Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

An Introduction to CSP.NET

Alex A. LEHMBERG and Martin N. OLSEN

Department of Computer Science, University of Copenhagen,
Universitetsparken 1, 2100, Copenhagen, Denmark.

{alex , nebelong} @diku.dk

Abstract. This paper reports on CSP.NET, developed over the last three months at the University of Copenhagen. CSP.NET is an object oriented CSP library designed to ease concurrent and distributed programming in Microsoft.NET 2.0. The library supports both shared memory multiprocessor systems and distributed-memory multicomputers and aims towards making the architecture transparent to the programmer. CSP.NET exploits the power of .NET Remoting to provide the distributed capabilities and like JCSP, CSP.NET relies exclusively on operating system threads. A Name Server and a workerpool are included in the library, both implemented as Windows Services. This paper presents CSP.NET from a users perspective and provides a tutorial along with some implementation details and performance tests.

Keywords. CSP library, Microsoft.NET, CSP.NET

Introduction

In as little as one year from now it will be very hard to get hold of computers with only one core. In order to increase performance significantly in future generation microprocessors all major manufacturers are taking the road of multiple cores on a chip, and multiple chips in a machine. These multi-chip multi-core machines will probably not run at much higher clock speeds than current machines, meaning that programs will have to use multiple threads of execution for any significant performance improvements to materialise.

As any skilled programmer will testify writing large error free concurrent programs in any mainstream programming language is extremely difficult at best. Threads, and various locks and synchronisation mechanisms, are the common constructs used to achieve concurrency, but they are all low level constructs unable to express the complex interactions in concurrent programs in a simple and secure manner.

Communicating Sequential Processes (CSP) [5] as a programming model builds on the CSP algebra and provides a series of higher level constructs that solves many of the problems inherent in traditional thread programming. CSP makes it easy to distinguish between deterministic and nondeterministic parts of concurrent programs, and makes synchronisation and concurrent execution relatively simple.

occam [9] is a language inspired by CSP, but CSP-like libraries for more widespread languages also exist. They include JCSP [4], JCSP.NET [6], CTJ [7], C++CSP [3] and C++CSP Networked [2]. This paper describes a new CSP library developed as a graduate student project at the University of Copenhagen under the supervision of Brian Vinter. The background was a graduate course in practical CSP programming taught during the spring of 2006.

CSP.NET is written for the Microsoft .NET platform, which in principle makes CSP available to programmers using any CLS-compliant language. CSP.NET is designed to run equally well on shared memory multiprocessor systems and distributed memory architec-

tures. It is designed to abstract away the underlying system and provides distributed versions of most of the implemented CSP operators. It is based on version 2.0 of the .NET framework and thus supports generics throughout. Remoting - the .NET equivalent of Java RMI - is used to provide the distributed capabilities of CSP.NET.

This paper describes CSP.NET from a users perspective and provides some insight into the implementation. Section 1 is a brief overview of the different elements in CSP.NET. Section 2 is a tutorial that demonstrates the use of CSP.NET through a series of examples. Section 3 provides some technical insight into the implementation of CSP.NET and finally section 4 demonstrates a CSP.NET implementation of the MonteCarlo Pi algorithm along with some interesting performance figures. Readers are assumed to have knowledge of programming and CSP.

CSP.NET can be downloaded from *www.cspdotnet.com* and we must stress that currently, not all features have been thoroughly tested. Thus the state of the program can best be described as a work in progress - we did say that thread programming was hard - and feedback is most welcome.

1. Library Details

CSP.NET is an object oriented implementation of CSP, designed to simplify concurrent and parallel programming on a Microsoft.NET 2.0 platform.

The API of CSP.NET is inspired by JCSP.NET, but the implementation is completely original. CSP.NET offers constructs like *barrier*, *bucket* and *parallel* but in contrast to JCSP.NET, CSP.NET provides both local and distributed implementations of these constructs. Furthermore the distributed channel ends and timers of CSP.NET differs from their counterparts in JCSP.NET. Every method in CSP.NET has been separately tested and the authors have used the library in several minor applications.

The entire documentation of CSP.NET is available at *www.cspdotnet.com* and this section gives a brief introduction to the main constructs in the library.

1.1. Processes

The object oriented approach implies that every CSP construct is implemented as a class in CSP.NET and hence a new process is constructed by creating a class that implements the *ICSProcess* interface.

Processes may be executed in parallel by using an instance of the *Parallel* class. Since all processes in a given instance of *Parallel* are executed locally, true parallelism will only occur on a multiprocessor machine. Otherwise the processes will be interleaved.

To ease distributed programming, CSP.NET provides a distributed Parallel class similar to the standard *Parallel* class, but *DistParallel* seeks to execute processes on remote machines by utilising the CSP.NET workerpools, see section 3.5.

1.2. Channels

So far we have discussed how to define and run several different processes, each containing sequential code, but as the name CSP implies processes must be able to interact. Interaction or process communication is managed by channels, which makes them a central part of any CSP implementation.

CSP.NET provides four distinct channels - *One2One*, *Any2One*, *One2Any* and *Any2Any*. These are all well known rendezvous channels that may be extended with buffers. The library comes with two predefined buffers and additional buffers can be defined by implementing the *IBuffer* interface. The available buffers are the standard FIFO buffer and an infinite buffer which, in theory, is able to hold an infinite number of elements.

1.2.1. Anonymous and Named Channels

CSP.NET channels are either anonymous or named. Named channels are by default distributed but can be declared as local while anonymous channels are always local. Distributed channel ends allow communication between processes residing on different machines or in different application domains on the same machine. Local channels only allow communication between processes in the same application domain. Distributed channel ends may be used for local communication but local channels are preferable due to their more efficient implementation, see section 3.2.

We have chosen not to implement a named *Any2Any* channel, but an anonymous *Any2Any* channel is available - see section 3.1.

1.2.2. Name Server

To use distributed channel ends a Name Server must be available on the network. The Name Server distinguishes channels by name, thus every named channel must have a unique name. Violations of this rule are not necessarily recognised by the Name Server but may result in erroneous programs.

The Name Server is provided as both a standard console application and as a Windows Service. The console version is configured though the command line while the service uses an XML configuration file.

1.2.3. Channel Communication

Channels in CSP.NET are generic and may be of any serializable data type, thus making it possible to send almost everything through a channel. But caution must be exercised - CSP.NET channels don't necessarily copy the data like the CSP-paradigm demands.

Distributed channels are call-by-value while local channels are call-by-reference. This is a tradeoff between safety and efficiency and, if preferred, copies can be made before sending data through a local channel.

1.3. Alternative

Alternatives permit the programmer to choose between multiple events - in CSP.NET known as guards. Four types of guards are available in CSP.NET - *One2One* channel, *Any2One* channel, *CSTimer* and *Skip*. Skips are always ready, timers are ready whenever a timeout occurs and the channels are ready if they contain data. The channels may be distributed channels residing on remote machines, while the timers and skips must be local.

To choose between ready guards *Alternative* provides two methods - *PriSelect* and *FairSelect*. The former always selects the guard with the highest priority while the latter guaranties a fair selection, meaning that every ready guard will be selected within n calls to *FairSelect*, where n is the number of ready guards. Like JCSP, C++CSP and KRoC, *FairSelect* in CSP.NET delivers unit time for each choice, regardless of the number of guards, provided that at least one guard is always pending. In CSP.NET the same applies to *PriSelect*.

1.4. Barriers and Buckets

CSP.NET provides *barriers* and *buckets* to synchronise multiple processes. Any process synchronising on a *barrier* will be blocked until all processes enrolled on the *barrier* has called the *Sync*-method and any process falling into a *bucket* will be blocked until another process call the *bucket*'s *Flush*-method.

Barriers and *buckets* are either anonymous or named and just like channels, anonymous *barriers* and *buckets* are local while named *barriers* and *buckets* can be local or distributed. Distributed *barriers* and *buckets* use the Name Server, meaning that every named *barrier* and *bucket* must have a unique name.

1.5. Workerpool Service

CSP.NET includes a workerpool Service along with the standard library. It's a Windows Service capable of running on any Microsoft.NET 2.0 platform. Once installed every CSP.NET program can use the service, by using the *DistParallel* class, which may be convenient, e.g. in grid-like programming.

The workerpool service needs information about port numbers, IP addresses etc. and to that end an XML configuration file is supplied. The configuration file is read each time the service is started.

2. Tutorial

This section demonstrates how to write some fairly simple programs using the CSP.NET library. We will start by implementing a workerpool and then move on to demonstrate the use of alternatives and distributed parallels in CSP.NET.

2.1. Workerpool

Our First CSP.NET program shows how to implement a workerpool. Note that the workerpool in this example has no relation to the workerpool Service provided by CSP.NET.

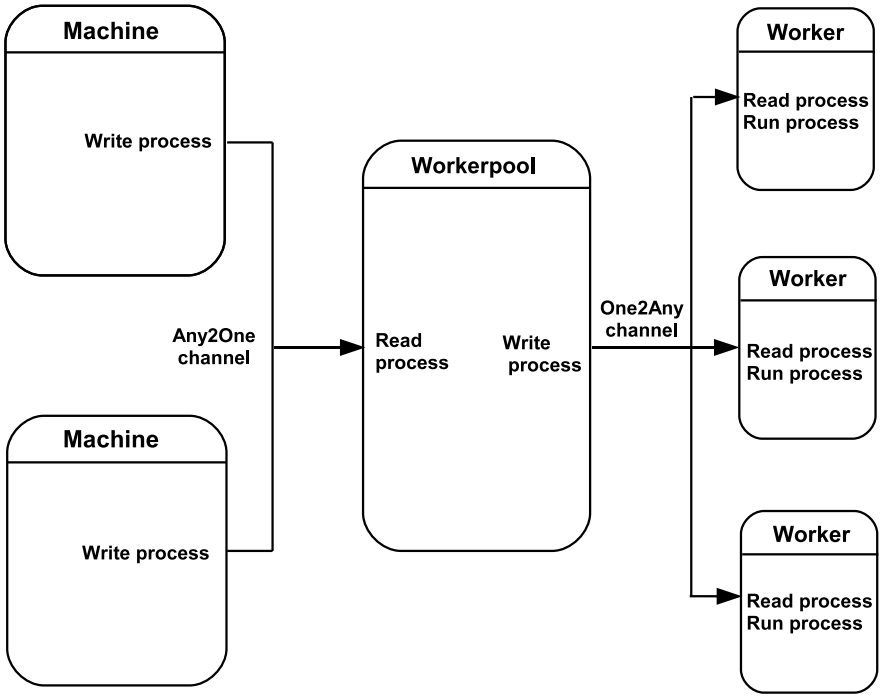

Figure 1. Workerpool structure

Figure 1 illustrates that we need a workerpool process connected to some worker processes through an *One2Any* channel and we also need an *Any2One* channel connecting the workerpool to the outside world. Translating the figure into a CSP.NET program is very easy. All we need is a workerpool process, some workers and of course a main method.

2.1.1. Workerpool Process

The workerpool process is shown in listing 1.

The implementation demonstrates process creation, channel connection/creation and channel communication in CSP.NET. Defining a process is very simple, just implement the *Run*-method on the *ICSProcess* interface.

```
class WorkerPool : ICSProcess
{
    private IChannelIn<ICSProcess> listenChannel;
    private IChannelOut<ICSProcess> workers;

    public WorkerPool(IChannelOut<ICSProcess> workerChannel)
    {
        workers = workerChannel;
    }

    public void Run()
    {
        listenChannel = Factory.GetAny2OneIn<ICSProcess>("WorkerPool");

        while (true)
        {
            ICSProcess p = listenChannel.Read();
            workers.Write(p);
        }
    }
}
```

Listing 1. Workerpool process code

The first thing to notice about the channels in the *WorkerPool* class is that they are declared as *IChannelIn* and *IChannelOut* channels. All CSP.NET channels implement these interfaces and hence the *WorkerPool* constructor will accept any of the four available channels. This is the standard way to declare channels in CSP.NET.

Connecting to a channel or creating a channel is often done through the static *Factory* class like the *listenchannel* in listing 1. In this particular case a named *Any2One* channel is created but the *Factory* class includes get-methods for every available channel.

Creating named channels involves two steps - creating the channel object, and connecting to the channel object. As explained in section 3.1 the channel is always created on the *One*-end of channels. That means that an *Any2One* channel is created by the reader and an *One2Any* channel is created by the writer. The *One2One* channel has two *One*-ends and is, by definition, created by the writer. The other end of the channel simply has to connect to the channel object. If it tries to connect to a channel before it has been created the process will block until the channel object is created by another process. To avoid deadlock when creating and connecting to named channels, it is recommended only to create named channels in the *run*-method of CSP processes.

Another notable point in listing 1 is the channel communication. The *Write*-method is used for writing data to a channel and the *Read*-method is used for reading data from a channel. Every channel in CSP.NET has a *Write*- and a *Read*-method.

2.1.2. Workers

With the workerpool process in place we move on to the implementation of the workers, which is shown in listing 2. The *Worker* process is similar to the *Workerpool* process and this is the typical appearance of processes in CSP.NET.

A *Worker* simply reads a process from a channel before executing it by calling it's *Run*-method. The process will run in the *Worker*'s thread of execution and no new processes will be accepted until the current process is done.

2.1.3. Main Method

The only thing left is our main program shown in listing 3.

The *Init*-method informs CSP.NET about the location of the Name Server(port number and IP-address) and registers the port number and IP-address of the current program. The

```
class Worker : ICSProcess
{
    private IChannelIn<ICSProcess> processes;
    private ICSProcess process;

    public Worker(IChannelIn<ICSProcess> listenChannel)
    {
        processes = listenChannel;
    }

    public void Run()
    {
        while (true)
        {
            process = processes.Read();
            process.Run();
        }
    }
}
```

Listing 2. Worker process code

```
class Program
{
    static void Main(string[] args)
    {
        CspManager.Init("9091","9090");
        One2AnyChannel<ICSProcess> chan = new One2AnyChannel<ICSProcess>();

        Worker[] workers = new Worker[3];
        for (int i = 0; i < 3; i++)
            workers[i] = new Worker(chan);

        WorkerPool workerPool = new WorkerPool(chan);

        new Parallel(new ICSProcess[]{workers[0], workers[1], workers[2],
                                      workerPool}).Run();
    }
}
```

Listing 3. Main program code

Init-method must be called in the beginning of every CSP.NET program. In this case the Name Server is running on the local machine and we only have to supply the port numbers.

As opposed to the *Workerpool* process that created a named channel the main program creates an anonymous channel. It's possible to create anonymous channels through the *Factory* class but normal instantiation is often used instead.

To get our workerpool running we simply instantiate the workerpool process and the worker processes in listing 1 and 2 before using the *Parallel* class to run them in parallel. Note that the *Parallel* class doesn't run processes on remote machines, but only locally.

2.2. Alternative and Distributed Parallel

Our second CSP.NET program demonstrates the *Alternative* class and the *DistParallel* class. It contains two *AddIntegers* processes that repeatedly add a sequence of numbers and return the result, and an *AltReader* process that reads and process the results. The processes are shown in listing 4 and 5.

Again we notice the familiar pattern of a CSP.NET process and we notice the *serializable* attribute used to make a class serializable. This is necessary in order to distribute the processes to remote machines.

```
[Serializable]
class AddIntegers : ICSProcess
{
    private int number;
    private string channelName;

    public AddIntegers(int num, string name)
    {
        number = num;
        channelName = name;

    }

    public void Run()
    {
        IChannelOut<int> result = Factory.GetOne2OneOut<int>(channelName);

        for(int j = 0; j < 100; j++)
        {
            int res = 0;
            for (int i = 1; i < number; i++)
                res += i;

            result.Write(res);
        }
    }
}
```

Listing 4. AddIntegers process code

Listing 5 demonstrates the use of the *Alternative* class described in section 1.3. Notice the presence of the *CSTimer*, causing the program to terminate if new data isn't available within one second.

The main program, shown in listing 6, is trivial. *DistParallel* distributes the processes to available remote machines running the CSP.NET workerpool Service. Like the *run*-method in *Parallel*, *DistParallel*'s *run*-method blocks until all processes has been executed once.

It's appropriate to use the *DistParallel* in a lot of applications, but there is of course a number of problems where automatic distribution of the processes aren't appropriate, e.g. peer two peer applications. CSP.NET includes the *DistParallel* class to be used when appropriate.

2.3. Transparency

As pointed out earlier one of the main objectives in CSP.NET is to keep the architecture transparent to the programmer and the last example program exhibits this transparency. The programmer doesn't have to pay any attention to the architecture because the *DistParallel* class will utilise remote workers if they are available, and otherwise create and use local workers.

3. Implementation Details

3.1. Distributed Applications and .NET Remoting

In CSP.NET distributed applications are not only applications residing on different machines and communicating through the network. They can also be applications that consist of multiple communicating programs on a single machine, or they can be a combination of the two. Regardless of the number of machines and programs involved, all communication between distributed applications is done through .NET remoting.

```
[Serializable]
class AltReader : ICSProcess
{
    public void Run()
    {
        AltingChannelIn<int> plus = Factory.GetOne2OneIn<int>("plus");
        AltingChannelIn<int> minus = Factory.GetOne2OneIn<int>("minus");

        CSTimer timer = new CSTimer();
        Alternative alt = new Alternative(new Guard[] { plus, minus,
                                                        timer });
        int result = 0;
        bool done = false;
        while (!done)
        {
            timer.RelativeTimeOut(1000);
            switch (alt.FairSelect())
            {
                case 0:
                    result += plus.Read();
                    break;
                case 1:
                    result -= minus.Read();
                    break;
                case 2:
                    done = true;
                    break;
            }
        }
    }
}
```

Listing 5. AltReader process code

```
class AltProgram
{
    static void Main(string[] args)
    {
        CspManager.Init("9092", "9090", "192.0.0.1","192.0.0.2");

        new DistParallel(new ICSProcess[]{new AltReader(),
                                          new AddIntegers(100, "plus"),
                                          new AddIntegers(100, "minus")
                                         }).Run();
    }
}
```

Listing 6. Distributed-parallel program code

Remoting is a simple way for programs running in one process to make objects accessible to programs in other processes, whether they reside on the same machine or on another machine on the network. Remoting is similar to Java RMI and is very easy to customise and extend. That is exactly what is done in CSP.NET in order to facilitate code transfer between machines. The structure of the remoting system in CSP.NET is shown in figure 2. In the CSP.NET case the formatter sink is a binary formatter that serializes all messages into a relatively compact binary format for efficient transport across the wire. The transport sink uses a simple TCP-channel which is fast and reliable.

Given the flexibility of Remoting it would be trivial to replace the Binary formatter with a SOAP formatter or the Transport sink with a HTTP channel. The CSP.NET sink and the CSP.NET proxy are discussed in section 3.3. All constructs in CSP.NET, that are in some way distributed, uses remoting behind the scenes. That goes for named channels like *One2One*,

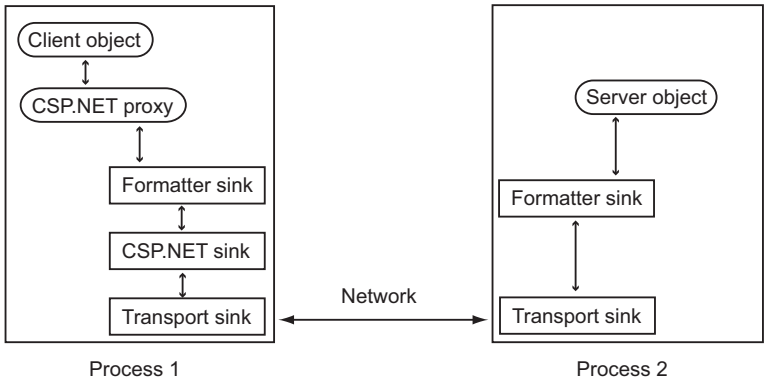

Figure 2. The remoting system structure in CSP.NET

Any2One and *One2Any* as well as for the named versions of *Barrier* and *Bucket*. With regard to the channels the real object - the server object - is always placed on the *One*-end and the proxies are created on the *Any*-ends. That means that processes on the *Any*-end can go out of scope without problems, as they only contain a proxy and not the real object. Only when the ”One” end goes out of scope does the channel cease to function. On channels where no *One*-end exists it is impossible to know where best to put the real object and the user runs the risk of breaking the channel every time a process using the channel goes out of scope. It’s possible to implement an *Any2Any* channel through a combination of an *Any2One* channel and an *One2Any* channel but the performance of such a channel would not be on par with the other channels in CSP.NET. That is the reason no named *Any2Any* Channel exists in CSP.NET.

While remoting has many advantages it is probably not as efficient as a tailor-made solution. In order to minimise the performance overhead we have chosen the fastest combination of formatter sink and transport sink. To further make sure that optimal speed is obtained wherever possible, CSP.NET offers a few mechanisms that optimise named constructs that do not cross process boundaries - see section 3.2.

3.2. Standalone Applications

When programming applications that are designed to run on more than one processor or machine, it is often impossible to know the exact runtime environment in advance. If a program is designed never to be run on multiple machines or communicate over the network there is no need to use named channels, barriers or buckets.

If it is uncertain whether the application will run on one or more machines it is wise to take that fact into consideration when designing the application. That would typically be done by using processes communicating through named channels and using named barriers and buckets in the parts of the program most likely to be run on separate machines. Alternatively it could mean using workerpools to offload heavy computations to other machines if present. Under normal circumstances that would mean poorer performance given the overhead incurred by the use of Remoting.

To ensure that all applications that use named constructs and workerpools run at optimum speed CSP.NET features a method called *CspManager.InitStandAlone*

By calling *CspManager.InitStandAlone* at the beginning of a program you tell CSP.NET that the application does not share named CSP.NET constructs with other applications. Nor will there be any remote workerpools available at runtime, and local workerpools will be used instead. That means that neither a stand-alone Name Server nor the use of Remoting is necessary, which results in maximum performance. *CspManager.InitStandAlone* is particularly useful when the program is designed for both distributed and non-distributed environments. In the non-distributed case the use of *CspManager.InitStandAlone* will yield the same performance as if the application had been written specifically for a single machine.

In distributed applications where *CspManager.Init* is called at the beginning of the program all named CSP.NET constructs are by default distributed. That means that they are created through the use of the stand-alone Name Server and communicate through Remoting. There might be situations, even in distributed applications, in which local named CSP.NET constructs are desirable. To accommodate such needs all named CSP.NET constructs can be created with a parameter specifying that they should bypass remoting and the Name Server Service.

3.3. Remote Code Transfer

The distributed nature of CSP.NET means that it is possible for a program on one machine to send an object over a distributed channel for another machine to use. If the object is an instance of an ordinary class any program that sends or receives that object would need the correct assembly, in order to compile. It is another matter if the channel is defined to transport objects implementing a specific interface and the programs on either side only invokes methods that are defined on that interface. Only the interface needs to be known for the program to compile - that would be the case with a channel transporting CSP processes implementing the *ICSProcess* interface. Even though the program may compile it will certainly throw an exception when run if the code that implements the interface is missing.

We want CSP.NET to handle exceptions thrown because of missing assemblies without the user noticing that anything is amiss. One way to solve the problem would be to continuously monitor objects sent through named channels. If the code for the object was unavailable on the other end of the channel the missing code would be sent over the channel prior to the actual object. Another possibility would be to always send the relevant code along with the object. The first option would mean a lot of unnecessary communication over the network and the second would mean sending a lot of redundant code.

To avoid chatty interfaces and redundant code transfers CSP.NET employs the flexibility of the Remoting system to deal with missing assembly files. As illustrated in figure 2 a custom proxy and a custom sink are inserted into the Remoting system on the client side. Any exception thrown because of missing code is caught in either the proxy or the sink. When that happens a recursive check is made of the missing assembly and any other referenced assemblies. All the missing assemblies are then copied to the machine on which they are needed. They are placed in a special location that CSP.NET always checks when looking for assemblies.

3.4. ThreadPool

At the heart of any implementation of CSP lies the management of the threads used when a set of CSP Processes are run in the *Parallel* construct. How the thread management is implemented depends on various considerations: how many threads do we want running concurrently, how transparent should the thread management be and what limitations do the OS impose on the use of threads.

The design of the thread management system in CSP.NET is very much dictated by the fact that it is meant to be a native .NET implementation of CSP. Lightweight threads - in Windows called fibers - are not available in the .NET API, which means that CSP.NET uses real heavyweight OS threads.

As thread creation and destruction are relatively heavy operations, CSP.NET implements a threadpool to manage busy and free threads. The *Parallel* class and the *CSPThreadPool* class are closely connected, as the threadpool makes sure that all CSP processes run in a *Parallel* are allocated a free thread. After a parallel run completes, all threads are returned to the threadpool but they are not released, unless the programmer explicitly requests the threadpool to do so. The principle of always freeing, but not releasing, threads that are not

currently executing means that a CSP.NET program will never use more threads than are executing concurrently.

The default, and recommended, stack size of threads in .NET is 1 MB. That means that programs with many CSP processes executing concurrently, requires lots of memory to run, compared to sequential programs. In CSP.NET it is possible to manually control the maximum stack size of new threads. That means that processes that are known to be frugal in their use of stack memory can be allocated less memory.

It would have been entirely possible to use the existing ThreadPool class in .NET but that would have meant less flexibility. The .NET threadpool does not allow for the programmer to specify neither maximum stack size nor thread priority, thus making it impossible to assign individual priorities to individual threads. By managing our own threadpool none of those limitations apply to CSP.NET.

3.5. Workerpool and Distributed Parallel

Workpools are often used in distributed applications to achieve load balancing and the workerpool[1] in CSP.NET is very similar to a standard centralised workpool [10]. It's implemented as a Windows Service and once started the Service will register a listenchannel and the number of available workers on the Name Server. The Name Server is responsible for managing the workers and hence every process in need of a worker must contact the Name Server, which is done implicitly through the *DistParallel* class.

Whenever a *DistParallel* object requests a worker from the Name Server a listenchannel, connecting the object to a workerpool with free workers, is returned and the number of available workers on the specific workerpool is decremented. The worker itself will read and run one *ICSProcess* before informing the Name Server that it is ready for new jobs/processes. At the same time, the worker informs the *DistParallel* object that it is done, hence allowing the *Run*-method to return when all processes have executed once. The Name Server will increment the number of available workers for the given workerpool.

By keeping score of available workers we ensure that processes only connect to workerpools with free workers thereby avoiding starvation. The downside of this approach is that we have to contact the Name Server in order to get free workers and we can only write one process to each worker.

4. Tests

Even though we attempt to demonstrate the performance of some aspects of CSP.NET, this section is by no means intended to be a thorough and comprehensive benchmark. No attempt is made to compare CSP.NET to other paradigms and only one small comparison is made with existing CSP libraries.

All single machine performance tests have been run on a machine containing: Pentium 4M 2 GHz processor, 512 MB ram, Windows XP SP2, .NET 2.0, Java 5.0. The distributed test was run on the single machine setup plus a second machine containing: Pentium M 2GHz, 1 GB ram, Windows XP SP2 and .NET 2.0. The two machines were connected directly through a 100 Mbps network.

4.1. Channel Performance

We have not done any extensive performance comparisons between CSP.NET and other implementations of CSP like JCSP, C++ CSP and KRoC(occam). Mostly because the main fo-

[1]We are deliberately using the term workerpool instead of workpool, since it's a pool of workers and not a pool of work/tasks.

cus has been ease of use and transparency rather than high performance, but also because comparing the performance of CSP.NET to C++ CSP and KRoC is irrelevant given the differences in functionality, see section 3.4. Given the similarities of CSP.NET and JCSP we have done one small test to illustrate the performance of the various channels. The test consist of three processes. *Process A* send an integer to *Process B* which reads the number and sends it to *Process C* which in turn reads the number and sends the number back to *Process A*. That means that *Process A* only sends another number when the previous one has passed through all the processes in the loop.

Channel type	**JCSP**(jre 5.0)	**CSP.NET**(.NET 2.0)
One2One	16975	12488
Any2One	17165	17825
One2Any	16854	12257
Any2Any	17125	17524

Table 1. Comparing the performance of different anonymous channel types in JCSP and CSP.NET. All times are in milliseconds.

Table 1 shows the results of the test. The numbers are in milliseconds and indicates the time it takes to send 500,000 integers through the loop.

The measurements reveal some clear differences between JCSP and CSP.NET. For all channels in JCSP the time is around 17 seconds, which is about the same as the *Any2One* and *Any2Any* channels in CSP.NET. The two remaining channels, *One2One* and *One2Any*, are somewhat faster in CSP.NET. It is impossible to tell if the differences in performance are due to the implementation of CSP.NET and JCSP, or if they are caused by differences between Java and .NET.

4.2. Performance Overhead

To measure the performance overhead incurred by using multiple threads on a uni-processor machine we have run a series of tests based on the Monte Carlo Pi algorithm. Monte Carlo Pi is well suited to measure performance overhead when going from sequential execution to concurrent execution because the algorithm can be divided into as many, almost independent parts, as there are threads of execution. Thus in an ideal world, with zero time context switches, the sequential and the parallel version should run equally fast, and the speedup using multiple processors should be linear or better - taking the increased number of cache hits into consideration. Listing 7 shows the sequential version and listing 8 shows the parallel version.

We have run two different Monte Carlo Pi tests. The first one runs Monte Carlo Pi for a fixed number of iterations, with varying numbers of threads - from 1 to 600. To fit 600 threads into memory we use a thread stack size of around 200 kilobytes. Table 2 shows the rather surprising results.

The concurrent version is consistently marginally faster than the sequential version, that runs in 112242 milliseconds, indicating that running a CSP.NET program with quite a number of threads on an idle machine is quite feasible. We should add that the machine is much more responsive when running the sequential test, meaning that machines that has to do other work than just computing Monte Carlo Pi might benefit from less threads being run.

It seems that the relatively long run time of the test program hides the overhead of thread creation and context switching. To better illustrate that the sequential version of Monte Carlo Pi is much faster at low iteration counts we have done a comparison shown in table 3. Here both the sequential and the parallel version of Monte Carlo Pi have been run at varying iteration counts. The parallel version has the number of threads fixed at 400 in all runs.

Iterations	Number of threads	Par Time
1,200,000,000	1	109371
1,200,000,000	50	110121
1,200,000,000	100	109989
1,200,000,000	200	110092
1,200,000,000	300	110579
1,200,000,000	400	110889
1,200,000,000	600	111480

Table 2. Monte Carlo Pi with fixed number of iterations and variable number of threads. All times are in milliseconds. Sequential time: 112242 milliseconds.

Iterations	Par Time	Seq Time
400	225	1
4000	224	1
40000	225	5
400000	262	42
4000000	595	378
40000000	3998	3798
400000000	37062	37258

Table 3. Parallel Monte Carlo Pi versus sequential Monte Carlo Pi at different iteration counts. Parallel version fixed at 400 threads. All times are in milliseconds.

Looking at table 3 it is evident that the cost of creating the threads completely dominates when the number of iterations is low. That makes perfect sense as creating one thread to do one simple calculation is a complete waste of time. But the numbers also confirm the fact that when the workload of each thread increases the overhead of thread creation will eventually become almost invisible.

4.3. Distributed Performance

To illustrate the performance overhead of going distributed, we have extended our Monte Carlo Pi test to run across two machines. The work is divided into two parts of equal size. One part is sent to a second machine for processing and the other part computed locally. When computation is done on the remote machine the data is sent back and used in the calculation of Pi.

Table 4 shows the results of the distributed version compared to the sequential version of Monte Carlo Pi. We varied the number of iterations while keeping a the number of threads fixed at 10 on each machine.

Iterations	Dist Time	Seq Time
80	497	1
800	782	1
8000	563	2
80000	817	8
800000	953	53
8000000	1214	745
80000000	4279	7281
800000000	38619	73117

Table 4. Distributed Monte Carlo Pi with variable number of iterations. Each machine in the distributed test uses 10 threads. All times are in milliseconds.

It is not surprising that the sequential version of Monte Carlo Pi is vastly superior when the number of iterations in each of the 20 processes is low. The overhead of remoting and network communication clearly dominates at low iteration counts, but the time is still well below one second. When the time exceeds a couple of seconds the distributed version comes into its own and is much faster than the sequential version. A speedup of 2 is almost achieved and even though the theoretical maximum speedup is unknown, the performance gain is significant. We have included the code for the distributed Monte Carlo Pi in the appendix to show an example of a distributed CSP.NET program that doesn't use workerpools and *DistParallel*. The program running on the server side is shown in listing 9 and the client program is shown in listing 10.

5. Conclusions

CSP.NET is a new implementation of the CSP paradigm suitable for both distributed-memory multicomputers and shared memory multiprocessor systems. A lot of functionality is provided in the library but some work remain e.g. robust error handling.

Future developments include channel poisoning known from C++CSP [3] and JCSP [1], user definable Name Servers and of course further work needs to done on the workerpool. *DistParallel* could also be extended to provide exactly the same methods and functionality as the normal *Parallel* class, making the boundary between distributed applications and local applications disappear completely. A thorough benchmark comparing CSP.NET to other libraries and paradigms would also be a good idea.

We hope that CSP.NET will introduce new programmers to the CSP paradigm and advocate CSP as the right choice for parallel and concurrent programming in Microsoft.NET. The library will be available on the website *www.cspdotnet.com*.

References

[1] Alastair R. Allen and Bernhard Sputh. JCSP-Poison: Safe Termination of CSP Process Networks. In *Communicating Process Architectures 2005*, pages 71–107. IOS Press, Amsterdam, Sept 2005.

[2] N.C.C. Brown. C++CSP Networked. In I.R. East, D. Duce, M. Green, J.M.R. Martin, and P.H. Welch, editors, *Communicating Process Architectures 2004*, pages 185–200. IOS Press, Amsterdam, 2004.

[3] N.C.C. Brown and P.H. Welch. An Introduction to the Kent C++CSP Library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156. IOS Press, Amsterdam, 2003.

[4] Communicating Sequential Processes for Java. www.cs.kent.ac.uk/projects/ofa/jcsp/.

[5] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[6] P.H.Welch, J.R.Aldous, and J.Foster. Csp networking for java (jcsp.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002.

[7] Nan C. Schaller, Gerald H. Hilderink, and Peter H. Welch. Using Java for Parallel Computing - JCSP versus CTJ. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226. IOS Press, Amsterdam, Sept 2000.

[8] Peter H. Welch and Brian Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 203–222. IOS Press, Amsterdam, Sept 2002.

[9] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166, Amsterdam, The Netherlands, 1996. World occam and Transputer User Group, IOS Press. ISBN: 90-5199-261-0.

[10] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

A. Monte Carlo Pi Source Code

A.1. Sequential Monte Carlo Pi

```
private static void CalcMonteCarloPiSeq()
{
    double x, y, area;
    int pi = 0;
    int i;
    Random r = new Random();

    for (i = 0; i < Max; i++)
    {
        x = r.NextDouble() * 2.0 - 1.0;
        y = r.NextDouble() * 2.0 - 1.0;

        if ((x * x + y * y) < 1)
            pi++;
    }
    area = 4.0 * (double)pi / (double)Max;
    Console.WriteLine("Seq Area: pi/Max " + pi + "/" +
                      Max + " = " + area);
}
```

Listing 7. Sequential Monte Carlo Pi

A.2. Parallel Monte Carlo Pi

```
public class Worker : ICSProcess
{
    long iters;
    int pi;
    Random r;
    Barrier ba;

    public Worker(long iterNum, int seed)
    {
        iters = iterNum;
        pi = 0;
        r = new Random(seed);
    }

    public Worker(long iterNum, int seed, Barrier b)
    {
        iters = iterNum;
        pi = 0;
        r = new Random(seed);
        ba = b;
    }

    public int getPI()
    {
        return pi;
    }

    public void Run()
    {
        double x, y;
        if(ba != null)
            ba.Sync();

        for (int i = 0; i < iters; i++)
        {
            x = r.NextDouble() * 2.0 - 1.0;
            y = r.NextDouble() * 2.0 - 1.0;
```

```
            if ((x * x + y * y) < 1)
                pi++;
        }
    }
}

private static void CalcMonteCarloPiPar(int num)
{

    Worker[] processes = new Worker[num];

    for (int i = 0; i < num; i++)
        processes[i] = new Worker(Max / num, 115 + i * 10);


    new Parallel(processes).Run();

    int pi = 0;
    for (int i = 0; i < num; i++)
        pi += processes[i].getPI();

    double area = 4.0 * (double)pi / (double)Max;
    Console.WriteLine("Par Area: pi/Max " + pi + "/" +
                      Max + " = " + area);
}
```

Listing 8. Parallel Monte Carlo Pi

A.3. Distributed Monte Carlo Pi – Server

```
[Serializable]
public class DistControl : ICSProcess
{
    int numThreads, pi;
    long iterNumPerWorker;
    string channelName;

    public DistControl(string name, int threads, long iterPerWorker)
    {
        numThreads = threads;
        iterNumPerWorker = iterPerWorker;
        channelName = name;
    }

    public void Run()
    {
        Worker[] processes = new Worker[numThreads];

        for (int i = 0; i < numThreads; i++)
            processes[i] = new Worker(iterNumPerWorker, 1763 + i * 10);

        new Parallel(processes).Run();

        for (int i = 0; i < numThreads; i++)
            pi += processes[i].getPI();

        IChannelOut<int> resultChannel =
            Factory.GetOne2OneOut<int>(channelName);
        resultChannel.Write(pi);
    }
}

public class LocalDistControl : ICSProcess
{
    string resultChannel;
    int numThreads, pi;
    long iterNumPerWorker;
    Barrier ba;
```

```
        public LocalDistControl(string name, int threads, long iter, Barrier b)
        {
            resultChannel = name;
            numThreads = threads;
            iterNumPerWorker = iter;
            ba = b;
        }

        public void Run()
        {
            IChannelOut<ICSProcess> work =
                Factory.GetOne2OneOut<ICSProcess>("workchannel");

            DistControl dc = new DistControl(resultChannel,
                numThreads, iterNumPerWorker);
            work.Write(dc);
            ba.Sync();
            IChannelIn<int> result = Factory.GetOne2OneIn<int>(resultChannel);
            pi = result.Read();
        }

        public int getPI()
        {
            return pi;
        }
    }

    private static void CalcMonteCarloPiDist(int numlocal, int numdist)
    {
        Worker[] processes = new Worker[numlocal];
        Barrier b = new Barrier(numlocal + 1);

        long localWork = Max / 2;
        for (int i = 0; i < numlocal; i++)
            processes[i] = new Worker(localWork / numlocal,115 + i * 10,b);

        Parallel p = new Parallel();
        long iterPerWorker = (Max-localWork)/numdist;
        LocalDistControl dc = new LocalDistControl("resultChannel5", numdist,
            iterPerWorker,b);
        p.AddProcess(dc);
        p.AddProcess(processes);

        p.Run();

        int pi = 0;
        for (int i = 0; i < numlocal; i++)
            pi += processes[i].getPI();
        pi += dc.getPI();

        double area = 4.0 * (double)pi / (double)Max;
        Console.WriteLine("DistPar Area: pi/Max " + pi + "/" + Max +
            " = " + area);
    }
```

Listing 9. Distributed Monte Carlo Pi – server side

A.4. Distributed Monte Carlo Pi – Client

```
    public class runTest : ICSProcess
    {
        string channelName;

        public runTest(string name)
        {
            channelName = name;
        }
```

```
        public void Run()
        {
            IChannelIn<ICSProcess> work =
                    Factory.GetOne2OneIn<ICSProcess>(channelName);
            ICSProcess p = work.Read();
            p.Run();
            Console.WriteLine("Work done");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            CspManager.Init("9097", "9090","192.0.0.1","192.0.0.2");
            Console.WriteLine("after Init");
            new Parallel(new ICSProcess[] { new runTest("workchannel")}).Run();
            Console.ReadKey();

        }
    }
```

Listing 10. Distributed Monte Carlo Pi – client side