Annals of R.S.C.B., ISSN:1583-6258, Vol. 25, Issue 3, 2021, Pages. 8924 - 8934
Received 16 February 2021; Accepted 08 March 2021.

Cache Problems in Parallel Computational Processes

OybekMallayev*,[1], Bunyodbek Anvarjonov[2], Mukhamedaminov Aziz[1]

[1]Tashkent university of Information technologies named after Muhammad al-Khwarizmi, Tashkent, Uzbekistan

[2]Tashkent State Transport University, Tashkent, Uzbekistan

info-oybek@rambler.ru

ABSTRACT

The article discusses the most common problems of computer parallel computing processes and ways to solve them. Examples of these issues are bandwidth, cache handling, memory conflicts, cache problems, false sharing, and memory consistency. Also presented are the problems that arise in the processes of information exchange between threads, and ways to solve them. Methods for solving these problems using the capabilities of parallel programming are proposed. In addition, the most important process in parallel programming was discussed - the problem of synchronization. Synchronization is a mechanism that allows you to impose restrictions on the flow order. By synchronizing, the relative order of the threads is regulated and any conflict between the threads that could lead to undesirable program behaviour is resolved.

1. Introduction

Concurrency in programs is a way to control the distribution of shared resources used simultaneously. Concurrency in programs is important for several reasons [1]:

• Parallelism allows [2] you to spend system resources the most in an efficient manner. Efficient use of resources is the key to increase the performance of computer systems [3].

• Many software issues allow for simple, competitive implementations. Competition is an abstraction of the implementation of software algorithms or applications that are inherently parallel.

It should be noted here that the terms parallel and competitive in the world of parallel programming [4] are not used interchangeably. When several software threads work in parallel, this means that active threads are executed simultaneously on different hardware resources (processing elements). Multiple program threads can be executed at the same time [5]. When several software threads work competitively, the execution of threads alternates on one hardware resource. Active threads are ready for execution, but only one thread can be executed at any given time. To achieve concurrency, the competitive operation of several hardware resources is required.

Threads are used by the operating system in large quantities for its own internal needs, so even if you are writing a single-threaded application, it will have many threads at run time [6]. All major programming languages today support the use of threads, be it imperative languages (C, Fortran, Pascal, Ada), object-oriented (C ++, Java, C #), functional (Lisp, Miranda, SML) or logical (Prolog).

Two synchronization options are widely used: mutual exclusion and conditional synchronization [7]. In the case of mutual exclusion, one thread blocks a critical section (a code area that contains common data), as a result, one or more threads wait for their turn to enter this area. This is useful when two or more threads share the same memory space and execute simultaneously [8].

Despite the fact that there are quite a few synchronization methods, only developers regularly use a few of them. The methods used are also determined to some extent by the programming environment [9].

When most of us do manual calculations, performance is limited by our ability to quickly read, rather than quickly read and write. Older microprocessors [10] had a similar limitation. In recent decades, microprocessors have added significantly more speed than memory. A single microprocessor core is capable of performing hundreds of operations in the time it takes to read or write a value to main memory [11]. The performance of the memory, rather than the processor, now often becomes the bottleneck of programs. Multi-core processors can exacerbate the problem if you do not take care of saving memory bus bandwidth and avoiding memory conflicts [12].

More rare data movement operations are a more subtle exercise than packaging, because the widely used programming languages do not have explicit commands for moving data between the kernel and memory. Data movement depends on the way the cores are read from memory and written to memory. There are two categories of interaction that should be considered: between kernels and memory and between kernels [13].

2. Statement of the problem. detecting problems that occur when synchronizing parallel processing of signals in multi-core processors

Parallel algorithms [14] and programs, methods for their implementation in multi-core processors are of great interest to scientists and specialists working in various fields, such as communications and control systems, radio engineering and electronics, acoustics and seismology, geophysics, broadcasting and television, measurement technology and instrumentation. This also includes relatively new directions in the creation of hardware and

Annals of R.S.C.B., ISSN:1583-6258, Vol. 25, Issue 3, 2021, Pages. 8924 - 8934
Received 16 February 2021; Accepted 08 March 2021.

software, such as processing audio and video signals, speech recognition, biometric systems, processing of dynamic images, multimedia learning systems [15].

Real-time operating conditions place increased demands on these systems in terms of processing speed and data transfer. The hardware environment of such systems cannot be bulky, multi-machine; compact, and often embedded and even mobile processing tools are required. For such requirements, the most effective solution is the use of multi-core processors with high-speed algorithms for processing, packaging and data transfer.

The following concepts are important for synchronizing parallel signal processing in multi-channel processors:

• Memory problems;

• Bandwidth;

• Work with the cache;

• Memory conflicts;

• Cache problems;

• False separation;

• Memory consistency;

For multi-core programs, working inside the cache becomes more difficult, because data is transferred not only between the kernel and memory, but also between the kernels. As with transfers to and from memory, major programming languages do not do this explicitly. Such transfers are performed implicitly because of read and write sequences from various cores. In this case, two types of data dependence arise:

• Read-write dependency. The kernel writes a cache line, and then another kernel reads it;

• Double-write dependency. The kernel writes a cache line, and then another kernel writes it.

As already noted in the discussion of time slicing problems, high performance is achieved when processors get most of their data from the cache, and not from the main memory. For sequential programs, modern caches usually work, well without any special tricks, although a little tweak does not hurt them either. In the case of parallel programming, caches hide much more serious problems.

The smallest unit of memory that two processors are lazy is the cache line (or sector). Two separate processors can share a cache line when both of them need to read it, but if the line is written to one cache and is read from the other, then it must be sent between the caches (even if the required addresses are not adjacent).

When executing a sequential program, the memory at any given time has a completely definite state. This is called consistent consistency. In parallel programs, it all depends on the point of view. Two writes to memory from a hardware stream from another stream can be seen in a different order. The reason is that when a hardware stream writes to memory, the data being written goes through a chain of buffers and caches before it reaches the main memory.

3. Solution Method

Some compilers support the #pragma pack directives, which pack structures even denser, probably by refusing any filling. However, such a very tight packaging can be ineffective because it causes unaligned downloads and saves, which can be significantly slower than aligned load and save.

Data transfer between the kernel and memory occurs in single-core processors, so minimizing the number of data movement operations is beneficial for sequential programs as well. There are many ways. For example, when dividing into blocks with ignoring the cache, the problem is recursively divided into smaller and smaller problems. Ultimately, the problems become so small that each of them is cached. This approach is described in. Another way to reduce cache size is to reorder the code. Sometimes for this it is enough just to rearrange the cycles. In other cases, more substantial restructuring is required.

In Lis.2 shows how a program can be restructured to adapt it to work with the cache. Instead of directly representing an imaginary sieve in the form of one large array, it is presented as a small window. Window size is approximately bytes. Restructuring requires that the original inner loop be stopped when it reaches the end of the window and restarted when the next window is processed. The striker array stores the indices of these paused loops and has an element for every prime number up to. Data structures grow much slower than n, and therefore are placed in a 106-byte cache even when n approaches values of the order of 1011. Obviously, it is almost impossible to place a composite array of 1011 bytes in memory (on most machines). A later discussion of sieve threading shows how to reduce the size of a composite array to instead of n bytes.

```
Inline long Strike (boll composite[],
long i, long stride, long limit ) {
for( ; i<=limit; i+=stride )
composite[i] = true;
return i; }
long CacheUnfriendlySieve( long n ) {
long count = 0;
long m = (long)sqrt((double)n);
bool* composite - new bool[n+l];
memset( composite. 0. n ):
for( long i=2; i<=m; ++i )
if( !composite[i] ) {
++count;
Strike(composite. 2*i, i, n );
}
for( long i=m+l; i<=n; ++i )
if( !composite[i] )
++count;
deleted composite;
return count;
}
```

Listing 1. Badly adapted for cache.

```
longCacheFriendlySieve( long n ) {
long count - 0;
long m - (long)sqrt((double)n);
bool* composite - new bool[n+l];
memset( composite. 0. n );
long* factor - new long[m];
long* striker - new long[m];
long n_factor - 0;
for( long i=2; i<=m; ++i )
if( !composite[i] ) {
++count;
striker[n_factor]=Strike(composite.2*i,i,m
);
factor[n_factor++] - i; }
for( long window=m+l; window<=n;
window+=m ) {
long limit = min(window+m-l,n);
for( long k=0; k<n_factor; ++k )
striker[k] - Strike( composite, striker[k],
factor[k], limit );
for( long i=window; i<=limit; ++i )
if( !composite[i] )
++count; }
delete[] striker;
deleted factor;
deleted composite;
return count; }
```

Listing2. Adapted for Cache.

Thus, if several cores only read the cache line and do not write to it, then the memory bandwidth is not consumed. Each core simply stores its own copy of the cache line.

Consider writing a multi-threaded version of the CacheFriendlySieve function in Lis.2. A good decomposition of the problem was the sequential filling of the factor array and subsequent parallel processing of the windows. The serial part will take O (Vic) time, which will have some insignificant effect on performance at large values of n. Parallel window processing requires some data to be shared. If you understand the essence of this use, it will help you write a parallel version.

• Once populated, the factor array is read-only. Therefore, each thread can share this array with others.

• The composite array is updated when finding prime numbers. However, updates are made in different windows, so they are unlikely to collide (excluding the borders of the windows falling into the cache line). Even better, the values in the window are only required when processing the window. The composite array no longer needs to be shared; instead, each stream can have its private part, which includes only the window of interest to it. This change is also useful for the serial version, since now the memory requirements for the sieve are reduced from 0 (n) to 0 (Vfl). This reduction makes it possible to calculate primes up to 1011 even on a 32-bit computer.

• The variable count is updated when primes are found. An atomic increment could be used, but this would lead to competition for memory. The best solution is the one shown in the example - let each thread do its own partial counting and add these partial counters at the end.

• The striker array is updated during window processing. Each thread will need its own private copy. The trick is that striker causes a dependency between windows on iteration of the loop. For each window, the initial striker value is the last value that the array had in the previous window. In order to break this dependence, the initial striker values must be re-calculated. This calculation is simple. The purpose of striker [k] is to track the current multiple of factor [k].

• The base variable is new in the parallel version. It tracks the start of the window for which the striker array is valid. If the base value is different from the beginning of the window being processed, this means that the stream must calculate striker again. This recalculation sets the initial value of striker [k] to the minimum multiple of factor [k], which is either in or out of the window. The multi-threaded sieve is shown in Lis.3 As a further improvement, which would halve the amount of work, look for only odd primes. This was not done in the examples, since it would complicate not following the problems a lot of threading.

```
longParallelSieve( long n ) {
long count - 0;
long m - (long)sqrt((double)n);
long n_factor - 0;
long* factor - new long[m];
#pragma ompparallel{
bool* composite - new bool[m+l];
long* striker - new long[m];
#pragma omp single {
memset( composite, 0, m );
for( long i«2; i<-m; ++i )
if( !composite[i] ) {
++count:
Strike(composite, 2*i, i, m );
factor[n factor++] - i; }}
long base - -1;
#pragma omp for reduction (+:count)
for( long windowm+1: window<=n;
window+»m ) {
memset( composite, 0, m );
if( base!=window ) {
for( long k=0; k<n_factor; ++k )
striker[k] - (base+factor[k]-l)/factor[k]
*factor[k] - base; }
long limit - min(window+m-l,n) - base;
for( long k=0; k<n_factor; ++k )
striker[k] – Strike(composite, striker[k],
factor[k], limit ) - m;
for( long i-0; i<-limit; ++i )
if( !composite[i] ) ++count;
base += m; }
<telete[] striker;
deleteC] composite; }
delete[] factor;
return count;
}
```

Listing 3. Parallel sieve.

Some compilers support alignment pragmas. Windows compilers have a declspec (align (n)) directive that allows you to specify alignment by the nth byte. With dynamic memory allocation, alignment can be achieved by providing additional padding bytes and returning a pointer to the next cache line in the block. In Lis.4 shows an example of such a memory allocation program. The CacheAlignedMalloc function uses the word immediately before the aligned block to store a pointer to the true base address of the block so that the CacheAlignedFree function can free the correct block. Note that if mallos returns a aligned pointer, then the CacheAlignedMallos function still rounds its top to the next cache line, since it needs the first cache line to store a pointer to a true base address.

```
// Allocation of a block of memory that starts a cache line
void* CacheAlignedMa1loc( size J: bytes, void* hint ) {
sizej: m - (cache line size in bytes);
assert( (m & m-l)=«0 ); // m must be a power of two
char* base - (char*)malloc(m+bytes);
// Round the pointer up to the next line.
char * result - (char*)((UIntPtr)(base+m)&-m);
// We write down where the block really begins.
((char**)result)[-l] - base;
return result; }
// Free the block allocated by the CacheAlignedMalloc function.
voidCacheAlignedFree( void* p ) {
```

```
// We restore the point of the real start of the block
char* base - ((byte**)p)[-l];
// The failure of the next call indicates that the memory was not allocated by the
CacheAlignedMa11os function..
assert( (void*)((UIntPtr)
(base+NFS_LineSize)&-NFSJ.ineSize) — p);
free( base );
}
```

Listing 4. A program for allocating memory blocks aligned on cache line boundaries.

It may not be obvious at all that there is always enough space in front of the aligned block to hold the pointer. Sufficiency of space depends on two assumptions:

• The size of the cache line is not less than the size of the pointer.

• A request of 11 so for a number of bytes equal to at least the size of the cache returns a pointer aligned on the border that is a multiple of size of (char *).

When writing portable code in a high-level language, the easiest way to reconcile memory is to use the synchronization primitives that exist in the language, which usually have built-in memory fencing mechanisms of the required type. Memory consistency problems arise only when programmers try to create their own synchronization primitives. If you need to implement your own synchronization, then the rules depend on the language and hardware.

The complexities of memory, cache, and pipeline interactions can seem daunting. Therefore, it can be helpful to imagine program memory divided into four types of regions:

1. Private area of the stream. Areas private for a given thread are never shared with other threads. The hardware thread tends to keep this memory in cache. Therefore, accessing private areas of the stream is very fast and does not consume bus bandwidth.

2. The general area of the stream is read-only. These areas are shared by multiple threads, but these threads never write to them.

3. The area of exclusive access. Exclusive areas are read and written, but are protected by a lock. As soon as a thread captures (locks) an area and starts working with data, the data is moved to the cache. After the area is freed, the data is moved back into memory, or it moves to the next hardware thread that grab the area.

4. Region of the Wild West. The wild west region is read and written by unsynchronized streams. Depending on how the locks are locked, these areas may contain the lock objects themselves, because by their nature, locks are accessed from unsynchronized threads, which are designed to synchronize the lock. Whether a locking object belongs to the Wild West region depends on whether the actual locked "content" is in that object, or it is just a pointer to another location (where the actually locked "content" is stored).

The key to successful parallel programming is good program decomposition. Consider the following key points when choosing a decomposition option:

· Keep the number of usable software currents consistent with the available number of hardware threads. Never hard-code the number of threads into your program code, leave this value as a tunable program parameter.

· Parallel programming for high productivity is always a search between too little and too much synchronization. Too little synchronization will lead to incorrect results, too much to slow acquisition.

· Use tools like Intel Thread Checker to detect race conditions.

· Keep your locks private. Do not lock the resource when calling code from another software package.

· Avoid deadlocks. To do this, seize resources in a consistent manner.

Annals of R.S.C.B., ISSN:1583-6258, Vol. 25, Issue 3, 2021, Pages. 8924 - 8934
Received 16 February 2021; Accepted 08 March 2021.

- When choosing a decomposition option for parallel execution, remember to consider memory bandwidth and contention issues. Pack your data tightly to reduce used bandwidth and cache space. Place data for different processors on different cache lines. Separate read-only shared data from written data.
- Distribute contentions using multiple distributed locks (where possible).
- Non-blocking algorithms have both advantages and disadvantages. They are especially difficult in languages that do not support garbage collection. Therefore, proceed with caution.
- Cache lines are quanta of information exchange between hardware threads.
- If you are writing your own synchronization code, understand the memory matching model used on your platform.

Serialization commands (such as atomic and memory fencing operations) can be useful, but are relatively expensive compared to other commands.

There is another major concurrency issue. This is called a trace buffer. To illustrate this point, consider the two streams shown in List. 5.

Listing5. Two threads write events to the trace buffer

```
unsigned _stdcall Thread (void *) {
m_global = do_wok ();
AddEntryToTraceBuffer (msg);}
unsigned _stdcall Thread2 (void *) {
Thread_local_data = m_global;
AddEntryToTraceBuffer (msg); }
```

It should now be clear what the problem is. There is a race condition between the two threads and access to the trace buffer. Thread 1 can write the global data value and then start registering this write event in the trace buffer, at the same time thread 2 can read this global value after writing, by registering this read event before the write event. Thus, the buffer data may not accurately reflect the actual sequence of events in the system.

One possible solution to the problem is to secure the operation that you want to register (as well as the subsequent access to the trace buffer) with a synchronization object. A thread might have requested exclusive access to the trace buffer when registering an event. After it finishes registering the event, it unlocks the trace buffer, allowing other threads to access the buffer. The described scheme is implemented in List. 6.

Listing6. Incorrect synchronization of trace buffer access

```
unsigned _stdcall Thread1 (void *){
LockTraceBuffer ();
m_global = do_work ();
ASdEntryToTraceBuffer (msg):
UnlockTraceBuffer (); }
unsigned _stdcall Thread2 (void *) {
LockTraceBuffer ();
Thread_local_data = m_global;
AddEntryToTraceBuffer (msg);
UnlockTraceBuffer (); }
```

This approach has several disadvantages. Using the "synchronization primitive" to secure access to the trace buffer can actually hide errors in the code, which discredits the very idea of using the trace buffer for debugging. Suppose the error the developer is looking for is a thread not having a read or write lock. By blocking access to the trace buffer, the developer protects a critical section of code that could be mistakenly left unprotected. Generally speaking, when tracking race conditions, the programmer should avoid synchronizing trace buffer access. If you are syncing access and your application is running, this is a hint that there may be a problem with the synchronization mechanism between threads.

The preferred method to overcome this limitation is to log the message before and after the event occurs.

4. Results

Research has been conducted on the above problems in multi-processor systems. The results of the study are presented in Table 1 below:

Table 1. Parallel process problems and their solutions.

Concepts of synchronizing	Solutions		
	Available method (solves the problem)	New method (solves the problem)	Achievement
memory problems;	85%	95%	10%
bandwidth;	75%	88%	13%
work with the cache;	80%	96%	16%
memory conflicts;	85%	98%	13%
cache problems;	80%	97%	17%
false separation;	75%	95%	20%
memory consistency;	90%	98%	8%

Table 1 shows the results and indicators of success in solving problem situations that arise in parallel processes using existing and proposed methods. For example, 85% of memory problems can be solved with the existing method. The proposed method is possible 95% solution. The increase was 10%. The maximum increase was 20%, and the minimum was 8%. New methods are currently being investigated to deal with false partitioning and caching problems, which are the most common problems in concurrent processes.

Conclusion

By logging pre- and post-event messages, the programmer can determine whether events have occurred as expected or not. If the messages before and after the event follow each other, then the developer can confidently assume that the events also happened one after the other. If the messages before and after the event alternate, then the order of occurrence of events is overridden, that is, events could occur in one of two sequences.

The trace buffer can be used to collect useful data about the sequence of operations occurring in a multithreaded application. More complex problems may require more advanced threading debugging tools. These means are discussed in the next article.

The location of the area may change during program execution. For example, a thread can privately create a lookup table and then publish its position to other threads so that it becomes a read-only table. Different types of regions should not be mixed on the same cache line, because there is a problem with false separation. For example, if you put the private stream and wild west data in one cache line, it will interfere with access to the stream's private area, since wild west data accesses will result in a ping-pong cache line situation.

In conclusion, we note that solving the most common problems of parallel programming using the methods proposed in the article is important when creating parallel programming tools. It is also possible to further improve the proposed methods based on modern processor architecture.

References

[1] V.P. Gergel. Vysokoproizvoditel'nyye vychisleniya dlya mnogoyadernykh mnogoprotsessornykh system, Textbook, NNGU, 2010, p. 364.

[2] Yu. E. Voskoboinikov, A. V. Gochakov, A. B. Kolker. Filtration of signals and images: Fourier and wavelet algorithms (with examples in Mathcad): monograph / Novosib. state architecture.-builds. un-t (Sibstrin). - Novosibirsk: NGASU (Sibstrin), 2010 .- 188 p.

[3] B.Chinnarao, M.M.Latha. An image denoising framework based on patch grouping in complex wavelet domain. -International Journal of Advanced Trends in Computer Science and Engineering, Volume 8, Issue 5, September-October 2019, Pages 2299-2306 DOI: 10.30534/ijatcse/2019/68852019

[4] H.N. Zaynidinov, O.U. Mallaev and B.B. Anvarjonov. A parallel algorithm for finding the human face in the image // IOP Conf. Series: Materials Science and Engineering, 862 Volume-5, May 2020, 052004, doi:10.1088/1757-899X/862/5/052004, https://doi.org/10.1088/1757-899X/862/5/052004

[5] H.N. Zaynidinov, O.U. Mallayev, I. Yusupov. Cubic Basic Splines and Parallel Algorithms // International Journal of Advanced Trends in Computer Science and Engineering (IJATCSE), India. http://www.warse.org/IJATCSE/static/pdf/file/ijatcse219932020.pdf

[6] Fei Xiongwei, Li Kenli, Yang Wangdong and Li Keqin. Analysis of energy efficiency of a parallel AES algorithm for CPU-GPU heterogeneous platforms. https://doi.org/10.1016/j.parco.2020.102621, Parallel Computing Volumes 94–95, June 2020, 102621.

[7] Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell and Eduard Ayguadé, Asynchronous runtime with distributed manager for task-based programming models. https://doi.org/10.1016/j.parco.2020. 102664. Parallel Computing Volume 97, September 2020, 102664.

[8] Rodolfo Pereira Araujo, Igor Machado Coelho and Leandro Augusto Justen Marzulo. A multi-improvement local search using dataflow and GPU to solve the minimum latency problem. https://doi.org/10.1016/j.parco.2020.102661Get rights and content, Parallel Computing Volume 98, October 2020, 102661.

[9] Jian Xiao, Min Long, Ce Yu, Xin Zhou and Li Ji. Performance optimization of non-equilibrium ionization simulations from MapReduce and GPU acceleration. https://doi.org/10.1016/j.parco.2020.102682, Parallel Computing Volume 98, October 2020, 102682.

[10] Javier Fernandez Gonzalez. Mastering Concurrency Programming with Java 9 - Second Edition: Fast, reactive and parallel application development, Paperback, Jul 17, 2017. p. 245.

[11] Peter Pacheco. An Introduction to Parallel Programming, Hardcover, Jan 21, 2011, p. 315.

[12] H.N. Zaynidinov, A.H. Yuldashev, A.B. Djumabayev. Estimation of the testing of students and their static treatment using a new software complex for adopted testing.

International journal of advanced research in science, engineering and technology. Vol 5. Issue 12.2019. -P.7701-7707.

[13] Caleb Hattingh. Using Asyncio in Python: Understanding Python's Asynchronous Programming Features, Feb 18, 2020, p. 254.

[14] Stephen Cleary. Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming 2nd Edition, Sep 10, 2019, p. 189.

[15] Brian Tuomanen. Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA. Paperback – November 27, 2018, p. 254.