# INVASIVE COMPUTING WITH iOMP

*M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, J. Weidendorfer*

Technische Universität München
Fakultät für Informatik
Boltzmannstr. 3, 85748 Garching, Germany

## ABSTRACT

This paper describes an extension to OpenMP for invasive computing. iOMP is being developed within the framework of the Transregional Collaborative Research Centre 89 (TRR89) funded by the German Research Foundation (DFG). Invasive computing allows the programmer to write resource aware parallel programs. The programs can specify to the resource management which and how many resource are required for execution. In addition, resource aware programs can adapt to a certain amount of resources available to their execution. iOMP is being developed in the context of scientific computing where novel algorithms tend to be more dynamic, e.g., by refining the grid dynamically to increase the quality of the solution in certain areas of the simulated domain. The main advantage of invasive computing in this context is the better utilization of resources compared to a static allocation.

***Index Terms***— Parallel Processing, High Performance Computing, Computer Languages

## 1. INTRODUCTION

iOMP is an extension to the OpenMP application programming interface (OMP) [5] to allow programming of resource aware applications. OpenMP itself has no direct support to manage resources and relies on the operating system (OS). iOMP extends OpenMP with concepts for dynamically allocating and freeing resources. This will allow an overall better machine utilization, higher efficiency, and a lower power consumption.

The idea of invasive computing is to enable the programmer to write resource aware programs. In contrast to the overall trend in programming interfaces to provide more and more abstract or high level interfaces and to automatically map those to the hardware, the goal of invasive computing is to enable the programmer to carefully optimize his program for the available resources. This resource awareness comes in two forms. First, the program can allocate and free resources according to the amount of available parallelism or the dynamic size of the data. Second, it can adapt itself to the size of resources that is available for its execution, for example, by selecting a different algorithm if less cores are available as were requested.

The focus of the Transregional Collaborative Research Centre 89 (TRR89) on Invasive Computing funded by the German Research Foundation (DFG) is on embedded architectures. The goal of the project is to develop an invasive chip which is a heterogeneous multicore chip based on tiles connect via an intelligent on-chip network. X10 was selected as the major programming interface for the invasive chip [6]. X10 is a PGAS language developed in the High Productivity Computing Systems (HPSC) initiative by IBM in the US [1]. In our project X10 was extended for invasive programming with two important concepts. The first one is a claim and the second an ilet. A *claim* is a set of resources which is allocated dynamically via *invade* operations. Additional resources are dynamically invaded and returned to the requesting application in form of a claim. The invaded resources are then infected with a computational task called an *ilet*. The ilet combines code and data for execution. It is written by the programmer and started on a claim via an *infect* operation. Once the application can no longer efficiently use all the allocated resources, the resources are freed via a *retreat* operation.

Invasive computing is an interesting concept for high performance computing (HPC) as well. Current supercomputers work in a space sharing mode, i.e. the nodes of the machine are statically partitioned and programs have exclusive access to their resources. Recent trends in the architecture of supercomputers as well as in algorithms used in applications running on those systems require a more flexible partitioning. Petascale computers consist of hundreds of thousands of cores requiring more and more parallelism to be exploited while modern applications are more and more dynamic, for example, the number of grid points are adapted to the currently computed solution. Thus, they consist of phases of more and less parallelism.

The idea is, to bring together the varying demand of parallelism with the extensive parallelism on hardware level via invasive computing. Adaptive applications do not statically allocate the maximum number of resources they can use, but allocate and free cores according to the available parallelism.

As a first step to enable invasive computing for HPC applications, we are developing iOMP at Technische Universität München. In a later phase of the project we will also look into extensions for MPI, the standard programming interface for distributed systems.

The following chapters specify the language extensions of iOMP, their implementation, as well as examples how they can be used in invasive programs. Section 2 gives a short recap of OMP. Section 3 defines some basic terms used in the language specification and specifies their relation to the terms used in X10. Section 4 presents the invasive language constructs in iOMP. An example is presented in Section 5. Section 6 outlines the current implementation. In Section 7 we demonstrate the benefits of invasive computing in the context of Tsunami simulation. Future extensions are presented in Section 8. The article ends with a discussion of related work and conclusions.

## 2. OPENMP

OpenMP is a standard programming interface for shared-memory parallelism in C, C++, and Fortran programs. OpenMP is implemented as a set of compiler directives, library routines, and environment variables [5].

The parallelization is based on directives that are inserted into the application's source code to define parallel regions that are executed in parallel. Such parallel regions are executed using a fork-join parallelization model. Applications start with a single thread (master thread) that executes the sequential portion of the application. When it hits a parallel region a team of threads is created that also includes the master thread. This operation can be seen as a fork of the execution. The body of the parallel region is then executed redundantly by all threads in the team. At the end of a parallel region all threads in the team are synchronized using a barrier and only the master continues with the subsequent sequential code. This operation can be seen as a join.

The body of a parallel region is executed redundantly by all threads of the team. OpenMP offers worksharing constructs allowing work distribution among all threads in a thread team. These constructs are in C/C++: `pragma omp for`, `pragma omp section` and `pragma omp task`. Listing 1 presents a typical OpenMP code snippet. First a parallel region is started in which one or more worksharing constructs are nested. The iterations of the worksharing loop are distributed blockwise onto the threads.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv)
{
   double a[16];

   #pragma omp parallel
   {
     #pragma omp for schedule(static)
     for(int i=0; i<16; i++)
        a[i] = omp_get_thread_num();
   }

   for(int i=0; i<16; i++)
      printf("%f ", a[i]);

   return 0;
}
```

Listing 1. OpenMP worksharing loop with static scheduling.

Each thread in a team has a unique identification number, the OpenMP thread number. This ID can be accessed within a parallel region using the OpenMP library call `omp_get_thread_num()`. The overall number of threads, the team size, is returned by `omp_get_num_threads()` call. All threads are sharing all data by definition in OpenMP. There is, however, the possibility to make variables and arrays private. This is necessary to solve data dependencies in order to keep the sequential semantics.

There are three ways to control the number of threads within a team. The most frequently used option is the definition of the environment variable OMP_NUM_THREADS. This environment variable is evaluated during the application start and is persistent for the whole lifetime of the application. The OpenMP library call `omp_set_num_threads(int size)` can be used to override the persistent value during the application runtime. The third way is to specify the team size for a parallel region via the `num_threads()` clause as part of the parallel region pragma.

## 3. IOMP TERMINOLOGY

The following terms will be used in the rest of the document:

**PE** : Modern multicore processors provide multiple cores on a single chip. Each of the cores can be multithreaded so that two or more hardware threads share the resources. We will use the term PE as a synonym for logical CPUs induced by the hardware threads.

**Claim** : We introduced claims in the previous section in the context of X10 as a set of resources and will use the term in the same way in iOMP. The major difference is that an iOMP program will work only with a single claim. This claim consists of all the PEs currently allocated for the program. Initially it consists of only a single PE but can grow and shrink under the control of the program and the system resource management.

**Invade** : The invade operation will allocate additional PEs. Which PEs and how many will be specified via constraints. iOMP in its current version supports only the single constraint PEQuantity. The number of requested PEs is the desired overall number of PEs in the claim. Thus, it consists of the already available PEs in the claim plus the additionally allocated PEs.

**Infect** : In iOMP we have no explicit infect operation. OpenMP threads will be automatically started and pinned on the PEs during the invade operation.

**Retreat** : The retreat operation allows to reduce the number of PEs in the claim. How many and which PEs are released is again specified by constraints. At least one PE has to remain in the claim until the termination of the program.

**Thread** : Threads are used in iOMP in the same way as in OMP. A parallel region is executed by threads that are created or taken from a pool.

**Team of threads** : In iOMP the size of the team executing a parallel region is implicitly defined and equal to the number of PEs allocated to the program, i.e. PEs in the claim.

## 4. iOMP INTERFACE

iOMP is implemented as a library in C++ using an object oriented approach that is similar to the invasive X10 implementation. There are several classes that are of interest for the user.

- Claim
- Constraint
- PEQuantity
- AND
- OR

These classes are defined in the following subsections.

### 4.1. Class `Claim`

In general a claim is a set of resources that can be used in an invasive application. In iOMP an application has only a single claim of resources. The claim consists of PEs and can dynamically grow and shrink according to invade and retreat operations. Initially the claim consists of only single PE and the master thread will be started on that PE.

The class `Claim` represents the claim of the iOMP program. It is a singleton since there will be only one claim during the entire execution. Access to the claim is only allowed in sequential code regions, i.e., outside of the lexical and dynamic extend of parallel regions. The claim of an iOMP application contains at least one PE. It provides four methods:

- invade
- retreat
- size
- toString

**void invade(Constraint const &c)** acquires new resources from the resource manager according to the user defined constraints. If there are not enough resources available it throws a NotEnoughResources exception. The invade will never lead to an implicit retreat from resources. Thus, the claim size will always be equal or larger than before the invade operation.

**void retreat(Constraint const &c)** reduces the claim in such a way that it fits to the constraint passed as parameter. The claim of the application has to consist of at least one PE after the execution of retreat. If no constraint is given, the claim will be reduced to one PE.

**int size()** returns the size of the claim.

**string toString()** creates an output string containing the identification of processing elements in the claim.

### 4.2. Class `Constraint and PEQuantity`

A constraint defines a certain restriction that is used when new resources are acquired. It can be seen as a set of rules to specify the developers wishes for new resources during the runtime of an application. The most obvious constraint on shared memory systems is the number of PEs. Other constraints might, for example, take into account the NUMA architecture of the machine and request PEs that are near to PEs already in the claim to optimize memory access. In addition to constraints requesting certain additional cores, the constraints can also specify other types of resources, e.g., memory.

Constraint definitions should allow alternatives, e.g. by using a range of numbers for PEs instead of using a fixed number. The resources available on system level are only known at runtime and might constantly be changing. Rigid constraints would reduce the number of possible choices drastically or make the constraint almost impossible to fulfil.

The Constraint class is actually the base class of a class hierarchy and can be extended in the future according to user requests. The first version of iOMP offers only the constraint PEQuantity. It defines the minimum and maximum quantity of PEs. The derived classes AND and OR allow to flexibly combine constraints.

## 5. iOMP EXAMPLES

The example in Listing 2 shows the usage of iOMP. The application's claim consists of a single PE when the application starts. In the initialization phase of an application there is in general only a limited amount of parallelism and no need for a big quantity of resources.

```
 1 #include <iostream>
   #include "Claim.h"
 3 #include "PEQuantity.h"

 5 int main() {

 7   Claim claim;
     int sum = 0;
 9
     /* acquire resources according to the given
          constraints */
11   claim.invade(PEQuantity(1,3));

13   /* print out the number of processing
         elements - from 1 to 3 PEs */
     std::cout << claim.toString() << std::endl;
15
     /* executing a parallel for loop on the
         given resources */
17   #pragma omp for reduce reduction(+:sum)
     for(int i=0; i < 100000; i++)
19     sum += i;

21   /* free resources and delete pinning */
     claim.retreat();
23
     /* print out the number of processing
         elements - 1 PE */
25   std::cout << claim.toString() << std::endl;
   }
```

Listing 2. The code demonstrates the usage of iOMP

The iOMP API calls are inserted before a parallel region is started. The invade method in line 11 modifies the claim according to the user constraint PEQuantity(1,3), which specifies that 1 to 3 PEs can be used in parallel regions, depending on the available resources. In line 17 a loop is parallelized with OpenMP. For the execution all invaded resources are used to speed up the execution. To be resource efficient it is necessary to know to which number of PEs the parallel region scales and always to stay below this threshold. After the execution is finished the retreat method is called, which reduces the claim to only one PE.

## 6. IMPLEMENTATION

The iOMP implementation consists of two parts: the resource manager and the iOMP library. The iOMP resource management approach is implemented as client/server architecture. There is one resource manager running on a shared memory system, which has a global overview of all resources. Each iOMP application is acting as a client and each resource request is handled by this resource manager. The communication between the application and the resource manager is implemented using Linux interprocess communication (IPC), i.e., message queues.

The resource manager has to be started before an application can be used with iOMP. It acts as service and manages the global resources. The iOMP application contacts the resource manager via the invade and retreat methods. In the current implementation, the manager keeps a global list of free PEs and, for each running application, the last requested maximum of PEs. When an application retreats from cores, not all the cores are immediately given to the application that executes an invade next, but the cores are evenly distributed across all running applications according to their previous requests and reserved until the next invade. This approach allows for the best utilization if all application use the cores with the same efficiency. In the future, we will extend this approach with scalability hints either specified by the application or automatically gathered by the runtime system (see Section 8).

The iOMP library implements the classes described above. The implementation is based on pinning the threads to the PEs in the claim. This is done via the Linux support for NUMA architectures available since Kernel version 2.6. Linux's default scheduler, the Completely Fair Scheduler (CFS), allows that pthreads can be executed on remote NUMA nodes, if it is allowed by the cpuset of the pthread. The cpuset is a flat data structure, a bit array, that represents all PEs in a Linux system. Setting the cpuset to PEs that are located on the same NUMA node, reduces the choices of the scheduler. An invade operation is now implemented by creating the same number of OpenMP threads and pinning a single thread to each of the PEs in the claim. We use the `omp_set_num_threads(int size)` OpenMP intrinsic to ensure that the right size of the thread team is used in the next parallel region.

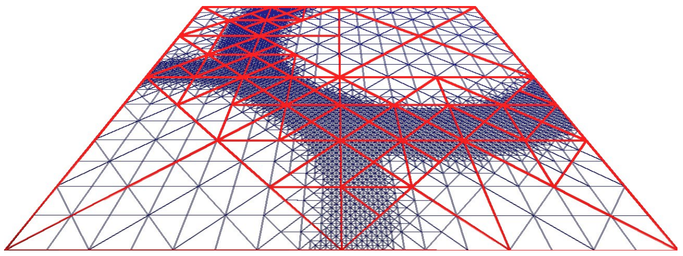

**Fig. 1**. Specific timestep and triangulation of a simulation with initially two waves. Partitions that are handled as individual OpenMP tasks are highlighted in red.

## 7. EXPERIMENTS

In order to demonstrate the advantages of invasive computing we implemented a shallow-water model, being the basic algorithm for current state-of-the art tsunami simulations. During the wave propagation through the simulation domain one of the error reduction criterias on grids is to refine the grid at wave fronts requesting high resolutions in this areas. Using such a fine grid in all areas of the simulation domain creates an enormous computation overhead leading to the demand for adaptive grids.

Our simulation framework[1] offers the capability of solving 2D hyperbolic problems on a fully-adaptive grid with dynamical refinement and coarsening. This simulation framework stores the underlying simulation grid in a memory efficient way using a linearized binary encoded tree structure. The adaptivity in this simulation uses the Sierpinski space filling curve (SFC) in combination with a stream and stack-based storage and communication scheme to create inherently cache efficient memory accesses [3, 2].

An invasive parallelization of the traversal of this SFC is one of our tasks in the TRR89 and is realized by a dynamic tree splitting approach creating by far more partitions than there are cores available on the system. A partition is split once a specific threshold on the number of grid cells in the partition is exceeded. This approach tries to balance the number of grid cells across the partitions. Kernels on the partitions, e.g. to advance the simulation by one time-step, are then executed by creating OpenMP tasks, as shown in Figure 1.

Since the number of cores can only be changed outside of a parallel region, the OMP parallel region comprises the loop body of the timestep loop as depicted in Listing 3. An *invade* is executed before each time-step. The constraint requests always the maximum number of cores. This enables the resource manager to optimize the number of cores for each individual simulation time-step based on the requests of other concurrent iOMP applications.

```
...
Claim claim;
for (int i = 0; i < max_timesteps; i++)
{
    claim.invade(PEQuantity(1, 1024));

    #pragma omp parallel
    #pragma omp single
    {
        // SINGLE SIMULATION TIMESTEP
        // using #pragma omp task
        ...
    }

    claim.retreat();
}
...
```

Listing 3. iOpenMP parallelized simulation

The scalability of our simulation strongly depends on the number of average grid cells during the simulation and increases with the problem size and the refinement. Running several iOMP applications in parallel improves the overall throughput since the application can operate with less cores with higher efficiency.

For the benchmark we used a square sized domain with an initial grid-refinement depth set to 12 and the maximum grid-refinement depth set to 18. We used 0th order basis functions with Lax-Friedrichs flux solvers for the underlying SWEs with an explicit 1st order time-stepping method. Initial conditions were set to a radial dam break scenario with simulation time-steps being executed until the coarsest resolution of the grid was reached.

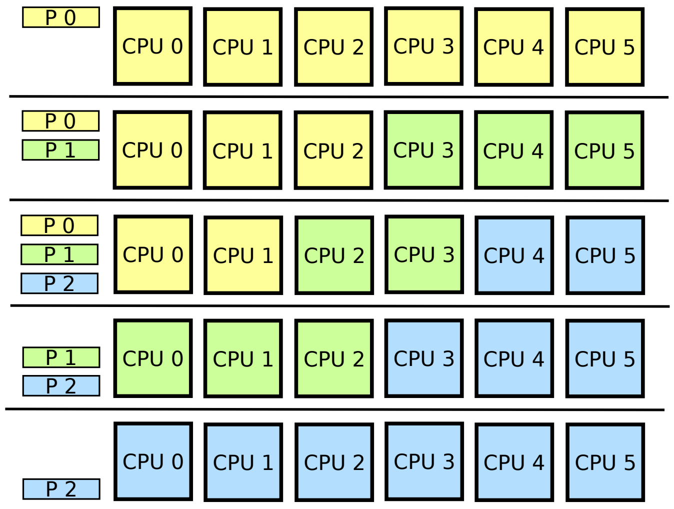

**Fig. 2**. Pinning of applications to cores.

[1]http://www5.in.tum.de/sierpi/

### 7.1. Measurements

We conducted our measurements on a ccNUMA quad-socket 10-core Intel Xeon E7 4850 (2.00 GHz) based machine.The resource manager was configured to evenly distribute the resources among all running applications. The simulations are started one after another and the resources are distributed by the resource manager. The first simulation gets all resources, since all resources are idle at the beginning.

A redistribution of the resources is triggered when the next simulation is started. The second simulation will try to invade additional cores but all the cores are still used. When the first application retreats from the core at the end of the time loop and invades again cores in the next iteration, it will only receive half of the cores since the resource management reserved the other half invade for the second simulation due to the last unsuccessful invade. The second one will receive those cores when it executes the next invade at the start of the next iteration.This behavior is shown in an example with 3 applications and 6 CPUs in Figure 2.

In this way we always utilize all 40 cores of our machine and improve the overall efficiency. Figure 3 shows the overall triangle throughput of all the concurrent Tsunami runs. Due to the invasive implementation of the application, the throughput increases when additional simulations are started dynamically.

iOMP applications do suffer some overhead introduced by the IPC communication between the client and the resource manager. For rather small problem sizes of our simulation we measured an overhead of about 20 percent. For simulations with big problem sizes the overhead is neglectable since the time spent in the parallel region dominates the overall time of the application.

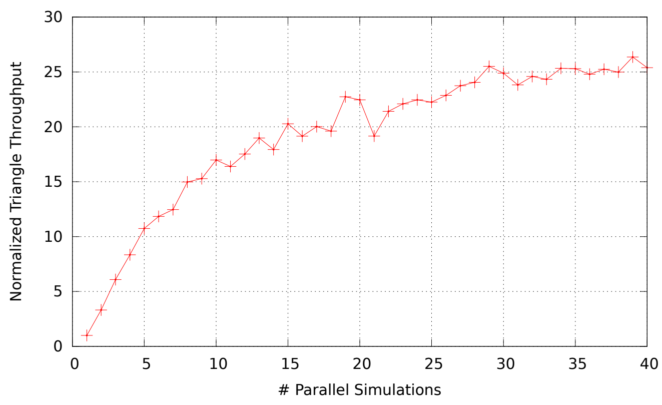

**Fig. 3**. Improved triangle throughput by executing more applications in parallel with appropriate pinning.

## 8. FUTURE WORK

During a Tsunami simulation the problem size strongly changes and therefore the scalability. Instead of having a uniform distribution of resources across all concurrent simulations it is more efficient to adapt the resources to the actual scalability of the individual simulations. Therefore the scalability information needs to be made available to the resource manager. This can either be done by the application itself via scalability hints or by enabling the runtime to measure the parallel efficiency on the fly.

The current version of iOMP does not support optimization of locality. Resources, i.e. PEs, are given to an application for exclusive usage, however, depending on the processor architecture PEs might share the last level cache, the bandwidth to the memory controller and also the interconnection bandwidth. To use resources efficiently it is crucial to minimize remote memory accesses and to minimize cache thrashing effects. In order to achieve this, it is necessary to know the exact system and PE architecture and the memory access patterns of the application.

Extracting all available information about the system architecture is possible, since it is provided by the Linux kernel. Identifying memory access patterns statically from applications is much harder and for data dependent access patterns useless. Instead of trying this approach, it is better to let the application developer give hints to the resource management about the locality requirements for the invaded cores.

## 9. RELATED WORK

Both OpenMP and MPI, as the major programming paradigms in HPC, favor a statical assignment of resources to programs, especially considering the number of cores assigned or the access to memory. Applications are currently not programmed to dynamically allocate and release resources. In a negative feedback loop, this also inhibits the development of more versatile programming models. As a result, programs with varying degree of parallelism either are run on the smallest number of processors that can efficiently be used all the time or on a larger set of processors with accepting that during some periods processors are not efficiently used.

The only notable alternative to MPI and OpenMP is the class of Partitioned Global Address Space languages (PGAS), for example, Co-Array Fortran [8], UPC [4], Chapel [7], and X10 [1]. These languages require the user to take care of data distribution but provide features to access remote data without explicit send and receive constructs. These languages also do not have support for dynamically changing the number of resources used.

Beyond the view of a single application, the management of the resources on an HPC system is currently often delegated to a batch scheduling system. The user submits jobs via batch scripts to the batch system which schedules the jobs on the machine in such a way that the resource utilization is optimized. All the large HPC systems are used in a space sharing mode where a set of processors is allocated exclusively for each job. As the set of processors is fixed for the entire execution, a pessimistic allocation of resources will waste performance on a global scale. An important question for the success of invasive computing in HPC is therefore also whether batch scheduling systems can and will be adapted to offer more flexible dynamic means for resource management.

## 10. CONCLUSIONS

In this paper we describe an approach to make OpenMP applications resource aware. Our extensions in iOMP enable to change the amount of resources available to an application during runtime and thus allows to adapt the resources to the actual needs of the application. We have defined a flexible interface to describe various resource requirements and alternatives of resources. The experiments show that the overall machine throughput can be drastically improved by dynamically distributing the cores to concurrent applications. Further work is required to investigate better metrics upon we can decide about the distribution of the resources, e.g. with respect to the application's scalability, the machine architecture, as well as the application's memory access patterns.

## 11. ACKNOWLEDGEMENT

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

## 12. REFERENCES

[1] Introducing x10. http://x10-lang.org/home/introduction.html.

[2] M. Bader, C. Böck, J. Schwaiger, and Csaba A. Vigh. Dynamically Adaptive Simulations with Minimal Memory Requirement - Solving the Shallow Water Equations Using Sierpinski Curves. *SIAM Journal of Scientific Computing*, 32(1):212–228, 2010.

[3] M. Bader, S. Schraufstetter, C. A Vigh, and J. Behrens. Memory Efficient Adaptive Mesh Generation and Implementation of Multigrid Algorithms Using Sierpinski Curves. *International Journal of Computational Science and Engineering*, 4(1):12–21, 2008.

[4] UPC Consortium. *UPC Language Specification V1.2*, 2005.

[5] OpenMP Forum. *OpenMP Application Program Interface, Version 3.1, July 2011*, 2011.

[6] Frank Hannig, Sascha Roloff, Gregor Snelting, Jürgen Teich, and Andreas Zwinkau. Resource-aware programming and simulation of mpsoc architectures through extension of x10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '11, pages 48–55, New York, NY, USA, 2011. ACM.

[7] Cray Inc. *Chapel Language Specification 0.702*, 2006.

[8] R.W. Numrich and J. Reid. Co-Array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.