Experiences Parallelizing a Web Server with OpenMP

J. Balart, A. Duran, M. Gonzàlez,
X. Martorell, E. Ayguadé, J. Labarta

CEPBA-IBM Research Institute
Computer Architecture Department
Universitat Politecnica de Catalunya

Outline

- Motivation
- The Boa web server
- Parallelizations
- Evaluation
- Experiences
- Conclusions

Motivation

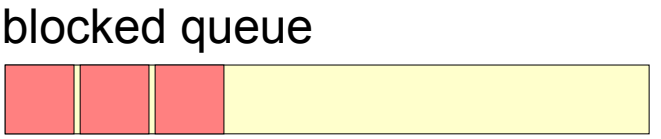
- OpenMP has been successful for numeric applications
  - The API has been influenced by these applications
- New parallel applications are emerging
  - with new needs
- Objective: Explore a new kind of application

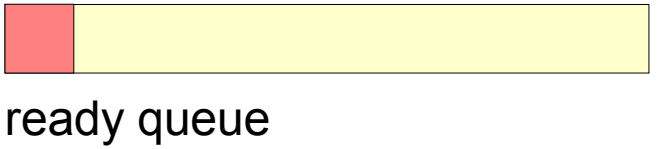
Boa

- Single threaded event-driven architecture
  - Does not use a thread/process by connection
- Multiplexes requests over a single thread
  - Round Robin scheduler
    - Two queues: ready & blocked
    - Requests are processed by chunks
- Uses non-blocking I/O for sockets
- Uses mmap for local files
  - Maintains a cache of open files to avoid remappping

Boa: main loop

```
for ( ; ; ) {
    process signals
    unblock requests
    accept new connections
    process ready requests
    select();
}
```



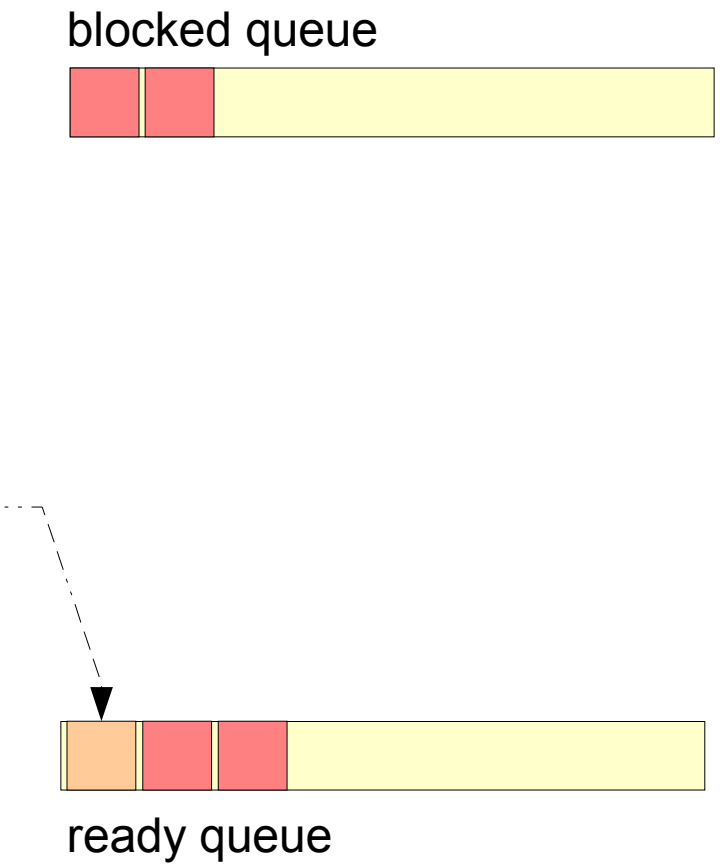


Boa: main loop

```
for ( ; ; ) {
    process signals
    unblock requests
    accept new connections
    process ready requests
    select();
}
```

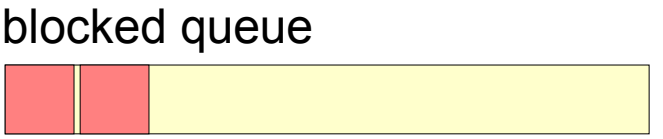
Boa: main loop

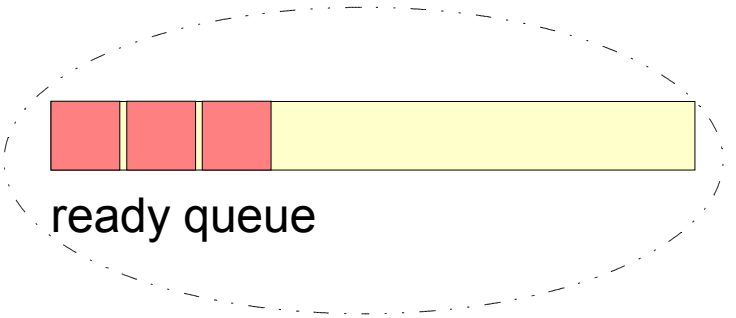
```
for ( ; ; ) {
    process signals
    unblock requests
    accept new connections
    process ready requests
    select();
}
```



Boa: main loop

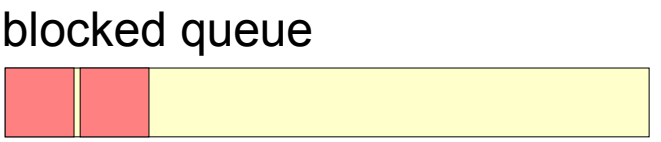
```
for ( ; ; ) {
    process signals
    unblock requests
    accept new connections
    process ready requests
    select();
}
```

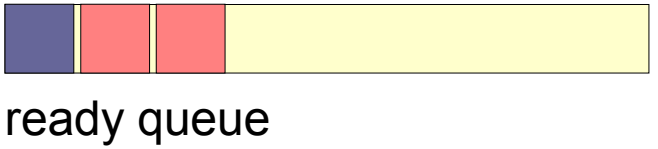




Boa: request processing loop

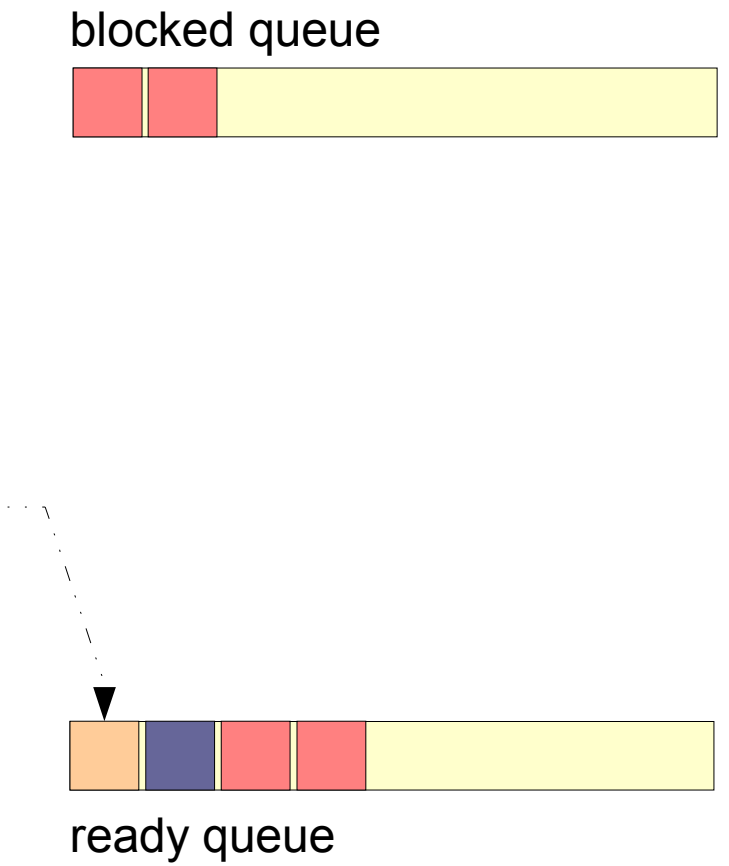
```
for each ready request {
    result = work_on(request);
    accept new connections
    if ( result == BLOCK )
        block(request);
    if ( result == FINISHED )
        free(request);
    else keep it ready;
}
```





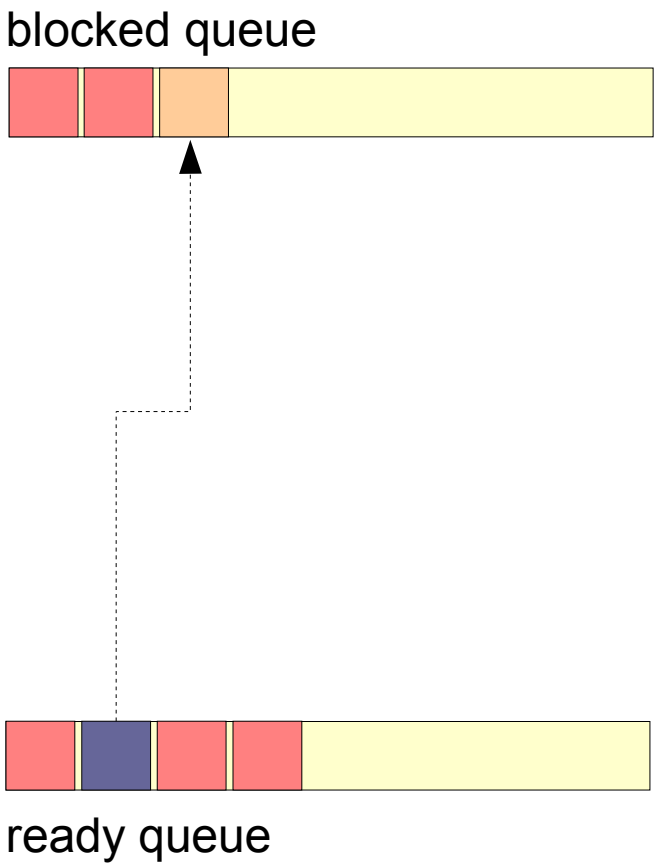
Boa: request processing loop

```
for each ready request {
    result = work_on(request);
    accept new connections
    if ( result == BLOCK )
        block(request);
    if ( result == FINISHED )
        free(request);
    else keep it ready;
}
```



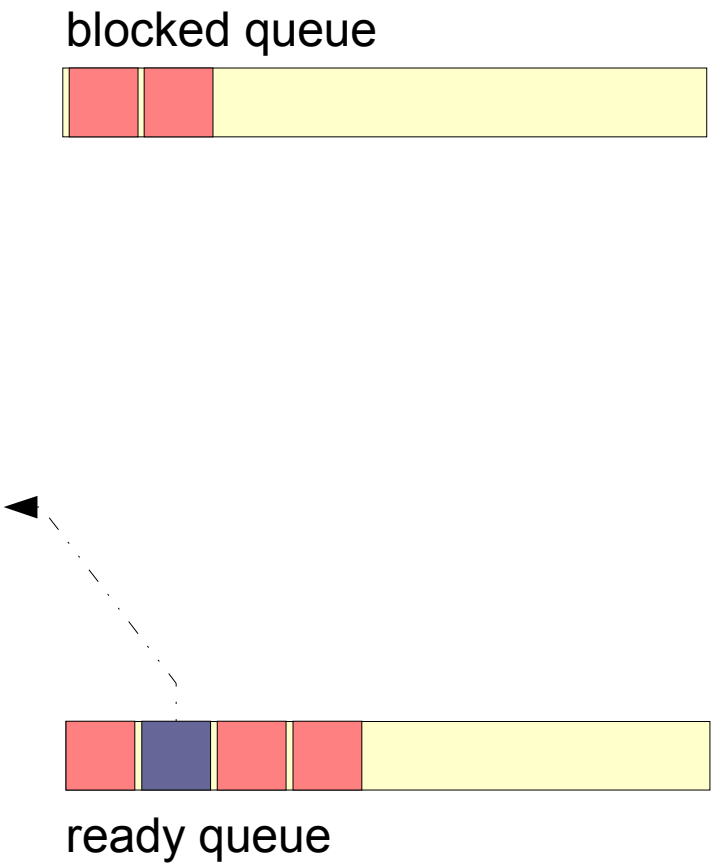
Boa: request processing loop

```
for each ready request {
    result = work_on(request);
    accept new connections
    if ( result == BLOCK )
        block(request);
    if ( result == FINISHED )
        free(request);
    else keep it ready;
}
```



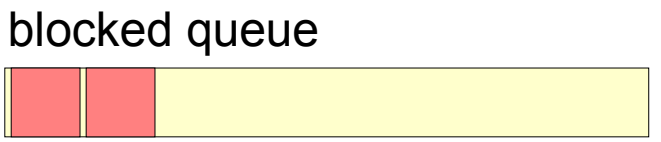
Boa: request processing loop

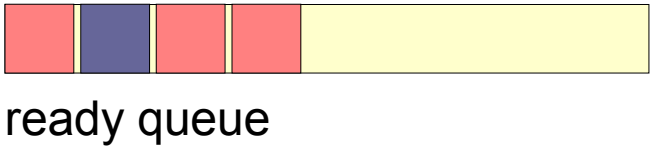
```
for each ready request {
    result = work_on(request);
    accept new connections
    if ( result == BLOCK )
        block(request);
    if ( result == FINISHED )
        free(request);
    else keep it ready;
}
```



Boa: request processing loop

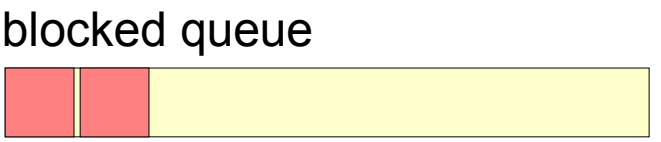
```
for each ready request {
    result = work_on(request);
    accept new connections
    if ( result == BLOCK )
        block(request);
    if ( result == FINISHED )
        free(request);
    else keep it ready;
}
```

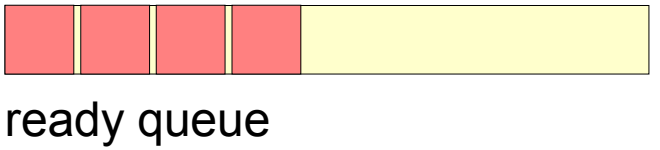




Boa: main loop

```
for ( ; ; ) {
    process signals
    unblock requests
    accept new connections
    process ready requests
    select();
}
```





Parallelization ...

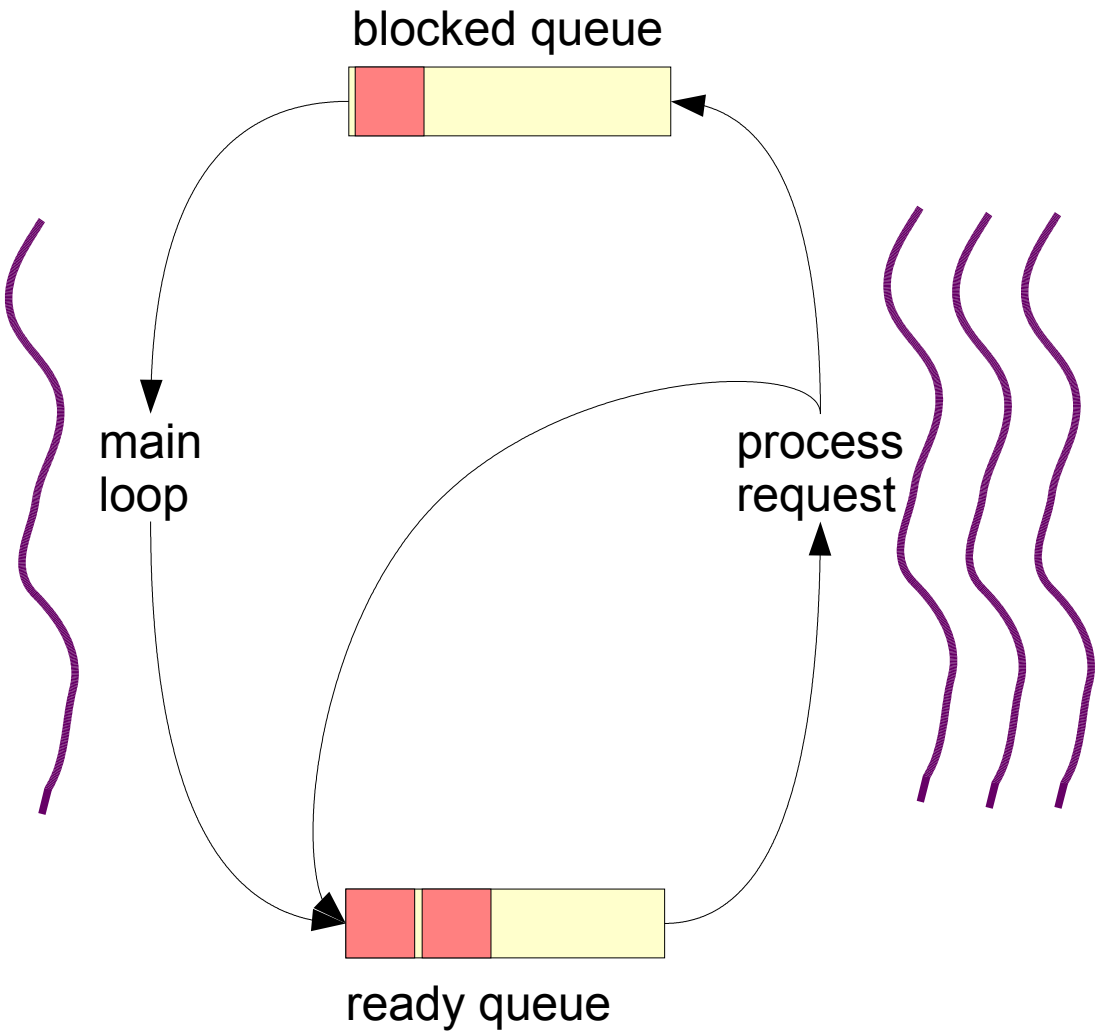
- Sources of parallelism
  - Computation of each request in parallel
  - Different tasks in parallel
    - serving requests
    - accepting new connections

Parallelization ...

- Common issues
  - Critical access was required for
    - global variables
    - manipulation of queues
    - access to the open files cache
    - server log files
  - A lot of static variables
    - false per-thread global variables
    - changed to an extra parameter

.... with pthreads

- Schema
  - One producer
  - N-1 consumers
- mutex locks for critical accesses

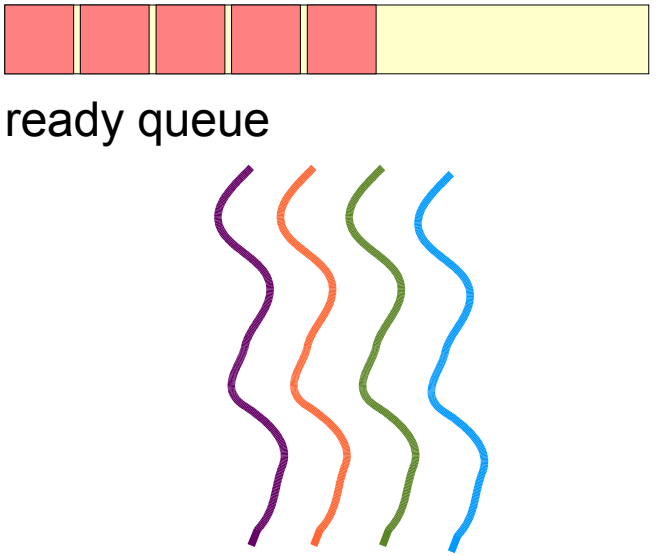


... with OpenMP

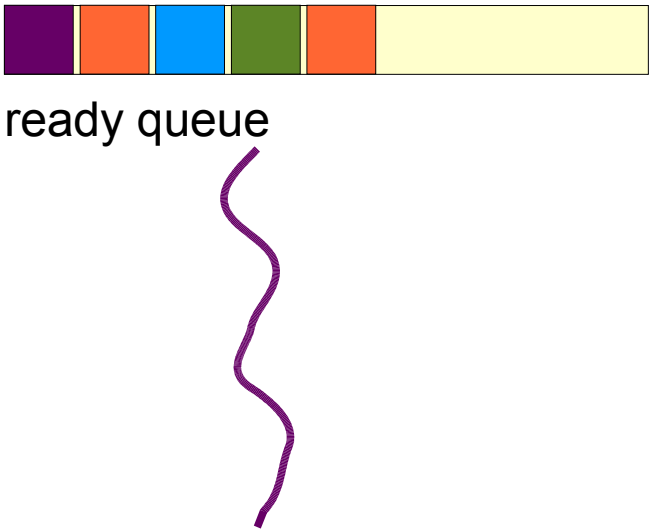
- Producer-consumer not easy in OpenMP
- Request processing loop parallelized
  - Needs to maintain the same number of elements inside the workshare
    - Splitted in two
  - Unbounded loop
    - do workshare cannot be used
    - single workshare with nowait used
- Critical sections and OpenMP locks used for critical accesses

... with OpenMP (II)

```
#pragma omp parallel
{
        for each request in the ready queue
#pragma omp single nowait
                request.result = work_on(request)
}
```




```
for each request in the ready queue
{
        if ( request.result == BLOCK ) block(request)
        else if ( request.result == FINISHED) free(request)
        else keep it in the queue
}
```



... with dynamic sections

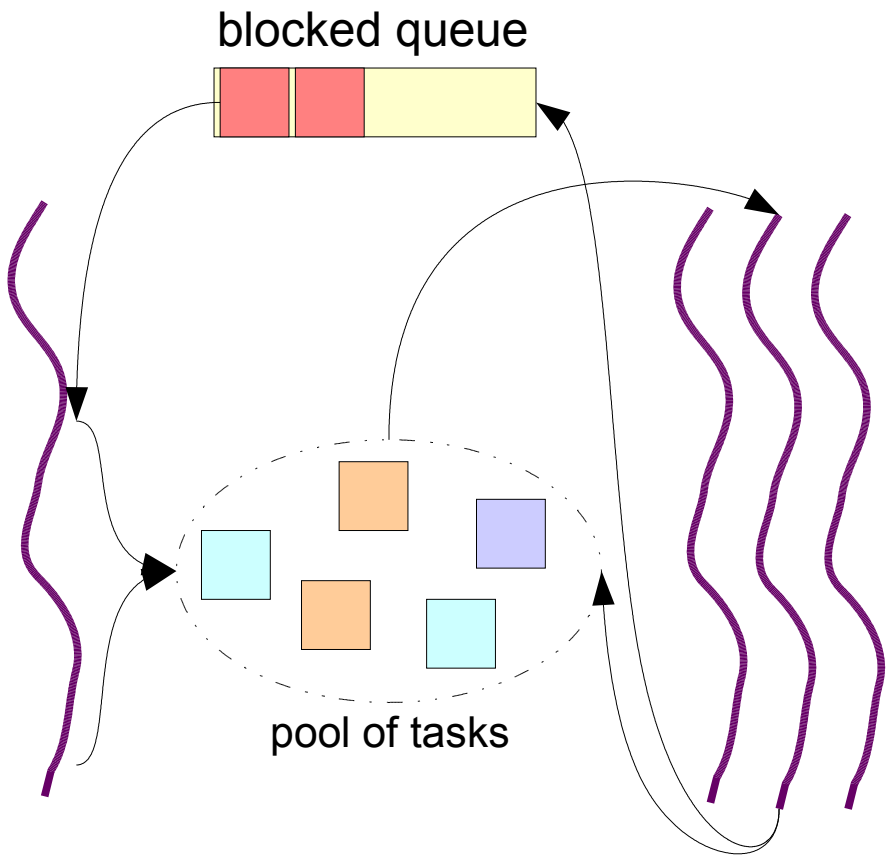
- Could be done without managing requests at application level?
- Available parallelism can be seen as collection of tasks
  - Dynamic sections can be used to express it

Dynamic sections

- Dynamic sections
  - A single thread executes the serial code
    - which can be seen as an implicit section too
  - Parallel tasks are created with section directives
  - Any thread can create new work
    - nesting of SECTION directive
  - Tasks are executed by any available thread

... with dynamic sections (II)

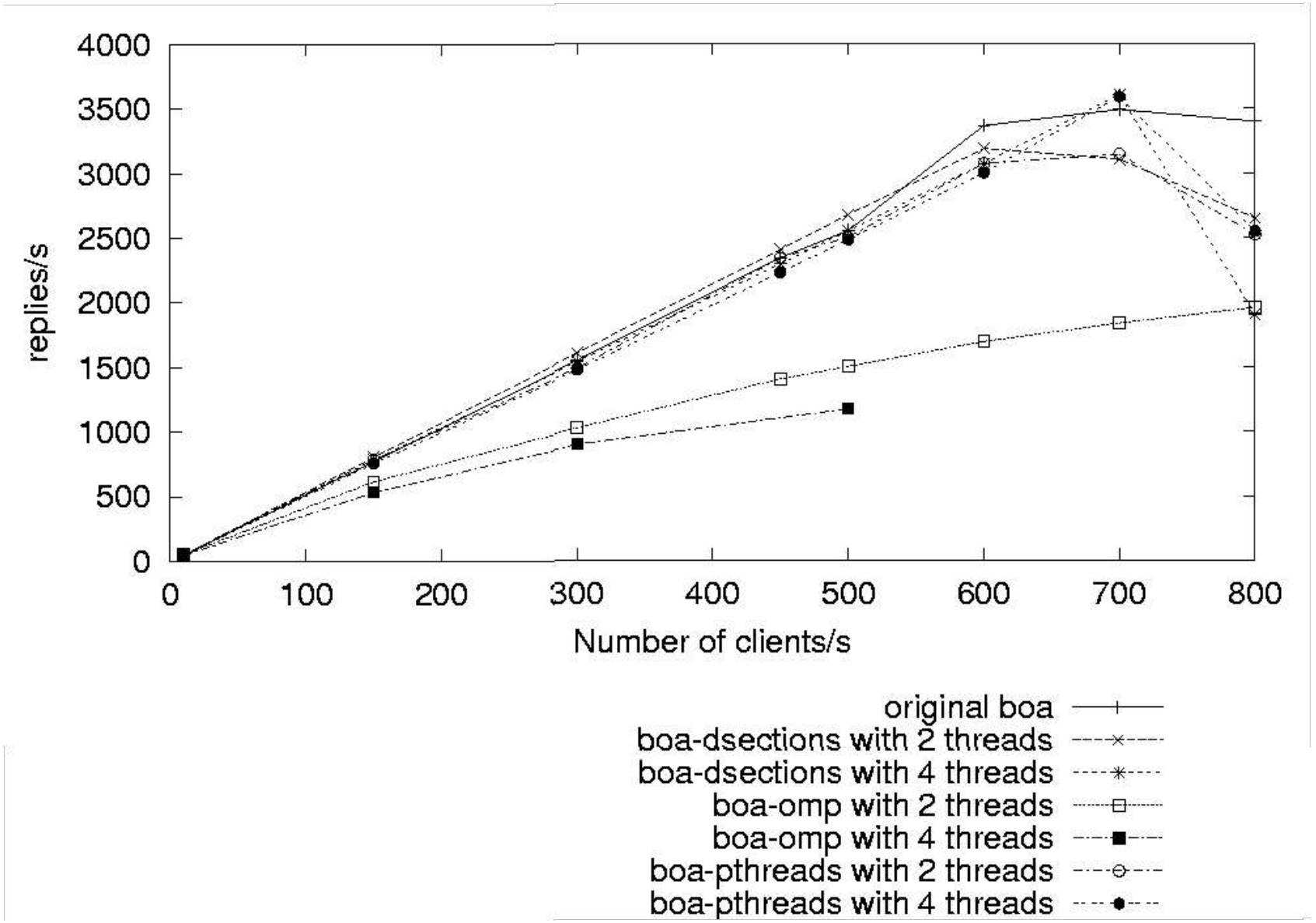
```
#pragma omp parallel
#pragma omp sections dynamic
while (1)  {
        foreach request in the blocked queue
                if ( dependences are met )
#pragma omp section captureprivate(request)
                        work_on(request)
        if ( new connection ) {
                accept it
#pragma omp section captureprivate(request)
                        work_on(request)
        }
        select()
}
```



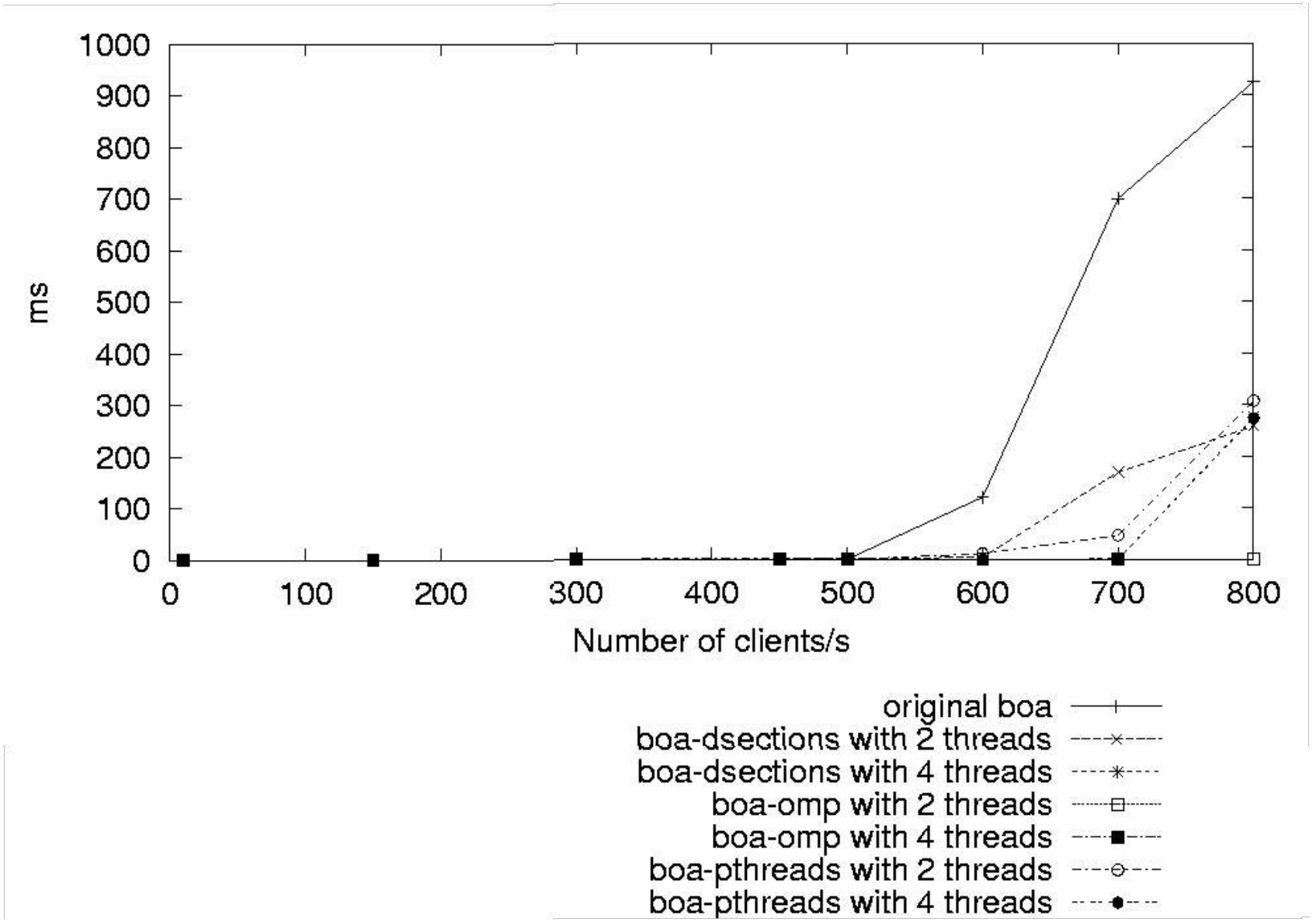
Evaluation

- Server: 4-wa Xeon at 1.4GHz with 2GB RAM
- Client: 2-way Xeon at 2.4GHz with 2GB RAM
- SO: Linux 2.6
- Network: Gigabit network
- Workload:
  - Surge workload
  - Static content requests with think time
  - Different loads of clients

Evaluation: Througput



Evaluation: Response time



Experiences

- Handling of static variables was a consuming tasks
  - tools can help
- Critical accesses
  - In general easier in OpenMP
  - But, when the same code applies to different data
    - Using locks, lock and unlock calls is as pthreads
    - Idea: have dynamically named critical sections
      - Example: #pragma omp critical (cache_lock[**i**])

Experiences (II)

- Pthread version
  - Much easier because of complex serial code
  - Overall effort: moderate
- OpenMP version
  - Much easier because of complex serial code
  - Few directives
  - Main difficulty: Correctness of single workshare
  - Reduction in performance because not enough parallelism was available

Experiences (III)

- Dynamic sections version
  - Simpler
    - did not use the request management of the serial version
  - Could easily handle different parallel tasks
    - pthreads code would grow in complexity
  - Good performance

Conclusions

- Web server could be parallelized with a handful of directives
  - but had bad performance
- Dynamic sections was also easier to use
  - matched pthreads performance

Future work

- Other web scenarios
  - SSL applications
  - Dynamic content
- Other applications
- Point-to-point synchronizations
  - wait/signal?