ● 23 ● 181 ● 353

1

There are three parts to parallel programming IMO: identify the parallelism and specify the parallelism. Identify=Breaking down the algorithm into concurrent chunks of work, specify=doing the actual coding/debugging. Identify is independent of which framework you will use to specify the parallelism and I don't think a framework can help there much. It comes with good understanding of your app, target platform, common parallel programming trade-offs (hardware latencies etc), and MOST importantly experience. Specify however can be discussed and this is what I try to answer below:

I have tried many frameworks (at school and work). Since you asked about multicores, which are all shared memory, I will stick with three shared memory frameworks I have used.

Pthreads (Its no really a framework, but definitely applicable):

Pro: -Pthreads is extremely general. To me, pthreads are like the assembly of parallel programming. You can code any paradigm in pthreads. -Its flexible so you can make it as high performance as you want. There are no inherent limitations to slow you down. You can write your own constructs and primitives and get as much speed as possible.

Con: -Requires you to do all the plumbing like managing work-queues, task distribution, yourself. -The actual syntax is ugly and your app often has a lot of extra code which makes code hard to write and then hard to read.

OpenMP:

Pros: -The code looks clean, plumbing and task-splitting is mostly down under the hood -Semi-flexible. It gives you several interesting options for work-scheduling

Cons: -Meant for simple for-loop like parallelism. (The newer Intel verion does support tasks too but the tasks are the same as Cilk). -Underlying structures may or may not be well-written for performance. GNUs implementation is just ok. Intel's ICC worked better for me but I would rather write some stuff myself for higher performance.

Cilk, Intel TBB, Apple GCD:

Pros: -Provably optimal underlying algorithms for task-level parallelism -Decent control of serial/parallel tasks -TBB also has a pipeline parallelism framework which I used (it isn't the best to be frank) -Eliminates the task of writing a lot of code for task-based systems which can be a big plus if you are short-handed

Cons: -Less control of underlying structures' performance. I know the Intel TBB has very poorly-performing underlying data structures, e.g., the work queue was bad (in 2008 when I saw it). -The code looks awful sometimes with all the keywords and buzzwords they want you to use -Requires reading a lot of references to understand their "flexible" APIs

Share Improve this answer

Follow

answered May 10, 2011 at 0:27

Aater Suleman

2,328 ● 18 ● 11

0

I'd use Java - its portable so future processors wont be a problem. I'd also code my application with layers seperating interface / logic and data (more like 3 tier web app) with standard mutex routines as a library (less debugging of parallel code). Remember that web servers scale to many processors really well and are the least painful path to multicore. Either that or look at the old Connection Machine model with a virtual processor tied to data.

Share Improve this answer

Follow

answered Sep 17, 2008 at 4:34

zurk

259 ● 1 ● 2 ● 5

0

Erlang is the more "mature solution" and is portable and open source. I fiddled around with Polyphonic C# , I don't know how it would be to program everyday in it.

There are libraries and extensions for almost every language/OS under the sun, google transactional memory . It's an interesting approach from MS.

Share Improve this answer

Follow

answered Oct 23, 2008 at 11:18

AlePani

519 ● 2 ● 7