Programming the CELL-BE using CSP

Kenneth SKOVHEDE [a,1] Morten N. LARSEN [a] and Brian VINTER [a],
[a] *eScience Center, Niels Bohr Institute, University of Copenhagen*

Abstract. The current trend in processor design seems to focus on using multiple cores, similar to a cluster-on-a-chip model. These processors are generally fast and power efficient, but due to their highly parallel nature, they are notoriously difficult to program for most scientists. One such processor is the CELL broadband engine (CELL-BE) which is known for its high performance, but also for a complex programming model which makes it difficult to exploit the architecture to its full potential. To address this difficulty, this paper proposes to change the programming model to use the principles of CSP design, thus making it simpler to program the CELL-BE and avoid livelocks, deadlocks and race conditions. The CSP model described here comprises a thread library for the synergistic processing elements (SPEs) and a simple channel based communication interface. To examine the scalability of the implementation, experiments are performed with both scientific computational cores and synthetic workloads. The implemented CSP model has a simple API and is shown to scale well for problems with significant computational requirements.

Keywords. CELL-BE, CSP, Programming

Introduction

The CELL-BE processor is an innovative architecture that attempts to tackle the problems, that prevent processors from achieving higher performance [1,2,3]. The limitations in traditional processors are primarily problems relating to heat, clock frequency and memory speed. Instead of using the traditional chip design, the CELL-BE consists of multiple units, effectively making it a cluster-on-a-chip processor with high interconnect speed. The CELL-BE processor consists of a single PowerPC (PPC) based processor connected to eight SPEs[1] through a 204.8 GB/s EIB[2] [4]. The computing power of a CELL-BE chip is well investigated [5,6], and a single CELL blade with two CELL-BE processors can yield as much as 460 GFLOPS [7] at one GFLOPS per Watt [7].

Unfortunately, the computing power comes at the price of a very complex programming model. As there is no cache coherent shared memory in the CELL-BE, the processes must explicitly transfer data between the units using a DMA model which resembles a form of memory mapped IO [8,4]. Furthermore to fully utilize the CELL-BE, the application must use task-, memory-, data- and instruction-level (SIMD[3]) parallelization [5]. A number of papers discuss various computational problems on the CELL-BE, illustrating that achieving good performance is possible, but the process is complex [5,9,10]. In this paper we focus on the communication patterns and disregard instruction-level and data parallelization methods because they depend on application specific computations and cannot be easily generalized.

C.A.R. Hoare introduced the CSP model in 1978, along with the concept of explicit communication through well-defined channels. Using only channel based communication, each

[1]Corresponding Author: E-mail: `skovhede@nbi.ku.dk`.
[1]Synergistic Processing Elements, a RISC based processor.
[2]Element Interconnect Bus.
[3]Single Instruction Multiple Data.

participating process becomes a sequential program [11,12]. It is possible to prove that a CSP based program is free from deadlocks and livelocks [11] using CSP algebra. Furthermore, CSP based programs are easy to understand, because the processes consist of sequential code and channels which handle communication between the processes. This normally means that the individual processes have very little code, but the total number of processes are very high.

This work uses the CSP design rules and not the CSP algebra itself. By using a CSP like interface, we can hide the underlying complexity from the programmer giving the illusion that all transfers are simply channel communications. We believe that this abstraction greatly simplifies the otherwise complex CELL-BE programming model. By adhering to the CSP model, the implementation automatically obtains properties from CSP, such as being free of race-conditions and having detectable deadlocks. Since the library does not use the CSP algebra, the programmer does not have to learn a new language but can still achieve many of the CSP benefits.

1. Related Work

A large number of programming models for the CELL-BE are available [13,14,15,16] illustrating the need for a simpler interface to the complex machine. Most general purpose libraries cannot be directly used on the CELL-BE, because the SPEs use a different instruction set than the PPC. Furthermore, the limited amount of memory available on the SPEs makes it difficult to load a general purpose library onto them.

1.1. Programming Libraries for the CELL-BE

The ALF [13] system allows the programmer to build a set of dependent tasks which are then scheduled and distributed automatically according to their dependencies. The OpenMP [14] and CellSs [15] systems provide automatic parallelization in otherwise sequential code through the use of code annotation.

As previously published [16], the Distributed Shared Memory for the CELL-BE (DSMCBE), is a distributed shared memory system that gives the programmer the “illusion” that the memory in a cluster of CELL-BE machines is shared. The channel based communication system described in this paper uses the communication system from DSMCBE, but does not use any DSM functionality. It is possible to use both communication models at the same time, however this is outside the scope of this paper.

The CellCSP [17] library shares the goals of the channel based system described in this paper but by scheduling independent processes with a focus on processes, rather than communication.

1.2. CSP Implementations

The Transterpreter [18] is a virtual machine that can run occam-π programs. By modifying the Transterpreter to run on the SPEs [19], it becomes possible to execute occam-π on the CELL-BE processor and also utilize the SPEs. The Transterpreter implementation that runs on the CELL-BE [19] has been extended to allow programs running in the virtual machine to access some of the SPE hardware. A similar project, trancell [20], allows a subset of occam-π to run on the SPU, by translating Extended Transputer Code to SPU binary code.

Using occam-π requires that the programmer learns and understands the occam-π programming language and model, and also requires that the programs are re-written in occam-π. The Transterpreter fro CELL-BE has an extension that allows callbacks to native code [19], which can mitigate this issue to some extent.

A number of other CSP implementations are available, such as C++CSP [21], JCSP [22] and PyCSP [23]. Although these may work on the CELL-BE processor they can currently only utilize the PPC and not the high performing SPEs. We have used the simplified channel interface in the newest version of PyCSP [24] as a basis for developing the channel communication interface. Since DSMCBE [16] is written in C, we have produced a flattened and non-object oriented interface.

2. Implementation

This section gives a short introduction to DSMCBE and describes some design and implementation details of the CSP library. For a more detailed description and evaluation of the DSMCBE system see previous work [16].

2.1. Distributed Shared Memory for the CELL-BE (DSMCBE)

As mentioned in the introduction, the basis for the implementation is the DSMCBE system. The main purpose of DSMCBE is to provide the user with a simple API that establishes a distributed shared memory system on the CELL-BE architecture. Apart from its main purpose, the underlying framework can also be adjusted to serve as a more generic platform for communication between the Power PC element (PPE) and the Synergistic Processing Elements (SPEs). Figure 1 shows the DSMCBE model along with the components involved. The DSMCBE system consists of four elements which we describe below:

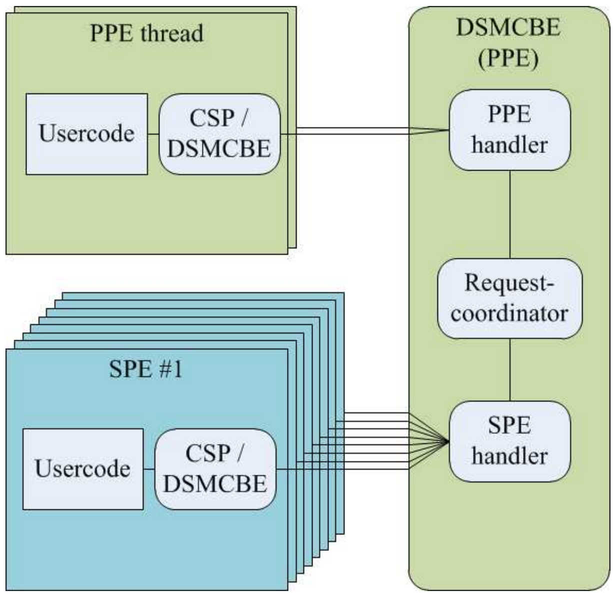

Figure 1. DSMCBE Internal Structure.

The DSMCBE PPE/SPE modules contains the DSMCBE functions which the programmer will call from the user code. To manipulate objects in the system, the programmer will use the functions from the modules to create, acquire and release objects. In addition the two modules are responsible for communicating with the main DSMCBE modules which are located on the PPC.

The PPE handler is responsible for handling communication between the PPC user code and the request coordinator (see below). Like the PPE handler, the SPE handler is responsible for handling communication between user code on the SPEs and the request coordinator (see below). However the SPE handler also manages allocation and deallocation of Local Store (LS) memory, which enables the SPE handler to perform memory management without interrupting the SPEs.

The DSMCBE library uses a single processing thread, called the request coordinator, which is responsible for servicing requests from the other modules. Components can then communicate with the request coordinator by supplying a target for the answer. Using this single thread approach makes it simpler to execute atomic operations and reduces the number of locks to a pair per participating component. Each PPC thread and SPE unit functions as a single component, which results in the request coordinator being unable to determine if the participant is a PPC thread or a SPE. As most requests must pass through the request coordinator, an obvious drawback to this method is that it easily becomes a bottleneck. With this communication framework it is easier to implement channel based communication, as the *Request Coordinator* can simply be extended to handle channel requests.

2.2. Extending DSMCBE with Channel Based Communication for CELL-BE

This section will describe how we propose to extend the DSMCBE model with channel based communication. We have used the DSMCBE system as a framework to ensure atomicity and enable memory transfers within the CELL-BE processor. The implementation does not use any DSM methods and consists of a separate set of function calls.

We have intentionally made the programming model very simple; it consists of only six functions:

- `dsmcbe_csp_channel_read`
- `dsmcbe_csp_channel_write`
- `dsmcbe_csp_item_create`
- `dsmcbe_csp_item_free`
- `dsmcbe_csp_channel_create`
- `dsmcbe_csp_channel_poison`

All functions return a status code which describes the outcome of the call.

2.2.1. Channel Communication

The basic idea in the communication model is to use channels to communicate. There are two operations defined for this: `dsmcbe_csp_channel_read` and `dsmcbe_csp_channel_wri-te`. As in other CSP implementations, the read and write operations block until a matching request arrives, making the operations a synchronized atomic event.

When writing to a channel, the calling process must supply a pointer to the data area. The result of a read operation is a pointer to a data area, as well as the size of the data area. After receiving a pointer the caller is free to read and write the contents of the area. As the area is exclusively owned by the process there is no possibility of a race condition. As it is possible to write arbitrary memory locations, when using C, it is the programmers responsibility not to use the data area after a call to write. Logically, the caller can consider the `dsmcbe_csp_channel_write` operation as transferring the data and ownership of the area to the recipient. After receiving a pointer from a read operation, and possibly modifying data area, the process may forward the pointer again using `dsmcbe_csp_channel_write`. As the reading process has exclusive ownership of the data area, it is also responsible for freeing the data area, if it is no longer needed.

The operation results in the same output regardless of which CELL-BE processor the call originates from. If both processes are in the same memory space the data is not copied ensuring maximal speed. If the data requires a transfer, the library will attempt to do so in the most efficient manner.

2.2.2. Transferable Items

The CELL-BE processor requires that data is aligned and have certain block sizes, a constraint that is not normally encountered by a programmer. We have chosen to expose a simple pair of functions that mimic the well-known `malloc` and `free` functions called `dsmcbe_csp_item_create` and `dsmcbe_csp_item_free`, respectively. A process wishing to communicate can allocate a block of memory by calling the `dsmcbe_csp_item_create` function and get a standard pointer to the allocated data area. The process is then free to write data into the allocated area. After a process has used a memory block, it can either forward the block to another channel, or release the resources held by calling `dsmcbe_csp_item_free`.

2.2.3. Channel Creation

When the programmer wants to use a channel it is necessary to create it by calling the `dsmcbe_csp_channel_create` method. To distinguish channels, the create function must be called with a unique number, similar to a channel name or channel object in other CSP systems. This channel number is used to uniquely identify the channel in all subsequent communication operations.

The create function allows the caller to set a buffer size on the channel, thus allowing the channel writers to write data into the channel without awaiting a matching reader. A buffer in the CSP model works by generating a sequence of processes where each process simply reads and writes an element. The number of processes in the chain determines the size of the buffer. The semantics of the implemented buffer are the same as a chain of processes, but the implementation uses a more efficient method with a queue.

The channel type specifies the expected use of the channel, with the following options: one-to-one, one-to-any, any-to-one, any-to-any and one-to-one-simple. Using the channel type it is possible to verify that the communication patterns correspond to the intended use. In situations where the participating processes do not change it is possible to enable "low overhead" communication by using the channel type one-to-one-simple. Section 2.2.8 describes this optimization in more detail.

A special convention borrowed from the DSMCBE model is that read or write operations on non-existing channels will cause the caller to block if the channel is not yet created. Since a program must call the create function exactly once for each channel, some start-up situations are difficult to handle without this convention. Once a process has created the channel, it processes all the pending operations as if they occurred after the channel creation.

2.2.4. Channel Poison

As all calls are blocking they can complicate the shutdown phase of a CSP network. The current CSP implementations support a channel poison state, which causes all pending and following operations on that channel to return the poison.

To poison a channel, a process calls `dsmcbe_csp_channel_poison` with the id of an existing channel. When using poison, it is important to check the return value of the read and write operations, as they may return the poison status. A macro named `CSP_SAFE_CALL` can be used to check the return value and exit the current function when encountered. However the programmer is still fully responsible for making the program handle and distribute poison correctly.

2.2.5. External Choice

As a read operation is blocking, it is not possible to wait for data on more than one channel, nor is it possible to probe a channel for its content. If a process could see whether or not a channel has content, a race condition could be introduced. Thereby a second process could read the item right after the probe, resulting in a blocking read.

To solve this issue, CSP uses the concept of external choice where a process can request data from multiple channels and then gets a response once a channel is ready. To use external choice, the process must call a variation of the `dsmcbe_csp_channel_read` function named dsmcbe_csp_channel_read_alt, where alt is short for “alternation”, the term used in C.A.R. Hoare’s original paper [25]. Using this function, the process can block for a read operation on multiple channels. When one of the channels has data, the data is returned, as with the normal read operation, along with the channel id of the originating channel. This way of dealing with reads ensures that race conditions cannot occur.

With the channel selection done externally, the calling process has no way of controlling which channel to read, should there be multiple available choices. To remedy this, the calling process must also specify what strategy to use if multiple channels are ready. The JCSP library offers three strategies: arbitrary, priority and fair. Arbitrary picks a channel at random whereas priority chooses the first available channel, prioritized by the order in which the channels are given. Fair selection keeps count of the number of times each channel has been selected and attempts to even out the usage of channels. The current implementation of CSP channels for CELL-BE only supports priority select, but the programmer can emulate the two other modes.

Similar to the read function, a function called dsmcbe_csp_channel_write_alt allows a process to write to the first available channel. This function also supports a selection strategy and returns the id of the channel written to. There is currently no mechanism to support the simultaneous selection of channel readers and writers, though there are other ways of engineering this.

2.2.6. Guards

To prevent a call from blocking, the calling function can supply a guard which is invoked when no data is available. The implementation defines a reserved channel number, called CSP_SKIP_GUARD which can be given as a channel id when requesting read or write from multiple channels. If the operation would otherwise block, the function returns a NULL pointer and CSP_SKIP_GUARD as the channel value.

Other CSP implementations also offer a time-out guard, which performs a skip, but only if the call blocks for a certain period. This functionality is not available in the current implementation, but could be added without much complication.

2.2.7. Processes for CELL-BE

The hardware in the CELL-BE is limited to a relatively low number of physical SPEs, which prevents the generation of a large number of CSP processes. To remedy this situation the implementation also supports running multiple processes on each SPE. Since the SPEs have little support for timed interrupts, the implementation is purely based on cooperative switching. To allow multiple processes on the SPE, we have used an approach similar to CELL-MT [26], basically implementing a user-mode thread library, but based on the standard C functions setjmp and longjmp.

The CSP threading library implements the main function, and allocates ABI compliant stacks for each of the processes when started. After setting up the multithreading environment, the scheduler is activated which transfers control to the first processes. Since the main function is implemented by the library, the user code must instead implement the dsmcbe_main function, which is activated for each process in turn. This means that all processes running on a single SPE must use the same dsmcbe_main function, but each process can call the function dsmcbe_thread_current_id and thus obtain a unique id, which can be used to determine what code the process will execute.

When a process is executing it can cooperatively yield control by calling dsmcbe_thread_yield, which will save the process state and transfer control to the next available process. Whenever a process is waiting for an API response, the library will automatically call a similar function called dsmcbe_thread_yield_ready. This function will yield if another process is ready to execute, meaning that it is not currently awaiting an API response. The effect of this is that each API call appears to be blocking, allowing the programmer to write a fully sequential program and transparently run multiple processes.

As there is no preemptive scheduling of threads, it is possible for a single process to prevent other processes from executing. This is a common trade-off between allowing the SPE to execute code at full speed, and ensuring progress in all processes. This can be remedied by inserting calls to `dsmcbe_thread_yield_ready` inside computationally heavy code, which allows the programmer to balance the single process execution and overall system progress in a fine grained manner.

The scheduler is a simple round-robin scheduler using a ready queue and a waiting queue. The number of threads possible is limited primarily by the amount of available LS memory, which is shared among program code, stack and data. The running time of the scheduler is $O(N)$ which we deem sufficient, given that all processes share the limited LS, making more than 8 processes per SPE unrealistic.

2.2.8. SPE-to-SPE Communication

Since the PPC is rarely a part of the actual problem solving, the memory blocks can often be transferred directly from SPE to SPE without transferring it into main memory.

If a SPE is writing to a buffered channel, the data may not be read immediately after the write. Thus, the SPE may run out of memory since the data is kept on the SPE in anticipation of a SPE-to-SPE transfer. To remedy this, the library will flush data to main memory if an allocation would fail. This is in effect a caching system, and as such it is subject to the regular benefits and drawbacks of a cache. One noticeable drawback is that due to the limited available memory, the SPEs are especially prone to memory fragmentation, which happens more often when using a cache, as the memory stays fully populated for longer periods.

If the channel is created with the type one-to-one-simple, the first communication will be used to determine the most efficient communication pattern, and thus remove some of the internal synchronization required. If two separate SPEs are communicating, this means that the communication will be handled locally in the *SPE Handler* shown in Figure 1, and thus eliminate the need to pass messages through the *Request Coordinator*.

A similar optimization is employed if two processes on the same SPE communicate. In this case the data is kept on the SPE, and all communication is handled locally on the SPE in the *DSMCBE SPE* module shown in Figure 1. Due to the limited amount of memory available on the SPE, data may be flushed out if the channel has large buffers or otherwise exhaust the available memory.

These optimizations can only work if the communication is done in a one-to-one fashion where the participating processes never change. Should the user code attempt to use such a channel in an unsupported manner, an error code will be returned.

2.2.9. Examples

To illustrate the usage of the channel-based communication Listing 1 shows four simple CSP processes. Listing 2 presents a simple example that uses the alternation method to read two channels and writes the sum to an output channel.

3. Experiments

When evaluating system performance, we focus mainly on the scalability aspect. If the system scales well, further optimizations may be made specific to the application, utilizing the SIMD capabilities of the SPEs. The source code for the experiments are available from `http://code.google.com/p/dsmcbe/`.

```c
#include <dsmcbe_csp.h>

int delta1(GUID in, GUID out) {
  void* value;

  while(1) {
    CSP_SAFE_CALL("read", dsmcbe_csp_channel_read(in, NULL, &value));
    CSP_SAFE_CALL("write", dsmcbe_csp_channel_write(out, value));
  }
}

int delta2(GUID in, GUID outA, GUID outB) {
  void* inValue, outValue;
  size_t size;

  while(1) {
    CSP_SAFE_CALL("read", dsmcbe_csp_channel_read(in, &size, &inValue));
    CSP_SAFE_CALL("allocate", dsmcbe_csp_item_create(&outValue, size));

    memcpy(outValue, inValue, size); //Copy contents as we need two copies

    CSP_SAFE_CALL("write A", dsmcbe_csp_channel_write(outA, inValue));
    CSP_SAFE_CALL("write B", dsmcbe_csp_channel_write(outB, outValue));
  }
}


int prefix(GUID in, GUID out, void* data) {
  CSP_SAFE_CALL("write", dsmcbe_csp_channel_write(out, data));

  return delta1(in, out);
}

int tail(GUID in, GUID out) {
  void* tmp;

  CSP_SAFE_CALL("read", dsmcbe_csp_channel_read(in, NULL, &tmp));
  CSP_SAFE_CALL("free", dsmcbe_csp_item_free(tmp));

  return delta1(in, out);
}
```

Listing 1. Four simple CSP processes.

```c
int add(GUID inA, GUID inB, GUID out)
{
  void *data1, *data2;

  GUID channelList[2];
  channelList[0] = inA;
  channelList[1] = inB;

  GUID chan;

  while(1)
  {
    dsmcbe_csp_channel_read_alt(CSP_ALT_MODE_PRIORITY, channelList, 2, &chan,
        NULL, &data1);
    dsmcbe_csp_channel_read(chan == inA ? inB : inA, NULL, &data2);

    *(int*)data1 = *((int*)data1) + *((int*)data2);

    dsmcbe_csp_item_free(data2);
    dsmcbe_csp_channel_write(out, data1);
  }
}
```

Listing 2. Reading from two channels with alternation read and external choice. To better fit the layout of the article the `CSP_SAFE_CALL` macro is omitted.

All experiments were performed on an IBM QS22 blade, which contains 2 connected CELL-BE processors, giving access to 4 PPE cores and 16 SPEs.

3.1. CommsTime

A common benchmark for any CSP implementation is the CommsTime application which sets up a ring of processes that simply forwards a single message. The conceptual setup is shown in Figure 2. This benchmark measures the communication overhead of the channel operations since there is almost no computation required in the processes. To better measure the scalability of the system, we have deviated slightly from the normal CommsTime implementation, by inserting extra successor processes as needed. This means that each extra participating process will add an extra channel, and thus and thus produce a longer communication ring.

Figure 3 shows the CommsTime when communicating among SPE processes. The PPE records the time between each received message, thus measuring the time it takes for the message to traverse the ring. The time shown is an average over 10 runs of 10.000 iterations. As can be seen, the times seems to stabilize around 80 μseconds when using one thread per SPE. When using two or more threads the times stabilizes around 38 μseconds, 27 μseconds, and 20 μseconds respectively. When using multiple threads, the communication is performed internally on the SPEs, which results in a minimal communication overhead causing the average communication overhead to decrease.

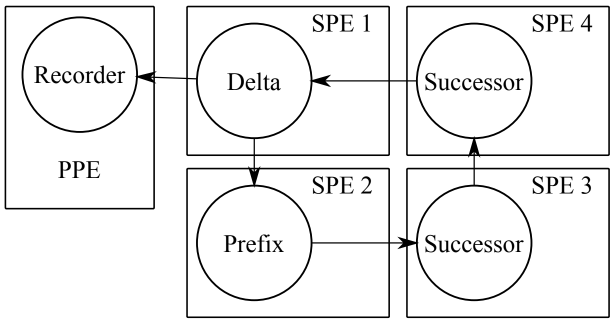

Figure 2. Conceptual setup for the CommsTime experiment with 4 SPEs.

We have executed the CommsTime sample from the JCSP library v.1.1rc4 on the PPE. The JCSP sample uses four processes in a setup similar to Figure 2 but with all processes placed on the PPE. Each communication took on average 63 μseconds which is slightly faster than our implementation, which runs at 145 μseconds on the PPE. Even though JCSP is faster, it does not utilize the SPEs, and cannot utilize the full potential of the CELL-BE.

3.2. Prototein Folding

Prototeins are a simplified 2D model of a protein, with only two amino acids and only 90 degree folds [27]. Folding a prototein is computationally simpler than folding a full protein, but exhibit the same computational characteristics. Prototein folding can be implemented with a bag-of-tasks type solution, illustrated in Figure 4, where partially folded prototeins are placed in the bag. The partially folded prototeins have no interdependencies, but may differ in required number of combinations and thus required computational time.

As seen in Figure 5 the problem scales very close to linearly with the number of SPEs, which is to be expected for this type of problem. This indicates that the communication latency is not a limiting factor, which also explains why the number of SPE threads have very little effect on the scalability.

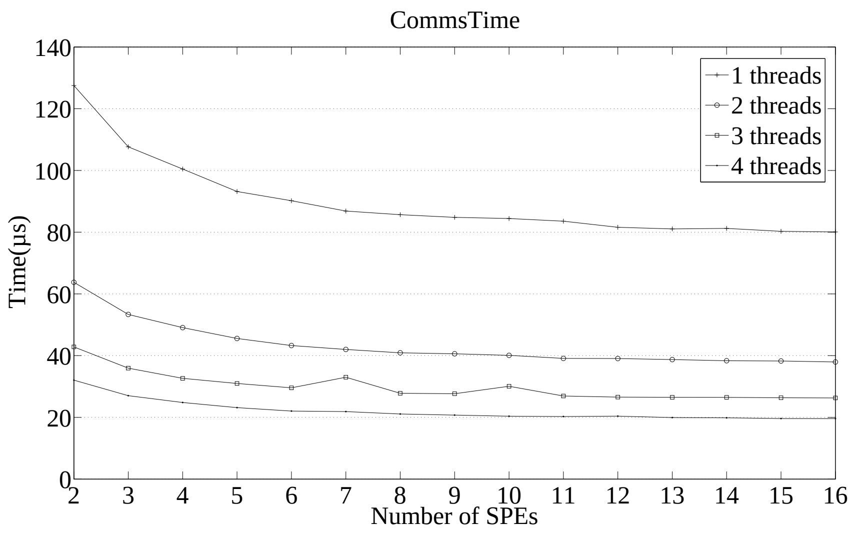

Figure 3. CommsTime using 2-16 SPEs with 1-4 threads per SPE.

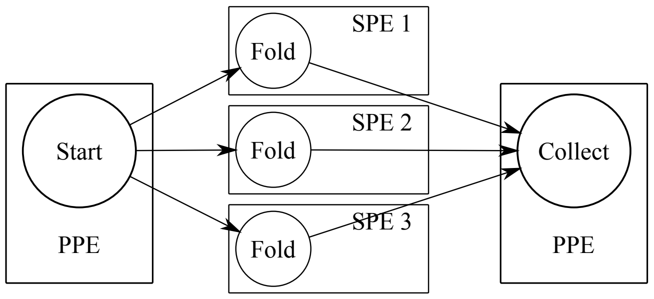

Figure 4. Conceptual setup for Prototein folding with 3 SPEs.

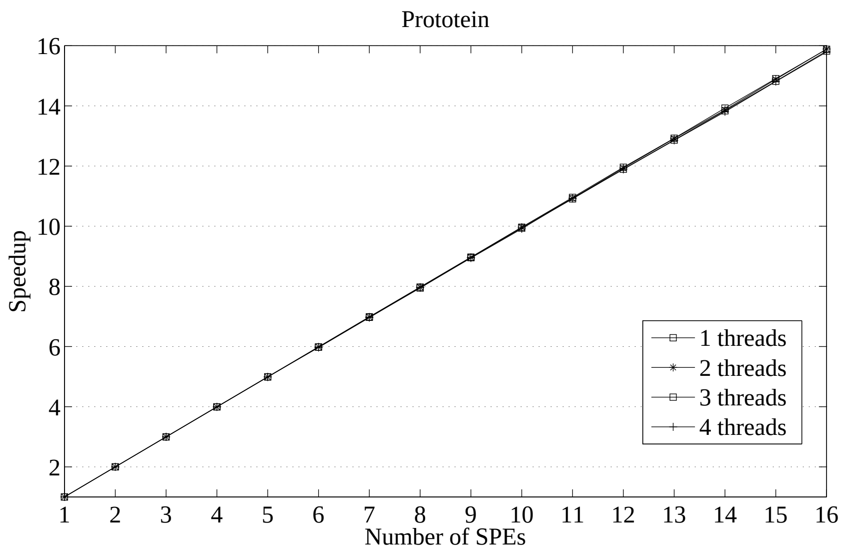

Figure 5. Speedup of prototein folding using 1-16 SPEs.

3.3. k Nearest Neighbors (kNN)

The kNN application is a port of a similar application written for PyCSP [28]. Where the PyCSP model is capable of handling an extreme number of concurrent processes, the library is limited by the number of available SPEs and the amount of threads each SPE can accommodate. Due to this, the source code for the two applications are hard to compare, but the overall approach and communication patterns are the same. Figure 6 shows a conceptual ring based setup for finding the kNN.

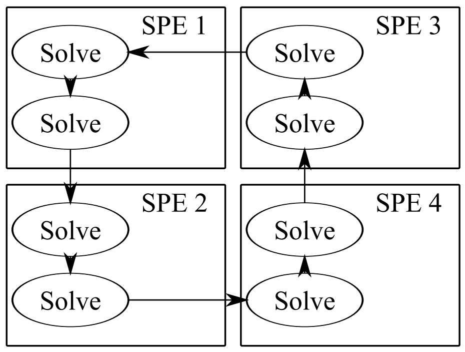

Figure 6. Conceptual setup for the kNN experiment with 4 SPEs, each running 2 threads.

This ring-based approach means that each process communicates only with its neighbor. To support arbitrary size problems, one of the channels are buffered. The underlying system will attempt to keep data on the SPE, in anticipation of a transfer, but as the SPE runs out of memory, the data will be swapped to main memory. This happens completely transparent to the process, but adds an unpredictable overhead to the communication. This construction allows us to run the same problem size on one to 16 SPEs.

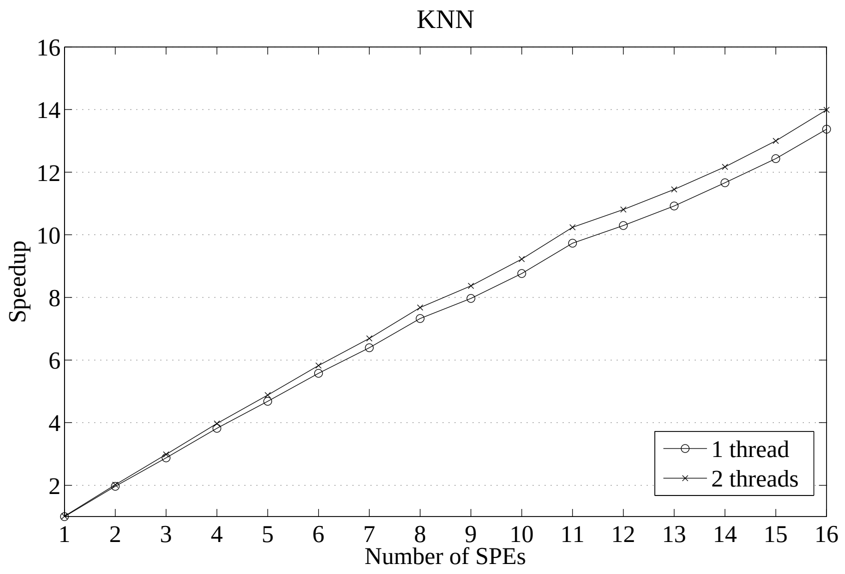

Figure 7. Speedup of the k Nearest Neighbors problem using 1-16 SPEs to search for 10 nearest neighbors in a set with 50k elements with 72 dimensions.

As seen in Figure 7 this does not scale linearly, but given the interdependencies we consider this to be a fairly good result. Figure 7 also shows that using threads to run multiple solver processes on each SPE offers a performance gain, even though the processes compete for the limited LS memory. This happens because the threads implement an implicit form of double buffering, allowing each SPE to mask communication delays with computation. The achieved speedup indicates that there is a good balance between the communication and computation performed in the experiment.

The speedup for both graphs is calculated based on the measured time for running the same problem size on a single SPE with a single solver thread.

The ring based communication model used in the kNN experiment is quite common for problems that use a n^2 approach. However, the scalability of such a setup is highly dependent on the amount of work required in each subtask. To quantify the communication to computation ratio required for a well-scaling system, we have developed a simple ring-based program that allows us to adjust the number of floating point operations performed between communications. The computation performed is adjustable and does not depend on the size of the transmitted data, allowing us to freely experiment with the computational workload. The setup for this communication system is shown in Figure 8. The setup is identical to the one used in the kNN experiment, but instead of having two communicating processes on the same SPE, the processes are spread out. This change cause the setup to loose the possibility for the very fast internal SPE communication channels, which causes more load on the PPE and thus gives a more realistic measurement for the communication delays.

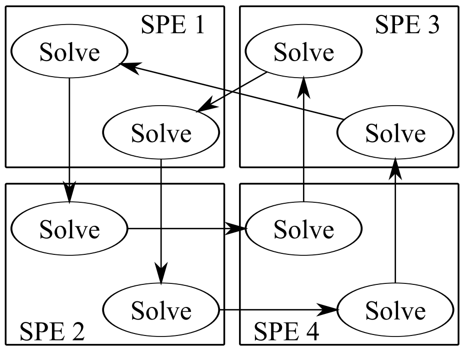

Figure 8. Conceptual setup for non-structured ring based communication.

As seen in Figure 9, the implementation scales well if computation performed in each ring iteration is around 100MFLOPS. Comparing the two graphs in Figure 9, shows that increasing the number of threads on the SPEs, results in a decrease in performance. This happens because the extra processes introduce more communication. This increase in communication causes a bigger strain on the PPE, which results in more latency than the processes hide. In other words, the threads cause more latency than they can hide in this setup.

The speedup for both graphs in Figure 9 are calculated based on measurements from a run with the same data size on a single SPE with a single thread.

Comparing the Communication to Computation experiment with the kNN experiment reveals that the use of optimized channels reduces the latency of requests to a level where the threads are unable to hide the remaining latency. In other words, the latency becomes so low, that the thread switching overhead is larger than the latency it attempts to hide. This is consistent with the results from the CommsTime experiment, which reveals that the communication time is very low when performing inter-SPE communication. This does not mean that the latency is as low as it can be, but it means that the extra communication generated by the threads increases the amount of latency that must be hidden.

4. Future Work

The main problem with any communication system is the overhead introduced by the communication. As the experiments show, this overhead exists but can be hidden because the CELL-BE and library are capable of performing the communication and computation simultaneously. But this hiding only works if the computational part of a program has a sufficient size. To remedy this, the communication overhead should be reduced significantly.

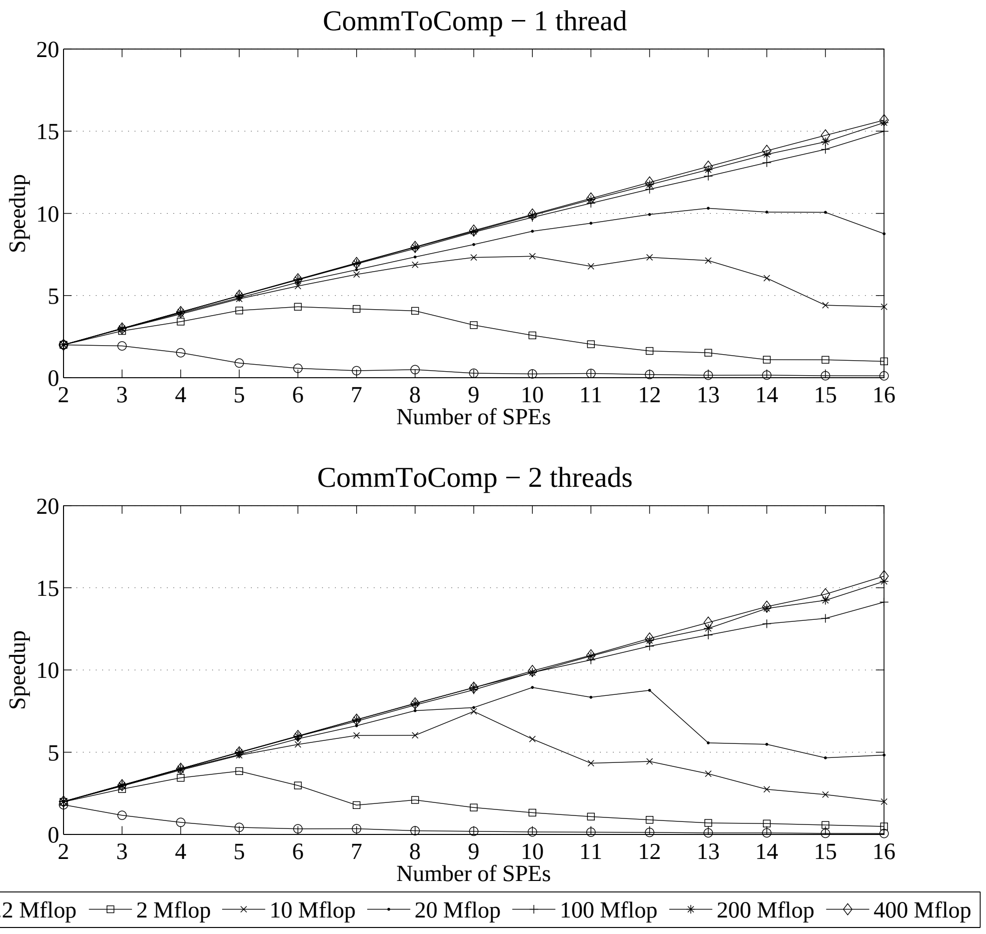

Figure 9. Communication To Computation ratio, 16 bytes of data.

The decision to use the request coordinator to handle the synchronization simplifies the implementation, but also introduces two performance problems. One problem is that if the system becomes overwhelmed with requests, the execution will be sequential, as the processes will only progress as fast as the request coordinator responds to messages. The other problem is that the requests pass through both the SPU handler and the request coordinator, which adds load to the system and latency to each communication operation.

4.1. Reduce Request Latency

Since the SPEs are the main workhorse of the CELL-BE, it makes sense to move much of the decision logic into the SPU handler rather than handle it in the request coordinator. The request coordinator is a legacy item from the DSM system, but there is nothing that prevents participating PPE processes from communicating directly with the SPU handler.

4.2. Increase Parallelism

Even if the request coordinator is removed completely, the PPE can still be overwhelmed with requests, which will make everything run sequentially rather than in parallel. It is not possible to completely remove a single synchronization point, but many communication operations involve exactly two processes. In the common case where these two processes reside on separate SPEs, it is possible to perform direct SPE-to-SPE communication through the use of signals and DMA transfers. If this is implemented, it will greatly reduce the load on the PPE for all the presented experiments.

4.3. Improve Performance of the SPU Handler

The current implementation uses a shared spinning thread that constantly checks for SPE and request coordinator messages. It is quite possible that this can be improved by using a thread for each SPE which uses the SPE events rather than spinning. Experiments performed for the DSMCBE [16] system show that improving the SPU handler can improve the overall system performance.

4.4. Improve Memory Exhaustion Handling

When the communication is handled by the SPEs internally, it is likely that they will run out of memory. If the SPU handler is involved, such situations are detected and handled gracefully. Since this is essentially a cache system, a cache policy can greatly improve the performance of the system, by selectively choosing which elements to remove from the LS and when such an operation is initiated.

4.5. Process Migration

The processes are currently bound to the SPE that started them, but it may turn out that the setup is ineffective and can be improved by moving communicating processes closer together, i.e. to the same SPE. There is limited support for this in the CELL-BE architecture itself, but the process state can be encapsulated to involve only the current thread stack and active objects. However, it may prove to be impossible to move a process, as data may occupy the same LS area. Since the C language uses pointers, the data locations cannot be changed during a switch from one SPE to another. One solution to this could be to allocate processes in *slots*, such as those used in CELL CSP [17].

4.6. Multiple Machines

The DSMCBE system already supports multiple machines, using standard TCP-IP communication. It would be desirable to also support multiple machines for CSP. The main challenge with multiple machines is to implement a well-scaling version of the alternation operations, because the involved channels can span multiple machines. This could use the cross-bar approach used in JCSP [29].

5. Conclusion

In this paper we have described a CSP inspired communication model and a thread library, that can help programmers handle the complex programming model on the CELL-BE. We have shown that even though the presented models introduce some overhead, it is possible to get good speedup for most problems. On the other hand Figure 9 shows, that if the computation to communication ratio is too low - meaning too little computation per communication, it is very hard to scale the problems to utilize all 16 SPEs. However we believe that for most programmers solving reasonable sized problems, the tools provided can significantly simplify the writing of programs for the CELL-BE architecture.

We have also shown that threads can be used to mask some latency, but at the same time they generate some latency, which limits their usefulness to certain problems.

DSMCBE and the communication model described in this paper is open source software under the LGPL license, and are available from `http://code.google.com/p/dsmcbe/`.

Acknowledgements

The authors acknowledge the Danish National Advanced Technology Foundation (grant number 09-067060) and the innovation consortium (grant number 09-052139) for supporting this research project. Furthermore the authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

References

[1] Wm. A. Wulf and Sally A. Mckee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23:20–24, 1995.

[2] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[3] Gordon E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[4] Thomas Chen. Cell Broadband Engine Architecture and its first implementation - A Performance View, 2005. `http://www.ibm.com/developerworks/power/library/pa-cellperf/`. Accessed 26 July 2010.

[5] Martin Rehr. Application Porting and Tuning on the Cell-BE Processor, 2008. `http://dk.migrid.org/public/doc/published_papers/nqueens.pdf`. Accessed 26 July 2010.

[6] Mohammed Jowkar. Exploring the Potential of the Cell Processor for High Performance Computing, 2007. `http://www.diku.dk/~rehr/cell/docs/mohammad_jowkar_thesis.pdf`. Accessed 26 July 2010.

[7] IBM. IBM Doubles Down on Cell Blade, 2007. `http://www-03.ibm.com/press/us/en/pressrelease/22258.wss`. Accessed 26 July 2010.

[8] IBM. Cell BE Programming Handbook Including PowerXCell 8i, 2008. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64%B3300257460006FD68D/$file/CellBE_PXCell_Handbook_v1.11_12May08_pub.pdf`. Accessed 26 July 2010.

[9] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008.

[10] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Solving Large Instances of Capacitated Vehicle Routing Problem over Cell BE. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 131–138, Washington, DC, USA, 2008. IEEE Computer Society.

[11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[12] A.W. Roscoe, C.A.R. Hoare, and R. Bird. *The theory and practice of concurrency*, volume 216. Citeseer, 1998.

[13] IBM. Accelerated Library Framework Programmer's Guide and API Reference, 2009. `http://public.dhe.ibm.com/software/dw/cell/ALF_Prog_Guide_API_v3.1.pdf`. Accessed 26 July 2010.

[14] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on Cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *ACM/IEEE CONFERENCE ON SUPERCOMPUTING*, page 86. ACM, 2006.

[16] Morten N. Larsen, Kenneth Skovhede, and Brian Vinter. Distributed Shared Memory for the Cell Broadband Engine (DSMCBE). In *ISPDC '09: Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*, pages 121–124, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Mads Alhof Kristiansen. CELL CSP Sourcecode, 2009. `http://code.google.com/p/cellcsp`. Accessed 26 July 2010.

[18] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106, Amsterdam, September 2004. IOS Press.

[19] Damian J. Dimmich, Christian L. Jacobsen, and Matthew C. Jadud. A Cell Transterpreter. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, volume 29 of *Concurrent Systems Engineering Series*, pages 215–224, Amsterdam, September 2006. IOS Press.

[20] Ulrik Schou Jørgensen and Espen Suenson. trancell - an Experimental ETC to Cell BE Translator. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 287–298, jul 2007.

[21] Alistair A. Mcewan, Steve Schneider, Wilson Ifill, Peter Welch, and Neil Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures, 2007.

[22] P. H. Welch, A. W. P. Bakkers (eds, and Nan C. Schaller. Using Java for Parallel Computing - JCSP versus CTJ. In *Communicating Process Architectures 2000*, pages 205–226, 2000.

[23] Otto J. Anshus, John Markus Bjørndalen, and Brian Vinter. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[24] Brian Vinter, John Markus Bjørndaln, and Rune Møllegaard Friborg. PyCSP Revisited, 2009. `http://pycsp.googlecode.com/files/paper-01.pdf`. Accessed 26 July 2010.

[25] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[26] Vicenç Beltran, David Carrera, Jordi Torres, and Eduard Ayguadé. CellMT: A cooperative multithreading library for the Cell/B.E. In *HiPC*, pages 245–253, 2009.

[27] Brian Hayes. Prototeins. *American Scientist*, 86(3):216–, 1998.

[28] Rune Møllegaard Friborg. PyCSP kNN implementation, 2010. `http://pycsp.googlecode.com/svn-history/r288/trunk/examples/kNN.py`. Accessed 26 July 2010.

[29] P.H. Welch and B. Vinter. Cluster Computing and JCSP Networking. *Communicating Process Architectures 2002*, 60:203–222, 2002.